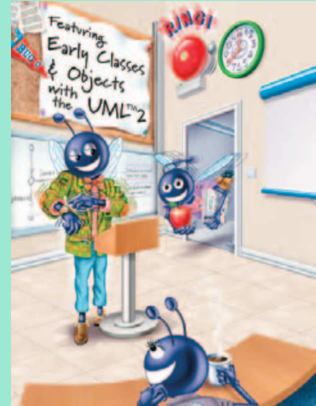
13 Object-Oriented Programming: Polymorphism



One Ring to rule them all,
One Ring to find them,
One Ring to bring them all
and in the darkness bind
them.

—John Ronald Reuel Tolkien

The silence often of pure
innocence
Persuades when speaking
fails.

—William Shakespeare

General propositions do not
decide concrete cases.

—Oliver Wendell Holmes

A philosopher of imposing
stature doesn't think in a
vacuum. Even his most
abstract ideas are, to some
extent, conditioned by what
is or is not known in the time
when he lives.

—Alfred North Whitehead

OBJECTIVES

In this chapter you will learn:

- What polymorphism is, how it makes programming more convenient, and how it makes systems more extensible and maintainable.
- To declare and use `virtual` functions to effect polymorphism.
- The distinction between abstract and concrete classes.
- To declare pure `virtual` functions to create abstract classes.
- How to use run-time type information (RTTI) with downcasting, `dynamic_cast`, `typeid` and `type_info`.
- How C++ implements `virtual` functions and dynamic binding "under the hood."
- How to use `virtual` destructors to ensure that all appropriate destructors run on an object.

Self-Review Exercises

13.1 Fill in the blanks in each of the following statements:

a) Treating a base-class object as a(n) ________ can cause errors.

ANS: derived-class object.

b) Polymorphism helps eliminate ________ logic.

ANS: `switch`.

c) If a class contains at least one pure `virtual` function, it is a(n) ________ class.

ANS: abstract.

d) Classes from which objects can be instantiated are called ________ classes.

ANS: concrete.

e) Operator ________ can be used to downcast base-class pointers safely.

ANS: `dynamic_cast`.

f) Operator `typeid` returns a reference to a(n) ________ object.

ANS: `type_info`.

g) ________ involves using a base-class pointer or reference to invoke `virtual` functions on base-class and derived-class objects.

ANS: Polymorphism.

h) Overridable functions are declared using keyword ________.

ANS: `virtual`.

i) Casting a base-class pointer to a derived-class pointer is called ________.

ANS: downcasting.

13.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) All `virtual` functions in an abstract base class must be declared as pure `virtual` functions.

ANS: False. An abstract base class can include virtual functions with implementations.

b) Referring to a derived-class object with a base-class handle is dangerous.

ANS: False. Referring to a base-class object with a derived-class handle is dangerous.

c) A class is made abstract by declaring that class `virtual`.

ANS: False. Classes are never declared `virtual`. Rather, a class is made abstract by including at least one pure virtual function in the class.

d) If a base class declares a pure `virtual` function, a derived class must implement that function to become a concrete class.

ANS: True.

e) Polymorphic programming can eliminate the need for `switch` logic.

ANS: True.

Exercises

13.3 How is it that polymorphism enables you to program "in the general" rather than "in the specific"? Discuss the key advantages of programming "in the general."

ANS: Polymorphism enables the programmer to concentrate on the common operations that are applied to objects of all the classes in a hierarchy. The general processing capabilities can be separated from any code that is specific to each class. Those general portions of the code can accommodate new classes without modification. In some polymorphic applications, only the code that creates the objects needs to be modified to extend the system with new classes.

13.4 Discuss the problems of programming with `switch` logic. Explain why polymorphism can be an effective alternative to using `switch` logic.

ANS: The main problems with programming using `switch` logic are extensibility and program maintainability. A program containing many `switch` statements is difficult to modify. A programmer will need to add or remove `case`s from many, possibly all, `switch` statements in the program as the types of objects in the program change. The polymorphic approach eliminates these issues, as new types can be added to a program with relative ease. Only the part of the program that creates objects needs to be aware of the newly added types. [*Note:* `switch` logic includes `if...else` statements which are more flexible than the `switch` statement.]

13.5 Distinguish between inheriting interface and inheriting implementation. How do inheritance hierarchies designed for inheriting interface differ from those designed for inheriting implementation?

ANS: When a class inherits implementation, it inherits previously defined functionality from another class. When a class inherits interface, it inherits the definition of what the interface to the new class type should be. The implementation is then provided by the programmer defining the new class type. Inheritance hierarchies designed for inheriting implementation are used to reduce the amount of new code that is being written. Such hierarchies are used to facilitate software reusability. Inheritance hierarchies designed for inheriting interface are used to write programs that perform generic processing of many class types. Such hierarchies are commonly used to facilitate software extensibility (i.e., new types can be added to the hierarchy without changing the generic processing capabilities of the program.)

13.6 What are `virtual` functions? Describe a circumstance in which `virtual` functions would be appropriate.

ANS: `virtual` functions are functions with the same function prototype that are defined throughout a class hierarchy. At least the base class occurrence of the function is preceded by the keyword `virtual`. Programmers use `virtual` function to enable generic processing of an entire class hierarchy of objects through a base-class pointer. If the program invokes a `virtual` function through a base-class pointer to a derived-class object, the program will choose the correct derived-class function dynamically (i.e., at execution time) based on the object type—not the pointer type. For example, in a shape hierarchy, all shapes can be drawn. If all shapes are derived from a base class `Shape` which contains a `virtual draw` function, then generic processing of the hierarchy can be performed by calling every shape's `draw` generically through a base class `Shape` pointer.

13.7 Distinguish between static binding and dynamic binding. Explain the use of `virtual` functions and the *vtable* in dynamic binding.

ANS: Static binding is performed at compile-time when a function is called via a specific object or via a pointer to an object. Dynamic binding is performed at run-time when a `virtual` function is called via a base-class pointer to a derived-class object (the object can be of any derived class). The `virtual` functions table (*vtable*) is used at run-time to enable the proper function to be called for the object to which the base-class pointer "points". Each class containing `virtual` functions has its own *vtable* that specifies where the `virtual` functions for that class are located. Every object of a class with `virtual` functions contains a hidden pointer to the class's *vtable*. When a `virtual` function is called via a base-class pointer, the hidden pointer is dereferenced to locate the *vtable*, then the *vtable* is searched for the proper function call.

13.8 Distinguish between `virtual` functions and pure `virtual` functions.

ANS: A `virtual` function must have a definition in the class in which it is declared. A pure `virtual` function does not provide a definition. A pure `virtual` function is appropriate when it does not make sense to provide an implementation for a function in a base class (i.e., some additional derived-class-specific data is required to implement the function in a meaningful manner). Classes derived directly from the abstract class must provide definitions for the inherited pure `virtual` functions to become a concrete class; otherwise, the derived class becomes an abstract class as well.

13.9 Suggest one or more levels of abstract base classes for the `Shape` hierarchy discussed in this chapter and shown in Fig. 12.3. (The first level is `Shape`, and the second level consists of the classes `TwoDimensionalShape` and `ThreeDimensionalShape`.)

ANS: In the `Shape` hierarchy, class `Shape` could be an abstract base class. In the second level of the hierarchy, `TwoDimensionalShape` and `ThreeDimensionalShape` could also be abstract base classes, as defining a function such as `draw` does not make sense for either a generic two-dimensional shape or a generic three-dimensional shape.

13.10 How does polymorphism promote extensibility?

ANS: Polymorphism makes programs more extensible by making all function calls generic. When a new class type with the appropriate `virtual` functions is added to the hierarchy, no changes need to be made to the generic function calls. Only client code that instantiates new objects must be modified to accommodate new types.

13.11 You have been asked to develop a flight simulator that will have elaborate graphical outputs. Explain why polymorphic programming would be especially effective for a problem of this nature.

ANS: A flight simulator most likely will involve displaying a number of different types of aircraft and objects on the screen. Suppose a programmer creates an abstract base class named `FlightSimulatorObject` with a pure `virtual` function `draw`. Derived classes could be created to represent the various on-screen elements, such as `Airplanes`, `Helicopters` and `Airports`. Each derived class would define `draw` so that it displays an appropriate image for that element. Polymorphism would allow the programmer to display all the on-screen elements in the flight simulator in a generic manner (i.e., the program can invoke function `draw` off base class `FlightSimulatorObject` pointers or references to derived-class objects. The flight simulator code responsible for displaying the elements on the screen would not need to worry about the details of drawing each type of element or about incorporating new types of on-screen elements. Each derived class handles the drawing details, and only the code that creates new objects would need to be modified to accommodate new types.

13.12 *(Payroll System Modification)* Modify the payroll system of Figs. 13.13–13.23 to include `private` data member `birthDate` in class `Employee`. Use class `Date` from Figs. 11.12–11.13 to represent an employee's birthday. Assume that payroll is processed once per month. Create a `vector` of `Employee` references to store the various employee objects. In a loop, calculate the payroll for each `Employee` (polymorphically), and add a $100.00 bonus to the person's payroll amount if the current month is the month in which the `Employee`'s birthday occurs.

ANS:

```
1   // Exercise 13.12 Solution: Date.h
2   // Date class definition.
3   #ifndef DATE_H
4   #define DATE_H
5
6   #include <iostream>
```

```
 7  using std::ostream;
 8
 9  class Date
10  {
11     friend ostream &operator<<( ostream &, const Date & );
12  public:
13     Date( int m = 1, int d = 1, int y = 1900 ); // default constructor
14     void setDate( int, int, int ); // set month, day, year
15     Date &operator++(); // prefix increment operator
16     Date operator++( int ); // postfix increment operator
17     const Date &operator+=( int ); // add days, modify object
18     bool leapYear( int ) const; // is date in a leap year?
19     bool endOfMonth( int ) const; // is date at the end of month?
20     int getMonth() const; // return the month of the date
21  private:
22     int month;
23     int day;
24     int year;
25
26     static const int days[]; // array of days per month
27     void helpIncrement(); // utility function for incrementing date
28  }; // end class Date
29
30  #endif
```

```
 1  // Exercise 13.12 Solution: Date.cpp
 2  // Date class member-function definitions.
 3  #include <iostream>
 4  #include "Date.h"
 5
 6  // initialize static member at file scope; one classwide copy
 7  const int Date::days[] =
 8     { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
 9
10  // Date constructor
11  Date::Date( int m, int d, int y )
12  {
13     setDate( m, d, y );
14  } // end Date constructor
15
16  // set month, day and year
17  void Date::setDate( int mm, int dd, int yy )
18  {
19     month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
20     year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;
21
22     // test for a leap year
23     if ( month == 2 && leapYear( year ) )
24        day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
25     else
26        day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
27  } // end function setDate
28
29  // overloaded prefix increment operator
```

```
30   Date &Date::operator++()
31   {
32      helpIncrement(); // increment date
33      return *this; // reference return to create an lvalue
34   } // end function operator++
35
36   // overloaded postfix increment operator; note that the
37   // dummy integer parameter does not have a parameter name
38   Date Date::operator++( int )
39   {
40      Date temp = *this; // hold current state of object
41      helpIncrement();
42
43      // return unincremented, saved, temporary object
44      return temp; // value return; not a reference return
45   } // end function operator++
46
47   // add specified number of days to date
48   const Date &Date::operator+=( int additionalDays )
49   {
50      for ( int i = 0; i < additionalDays; i++ )
51         helpIncrement();
52
53      return *this; // enables cascading
54   } // end function operator+=
55
56   // if the year is a leap year, return true; otherwise, return false
57   bool Date::leapYear( int testYear ) const
58   {
59      if ( testYear % 400 == 0 ||
60         ( testYear % 100 != 0 && testYear % 4 == 0 ) )
61         return true; // a leap year
62      else
63         return false; // not a leap year
64   } // end function leapYear
65
66   // return the month of the date
67   int Date::getMonth() const
68   {
69      return month;
70   } // end function getMonth
71
72   // determine whether the day is the last day of the month
73   bool Date::endOfMonth( int testDay ) const
74   {
75      if ( month == 2 && leapYear( year ) )
76         return testDay == 29; // last day of Feb. in leap year
77      else
78         return testDay == days[ month ];
79   } // end function endOfMonth
80
81   // function to help increment the date
82   void Date::helpIncrement()
83   {
84      // day is not end of month
```

```
85     if ( !endOfMonth( day ) )
86        day++; // increment day
87     else
88        if ( month < 12 ) // day is end of month and month < 12
89        {
90           month++; // increment month
91           day = 1; // first day of new month
92        } // end if
93        else // last day of year
94        {
95           year++; // increment year
96           month = 1; // first month of new year
97           day = 1; // first day of new month
98        } // end else
99  } // end function helpIncrement
100
101 // overloaded output operator
102 ostream &operator<<( ostream &output, const Date &d )
103 {
104    static char *monthName[ 13 ] = { "", "January", "February",
105       "March", "April", "May", "June", "July", "August",
106       "September", "October", "November", "December" };
107    output << monthName[ d.month ] << ' ' << d.day << ", " << d.year;
108    return output; // enables cascading
109 } // end function operator<<
```

```
1  // Exercise 13.12 Solution: 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 std::string;
8
9  #include "Date.h" // Date class definition
10
11 class Employee
12 {
13 public:
14    Employee( const string &, const string &, const string &,
15       int, int, int );
16
17    void setFirstName( const string & ); // set first name
18    string getFirstName() const; // return first name
19
20    void setLastName( const string & ); // set last name
21    string getLastName() const; // return last name
22
23    void setSocialSecurityNumber( const string & ); // set SSN
24    string getSocialSecurityNumber() const; // return SSN
25
26    void setBirthDate( int, int, int ); // set birthday
27    Date getBirthDate() const; // return birthday
28
```

```
29     // pure virtual function makes Employee abstract base class
30     virtual double earnings() const = 0; // pure virtual
31     virtual void print() const; // virtual
32  private:
33     string firstName;
34     string lastName;
35     string socialSecurityNumber;
36     Date birthDate; // the Employee's birthday
37  }; // end class Employee
38
39  #endif // EMPLOYEE_H
```

```
1   // Exercise 13.12 Solution: Employee.cpp
2   // Abstract-base-class Employee member-function definitions.
3   // Note: No definitions are given for pure virtual functions.
4   #include <iostream>
5   using std::cout;
6
7   #include "Employee.h" // Employee class definition
8
9   // constructor
10  Employee::Employee( const string &first, const string &last,
11     const string &ssn, int month, int day, int year )
12     : firstName( first ), lastName( last ), socialSecurityNumber( ssn ),
13       birthDate( month, day, year )
14  {
15     // empty body
16  } // end Employee constructor
17
18  // set first name
19  void Employee::setFirstName( const string &first )
20  {
21     firstName = first;
22  } // end function setFirstName
23
24  // return first name
25  string Employee::getFirstName() const
26  {
27     return firstName;
28  } // end function getFirstName
29
30  // set last name
31  void Employee::setLastName( const string &last )
32  {
33     lastName = last;
34  } // end function setLastName
35
36  // return last name
37  string Employee::getLastName() const
38  {
39     return lastName;
40  } // end function getLastName
41
42  // set social security number
```

```
43  void Employee::setSocialSecurityNumber( const string &ssn )
44  {
45     socialSecurityNumber = ssn; // should validate
46  } // end function setSocialSecurityNumber
47
48  // return social security number
49  string Employee::getSocialSecurityNumber() const
50  {
51     return socialSecurityNumber;
52  } // end function getSocialSecurityNumber
53
54  // set birthday
55  void Employee::setBirthDate( int month, int day, int year )
56  {
57     birthDate.setDate( month, day, year );
58  } // end function setBirthDate
59
60  // return birthday
61  Date Employee::getBirthDate() const
62  {
63     return birthDate;
64  } // end function getBirthDate
65
66  // print Employee's information (virtual, but not pure virtual)
67  void Employee::print() const
68  {
69     cout << getFirstName() << ' ' << getLastName()
70        << "\nbirthday: " << getBirthDate()
71        << "\nsocial security number: " << getSocialSecurityNumber();
72  } // end function print
```

```
1   // Exercise 13.12 Solution: 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 &, int, int, int, 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
```

```
1  // Exercise 13.12 Solution: SalariedEmployee.cpp
2  // SalariedEmployee class member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "SalariedEmployee.h" // SalariedEmployee class definition
7
8  // constructor
9  SalariedEmployee::SalariedEmployee( const string &first,
10    const string &last, const string &ssn, int month, int day, int year,
11    double salary )
12    : Employee( first, last, ssn, month, day, year )
13 {
14    setWeeklySalary( salary );
15 } // end SalariedEmployee constructor
16
17 // set salary
18 void SalariedEmployee::setWeeklySalary( double salary )
19 {
20    weeklySalary = ( salary < 0.0 ) ? 0.0 : salary;
21 } // end function setWeeklySalary
22
23 // return salary
24 double SalariedEmployee::getWeeklySalary() const
25 {
26    return weeklySalary;
27 } // end function getWeeklySalary
28
29 // calculate earnings;
30 // override pure virtual function earnings in Employee
31 double SalariedEmployee::earnings() const
32 {
33    return getWeeklySalary();
34 } // end function earnings
35
36 // print SalariedEmployee's information
37 void SalariedEmployee::print() const
38 {
39    cout << "salaried employee: ";
40    Employee::print(); // reuse abstract base-class print function
41    cout << "\nweekly salary: " << getWeeklySalary();
42 } // end function print
```

```
1  // Exercise 13.12 Solution: 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     HourlyEmployee( const string &, const string &,
12        const string &, int, int, int, double = 0.0, double = 0.0 );
13  
14     void setWage( double ); // set hourly wage
15     double getWage() const; // return hourly wage
16  
17     void setHours( double ); // set hours worked
18     double getHours() const; // return hours worked
19  
20     // keyword virtual signals intent to override
21     virtual double earnings() const; // calculate earnings
22     virtual void print() const; // print HourlyEmployee object
23  private:
24     double wage; // wage per hour
25     double hours; // hours worked for week
26  }; // end class HourlyEmployee
27  
28  #endif // HOURLY_H
```

```
1   // Exercise 13.12 Solution: HourlyEmployee.cpp
2   // HourlyEmployee class member-function definitions.
3   #include <iostream>
4   using std::cout;
5   
6   #include "HourlyEmployee.h" // HourlyEmployee class definition
7   
8   // constructor
9   HourlyEmployee::HourlyEmployee( const string &first, const string &last,
10     const string &ssn, int month, int day, int year,
11     double hourlyWage, double hoursWorked )
12     : Employee( first, last, ssn, month, day, year )
13  {
14     setWage( hourlyWage ); // validate hourly wage
15     setHours( hoursWorked ); // validate hours worked
16  } // end HourlyEmployee constructor
17  
18  // set wage
19  void HourlyEmployee::setWage( double hourlyWage )
20  {
21     wage = ( hourlyWage < 0.0 ? 0.0 : hourlyWage );
22  } // end function setWage
23  
24  // return wage
25  double HourlyEmployee::getWage() const
26  {
27     return wage;
28  } // end function getWage
29  
30  // set hours worked
31  void HourlyEmployee::setHours( double hoursWorked )
32  {
33     hours = ( ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) ) ?
```

```
34          hoursWorked : 0.0 );
35    } // end function setHours
36
37    // return hours worked
38    double HourlyEmployee::getHours() const
39    {
40       return hours;
41    } // end function getHours
42
43    // calculate earnings;
44    // override pure virtual function earnings in Employee
45    double HourlyEmployee::earnings() const
46    {
47       if ( getHours() <= 40 ) // no overtime
48          return getWage() * getHours();
49       else
50          return 40 * getWage() + ( ( getHours() - 40 ) * getWage() * 1.5 );
51    } // end function earnings
52
53    // print HourlyEmployee's information
54    void HourlyEmployee::print() const
55    {
56       cout << "hourly employee: ";
57       Employee::print(); // code reuse
58       cout << "\nhourly wage: " << getWage() <<
59          "; hours worked: " << getHours();
60    } // end function print
```

```
 1    // Exercise 13.12 Solution: 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 &, int, int, int, 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
23    private:
24       double grossSales; // gross weekly sales
25       double commissionRate; // commission percentage
26    }; // end class CommissionEmployee
```

```
27
28 #endif // COMMISSION_H
```

```
1  // Exercise 13.12 Solution: CommissionEmployee.cpp
2  // CommissionEmployee class member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8  // constructor
9  CommissionEmployee::CommissionEmployee( const string &first,
10    const string &last, const string &ssn, int month, int day, int year,
11    double sales, double rate )
12    : Employee( first, last, ssn, month, day, year )
13 {
14    setGrossSales( sales );
15    setCommissionRate( rate );
16 } // end CommissionEmployee constructor
17
18 // set commission rate
19 void CommissionEmployee::setCommissionRate( double rate )
20 {
21     commissionRate = ( ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0 );
22 } // end function setCommissionRate
23
24 // return commission rate
25 double CommissionEmployee::getCommissionRate() const
26 {
27     return commissionRate;
28 } // end function getCommissionRate
29
30 // set gross sales amount
31 void CommissionEmployee::setGrossSales( double sales )
32 {
33    grossSales = ( ( sales < 0.0 ) ? 0.0 : sales );
34 } // end function setGrossSales
35
36 // return gross sales amount
37 double CommissionEmployee::getGrossSales() const
38 {
39     return grossSales;
40 } // end function getGrossSales
41
42 // calculate earnings;
43 // override pure virtual function earnings in Employee
44 double CommissionEmployee::earnings() const
45 {
46    return getCommissionRate() * getGrossSales();
47 } // end function earnings
48
49 // print CommissionEmployee's information
50 void CommissionEmployee::print() const
51 {
```

```
52       cout << "commission employee: ";
53       Employee::print(); // code reuse
54       cout << "\ngross sales: " << getGrossSales()
55          << "; commission rate: " << getCommissionRate();
56    } // end function print
```

```
1    // Exercise 13.12 Solution: BasePlusCommissionEmployee.h
2    // BasePlusCommissionEmployee class derived from Employee.
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 &, int, int, int, double = 0.0, double = 0.0,
13         double = 0.0 );
14
15      void setBaseSalary( double ); // set base salary
16      double getBaseSalary() const; // return base salary
17
18      // keyword virtual signals intent to override
19      virtual double earnings() const; // calculate earnings
20      virtual void print() const; // print BasePlusCommissionEmployee object
21   private:
22      double baseSalary; // base salary per week
23   }; // end class BasePlusCommissionEmployee
24
25   #endif // BASEPLUS_H
```

```
1    // Exercise 13.12 Solution: BasePlusCommissionEmployee.cpp
2    // BasePlusCommissionEmployee member-function definitions.
3    #include <iostream>
4    using std::cout;
5
6    // BasePlusCommissionEmployee class definition
7    #include "BasePlusCommissionEmployee.h"
8
9    // constructor
10   BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11      const string &first, const string &last, const string &ssn,
12      int month, int day, int year, double sales,
13      double rate, double salary )
14      : CommissionEmployee( first, last, ssn, month, day, year, sales, rate )
15   {
16      setBaseSalary( salary ); // validate and store base salary
17   } // end BasePlusCommissionEmployee constructor
18
19   // set base salary
20   void BasePlusCommissionEmployee::setBaseSalary( double salary )
21   {
```

```
22      baseSalary = ( ( salary < 0.0 ) ? 0.0 : salary );
23   } // end function setBaseSalary
24
25   // return base salary
26   double BasePlusCommissionEmployee::getBaseSalary() const
27   {
28       return baseSalary;
29   } // end function getBaseSalary
30
31   // calculate earnings;
32   // override pure virtual function earnings in Employee
33   double BasePlusCommissionEmployee::earnings() const
34   {
35       return getBaseSalary() + CommissionEmployee::earnings();
36   } // end function earnings
37
38   // print BasePlusCommissionEmployee's information
39   void BasePlusCommissionEmployee::print() const
40   {
41      cout << "base-salaried ";
42      CommissionEmployee::print(); // code reuse
43      cout << "; base salary: " << getBaseSalary();
44   } // end function print
```

```
1    // Exercise 13.12 Solution: ex13_12.cpp
2    // Processing Employee derived-class objects polymorphically.
3    #include <iostream>
4    using std::cout;
5    using std::endl;
6    using std::fixed;
7
8    #include <iomanip>
9    using std::setprecision;
10
11   #include <vector>
12   using std::vector;
13
14   #include <typeinfo>
15
16   #include <ctime>
17   using std::time_t;
18   using std::time;
19   using std::localtime;
20   using std::strftime;
21
22   #include <cstdlib>
23   using std::atoi;
24
25   // include definitions of classes in Employee hierarchy
26   #include "Employee.h"
27   #include "SalariedEmployee.h"
28   #include "HourlyEmployee.h"
29   #include "CommissionEmployee.h"
30   #include "BasePlusCommissionEmployee.h"
```

```
31
32  int determineMonth(); // prototype of function that returns current month
33
34  int main()
35  {
36     // set floating-point output formatting
37     cout << fixed << setprecision( 2 );
38
39     // create vector of four base-class pointers
40     vector < Employee * > employees( 4 );
41
42     // initialize vector with Employees
43     employees[ 0 ] = new SalariedEmployee(
44        "John", "Smith", "111-11-1111", 6, 15, 1944, 800 );
45     employees[ 1 ] = new HourlyEmployee(
46        "Karen", "Price", "222-22-2222", 12, 29, 1960, 16.75, 40 );
47     employees[ 2 ] = new CommissionEmployee(
48        "Sue", "Jones", "333-33-3333", 9, 8, 1954, 10000, .06 );
49     employees[ 3 ] = new BasePlusCommissionEmployee(
50        "Bob", "Lewis", "444-44-4444", 3, 2, 1965, 5000, .04, 300 );
51
52     int month = determineMonth();
53
54     cout << "Employees processed polymorphically via dynamic binding:\n\n";
55
56     for ( size_t i = 0; i < employees.size(); i++ )
57     {
58        employees[ i ]->print(); // output employee information
59        cout << endl;
60
61        // downcast pointer
62        BasePlusCommissionEmployee *derivedPtr =
63           dynamic_cast < BasePlusCommissionEmployee * >
64              ( employees[ i ] );
65
66        // determine whether element points to base-salaried
67        // commission employee
68        if ( derivedPtr != 0 ) // 0 if not a BasePlusCommissionEmployee
69        {
70           double oldBaseSalary = derivedPtr->getBaseSalary();
71           cout << "old base salary: $" << oldBaseSalary << endl;
72           derivedPtr->setBaseSalary( 1.10 * oldBaseSalary );
73           cout << "new base salary with 10% increase is: $"
74              << derivedPtr->getBaseSalary() << endl;
75        } // end if
76
77        // get current employee's birthday
78        Date birthday = employees[ i ]->getBirthDate();
79
80        // if current month is employee's birthday month, add $100 to salary
81        if ( birthday.getMonth() == month )
82           cout << "HAPPY BIRTHDAY!\nearned $"
83              << ( employees[ i ]->earnings() + 100.0 ) << endl;
84        else
85           cout << "earned $" << employees[ i ]->earnings() << endl;
```

```
86
87          cout << endl;
88       } // end for
89
90       // release objects pointed to by vector's elements
91       for ( size_t j = 0; j < employees.size(); j++ )
92       {
93          // output class name
94          cout << "deleting object of "
95             << typeid( *employees[ j ] ).name() << endl;
96
97          delete employees[ j ];
98       } // end for
99
100      return 0;
101   } // end main
102
103   // Determine the current month using standard library functions of ctime
104   int determineMonth()
105   {
106      time_t currentTime;
107      char monthString[ 3 ];
108      time( &currentTime );
109      strftime( monthString, 3, "%m", localtime( &currentTime ) );
110      return atoi( monthString );
111   } // end function determineMonth
```

```
Employees processed polymorphically via dynamic binding:

salaried employee: John Smith
birthday: June 15, 1944
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
birthday: December 29, 1960
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
HAPPY BIRTHDAY!
earned $770.00

commission employee: Sue Jones
birthday: September 8, 1954
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
birthday: March 2, 1965
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
```

13.13 *(Shape Hierarchy)* Implement the `Shape` hierarchy designed in Exercise 12.7 (which is based on the hierarchy in Fig. 12.3). Each `TwoDimensionalShape` should contain function `getArea` to calculate the area of the two-dimensional shape. Each `ThreeDimensionalShape` should have member functions `getArea` and `getVolume` to calculate the surface area and volume of the three-dimensional shape, respectively. Create a program that uses a `vector` of `Shape` pointers to objects of each concrete class in the hierarchy. The program should print the object to which each `vector` element points. Also, in the loop that processes all the shapes in the `vector`, determine whether each shape is a `TwoDimensionalShape` or a `ThreeDimensionalShape`. If a shape is a `TwoDimensionalShape`, display its area. If a shape is a `ThreeDimensionalShape`, display its area and volume.

ANS:

```
1  // Exercise 13.13 Solution: Shape.h
2  // Definition of base-class Shape
3  #ifndef SHAPE_H
4  #define SHAPE_H
5
6  #include <iostream>
7  using std::ostream;
8
```

```
 9  class Shape {
10     friend ostream & operator<<( ostream &, Shape & );
11  public:
12     Shape( double = 0.0, double = 0.0 ); // default constructor
13     double getCenterX() const; // return x from coordinate pair
14     double getCenterY() const; // return y from coordinate pair
15     virtual void print() const = 0; // output Shape object
16  protected:
17     double xCenter; // x part of coordinate pair
18     double yCenter; // y part of coordinate pair
19  }; // end class Shape
20
21  #endif
```

```
 1  // Exercise 13.13 Solution: Shape.cpp
 2  // Member and friend definitions for class Shape
 3
 4  #include "Shape.h"
 5
 6  // default constructor
 7  Shape::Shape( double x, double y )
 8  {
 9     xCenter = x;
10     yCenter = y;
11  } // end Shape constructor
12
13  // return x from coordinate pair
14  double Shape::getCenterX() const
15  {
16     return xCenter;
17  } // end function getCenterX
18
19  // return y from coordinate pair
20  double Shape::getCenterY() const
21  {
22     return yCenter;
23  } // end function getCenterY
24
25  // overloaded output operator
26  ostream & operator<<( ostream &out, Shape &s )
27  {
28     s.print();
29     return out;
30  } // end overloaded output operator function
```

```
 1  // Exercise 13.13 Solution: TwoDimensionalShape.h
 2  // Definition of class TwoDimensionalShape
 3  #ifndef TWODIM_H
 4  #define TWODIM_H
 5
 6  #include "Shape.h"
 7
 8  class TwoDimensionalShape : public Shape
```

```
9   {
10  public:
11     // default constructor
12     TwoDimensionalShape( double x, double y ) : Shape( x, y ) { }
13
14     virtual double getArea() const = 0; // area of TwoDimensionalShape
15  }; // end class TwoDimensionalShape
16
17  #endif
```

```
1   // Exercise 13.13 Solution: ThreeDimensionalShape.h
2   // Definition of class ThreeDimensionalShape
3   #ifndef THREEDIM_H
4   #define THREEDIM_H
5
6   #include "Shape.h"
7
8   class ThreeDimensionalShape : public Shape
9   {
10  public:
11     // default constructor
12     ThreeDimensionalShape( double x, double y ) : Shape( x, y ) { }
13
14     virtual double getArea() const = 0; // area of 3-dimensional shape
15     virtual double getVolume() const = 0; // volume of 3-dimensional shape
16  }; // end class ThreeDimensionalShape
17
18  #endif
```

```
1   // Exercise 13.13 Solution: Circle.h
2   // Definition of class Circle
3   #ifndef CIRCLE_H
4   #define CIRCLE_H
5
6   #include "TwoDimensionalShape.h"
7
8   class Circle : public TwoDimensionalShape
9   {
10  public:
11     // default consturctor
12     Circle( double = 0.0, double = 0.0, double = 0.0 );
13
14     virtual double getRadius() const; // return radius
15     virtual double getArea() const; // return area
16     void print() const; // output Circle object
17  private:
18     double radius; // Circle's radius
19  }; // end class Circle
20
21  #endif
```

```
1  // Exercise 13.13 Solution: Circle.cpp
2  // Member-function definitions for class Circle
3  #include <iostream>
4  using std::cout;
5
6  #include "Circle.h"
7
8  // default constructor
9  Circle::Circle( double r, double x, double y )
10    : TwoDimensionalShape( x, y )
11 {
12    radius = ( ( r > 0.0 ) ? r : 0.0 );
13 } // end Circle constructor
14
15 // return radius of circle object
16 double Circle::getRadius() const
17 {
18    return radius;
19 } // end function getRadius
20
21 // return area of circle object
22 double Circle::getArea() const
23 {
24    return 3.14159 * radius * radius;
25 } // end function getArea
26
27 // output circle object
28 void Circle::print() const
29 {
30    cout << "Circle with radius " << radius << "; center at ("
31       << xCenter << ", " << yCenter << ")";
32 } // end function print
```

```
1  // Exercise 13.13 Solution: Square.h
2  // Definition of class Square
3  #ifndef SQUARE_H
4  #define SQUARE_H
5
6  #include "TwoDimensionalShape.h"
7
8  class Square : public TwoDimensionalShape
9  {
10 public:
11    // default constructor
12    Square( double = 0.0, double = 0.0, double = 0.0 );
13
14    virtual double getSideLength() const; // return lenght of sides
15    virtual double getArea() const; // return area of Square
16    void print() const; // output Square object
17 private:
18    double sideLength; // length of sides
19 }; // end class Square
20
```

```
21 #endif
```

```
1  // Exercise 13.13 Solution: Square.cpp
2  // Member-function definitions for class Square
3  #include <iostream>
4  using std::cout;
5
6  #include "Square.h"
7
8  // default constructor
9  Square::Square( double s, double x, double y )
10    : TwoDimensionalShape( x, y )
11 {
12    sideLength = ( ( s > 0.0 ) ? s : 0.0 );
13 } // end Square constructor
14
15 // return side of Square
16 double Square::getSideLength() const
17 {
18    return sideLength;
19 } // end function getSideLength
20
21 // return area of Square
22 double Square::getArea() const
23 {
24    return sideLength * sideLength;
25 } // end function getArea
26
27 // output Square object
28 void Square::print() const
29 {
30    cout << "Square with side length " << sideLength << "; center at ("
31       << xCenter << ", " << yCenter << ")";
32 } // end function print
```

```
1  // Exercise 13.13 Solution: Sphere.h
2  // Definition of class Sphere
3  #ifndef SPHERE_H
4  #define SPHERE_H
5
6  #include "ThreeDimensionalShape.h"
7
8  class Sphere : public ThreeDimensionalShape
9  {
10 public:
11    // default constructor
12    Sphere( double = 0.0, double = 0.0, double = 0.0 );
13
14    virtual double getArea() const; // return area of Sphere
15    virtual double getVolume() const; // return volume of Sphere
16    double getRadius() const; // return radius of Sphere
17    void print() const; // output Sphere object
18 private:
```

```
19       double radius; // radius of Sphere
20    }; // end class Sphere
21
22    #endif
```

```
 1    // Exercise 13.13 Solution: Sphere.cpp
 2    // Member-function definitions for class Sphere
 3    #include <iostream>
 4    using std::cout;
 5
 6    #include "Sphere.h"
 7
 8    // default constructor
 9    Sphere::Sphere( double r, double x, double y )
10       : ThreeDimensionalShape( x, y )
11    {
12       radius = ( ( r > 0.0 ) ? r : 0.0 );
13    } // end Sphere constructor
14
15    // return area of Sphere
16    double Sphere::getArea() const
17    {
18       return 4.0 * 3.14159 * radius * radius;
19    } // end function getArea
20
21    // return volume of Sphere
22    double Sphere::getVolume() const
23    {
24       return ( 4.0 / 3.0 ) * 3.14159 * radius * radius * radius;
25    } // end function getVolume
26
27    // return radius of Sphere
28    double Sphere::getRadius() const
29    {
30       return radius;
31    } // end function getRadius
32
33    // output Sphere object
34    void Sphere::print() const
35    {
36       cout << "Sphere with radius " << radius << "; center at ("
37          << xCenter << ", " << yCenter << ")";
38    } // end function print
```

```
 1    // Exercise 13.13 Solution: Cube.h
 2    // Definition of class Cube
 3    #ifndef CUBE_H
 4    #define CUBE_H
 5
 6    #include "ThreeDimensionalShape.h"
 7
 8    class Cube : public ThreeDimensionalShape
 9    {
```

```
10 public:
11    // default constructor
12    Cube( double = 0.0, double = 0.0, double = 0.0 );
13
14    virtual double getArea() const; // return area of Cube object
15    virtual double getVolume() const; // return volume of Cube object
16    double getSideLength() const; // return length of sides
17    void print() const; // output Cube object
18 private:
19    double sideLength; // length of sides of Cube
20 }; // end class Cube
21
22 #endif
```

```
1  // Exercise 13.13 Solution: Cube.cpp
2  // Member-function definitions for class Cube
3  #include <iostream>
4  using std::cout;
5
6  #include "Cube.h"
7
8  // default constructor
9  Cube::Cube( double s, double x, double y )
10    : ThreeDimensionalShape( x, y )
11 {
12    sideLength = ( ( s > 0.0 ) ? s : 0.0 );
13 } // end Cube constructor
14
15 // return area of Cube
16 double Cube::getArea() const
17 {
18    return 6 * sideLength * sideLength;
19 } // end function getArea
20
21 // return volume of Cube
22 double Cube::getVolume() const
23 {
24    return sideLength * sideLength * sideLength;
25 } // end function getVolume
26
27 // return length of sides
28 double Cube::getSideLength() const
29 {
30    return sideLength;
31 } // end function getSideLength
32
33 // output Cube object
34 void Cube::print() const
35 {
36    cout << "Cube with side length " << sideLength << "; center at ("
37       << xCenter << ", " << yCenter << ")";
38 } // end function print
```

```
1  // Exercise 13.13 Solution: ex13_13.cpp
2  // Driver to test Shape hierarchy
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <vector>
8  using std::vector;
9
10 #include <typeinfo>
11
12 #include "Shape.h"
13 #include "TwoDimensionalShape.h"
14 #include "ThreeDimensionalShape.h"
15 #include "Circle.h"
16 #include "Square.h"
17 #include "Sphere.h"
18 #include "Cube.h"
19
20 int main()
21 {
22    // create vector shapes
23    vector < Shape * > shapes( 4 );
24
25    // initialize vector with Shapes
26    shapes[ 0 ] = new Circle( 3.5, 6, 9 );
27    shapes[ 1 ] = new Square( 12, 2, 2 );
28    shapes[ 2 ] = new Sphere( 5, 1.5, 4.5 );
29    shapes[ 3 ] = new Cube( 2.2 );
30
31    // output Shape objects and display area and volume as appropriate
32    for ( int i = 0; i < 4; i++ )
33    {
34       cout << *( shapes[ i ] ) << endl;
35
36       // downcast pointer
37       TwoDimensionalShape *twoDimensionalShapePtr =
38          dynamic_cast < TwoDimensionalShape * > ( shapes[ i ] );
39
40       // if Shape is a TwoDimensionalShape, display its area
41       if ( twoDimensionalShapePtr != 0 )
42          cout << "Area: " << twoDimensionalShapePtr->getArea() << endl;
43
44       // downcast pointer
45       ThreeDimensionalShape *threeDimensionalShapePtr =
46          dynamic_cast < ThreeDimensionalShape * > ( shapes[ i ] );
47
48       // if Shape is a ThreeDimensionalShape, display its area and volume
49       if ( threeDimensionalShapePtr != 0 )
50          cout << "Area: " << threeDimensionalShapePtr->getArea()
51             << "\nVolume: " << threeDimensionalShapePtr->getVolume()
52             << endl;
53
54       cout << endl;
55    } // end for
```

```
56
57      return 0;
58   } // end main
```

```
Circle with radius 3.5; center at (6, 9)
Area: 38.4845

Square with side length 12; center at (2, 2)
Area: 144

Sphere with radius 5; center at (1.5, 4.5)
Area: 314.159
Volume: 523.598

Cube with side length 2.2; center at (0, 0)
Area: 29.04
Volume: 10.648
```

13.14 *(Polymorphic Screen Manager Using `Shape` Hierarchy)* Develop a basic graphics package. Use the `Shape` hierarchy implemented in Exercise 13.13. Limit yourself to two-dimensional shapes such as squares, rectangles, triangles and circles. Interact with the user. Let the user specify the position, size, shape and fill characters to be used in drawing each shape. The user can specify more than one of the same shape. As you create each shape, place a `Shape *` pointer to each new `Shape` object into an array. Each `Shape` class should now have its own `draw` member function. Write a polymorphic screen manager that walks through the array, sending `draw` messages to each object in the array to form a screen image. Redraw the screen image each time the user specifies an additional shape.

13.15 *(`Package` Inheritance Hierarchy)* Use the `Package` inheritance hierarchy created in Exercise 12.9 to create a program that displays the address information and calculates the shipping costs for several `Package`s. The program should contain a `vector` of `Package` pointers to objects of classes `TwoDayPackage` and `OvernightPackage`. Loop through the `vector` to process the `Package`s polymorphically. For each `Package`, invoke *get* functions to obtain the address information of the sender and the recipient, then print the two addresses as they would appear on mailing labels. Also, call each `Package`'s `calculateCost` member function and print the result. Keep track of the total shipping cost for all `Package`s in the `vector`, and display this total when the loop terminates.

ANS: [*Note:* To achieve polymorphic behavior in the `Package` hierarchy, each class definition must declare the `calculateCost` member function as a `virtual` function.]

```
1   // Exercise 13.15 Solution: Package.h
2   // Definition of base class Package.
3   #ifndef PACKAGE_H
4   #define PACKAGE_H
5
6   #include <string>
7   using std::string;
8
9   class Package
10  {
11  public:
12     // constructor initiliazes data members
13     Package( const string &, const string &, const string &,
14        const string &, int, const string &, const string &, const string &,
```

```
15        const string &, int, double, double );
16
17     void setSenderName( const string & ); // set sender's name
18     string getSenderName() const; // return sender's name
19     void setSenderAddress( const string & ); // set sender's address
20     string getSenderAddress() const; // return sender's address
21     void setSenderCity( const string & ); // set sender's city
22     string getSenderCity() const; // return sender's city
23     void setSenderState( const string & ); // set sender's state
24     string getSenderState() const; // return sender's state
25     void setSenderZIP( int ); // set sender's ZIP code
26     int getSenderZIP() const; // return sender's ZIP code
27     void setRecipientName( const string & ); // set recipient's name
28     string getRecipientName() const; // return recipient's name
29     void setRecipientAddress( const string & ); // set recipient's address
30     string getRecipientAddress() const; // return recipient's address
31     void setRecipientCity( const string & ); // set recipient's city
32     string getRecipientCity() const; // return recipient's city
33     void setRecipientState( const string & ); // set recipient's state
34     string getRecipientState() const; // return recipient's state
35     void setRecipientZIP( int ); // set recipient's ZIP code
36     int getRecipientZIP() const; // return recipient's ZIP code
37     void setWeight( double ); // validate and store weight
38     double getWeight() const; // return weight of package
39     void setCostPerOunce( double ); // validate and store cost per ounce
40     double getCostPerOunce() const; // return cost per ounce
41
42     virtual double calculateCost() const; // calculate shipping cost
43  private:
44     // data members to store sender and recipient's address information
45     string senderName;
46     string senderAddress;
47     string senderCity;
48     string senderState;
49     int senderZIP;
50     string recipientName;
51     string recipientAddress;
52     string recipientCity;
53     string recipientState;
54     int recipientZIP;
55
56     double weight; // weight of the package
57     double costPerOunce; // cost per ounce to ship the package
58  }; // end class Package
59
60  #endif
```

```
1   // Exercise 13.15 Solution: Package.cpp
2   // Member-function definitions of class Package.
3
4   #include "Package.h" // Package class definition
5
6   // constructor initiliazes data members
7   Package::Package( const string &sName, const string &sAddress,
```

```
8       const string &sCity, const string &sState, int sZIP,
9       const string &rName, const string &rAddress, const string &rCity,
10      const string &rState, int rZIP, double w, double cost )
11      : senderName( sName ), senderAddress( sAddress ), senderCity( sCity ),
12        senderState( sState ), senderZIP( sZIP ), recipientName( rName ),
13        recipientAddress( rAddress ), recipientCity( rCity ),
14        recipientState( rState ), recipientZIP( rZIP )
15   {
16      setWeight( w ); // validate and store weight
17      setCostPerOunce( cost ); // validate and store cost per ounce
18   } // end Package constructor
19
20   // set sender's name
21   void Package::setSenderName( const string &name )
22   {
23       senderName = name;
24   } // end function setSenderName
25
26   // return sender's name
27   string Package::getSenderName() const
28   {
29      return senderName;
30   } // end function getSenderName
31
32   // set sender's address
33   void Package::setSenderAddress( const string &address )
34   {
35      senderAddress = address;
36   } // end function setSenderAddress
37
38   // return sender's address
39   string Package::getSenderAddress() const
40   {
41      return senderAddress;
42   } // end function getSenderAddress
43
44   // set sender's city
45   void Package::setSenderCity( const string &city )
46   {
47      senderCity = city;
48   } // end function setSenderCity
49
50   // return sender's city
51   string Package::getSenderCity() const
52   {
53      return senderCity;
54   } // end function getSenderCity
55
56   // set sender's state
57   void Package::setSenderState( const string &state )
58   {
59      senderState = state;
60   } // end function setSenderState
61
62   // return sender's state
```

```
63  string Package::getSenderState() const
64  {
65     return senderState;
66  } // end function getSenderState
67
68  // set sender's ZIP code
69  void Package::setSenderZIP( int zip )
70  {
71     senderZIP = zip;
72  } // end function setSenderZIP
73
74  // return sender's ZIP code
75  int Package::getSenderZIP() const
76  {
77     return senderZIP;
78  } // end function getSenderZIP
79
80  // set recipient's name
81  void Package::setRecipientName( const string &name )
82  {
83      recipientName = name;
84  } // end function setRecipientName
85
86  // return recipient's name
87  string Package::getRecipientName() const
88  {
89     return recipientName;
90  } // end function getRecipientName
91
92  // set recipient's address
93  void Package::setRecipientAddress( const string &address )
94  {
95     recipientAddress = address;
96  } // end function setRecipientAddress
97
98  // return recipient's address
99  string Package::getRecipientAddress() const
100 {
101    return recipientAddress;
102 } // end function getRecipientAddress
103
104 // set recipient's city
105 void Package::setRecipientCity( const string &city )
106 {
107    recipientCity = city;
108 } // end function setRecipientCity
109
110 // return recipient's city
111 string Package::getRecipientCity() const
112 {
113    return recipientCity;
114 } // end function getRecipientCity
115
116 // set recipient's state
117 void Package::setRecipientState( const string &state )
```

```
118 {
119    recipientState = state;
120 } // end function setRecipientState
121
122 // return recipient's state
123 string Package::getRecipientState() const
124 {
125    return recipientState;
126 } // end function getRecipientState
127
128 // set recipient's ZIP code
129 void Package::setRecipientZIP( int zip )
130 {
131    recipientZIP = zip;
132 } // end function setRecipientZIP
133
134 // return recipient's ZIP code
135 int Package::getRecipientZIP() const
136 {
137    return recipientZIP;
138 } // end function getRecipientZIP
139
140 // validate and store weight
141 void Package::setWeight( double w )
142 {
143    weight = ( w < 0.0 ) ? 0.0 : w;
144 } // end function setWeight
145
146 // return weight of package
147 double Package::getWeight() const
148 {
149    return weight;
150 } // end function getWeight
151
152 // validate and store cost per ounce
153 void Package::setCostPerOunce( double cost )
154 {
155    costPerOunce = ( cost < 0.0 ) ? 0.0 : cost;
156 } // end function setCostPerOunce
157
158 // return cost per ounce
159 double Package::getCostPerOunce() const
160 {
161    return costPerOunce;
162 } // end function getCostPerOunce
163
164 // calculate shipping cost for package
165 double Package::calculateCost() const
166 {
167    return getWeight() * getCostPerOunce();
168 } // end function calculateCost
```

```
1 // Exercise 13.15 Solution: TwoDayPackage.h
2 // Definition of derived class TwoDayPackage.
```

```
3   #ifndef TWODAY_H
4   #define TWODAY_H
5
6   #include "Package.h" // Package class definition
7
8   class TwoDayPackage : public Package
9   {
10  public:
11     TwoDayPackage( const string &, const string &, const string &,
12        const string &, int, const string &, const string &, const string &,
13        const string &, int, double, double, double );
14
15     void setFlatFee( double ); // set flat fee for two-day-delivery service
16     double getFlatFee() const; // return flat fee
17
18     virtual double calculateCost() const; // calculate shipping cost
19  private:
20     double flatFee; // flat fee for two-day-delivery service
21  }; // end class TwoDayPackage
22
23  #endif
```

```
1   // Exercise 13.15 Solution: TwoDayPackage.cpp
2   // Member-function definitions of class TwoDayPackage.
3
4   #include "TwoDayPackage.h" // TwoDayPackage class definition
5
6   // constructor
7   TwoDayPackage::TwoDayPackage( const string &sName,
8      const string &sAddress, const string &sCity, const string &sState,
9      int sZIP, const string &rName, const string &rAddress,
10     const string &rCity, const string &rState, int rZIP,
11     double w, double cost, double fee )
12     : Package( sName, sAddress, sCity, sState, sZIP,
13        rName, rAddress, rCity, rState, rZIP, w, cost )
14  {
15     setFlatFee( fee );
16  } // end TwoDayPackage constructor
17
18  // set flat fee
19  void TwoDayPackage::setFlatFee( double fee )
20  {
21     flatFee = ( fee < 0.0 ) ? 0.0 : fee;
22  } // end function setFlatFee
23
24  // return flat fee
25  double TwoDayPackage::getFlatFee() const
26  {
27     return flatFee;
28  } // end function getFlatFee
29
30  // calculate shipping cost for package
31  double TwoDayPackage::calculateCost() const
32  {
```

```
33       return Package::calculateCost() + getFlatFee();
34    } // end function calculateCost
```

```
1     // Exercise 13.15 Solution: OvernightPackage.h
2     // Definition of derived class OvernightPackage.
3     #ifndef OVERNIGHT_H
4     #define OVERNIGHT_H
5
6     #include "Package.h" // Package class definition
7
8     class OvernightPackage : public Package
9     {
10    public:
11       OvernightPackage( const string &, const string &, const string &,
12          const string &, int, const string &, const string &, const string &,
13          const string &, int, double, double, double );
14
15       void setOvernightFeePerOunce( double ); // set overnight fee
16       double getOvernightFeePerOunce() const; // return overnight fee
17
18       virtual double calculateCost() const; // calculate shipping cost
19    private:
20       double overnightFeePerOunce; // fee per ounce for overnight delivery
21    }; // end class OvernightPackage
22
23    #endif
```

```
1     // Exercise 13.15 Solution: OvernightPackage.cpp
2     // Member-function definitions of class OvernightPackage.
3
4     #include "OvernightPackage.h" // OvernightPackage class definition
5
6     // constructor
7     OvernightPackage::OvernightPackage( const string &sName,
8        const string &sAddress, const string &sCity, const string &sState,
9        int sZIP, const string &rName, const string &rAddress,
10       const string &rCity, const string &rState, int rZIP,
11       double w, double cost, double fee )
12       : Package( sName, sAddress, sCity, sState, sZIP,
13          rName, rAddress, rCity, rState, rZIP, w, cost )
14    {
15       setOvernightFeePerOunce( fee ); // validate and store overnight fee
16    } // end OvernightPackage constructor
17
18    // set overnight fee
19    void OvernightPackage::setOvernightFeePerOunce( double overnightFee )
20    {
21       overnightFeePerOunce = ( overnightFee < 0.0 ) ? 0.0 : overnightFee;
22    } // end function setOvernightFeePerOunce
23
24    // return overnight fee
25    double OvernightPackage::getOvernightFeePerOunce() const
26    {
```

```
27       return overnightFeePerOunce;
28    } // end function getOvernghtFeePerOunce
29
30    // calculate shipping cost for package
31    double OvernightPackage::calculateCost() const
32    {
33       return getWeight() * ( getCostPerOunce() + getOvernightFeePerOunce() );
34    } // end function calculateCost
```

```
 1    // Exercise 13.15 Solution: ex13_15.cpp
 2    // Processing Packages polymorphically.
 3    #include <iostream>
 4    using std::cout;
 5    using std::endl;
 6
 7    #include <iomanip>
 8    using std::setprecision;
 9    using std::fixed;
10
11    #include <vector>
12    using std::vector;
13
14    #include "Package.h" // Package class definition
15    #include "TwoDayPackage.h" // TwoDayPackage class definition
16    #include "OvernightPackage.h" // OvernightPackage class definition
17
18    int main()
19    {
20       // create vector packages
21       vector < Package * > packages( 3 );
22
23       // initialize vector with Packages
24       packages[ 0 ] = new Package( "Lou Brown", "1 Main St", "Boston", "MA",
25          11111, "Mary Smith", "7 Elm St", "New York", "NY", 22222, 8.5, .5 );
26       packages[ 1 ] = new TwoDayPackage( "Lisa Klein", "5 Broadway",
27          "Somerville", "MA", 33333, "Bob George", "21 Pine Rd", "Cambridge",
28          "MA", 44444, 10.5, .65, 2.0 );
29       packages[ 2 ] = new OvernightPackage( "Ed Lewis", "2 Oak St", "Boston",
30          "MA", 55555, "Don Kelly", "9 Main St", "Denver", "CO", 66666,
31          12.25, .7, .25 );
32
33       double totalShippingCost = 0.0;
34
35       cout << fixed << setprecision( 2 );
36
37       // print each package's information and cost
38       for ( size_t i = 0; i < packages.size(); i++ )
39       {
40          cout << "Package " << i + 1 << "\n\nSender:\n"
41             << packages[ i ]->getSenderName() << '\n'
42             << packages[ i ]->getSenderAddress() << '\n'
43             << packages[ i ]->getSenderCity() << ", "
44             << packages[ i ]->getSenderState() << ' '
45             << packages[ i ]->getSenderZIP();
```

```
46        cout << "\n\nRecipient:\n" << packages[ i ]->getRecipientName()
47           << '\n' << packages[ i ]->getRecipientAddress() << '\n'
48           << packages[ i ]->getRecipientCity() << ", "
49           << packages[ i ]->getRecipientState() << ' '
50           << packages[ i ]->getRecipientZIP();
51
52        double cost = packages[ i ]->calculateCost();
53        cout << "\n\nCost: $" << cost << "\n\n";
54        totalShippingCost += cost; // add this Package's cost to total
55     } // end for
56
57     cout << "Total shipping cost: $" << totalShippingCost << endl;
58     return 0;
59  } // end main
```

```
Package 1

Sender:
Lou Brown
1 Main St
Boston, MA 11111

Recipient:
Mary Smith
7 Elm St
New York, NY 22222

Cost: $4.25

Package 2

Sender:
Lisa Klein
5 Broadway
Somerville, MA 33333

Recipient:
Bob George
21 Pine Rd
Cambridge, MA 44444

Cost: $8.82

Package 3

Sender:
Ed Lewis
2 Oak St
Boston, MA 55555

Recipient:
Don Kelly
9 Main St
Denver, CO 66666

Cost: $11.64

Total shipping cost: $24.71
```

13.16 *(Polymorphic Banking Program Using `Account` Hierarchy)* Develop a polymorphic banking program using the `Account` hierarchy created in Exercise 12.10. Create a `vector` of `Account` pointers to `SavingsAccount` and `CheckingAccount` objects. For each `Account` in the `vector`, allow the user to specify an amount of money to withdraw from the `Account` using member function `debit` and an amount of money to deposit into the `Account` using member function `credit`. As you process each `Account`, determine its type. If an `Account` is a `SavingsAccount`, calculate the amount of interest

owed to the `Account` using member function `calculateInterest`, then add the interest to the account balance using member function `credit`. After processing an `Account`, print the updated account balance obtained by invoking base class member function `getBalance`.

ANS: [*Note:* To achieve polymorphic behavior in the `Account` hierarchy, each class definition must declare the `debit` and `credit` member functions as `virtual` functions.]

```
1   // Solution 13.16 Solution: Account.h
2   // Definition of Account class.
3   #ifndef ACCOUNT_H
4   #define ACCOUNT_H
5
6   class Account
7   {
8   public:
9      Account( double ); // constructor initializes balance
10     virtual void credit( double ); // add an amount to the account balance
11     virtual bool debit( double ); // subtract an amount from the balance
12     void setBalance( double ); // sets the account balance
13     double getBalance(); // return the account balance
14  private:
15     double balance; // data member that stores the balance
16  }; // end class Account
17
18  #endif
```

```
1   // Exercise 13.16 Solution: Account.cpp
2   // Member-function definitions for class Account.
3   #include <iostream>
4   using std::cout;
5   using std::endl;
6
7   #include "Account.h" // include definition of class Account
8
9   // Account constructor initializes data member balance
10  Account::Account( double initialBalance )
11  {
12     // if initialBalance is greater than or equal to 0.0, set this value
13     // as the balance of the Account
14     if ( initialBalance >= 0.0 )
15        balance = initialBalance;
16     else // otherwise, output message and set balance to 0.0
17     {
18        cout << "Error: Initial balance cannot be negative." << endl;
19        balance = 0.0;
20     } // end if...else
21  } // end Account constructor
22
23  // credit (add) an amount to the account balance
24  void Account::credit( double amount )
25  {
26     balance = balance + amount; // add amount to balance
27  } // end function credit
28
```

```
29  // debit (subtract) an amount from the account balance
30  // return bool indicating whether money was debited
31  bool Account::debit( double amount )
32  {
33     if ( amount > balance ) // debit amount exceeds balance
34     {
35        cout << "Debit amount exceeded account balance." << endl;
36        return false;
37     } // end if
38     else // debit amount does not exceed balance
39     {
40        balance = balance - amount;
41        return true;
42     } // end else
43  } // end function debit
44
45  // set the account balance
46  void Account::setBalance( double newBalance )
47  {
48     balance = newBalance;
49  } // end function setBalance
50
51  // return the account balance
52  double Account::getBalance()
53  {
54     return balance;
55  } // end function getBalance
```

```
1   // Exercise 13.16 Solution: SavingsAccount.h
2   // Definition of SavingsAccount class.
3   #ifndef SAVINGS_H
4   #define SAVINGS_H
5
6   #include "Account.h" // Account class definition
7
8   class SavingsAccount : public Account
9   {
10  public:
11     // constructor initializes balance and interest rate
12     SavingsAccount( double, double );
13
14     double calculateInterest(); // determine interest owed
15  private:
16     double interestRate; // interest rate (percentage) earned by account
17  }; // end class SavingsAccount
18
19  #endif
```

```
1   // Exercise 13.16 Solution: SavingsAccount.cpp
2   // Member-function definitions for class SavingsAccount.
3
4   #include "SavingsAccount.h" // SavingsAccount class definition
5
```

```
 6  // constructor initializes balance and interest rate
 7  SavingsAccount::SavingsAccount( double initialBalance, double rate )
 8     : Account( initialBalance ) // initialize base class
 9  {
10     interestRate = ( rate < 0.0 ) ? 0.0 : rate; // set interestRate
11  } // end SavingsAccount constructor
12
13  // return the amount of interest earned
14  double SavingsAccount::calculateInterest()
15  {
16     return getBalance() * interestRate;
17  } // end function calculateInterest
```

```
 1  // Exercise 13.16 Solution: CheckingAccount.h
 2  // Definition of CheckingAccount class.
 3  #ifndef CHECKING_H
 4  #define CHECKING_H
 5
 6  #include "Account.h" // Account class definition
 7
 8  class CheckingAccount : public Account
 9  {
10  public:
11     // constructor initializes balance and transaction fee
12     CheckingAccount( double, double );
13
14     virtual void credit( double ); // redefined credit function
15     virtual bool debit( double ); // redefined debit function
16  private:
17     double transactionFee; // fee charged per transaction
18
19     // utility function to charge fee
20     void chargeFee();
21  }; // end class CheckingAccount
22
23  #endif
```

```
 1  // Exercise 13.16 Solution: CheckingAccount.cpp
 2  // Member-function definitions for class CheckingAccount.
 3  #include <iostream>
 4  using std::cout;
 5  using std::endl;
 6
 7  #include "CheckingAccount.h" // CheckingAccount class definition
 8
 9  // constructor initializes balance and transaction fee
10  CheckingAccount::CheckingAccount( double initialBalance, double fee )
11     : Account( initialBalance ) // initialize base class
12  {
13     transactionFee = ( fee < 0.0 ) ? 0.0 : fee; // set transaction fee
14  } // end CheckingAccount constructor
15
16  // credit (add) an amount to the account balance and charge fee
```

```
17   void CheckingAccount::credit( double amount )
18   {
19      Account::credit( amount ); // always succeeds
20      chargeFee();
21   } // end function credit
22
23   // debit (subtract) an amount from the account balance and charge fee
24   bool CheckingAccount::debit( double amount )
25   {
26      bool success = Account::debit( amount ); // attempt to debit
27
28      if ( success ) // if money was debited, charge fee and return true
29      {
30         chargeFee();
31         return true;
32      } // end if
33      else // otherwise, do not charge fee and return false
34         return false;
35   } // end function debit
36
37   // subtract transaction fee
38   void CheckingAccount::chargeFee()
39   {
40      Account::setBalance( getBalance() - transactionFee );
41      cout << "$" << transactionFee << " transaction fee charged." << endl;
42   } // end function chargeFee
```

```
1    // Exercise 13.16 Solution: ex13_16.cpp
2    // Processing Accounts polymorphically.
3    #include <iostream>
4    using std::cout;
5    using std::cin;
6    using std::endl;
7
8    #include <iomanip>
9    using std::setprecision;
10   using std::fixed;
11
12   #include <vector>
13   using std::vector;
14
15   #include "Account.h" // Account class definition
16   #include "SavingsAccount.h" // SavingsAccount class definition
17   #include "CheckingAccount.h" // CheckingAccount class definition
18
19   int main()
20   {
21      // create vector accounts
22      vector < Account * > accounts( 4 );
23
24      // initialize vector with Accounts
25      accounts[ 0 ] = new SavingsAccount( 25.0, .03 );
26      accounts[ 1 ] = new CheckingAccount( 80.0, 1.0 );
27      accounts[ 2 ] = new SavingsAccount( 200.0, .015 );
```

```
28       accounts[ 3 ] = new CheckingAccount( 400.0, .5 );
29
30       cout << fixed << setprecision( 2 );
31
32       // loop through vector, prompting user for debit and credit amounts
33       for ( size_t i = 0; i < accounts.size(); i++ )
34       {
35          cout << "Account " << i + 1 << " balance: $"
36             << accounts[ i ]->getBalance();
37
38          double withdrawalAmount = 0.0;
39          cout << "\nEnter an amount to withdraw from Account " << i + 1
40             << ": ";
41          cin >> withdrawalAmount;
42          accounts[ i ]->debit( withdrawalAmount ); // attempt to debit
43
44          double depositAmount = 0.0;
45          cout << "Enter an amount to deposit into Account " << i + 1
46             << ": ";
47          cin >> depositAmount;
48          accounts[ i ]->credit( depositAmount ); // credit amount to Account
49
50          // downcast pointer
51          SavingsAccount *savingsAccountPtr =
52             dynamic_cast < SavingsAccount * > ( accounts[ i ] );
53
54          // if Account is a SavingsAccount, calculate and add interest
55          if ( savingsAccountPtr != 0 )
56          {
57             double interestEarned = savingsAccountPtr->calculateInterest();
58             cout << "Adding $" << interestEarned << " interest to Account "
59                << i + 1 << " (a SavingsAccount)" << endl;
60             savingsAccountPtr->credit( interestEarned );
61          } // end if
62
63          cout << "Updated Account " << i + 1 << " balance: $"
64             << accounts[ i ]->getBalance() << "\n\n";
65       } // end for
66
67       return 0;
68    } // end main
```

```
Account 1 balance: $25.00
Enter an amount to withdraw from Account 1: 15.00
Enter an amount to deposit into Account 1: 10.50
Adding $0.61 interest to Account 1 (a SavingsAccount)
Updated Account 1 balance: $21.11

Account 2 balance: $80.00
Enter an amount to withdraw from Account 2: 90.00
Debit amount exceeded account balance.
Enter an amount to deposit into Account 2: 45.00
$1.00 transaction fee charged.
Updated Account 2 balance: $124.00

Account 3 balance: $200.00
Enter an amount to withdraw from Account 3: 75.50
Enter an amount to deposit into Account 3: 300.00
Adding $6.37 interest to Account 3 (a SavingsAccount)
Updated Account 3 balance: $430.87

Account 4 balance: $400.00
Enter an amount to withdraw from Account 4: 56.81
$0.50 transaction fee charged.
Enter an amount to deposit into Account 4: 37.83
$0.50 transaction fee charged.
Updated Account 4 balance: $380.02
```