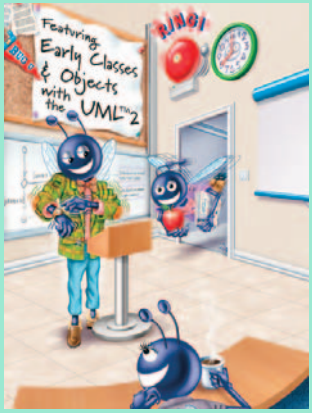
For the exclusive use of adopters of the book C++ How to Program, 5th Edition,
by Deitel and Deitel. ISBN 0-13-185757-6.

12

Object-Oriented Programming: Inheritance



Say not you know another entirely, till you have divided an inheritance with him.

—Johann Kasper Lavater

This method is to define as the number of a class the class of all classes similar to the given class.

—Bertrand Russell

Good as it is to inherit a library, it is better to collect one.

—Augustine Birrell

Save base authority from others' books.

—William Shakespeare

OBJECTIVES

In this chapter you will learn:

- To create classes by inheriting from existing classes.
- How inheritance promotes software reuse.
- The notions of base classes and derived classes and the relationships between them.
- The `protected` member access specifier.
- The use of constructors and destructors in inheritance hierarchies.
- The differences between `public`, `protected` and `private` inheritance.
- The use of inheritance to customize existing software.

Self-Review Exercises

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

a) _____ is a form of software reuse in which new classes absorb the data and behaviors of existing classes and embellish these classes with new capabilities.

ANS: Inheritance.

b) A base class's _____ members can be accessed only in the base-class definition or in derived-class definitions.

ANS: `protected`.

c) In a(n) _____ relationship, an object of a derived class also can be treated as an object of its base class.

ANS: *is-a* or inheritance.

d) In a(n) _____ relationship, a class object has one or more objects of other classes as members.

ANS: *has-a* or composition or aggregation.

e) In single inheritance, a class exists in a(n) _____ relationship with its derived classes.

ANS: hierarchical.

f) A base class's _____ members are accessible within that base class and anywhere that the program has a handle to an object of that base class or to an object of one of its derived classes.

ANS: `public`.

g) A base class's `protected` access members have a level of protection between those of `public` and _____ access.

ANS: `private`.

h) C++ provides for _____, which allows a derived class to inherit from many base classes, even if these base classes are unrelated.

ANS: multiple inheritance.

i) When an object of a derived class is instantiated, the base class's _____ is called implicitly or explicitly to do any necessary initialization of the base-class data members in the derived-class object.

ANS: constructor.

j) When deriving a class from a base class with `public` inheritance, `public` members of the base class become _____ members of the derived class, and `protected` members of the base class become _____ members of the derived class.

ANS: `public`, `protected`.

k) When deriving a class from a base class with `protected` inheritance, `public` members of the base class become _____ members of the derived class, and `protected` members of the base class become _____ members of the derived class.

ANS: `protected`, `protected`.

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

a) Base-class constructors are not inherited by derived classes.

ANS: True.

b) A *has-a* relationship is implemented via inheritance.

ANS: False. A *has-a* relationship is implemented via composition. An *is-a* relationship is implemented via inheritance.

c) A `Car` class has an *is-a* relationship with the `SteeringWheel` and `Brakes` classes.

ANS: False. This is an example of a *has-a* relationship. Class `Car` has an *is-a* relationship with class `Vehicle`.

d) Inheritance encourages the reuse of proven high-quality software.

ANS: True.

e) When a derived-class object is destroyed, the destructors are called in the reverse order of the constructors.

ANS: True.

Exercises

12.3 Many programs written with inheritance could be written with composition instead, and vice versa. Rewrite class `BasePlusCommissionEmployee` of the `CommissionEmployee–BasePlusCommissionEmployee` hierarchy to use composition rather than inheritance. After you do this, assess the relative merits of the two approaches for designing classes `CommissionEmployee` and `BasePlusCommissionEmployee`, as well as for object-oriented programs in general. Which approach is more natural? Why?

ANS: For a relatively short program like this one, either approach is acceptable. But as programs become larger with more and more objects being instantiated, inheritance becomes preferable because it makes the program easier to modify and promotes the reuse of code. The inheritance approach is more natural because a base-salaried commission employee *is a* commission employee. Composition is defined by the "has-a" relationship, and clearly it would be strange to say that "a base-salaried commission employee *has a* commission employee."

```
1  // Exercise 12.3 Solution: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class using composition.
3  #ifndef BASEPLUS_H
4  #define BASEPLUS_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
9  #include "CommissionEmployee.h" // CommissionEmployee class definition
10
11 class BasePlusCommissionEmployee
12 {
13 public:
14    BasePlusCommissionEmployee( const string &, const string &,
15       const string &, double = 0.0, double = 0.0, double = 0.0 );
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 setGrossSales( double ); // set gross sales amount
27    double getGrossSales() const; // return gross sales amount
28
29    void setCommissionRate( double ); // set commission rate
30    double getCommissionRate() const; // return commission rate
31
32    void setBaseSalary( double ); // set base salary
33    double getBaseSalary() const; // return base salary
```

```
34
35    double earnings() const; // calculate earnings
36    void print() const; // print BasePlusCommissionEmployee object
37 private:
38    double baseSalary; // base salary
39    CommissionEmployee commissionEmployee; // composed object
40 }; // end class BasePlusCommissionEmployee
41
42 #endif
```

```
1  // Exercise 12.3 Solution: BasePlusCommissionEmployee.cpp
2  // Member-function definitions of class BasePlusCommissionEmployee
3  // using composition.
4  #include <iostream>
5  using std::cout;
6
7  // BasePlusCommissionEmployee class definition
8  #include "BasePlusCommissionEmployee.h"
9
10 // constructor
11 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
12    const string &first, const string &last, const string &ssn,
13    double sales, double rate, double salary )
14    // initialize composed object
15    : commissionEmployee( first, last, ssn, sales, rate )
16 {
17    setBaseSalary( salary ); // validate and store base salary
18 } // end BasePlusCommissionEmployee constructor
19
20 // set commission employee's first name
21 void BasePlusCommissionEmployee::setFirstName( const string &first )
22 {
23    commissionEmployee.setFirstName( first );
24 } // end function setFirstName
25
26 // return commission employee's first name
27 string BasePlusCommissionEmployee::getFirstName() const
28 {
29    return commissionEmployee.getFirstName();
30 } // end function getFirstName
31
32 // set commission employee's last name
33 void BasePlusCommissionEmployee::setLastName( const string &last )
34 {
35    commissionEmployee.setLastName( last );
36 } // end function setLastName
37
38 // return commission employee's last name
39 string BasePlusCommissionEmployee::getLastName() const
40 {
41    return commissionEmployee.getLastName();
42 } // end function getLastName
43
44 // set commission employee's social security number
```

```
45 void BasePlusCommissionEmployee::setSocialSecurityNumber(
46    const string &ssn )
47 {
48    commissionEmployee.setSocialSecurityNumber( ssn );
49 } // end function setSocialSecurityNumber
50
51 // return commission employee's social security number
52 string BasePlusCommissionEmployee::getSocialSecurityNumber() const
53 {
54    return commissionEmployee.getSocialSecurityNumber();
55 } // end function getSocialSecurityNumber
56
57 // set commission employee's gross sales amount
58 void BasePlusCommissionEmployee::setGrossSales( double sales )
59 {
60    commissionEmployee.setGrossSales( sales );
61 } // end function setGrossSales
62
63 // return commission employee's gross sales amount
64 double BasePlusCommissionEmployee::getGrossSales() const
65 {
66    return commissionEmployee.getGrossSales();
67 } // end function getGrossSales
68
69 // set commission employee's commission rate
70 void BasePlusCommissionEmployee::setCommissionRate( double rate )
71 {
72    commissionEmployee.setCommissionRate( rate );
73 } // end function setCommissionRate
74
75 // return commission employee's commission rate
76 double BasePlusCommissionEmployee::getCommissionRate() const
77 {
78    return commissionEmployee.getCommissionRate();
79 } // end function getCommissionRate
80
81 // set base salary
82 void BasePlusCommissionEmployee::setBaseSalary( double salary )
83 {
84    baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
85 } // end function setBaseSalary
86
87 // return base salary
88 double BasePlusCommissionEmployee::getBaseSalary() const
89 {
90    return baseSalary;
91 } // end function getBaseSalary
92
93 // calculate earnings
94 double BasePlusCommissionEmployee::earnings() const
95 {
96    return getBaseSalary() + commissionEmployee.earnings();
97 } // end function earnings
98
99 // print BasePlusCommissionEmployee object
```

```
100 void BasePlusCommissionEmployee::print() const
101 {
102    cout << "base-salaried ";
103
104    // invoke composed CommissionEmployee object's print function
105    commissionEmployee.print();
106
107    cout << "\nbase salary: " << getBaseSalary();
108 } // end function print
```

12.4 Discuss the ways in which inheritance promotes software reuse, saves time during program development and helps prevent errors.

ANS: Inheritance allows developers to create derived classes that reuse code declared already in a base class. Avoiding the duplication of common functionality between several classes by building an inheritance hierarchy to contain the classes can save developers a considerable amount of time. Similarly, placing common functionality in a single base class, rather than duplicating the code in multiple unrelated classes, helps prevent the same errors from appearing in multiple source-code files. If several classes each contain duplicate code containing an error, the software developer has to spend time correcting each source-code file with the error. However, if these classes take advantage of inheritance, and the error occurs in the common functionality of the base class, the software developer needs to modify only the base class's code.

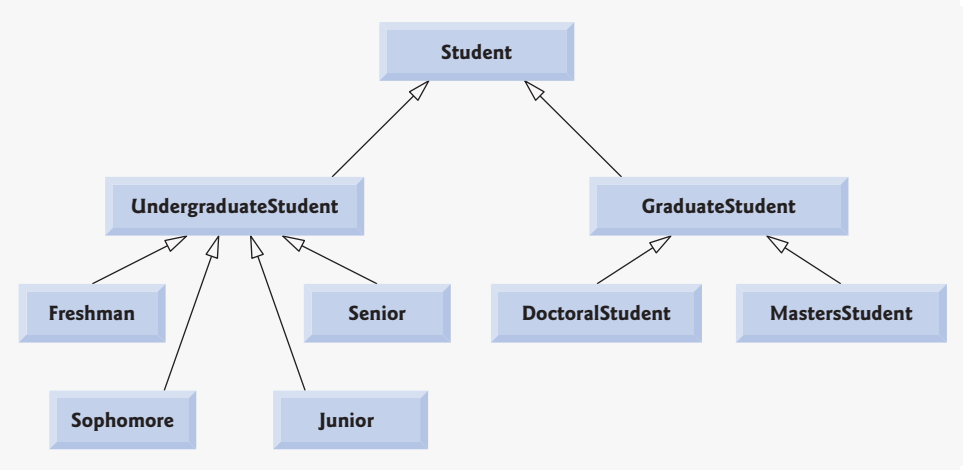
12.5 Some programmers prefer not to use `protected` access because they believe it breaks the encapsulation of the base class. Discuss the relative merits of using `protected` access vs. using `private` access in base classes.

ANS: `private` data members are hidden in the base class and are accessible only through the `public` or `protected` member functions of the base class. Using `protected` access enables the derived class to manipulate the `protected` members without using the access functions of the base class. If the base class members are `private`, the member functions of the base class must be used to access the data. This may result in a decrease in performance due to the extra function calls, yet accessing and modifying `private` data in this indirect manner helps ensure that the data in the base class remains consistent.

12.6 Draw an inheritance hierarchy for students at a university similar to the hierarchy shown in Fig. 12.2. Use `Student` as the base class of the hierarchy, then include classes `UndergraduateStudent` and `GraduateStudent` that derive from `Student`. Continue to extend the hierarchy as deep (i.e., as many levels) as possible. For example, `Freshman`, `Sophomore`, `Junior` and `Senior` might derive from `UndergraduateStudent`, and `DoctoralStudent` and `MastersStudent` might derive from `Graduate-`

`Student`. After drawing the hierarchy, discuss the relationships that exist between the classes. [*Note:* You do not need to write any code for this exercise.]

ANS:

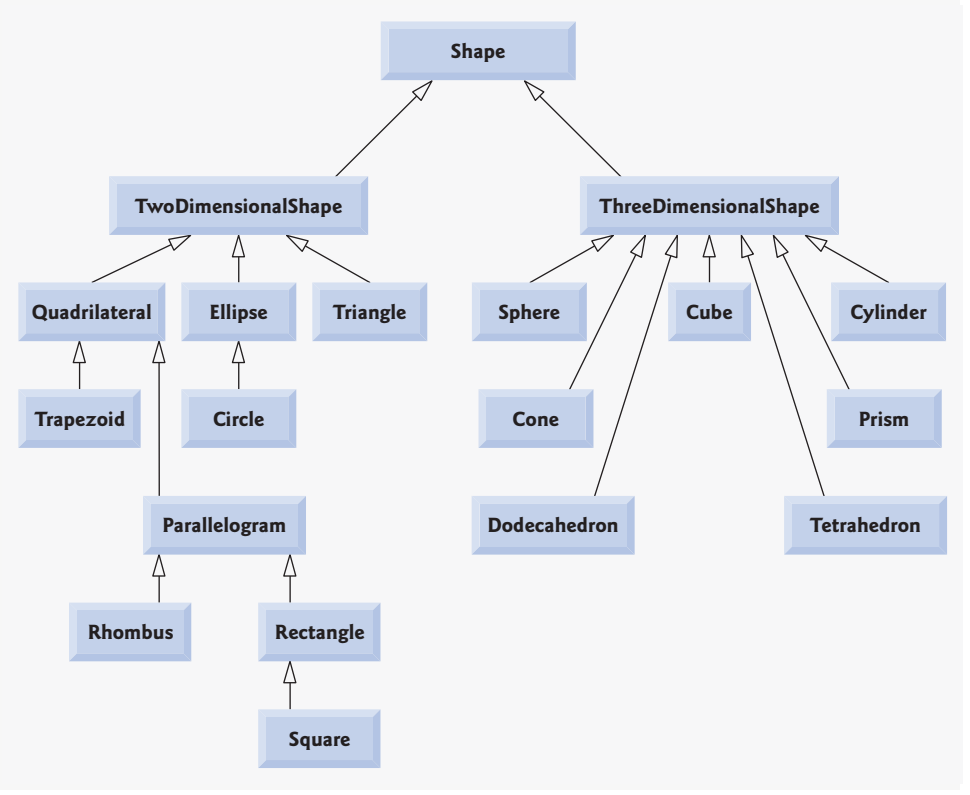


This hierarchy contains many "is-a" (inheritance) relationships. An `UndergraduateStudent` *is a* `Student`. A `GraduateStudent` *is a* `Student` too. Each of the classes `Freshman`, `Sophomore`, `Junior` and `Senior` *is an* `UndergraduateStudent` and *is a* `Student`. Each of the classes `DoctoralStudent` and `MastersStudent` *is a* `GraduateStudent` and *is a* `Student`.

12.7 The world of shapes is much richer than the shapes included in the inheritance hierarchy of Fig. 12.3. Write down all the shapes you can think of—both two-dimensional and three-dimensional—and form them into a more complete `Shape` hierarchy with as many levels as possible. Your hierarchy should have base class `Shape` from which class `TwoDimensionalShape` and class `ThreeDi-`

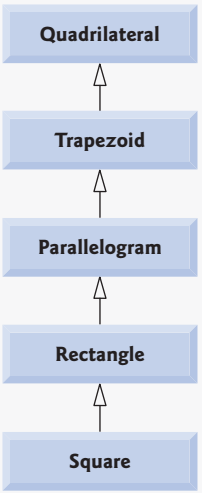
`mensionalShape` are derived. [*Note:* You do not need to write any code for this exercise.] We will use this hierarchy in the exercises of Chapter 13 to process a set of distinct shapes as objects of base-class `Shape`. (This technique, called polymorphism, is the subject of Chapter 13.)

ANS: [*Note:* Solutions may vary.]



12.8 Draw an inheritance hierarchy for classes `Quadrilateral`, `Trapezoid`, `Parallelogram`, `Rectangle` and `Square`. Use `Quadrilateral` as the base class of the hierarchy. Make the hierarchy as deep as possible.

ANS:



12.9 (*`Package` Inheritance Hierarchy*) Package-delivery services, such as FedEx®, DHL® and UPS®, offer a number of different shipping options, each with specific costs associated. Create an inheritance hierarchy to represent various types of packages. Use `Package` as the base class of the hierarchy, then include classes `TwoDayPackage` and `OvernightPackage` that derive from `Package`. Base class `Package` should include data members representing the name, address, city, state and ZIP code for both the sender and the recipient of the package, in addition to data members that store the weight (in ounces) and cost per ounce to ship the package. `Package`'s constructor should initialize these data members. Ensure that the weight and cost per ounce contain positive values. `Package` should provide a `public` member function `calculateCost` that returns a `double` indicating the cost associated with shipping the package. `Package`'s `calculateCost` function should determine the cost by multiplying the weight by the cost per ounce. Derived class `TwoDayPackage` should inherit the functionality of base class `Package`, but also include a data member that represents a flat fee that the shipping company charges for two-day-delivery service. `TwoDayPackage`'s constructor should receive a value to initialize this data member. `TwoDayPackage` should redefine member function `calculateCost` so that it computes the shipping cost by adding the flat fee to the weight-based cost calculated by base class `Package`'s `calculateCost` function. Class `OvernightPackage` should inherit directly from class `Package` and contain an additional data member representing an additional fee per ounce charged for overnight-delivery service. `OvernightPackage` should redefine member function `calculateCost` so that it adds the additional fee per ounce to the standard cost per ounce before calculating the shipping cost. Write a test program that creates objects of each type of `Package` and tests member function `calculateCost`.

ANS:

```
1  // Exercise 12.9 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    double calculateCost() const; // calculate shipping cost for package
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 12.9 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;
```

```
} // end function getCostPerOunce

// calculate shipping cost for package
double Package::calculateCost() const
{
   return getWeight() * getCostPerOunce();
} // end function calculateCost
```

```
// Exercise 12.9 Solution: TwoDayPackage.h
// Definition of derived class TwoDayPackage.
#ifndef TWODAY_H
#define TWODAY_H

#include "Package.h" // Package class definition

class TwoDayPackage : public Package
{
public:
   TwoDayPackage( const string &, const string &, const string &,
      const string &, int, const string &, const string &, const string &,
      const string &, int, double, double, double );

   void setFlatFee( double ); // set flat fee for two-day-delivery service
   double getFlatFee() const; // return flat fee

   double calculateCost() const; // calculate shipping cost for package
private:
   double flatFee; // flat fee for two-day-delivery service
}; // end class TwoDayPackage

#endif
```

```
// Exercise 12.9 Solution: TwoDayPackage.cpp
// Member-function definitions of class TwoDayPackage.

#include "TwoDayPackage.h" // TwoDayPackage class definition

// constructor
TwoDayPackage::TwoDayPackage( const string &sName,
   const string &sAddress, const string &sCity, const string &sState,
   int sZIP, const string &rName, const string &rAddress,
   const string &rCity, const string &rState, int rZIP,
   double w, double cost, double fee )
   : Package( sName, sAddress, sCity, sState, sZIP,
      rName, rAddress, rCity, rState, rZIP, w, cost )
{
   setFlatFee( fee );
} // end TwoDayPackage constructor

// set flat fee
void TwoDayPackage::setFlatFee( double fee )
{
   flatFee = ( fee < 0.0 ) ? 0.0 : fee;
```

```
} // end function setFlatFee

// return flat fee
double TwoDayPackage::getFlatFee() const
{
   return flatFee;
} // end function getFlatFee

// calculate shipping cost for package
double TwoDayPackage::calculateCost() const
{
   return Package::calculateCost() + getFlatFee();
} // end function calculateCost
```

```
// Exercise 12.9 Solution: OvernightPackage.h
// Definition of derived class OvernightPackage.
#ifndef OVERNIGHT_H
#define OVERNIGHT_H

#include "Package.h" // Package class definition

class OvernightPackage : public Package
{
public:
   OvernightPackage( const string &, const string &, const string &,
      const string &, int, const string &, const string &, const string &,
      const string &, int, double, double, double );

   void setOvernightFeePerOunce( double ); // set overnight fee
   double getOvernightFeePerOunce() const; // return overnight fee

   double calculateCost() const; // calculate shipping cost for package
private:
   double overnightFeePerOunce; // fee per ounce for overnight delivery
}; // end class OvernightPackage

#endif
```

```
// Exercise 12.9 Solution: OvernightPackage.cpp
// Member-function definitions of class OvernightPackage.

#include "OvernightPackage.h" // OvernightPackage class definition

// constructor
OvernightPackage::OvernightPackage( const string &sName,
   const string &sAddress, const string &sCity, const string &sState,
   int sZIP, const string &rName, const string &rAddress,
   const string &rCity, const string &rState, int rZIP,
   double w, double cost, double fee )
   : Package( sName, sAddress, sCity, sState, sZIP,
      rName, rAddress, rCity, rState, rZIP, w, cost )
{
   setOvernightFeePerOunce( fee ); // validate and store overnight fee
```

```
} // end OvernightPackage constructor

// set overnight fee
void OvernightPackage::setOvernightFeePerOunce( double overnightFee )
{
   overnightFeePerOunce = ( overnightFee < 0.0 ) ? 0.0 : overnightFee;
} // end function setOvernightFeePerOunce

// return overnight fee
double OvernightPackage::getOvernightFeePerOunce() const
{
   return overnightFeePerOunce;
} // end function getOvernghtFeePerOunce

// calculate shipping cost for package
double OvernightPackage::calculateCost() const
{
   return getWeight() * ( getCostPerOunce() + getOvernightFeePerOunce() );
} // end function calculateCost
```

```
// Exercise 12.9 Solution: ex12_09.cpp
// Driver program for Package hierarchy.
#include <iostream>
using std::cout;
using std::endl;

#include <iomanip>
using std::setprecision;
using std::fixed;

#include "Package.h" // Package class definition
#include "TwoDayPackage.h" // TwoDayPackage class definition
#include "OvernightPackage.h" // OvernightPackage class definition

int main()
{
   Package package1( "Lou Brown", "1 Main St", "Boston", "MA", 11111,
      "Mary Smith", "7 Elm St", "New York", "NY", 22222, 8.5, .5 );
   TwoDayPackage package2( "Lisa Klein", "5 Broadway", "Somerville", "MA",
      33333, "Bob George", "21 Pine Rd", "Cambridge", "MA", 44444,
      10.5, .65, 2.0 );
   OvernightPackage package3( "Ed Lewis", "2 Oak St", "Boston", "MA",
      55555, "Don Kelly", "9 Main St", "Denver", "CO", 66666,
      12.25, .7, .25 );

   cout << fixed << setprecision( 2 );

   // print each package's information and cost
   cout << "Package 1:\n\nSender:\n" << package1.getSenderName()
      << '\n' << package1.getSenderAddress() << '\n'
      << package1.getSenderCity() << ", " << package1.getSenderState()
      << ' ' << package1.getSenderZIP();
   cout << "\n\nRecipient:\n" << package1.getRecipientName()
      << '\n' << package1.getRecipientAddress() << '\n'
```

```
35          << package1.getRecipientCity() << ", "
36          << package1.getRecipientState() << ' '
37          << package1.getRecipientZIP();
38       cout << "\n\nCost: $" << package1.calculateCost() << endl;
39
40       cout << "\nPackage 2:\n\nSender:\n" << package2.getSenderName()
41          << '\n' << package2.getSenderAddress() << '\n'
42          << package2.getSenderCity() << ", " << package2.getSenderState()
43          << ' ' << package2.getSenderZIP();
44       cout << "\n\nRecipient:\n" << package2.getRecipientName()
45          << '\n' << package2.getRecipientAddress() << '\n'
46          << package2.getRecipientCity() << ", "
47          << package2.getRecipientState() << ' '
48          << package2.getRecipientZIP();
49       cout << "\n\nCost: $" << package2.calculateCost() << endl;
50
51       cout << "\nPackage 3:\n\nSender:\n" << package3.getSenderName()
52          << '\n' << package3.getSenderAddress() << '\n'
53          << package3.getSenderCity() << ", " << package3.getSenderState()
54          << ' ' << package3.getSenderZIP();
55       cout << "\n\nRecipient:\n" << package3.getRecipientName()
56          << '\n' << package3.getRecipientAddress() << '\n'
57          << package3.getRecipientCity() << ", "
58          << package3.getRecipientState() << ' '
59          << package3.getRecipientZIP();
60       cout << "\n\nCost: $" << package3.calculateCost() << endl;
61       return 0;
62    } // 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
```

12.10 *(`Account` Inheritance Hierarchy)* Create an inheritance hierarchy that a bank might use to represent customers' bank accounts. All customers at this bank can deposit (i.e., credit) money into their accounts and withdraw (i.e., debit) money from their accounts. More specific types of accounts also exist. Savings accounts, for instance, earn interest on the money they hold. Checking accounts, on the other hand, charge a fee per transaction (i.e., credit or debit).

Create an inheritance hierarchy containing base class `Account` and derived classes `SavingsAccount` and `CheckingAccount` that inherit from class `Account`. Base class `Account` should include one data member of type `double` to represent the account balance. The class should provide a constructor that receives an initial balance and uses it to initialize the data member. The constructor should validate the initial balance to ensure that it is greater than or equal to `0.0`. If not, the balance should be set to `0.0` and the constructor should display an error message, indicating that the initial balance was invalid. The class should provide three member functions. Member function

`credit` should add an amount to the current balance. Member function `debit` should withdraw money from the `Account` and ensure that the debit amount does not exceed the `Account`'s balance. If it does, the balance should be left unchanged and the function should print the message `"Debit amount exceeded account balance."` Member function `getBalance` should return the current balance.

Derived class `SavingsAccount` should inherit the functionality of an `Account`, but also include a data member of type `double` indicating the interest rate (percentage) assigned to the `Account`. `SavingsAccount`'s constructor should receive the initial balance, as well as an initial value for the `SavingsAccount`'s interest rate. `SavingsAccount` should provide a `public` member function `calculateInterest` that returns a `double` indicating the amount of interest earned by an account. Member function `calculateInterest` should determine this amount by multiplying the interest rate by the account balance. [*Note:* `SavingsAccount` should inherit member functions `credit` and `debit` as is without redefining them.]

Derived class `CheckingAccount` should inherit from base class `Account` and include an additional data member of type `double` that represents the fee charged per transaction. `CheckingAccount`'s constructor should receive the initial balance, as well as a parameter indicating a fee amount. Class `CheckingAccount` should redefine member functions `credit` and `debit` so that they subtract the fee from the account balance whenever either transaction is performed successfully. `CheckingAccount`'s versions of these functions should invoke the base-class `Account` version to perform the updates to an account balance. `CheckingAccount`'s `debit` function should charge a fee only if money is actually withdrawn (i.e., the debit amount does not exceed the account balance). [*Hint:* Define `Account`'s `debit` function so that it returns a `bool` indicating whether money was withdrawn. Then use the return value to determine whether a fee should be charged.]

After defining the classes in this hierarchy, write a program that creates objects of each class and tests their member functions. Add interest to the `SavingsAccount` object by first invoking its `calculateInterest` function, then passing the returned interest amount to the object's `credit` function.

ANS:

```
// Solution 12.10 Solution: Account.h
// Definition of Account class.
#ifndef ACCOUNT_H
#define ACCOUNT_H

class Account
{
public:
   Account( double ); // constructor initializes balance
   void credit( double ); // add an amount to the account balance
   bool debit( double ); // subtract an amount from the account balance
   void setBalance( double ); // sets the account balance
   double getBalance(); // return the account balance
private:
   double balance; // data member that stores the balance
}; // end class Account

#endif
```

```
// Exercise 12.10 Solution: Account.cpp
// 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
```

```
// Exercise 12.10 Solution: SavingsAccount.h
// Definition of SavingsAccount class.
#ifndef SAVINGS_H
#define SAVINGS_H

#include "Account.h" // Account class definition

class SavingsAccount : public Account
{
public:
   // constructor initializes balance and interest rate
   SavingsAccount( double, double );

   double calculateInterest(); // determine interest owed
private:
   double interestRate; // interest rate (percentage) earned by account
}; // end class SavingsAccount

#endif
```

```
// Exercise 12.10 Solution: SavingsAccount.cpp
// Member-function definitions for class SavingsAccount.

#include "SavingsAccount.h" // SavingsAccount class definition

// constructor initializes balance and interest rate
SavingsAccount::SavingsAccount( double initialBalance, double rate )
   : Account( initialBalance ) // initialize base class
{
   interestRate = ( rate < 0.0 ) ? 0.0 : rate; // set interestRate
} // end SavingsAccount constructor

// return the amount of interest earned
double SavingsAccount::calculateInterest()
{
   return getBalance() * interestRate;
} // end function calculateInterest
```

```
// Exercise 12.10 Solution: CheckingAccount.h
// Definition of CheckingAccount class.
#ifndef CHECKING_H
#define CHECKING_H

#include "Account.h" // Account class definition

class CheckingAccount : public Account
{
public:
   // constructor initializes balance and transaction fee
   CheckingAccount( double, double );

   void credit( double ); // redefined credit function
```

```
15    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 12.10 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
```

```
// Exercise 12.10 Solution: ex12_10.cpp
// Test program for Account hierarchy.
#include <iostream>
using std::cout;
using std::endl;

#include <iomanip>
using std::setprecision;
using std::fixed;

#include "Account.h" // Account class definition
#include "SavingsAccount.h" // SavingsAccount class definition
#include "CheckingAccount.h" // CheckingAccount class definition

int main()
{
   Account account1( 50.0 ); // create Account object
   SavingsAccount account2( 25.0, .03 ); // create SavingsAccount object
   CheckingAccount account3( 80.0, 1.0 ); // create CheckingAccount object

   cout << fixed << setprecision( 2 );

   // display initial balance of each object
   cout << "account1 balance: $" << account1.getBalance() << endl;
   cout << "account2 balance: $" << account2.getBalance() << endl;
   cout << "account3 balance: $" << account3.getBalance() << endl;

   cout << "\nAttempting to debit $25.00 from account1." << endl;
   account1.debit( 25.0 ); // try to debit $25.00 from account1
   cout << "\nAttempting to debit $30.00 from account2." << endl;
   account2.debit( 30.0 ); // try to debit $30.00 from account2
   cout << "\nAttempting to debit $40.00 from account3." << endl;
   account3.debit( 40.0 ); // try to debit $40.00 from account3

   // display balances
   cout << "\naccount1 balance: $" << account1.getBalance() << endl;
   cout << "account2 balance: $" << account2.getBalance() << endl;
   cout << "account3 balance: $" << account3.getBalance() << endl;

   cout << "\nCrediting $40.00 to account1." << endl;
   account1.credit( 40.0 ); // credit $40.00 to account1
   cout << "\nCrediting $65.00 to account2." << endl;
   account2.credit( 65.0 ); // credit $65.00 to account2
   cout << "\nCrediting $20.00 to account3." << endl;
   account3.credit( 20.0 ); // credit $20.00 to account3

   // display balances
   cout << "\naccount1 balance: $" << account1.getBalance() << endl;
   cout << "account2 balance: $" << account2.getBalance() << endl;
   cout << "account3 balance: $" << account3.getBalance() << endl;

   // add interest to SavingsAccount object account2
   double interestEarned = account2.calculateInterest();
   cout << "\nAdding $" << interestEarned << " interest to account2."
      << endl;
```

```
56      account2.credit( interestEarned );
57
58      cout << "\nNew account2 balance: $" << account2.getBalance() << endl;
59      return 0;
60   } // end main
```

```
account1 balance: $50.00
account2 balance: $25.00
account3 balance: $80.00

Attempting to debit $25.00 from account1.

Attempting to debit $30.00 from account2.
Debit amount exceeded account balance.

Attempting to debit $40.00 from account3.
$1.00 transaction fee charged.

account1 balance: $25.00
account2 balance: $25.00
account3 balance: $39.00

Crediting $40.00 to account1.

Crediting $65.00 to account2.

Crediting $20.00 to account3.
$1.00 transaction fee charged.

account1 balance: $65.00
account2 balance: $90.00
account3 balance: $58.00

Adding $2.70 interest to account2.

New account2 balance: $92.70
```