CHAPTER 15

Inheritance, Polymorphism, and Virtual Functions

TOPICS

15.1 What Is Inheritance?
15.2 Protected Members and Class Access
15.3 Constructors and Destructors in Base and Derived Classes
15.4 Redefining Base Class Functions
15.5 Class Hierarchies
15.6 Polymorphism and Virtual Member Functions
15.7 Abstract Base Classes and Pure Virtual Functions
15.8 Multiple Inheritance

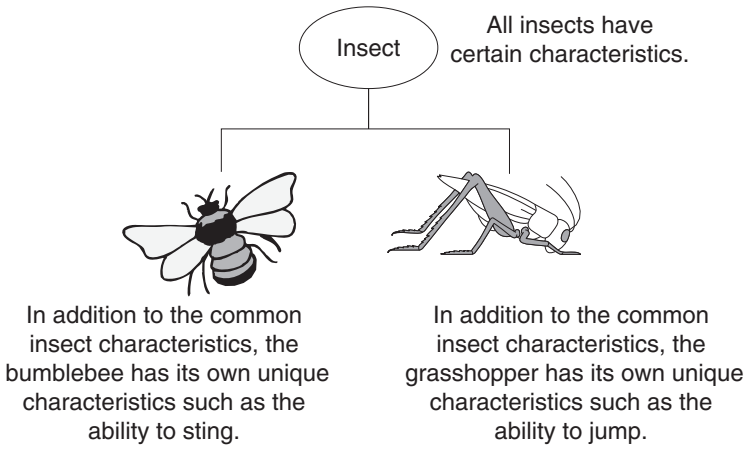
15.1 What Is Inheritance?

CONCEPT: **Inheritance allows a new class to be based on an existing class. The new class inherits all the member variables and functions (except the constructors and destructor) of the class it is based on.**

Generalization and Specialization

In the real world you can find many objects that are specialized versions of other more general objects. For example, the term "insect" describes a very general type of creature with numerous characteristics. Because grasshoppers and bumblebees are insects, they have all the general characteristics of an insect. In addition, they have special characteristics of their own. For example, the grasshopper has its jumping ability, and the bumblebee has its stinger. Grasshoppers and bumblebees are specialized versions of an insect. This is illustrated in Figure 15-1.

Figure 15-1



Inheritance and the "Is a" Relationship

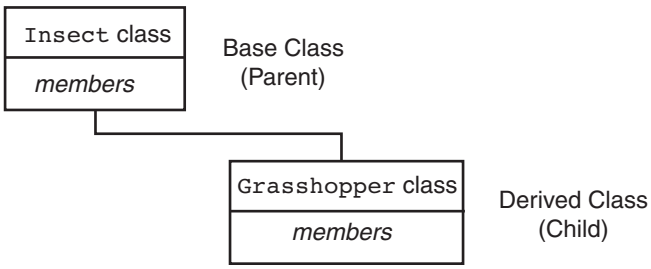
When one object is a specialized version of another object, there is an *"is a" relationship* between them. For example, a grasshopper *is an* insect. Here are a few other examples of the "is a" relationship.

- A poodle *is a* dog.
- A car *is a* vehicle.
- A tree *is a* plant.
- A rectangle *is a* shape.
- A football player *is an* athlete.

When an "is a" relationship exists between classes, it means that the specialized class has all of the characteristics of the general class, plus additional characteristics that make it special. In object-oriented programming, *inheritance* is used to create an "is a" relationship between classes.

Inheritance involves a base class and a derived class. The *base class* is the general class and the *derived class* is the specialized class. The derived class is based on, or derived from, the base class. You can think of the base class as the parent and the derived class as the child. This is illustrated in Figure 15-2.

Figure 15-2



The derived class inherits the member variables and member functions of the base class without any of them being rewritten. Furthermore, new member variables and functions may be added to the derived class to make it more specialized than the base class.

Let's look at an example of how inheritance can be used. Most teachers assign various graded activities for their students to complete. A graded activity can receive a numeric score such as 70, 85, 90, and so on, and a letter grade such as A, B, C, D, or F. The following `GradedActivity` class is designed to hold the numeric score and letter grade of a graded activity. When a numeric score is stored by the class, it automatically determines the letter grade. (These files are stored in the Student Source Code Folder `Chapter 15\GradedActivity Version 1`.)

Contents of `GradedActivity.h` (Version 1)

```
#ifndef GRADEDACTIVITY_H
#define GRADEDACTIVITY_H

// GradedActivity class declaration

class GradedActivity
{
private:
   double score; // To hold the numeric score
public:
   // Default constructor
   GradedActivity()
      { score = 0.0; }

   // Constructor
   GradedActivity(double s)
      { score = s; }

   // Mutator function
   void setScore(double s)
      { score = s; }

   // Accessor functions
   double getScore() const
      { return score; }

   char getLetterGrade() const;
};
#endif
```

Contents of `GradedActivity.cpp` (Version 1)

```
#include "GradedActivity.h"

//*****************************************************
// Member function GradedActivity::getLetterGrade     *
//*****************************************************

```

```
 7  char GradedActivity::getLetterGrade() const
 8  {
 9     char letterGrade; // To hold the letter grade
10
11     if (score > 89)
12        letterGrade = 'A';
13     else if (score > 79)
14        letterGrade = 'B';
15     else if (score > 69)
16        letterGrade = 'C';
17     else if (score > 59)
18        letterGrade = 'D';
19     else
20        letterGrade = 'F';
21
22     return letterGrade;
23  }
```

The `GradedActivity` class has a default constructor that initializes the `score` member variable to 0.0. A second constructor accepts an argument for `score`. The `setScore` member function also accepts an argument for the `score` variable, and the `getLetterGrade` member function returns the letter grade that corresponds to the value in `score`. Program 15-1 demonstrates the `GradedActivity` class. (This file is also stored in the Student Source Code Folder `Chapter 15\GradedActivity Version 1`.)

Program 15-1

```
 1  // This program demonstrates the GradedActivity class.
 2  #include <iostream>
 3  #include "GradedActivity.h"
 4  using namespace std;
 5
 6  int main()
 7  {
 8     double testScore;  // To hold a test score
 9
10     // Create a GradedActivity object for the test.
11     GradedActivity test;
12
13     // Get a numeric test score from the user.
14     cout << "Enter your numeric test score: ";
15     cin >> testScore;
16
17     // Store the numeric score in the test object.
18     test.setScore(testScore);
19
20     // Display the letter grade for the test.
21     cout << "The grade for that test is "
22          << test.getLetterGrade() << endl;
23
24     return 0;
25  }
```

Program Output with Example Input Shown in Bold

```
Enter your numeric test score: 89 [Enter]
The grade for that test is B
```

Program Output with Different Example Input Shown in Bold

```
Enter your numeric test score: 75 [Enter]
The grade for that test is C
```

The `GradedActivity` class represents the general characteristics of a student's graded activity. Many different types of graded activities exist, however, such as quizzes, midterm exams, final exams, lab reports, essays, and so on. Because the numeric scores might be determined differently for each of these graded activities, we can create derived classes to handle each one. For example, the following code shows the `FinalExam` class, which is derived from the `GradedActivity` class. It has member variables for the number of questions on the exam (`numQuestions`), the number of points each question is worth (`pointsEach`), and the number of questions missed by the student (`numMissed`). These files are also stored in the Student Source Code Folder `Chapter 15\GradedActivity Version 1`.

Contents of `FinalExam.h`

```
1  #ifndef FINALEXAM_H
2  #define FINALEXAM_H
3  #include "GradedActivity.h"
4
5  class FinalExam : public GradedActivity
6  {
7  private:
8     int numQuestions;    // Number of questions
9     double pointsEach;   // Points for each question
10    int numMissed;       // Number of questions missed
11 public:
12    // Default constructor
13    FinalExam()
14       { numQuestions = 0;
15         pointsEach = 0.0;
16         numMissed = 0; }
17
18    // Constructor
19    FinalExam(int questions, int missed)
20       { set(questions, missed); }
21
22    // Mutator function
23    void set(int, int);  // Defined in FinalExam.cpp
24
25    // Accessor functions
26    double getNumQuestions() const
27       { return numQuestions; }
28
29    double getPointsEach() const
30       { return pointsEach; }
31
```

```
32      int getNumMissed() const
33         { return numMissed; }
34  };
35  #endif
```

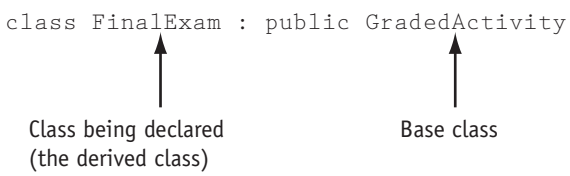
Contents of `FinalExam.cpp`

```
 1  #include "FinalExam.h"
 2
 3  //*******************************************************
 4  // set function                                         *
 5  // The parameters are the number of questions and the   *
 6  // number of questions missed.                          *
 7  //*******************************************************
 8
 9  void FinalExam::set(int questions, int missed)
10  {
11     double numericScore;  // To hold the numeric score
12
13     // Set the number of questions and number missed.
14     numQuestions = questions;
15     numMissed = missed;
16
17     // Calculate the points for each question.
18     pointsEach = 100.0 / numQuestions;
19
20     // Calculate the numeric score for this exam.
21     numericScore = 100.0 - (missed * pointsEach);
22
23     // Call the inherited setScore function to set
24     // the numeric score.
25     setScore(numericScore);
26  }
```

The only new notation in this code is in line 5 of the `FinalExam.h` file, which reads

```
class FinalExam : public GradedActivity
```

This line indicates the name of the class being declared and the name of the base class it is derived from. `FinalExam` is the name of the class being declared, and `GradedActivity` is the name of the base class it inherits from.



If we want to express the relationship between the two classes, we can say that a `FinalExam` *is a* `GradedActivity`.

The word `public`, which precedes the name of the base class in line 5 of the `FinalExam.h` file, is the *base class access specification*. It affects how the members of the base class are inherited by the derived class. When you create an object of a derived class, you can think

of it as being built on top of an object of the base class. The members of the base class object become members of the derived class object. How the base class members appear in the derived class is determined by the base class access specification.

Although we will discuss this topic in more detail in the next section, let's see how it works in this example. The `GradedActivity` class has both private members and public members. The `FinalExam` class is derived from the `GradedActivity` class, using `public` access specification. This means that the public members of the `GradedActivity` class will become public members of the `FinalExam` class. The private members of the `GradedActivity` class cannot be accessed directly by code in the `FinalExam` class. Although the private members of the `GradedActivity` class are inherited, it's as though they are invisible to the code in the `FinalExam` class. They can only be accessed by the member functions of the `GradedActivity` class. Here is a list of the members of the `FinalExam` class:

Private Members:

`int numQuestions`	Declared in the `FinalExam` class
`double pointsEach`	Declared in the `FinalExam` class
`int numMissed`	Declared in the `FinalExam` class

Public Members:

`FinalExam()`	Defined in the `FinalExam` class
`FinalExam(int, int)`	Defined in the `FinalExam` class
`set(int, int)`	Defined in the `FinalExam` class
`getNumQuestions()`	Defined in the `FinalExam` class
`getPointsEach()`	Defined in the `FinalExam` class
`getNumMissed()`	Defined in the `FinalExam` class
`setScore(double)`	Inherited from `GradedActivity`
`getScore()`	Inherited from `GradedActivity`
`getLetterGrade()`	Inherited from `GradedActivity`

The `GradedActivity` class has one private member, the variable `score`. Notice that it is not listed as a member of the `FinalExam` class. It is still inherited by the derived class, but because it is a private member of the base class, only member functions of the base class may access it. It is truly private to the base class. Because the functions `setScore`, `getScore`, and `getLetterGrade` are public members of the base class, they also become public members of the derived class.

You will also notice that the `GradedActivity` class constructors are not listed among the members of the `FinalExam` class. Although the base class constructors still exist, it makes sense that they are not members of the derived class because their purpose is to construct objects of the base class. In the next section we discuss in more detail how base class constructors operate.

Let's take a closer look at the `FinalExam` class constructors. The default constructor appears in lines 13 through 16 of the `FinalExam.h` file. It simply assigns 0 to each of the class's member variables. Another constructor appears in lines 19 through 20. This constructor accepts two arguments, one for the number of questions on the exam, and one for the number of questions missed. This constructor merely passes those values as arguments to the `set` function.

The `set` function is defined in `FinalExam.cpp`. It accepts two arguments: the number of questions on the exam, and the number of questions missed by the student. In lines 14 and 15 these values are assigned to the `numQuestions` and `numMissed` member variables. In line 18 the number of points for each question is calculated. In line 21 the numeric test score is calculated. In line 25, the last statement in the function reads:

```
setScore(numericScore);
```

This is a call to the `setScore` function. Although no `setScore` function appears in the `FinalExam` class, it is inherited from the `GradedActivity` class. Program 15-2 demonstrates the `FinalExam` class.

Program 15-2

```
 1  // This program demonstrates a base class and a derived class.
 2  #include <iostream>
 3  #include <iomanip>
 4  #include "FinalExam.h"
 5  using namespace std;
 6
 7  int main()
 8  {
 9     int questions; // Number of questions on the exam
10     int missed;    // Number of questions missed by the student
11
12     // Get the number of questions on the final exam.
13     cout << "How many questions are on the final exam? ";
14     cin >> questions;
15
16     // Get the number of questions the student missed.
17     cout << "How many questions did the student miss? ";
18     cin >> missed;
19
20     // Define a FinalExam object and initialize it with
21     // the values entered.
22     FinalExam test(questions, missed);
23
24     // Display the test results.
25     cout << setprecision(2);
26     cout << "\nEach question counts " << test.getPointsEach()
27          << " points.\n";
28     cout << "The exam score is " << test.getScore() << endl;
29     cout << "The exam grade is " << test.getLetterGrade() << endl;
30
31     return 0;
32  }
```

Program Output with Example Input Shown in Bold

```
How many questions are on the final exam? 20 [Enter]
How many questions did the student miss? 3 [Enter]

Each question counts 5 points.
The exam score is 85
The exam grade is B
```

Notice in lines 28 and 29 that the public member functions of the `GradedActivity` class may be directly called by the `test` object:

```
cout << "The exam score is " << test.getScore() << endl;
cout << "The exam grade is " << test.getLetterGrade() << endl;
```

The `getScore` and `getLetterGrade` member functions are inherited as public members of the `FinalExam` class, so they may be accessed like any other public member.

Inheritance does not work in reverse. It is not possible for a base class to call a member function of a derived class. For example, the following classes will not compile in a program because the `BadBase` constructor attempts to call a function in its derived class:

```
class BadBase
{
   private:
      int x;
   public:
      BadBase() { x = getVal(); }  // Error!
};

class Derived : public BadBase
{
   private:
      int y;
   public:
      Derived(int z) { y = z; }
      int getVal() { return y; }
};
```

Checkpoint

15.1 Here is the first line of a class declaration. Circle the name of the base class:

```
class Truck : public Vehicle
```

15.2 Circle the name of the derived class in the following declaration line:

```
class Truck : public Vehicle
```

15.3 Suppose a program has the following class declarations:

```
class Shape
{
private:
   double area;
public:
   void setArea(double a)
      { area = a; }

   double getArea()
      { return area; }
};

class Circle : public Shape
{
private:
   double radius;
```

```
public:
   void setRadius(double r)
      { radius = r;
        setArea(3.14 * r * r); }
   double getRadius()
      { return radius; }
};
```

Answer the following questions concerning these classes:

A) When an object of the `Circle` class is created, what are its private members?

B) When an object of the `Circle` class is created, what are its public members?

C) What members of the `Shape` class are not accessible to member functions of the `Circle` class?

15.2 Protected Members and Class Access

CONCEPT: **Protected members of a base class are like private members, but they may be accessed by derived classes. The base class access specification determines how private, public, and protected base class members are accessed when they are inherited by the derived classes.**

Until now you have used two access specifications within a class: `private` and `public`. C++ provides a third access specification, `protected`. Protected members of a base class are like private members, except they may be accessed by functions in a derived class. To the rest of the program, however, protected members are inaccessible.

The following code shows a modified version of the `GradedActivity` class declaration. The private member of the class has been made protected. This file is stored in the Student Source Code Folder `Chapter 15\GradedActivity Version 2`. The implementation file, `GradedActivity.cpp` has not changed, so it is not shown again in this example.

Contents of `GradedActivity.h` (Version 2)

```
 1  #ifndef GRADEDACTIVITY_H
 2  #define GRADEDACTIVITY_H
 3
 4  // GradedActivity class declaration
 5
 6  class GradedActivity
 7  {
 8  protected:
 9     double score;  // To hold the numeric score
10  public:
11     // Default constructor
12     GradedActivity()
13        { score = 0.0; }
14
15     // Constructor
16     GradedActivity(double s)
17        { score = s; }
18
```

```
19      // Mutator function
20      void setScore(double s)
21         { score = s; }
22
23      // Accessor functions
24      double getScore() const
25         { return score; }
26
27      char getLetterGrade() const;
28   };
29   #endif
```

Now we will look at a modified version of the `FinalExam` class, which is derived from this version of the `GradedActivity` class. This version of the `FinalExam` class has a new member function named `adjustScore`. This function directly accesses the `GradedActivity` class's `score` member variable. If the content of the `score` variable has a fractional part of 0.5 or greater, the function rounds `score` up to the next whole number. The `set` function calls the `adjustScore` function after it calculates the numeric score. (These files are stored in the Student Source Code Folder `Chapter 15\GradedActivity Version 2`.)

Contents of `FinalExam.h` (Version 2)

```
1   #ifndef FINALEXAM_H
2   #define FINALEXAM_H
3   #include "GradedActivity.h"
4
5   class FinalExam : public GradedActivity
6   {
7   private:
8      int numQuestions;    // Number of questions
9      double pointsEach;   // Points for each question
10     int numMissed;       // Number of questions missed
11  public:
12     // Default constructor
13     FinalExam()
14        { numQuestions = 0;
15          pointsEach = 0.0;
16          numMissed = 0; }
17
18     // Constructor
19     FinalExam(int questions, int missed)
20        { set(questions, missed); }
21
22     // Mutator functions
23     void set(int, int); // Defined in FinalExam.cpp
24     void adjustScore(); // Defined in FinalExam.cpp
25
26     // Accessor functions
27     double getNumQuestions() const
28        { return numQuestions; }
29
30     double getPointsEach() const
31        { return pointsEach; }
32
```

```
33      int getNumMissed() const
34         { return numMissed; }
35  };
36  #endif
```

Contents of `FinalExam.cpp` (Version 2)

```
 1  #include "FinalExam.h"
 2
 3  //*********************************************************
 4  // set function                                           *
 5  // The parameters are the number of questions and the     *
 6  // number of questions missed.                            *
 7  //*********************************************************
 8
 9  void FinalExam::set(int questions, int missed)
10  {
11     double numericScore;  // To hold the numeric score
12
13     // Set the number of questions and number missed.
14     numQuestions = questions;
15     numMissed = missed;
16
17     // Calculate the points for each question.
18     pointsEach = 100.0 / numQuestions;
19
20     // Calculate the numeric score for this exam.
21     numericScore = 100.0 - (missed * pointsEach);
22
23     // Call the inherited setScore function to set
24     // the numeric score.
25     setScore(numericScore);
26
27     // Call the adjustScore function to adjust
28     // the score.
29     adjustScore();
30  }
31
32  //*************************************************************
33  // Definition of Test::adjustScore. If score is within        *
34  // 0.5 points of the next whole point, it rounds the score up *
35  // and recalculates the letter grade.                         *
36  //*************************************************************
37
38  void FinalExam::adjustScore()
39  {
40     double fraction = score - static_cast<int>(score);
41
42     if (fraction >= 0.5)
43     {
44        // Adjust the score variable in the GradedActivity class.
45        score += (1.0 - fraction);
46     }
47  }
```

Program 15-3 demonstrates these versions of the `GradedActivity` and `FinalExam` classes. (This file is also stored in the Student Source Code Folder `Chapter 15\GradedActivity Version 2`.)

Program 15-3

```
1  // This program demonstrates a base class with a
2  // protected member.
3  #include <iostream>
4  #include <iomanip>
5  #include "FinalExam.h"
6  using namespace std;
7
8  int main()
9  {
10    int questions; // Number of questions on the exam
11    int missed;    // Number of questions missed by the student
12
13    // Get the number of questions on the final exam.
14    cout << "How many questions are on the final exam? ";
15    cin >> questions;
16
17    // Get the number of questions the student missed.
18    cout << "How many questions did the student miss? ";
19    cin >> missed;
20
21    // Define a FinalExam object and initialize it with
22    // the values entered.
23    FinalExam test(questions, missed);
24
25    // Display the adjusted test results.
26    cout << setprecision(2) << fixed;
27    cout << "\nEach question counts "
28         << test.getPointsEach() << " points.\n";
29    cout << "The adjusted exam score is "
30         << test.getScore() << endl;
31    cout << "The exam grade is "
32         << test.getLetterGrade() << endl;
33
34    return 0;
35 }
```

Program Output with Example Input Shown in Bold

```
How many questions are on the final exam? 16 [Enter]
How many questions did the student miss? 5 [Enter]

Each question counts 6.25 points.
The adjusted exam score is 69.00
The exam grade is D
```

The program works as planned. In the example run, the student missed five questions, which are worth 6.25 points each. The unadjusted score would be 68.75. The score was adjusted to 69.

More About Base Class Access Specification

The first line of the `FinalExam` class declaration reads:

```
class FinalExam : public GradedActivity
```

This declaration gives public access specification to the base class. Recall from our earlier discussion that base class access specification affects how inherited base class members are accessed. Be careful not to confuse base class access specification with member access specification. Member access specification determines how members that are *defined* within the class are accessed. Base class access specification determines how *inherited* members are accessed.

When you create an object of a derived class, it inherits the members of the base class. The derived class can have its own private, protected, and public members, but what is the access specification of the inherited members? This is determined by the base class access specification. Table 15-1 summarizes how base class access specification affects the way that base class members are inherited.

Table 15-1

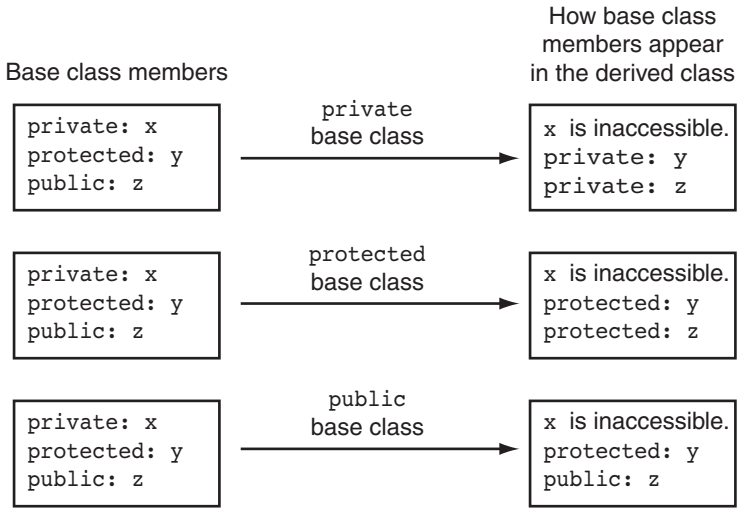
Base Class Access Specification	How Members of the Base Class Appear in the Derived Class
`private`	Private members of the base class are inaccessible to the derived class.
	Protected members of the base class become private members of the derived class.
	Public members of the base class become private members of the derived class.
`protected`	Private members of the base class are inaccessible to the derived class.
	Protected members of the base class become protected members of the derived class.
	Public members of the base class become protected members of the derived class.
`public`	Private members of the base class are inaccessible to the derived class.
	Protected members of the base class become protected members of the derived class.
	Public members of the base class become public members of the derived class.

As you can see from Table 15-1, class access specification gives you a great deal of flexibility in determining how base class members will appear in the derived class. Think of a base class's access specification as a filter that base class members must pass through when becoming inherited members of a derived class. This is illustrated in Figure 15-3.

NOTE: If the base class access specification is left out of a declaration, the default access specification is `private`. For example, in the following declaration, `Grade` is declared as a `private` base class:

```
class Test : Grade
```

Figure 15-3



Checkpoint

15.4 What is the difference between private members and protected members?

15.5 What is the difference between member access specification and class access specification?

15.6 Suppose a program has the following class declaration:

```
// Declaration of CheckPoint class.
class CheckPoint
{
   private:
      int a;
   protected:
      int b;
      int c;
      void setA(int x) { a = x;}
   public:
      void setB(int y) { b = y;}
      void setC(int z) { c = z;}
};
```

Answer the following questions regarding the class:

A) Suppose another class, `Quiz`, is derived from the `CheckPoint` class. Here is the first line of its declaration:

```
class Quiz : private CheckPoint
```

Indicate whether each member of the `CheckPoint` class is `private`, `protected`, `public`, or inaccessible:

```
a
b
c
setA
setB
setC
```

B) Suppose the `Quiz` class, derived from the `CheckPoint` class, is declared as

```
class Quiz : protected Checkpoint
```

Indicate whether each member of the `CheckPoint` class is `private`, `protected`, `public`, or inaccessible:

```
a
b
c
setA
setB
setC
```

C) Suppose the `Quiz` class, derived from the `CheckPoint` class, is declared as

```
class Quiz : public Checkpoint
```

Indicate whether each member of the `CheckPoint` class is `private`, `protected`, `public`, or inaccessible:

```
a
b
c
setA
setB
setC
```

D) Suppose the `Quiz` class, derived from the `CheckPoint` class, is declared as

```
class Quiz : Checkpoint
```

Is the `CheckPoint` class a `private`, `public`, or `protected` base class?

15.3 Constructors and Destructors in Base and Derived Classes

CONCEPT: **The base class's constructor is called before the derived class's constructor. The destructors are called in reverse order, with the derived class's destructor being called first.**

In inheritance, the base class constructor is called before the derived class constructor. Destructors are called in reverse order. Program 15-4 shows a simple set of demonstration classes, each with a default constructor and a destructor. The `DerivedClass` class is derived from the `BaseClass` class. Messages are displayed by the constructors and destructors to demonstrate when each is called.

Program 15-4

```
1  // This program demonstrates the order in which base and
2  // derived class constructors and destructors are called.
3  #include <iostream>
4  using namespace std;
5
6  //********************************
7  // BaseClass declaration         *
8  //********************************
9
```

```
10  class BaseClass
11  {
12  public:
13     BaseClass()  // Constructor
14        { cout << "This is the BaseClass constructor.\n"; }
15
16     ~BaseClass()  // Destructor
17        { cout << "This is the BaseClass destructor.\n"; }
18  };
19
20  //*********************************
21  // DerivedClass declaration      *
22  //*********************************
23
24  class DerivedClass : public BaseClass
25  {
26  public:
27     DerivedClass()  // Constructor
28        { cout << "This is the DerivedClass constructor.\n"; }
29
30     ~DerivedClass()  // Destructor
31        { cout << "This is the DerivedClass destructor.\n"; }
32  };
33
34  //*********************************
35  // main function                 *
36  //*********************************
37
38  int main()
39  {
40     cout << "We will now define a DerivedClass object.\n";
41
42     DerivedClass object;
43
44     cout << "The program is now going to end.\n";
45     return 0;
46  }
```

Program Output

```
We will now define a DerivedClass object.
This is the BaseClass constructor.
This is the DerivedClass constructor.
The program is now going to end.
This is the DerivedClass destructor.
This is the BaseClass destructor.
```

Passing Arguments to Base Class Constructors

In Program 15-4, both the base class and derived class have default constructors, that are called automatically. But what if the base class's constructor takes arguments? What if there is more than one constructor in the base class? The answer to these questions is to let the derived class constructor pass arguments to the base class constructor. For example, consider the following class:

Contents of `Rectangle.h`

```
1  #ifndef RECTANGLE_H
2  #define RECTANGLE_H
3
4  class Rectangle
5  {
6  private:
7     double width;
8     double length;
9  public:
10    // Default constructor
11    Rectangle()
12       { width = 0.0;
13         length = 0.0; }
14
15    // Constructor #2
16    Rectangle(double w, double len)
17       { width = w;
18         length = len; }
19
20    double getWidth() const
21       { return width; }
22
23    double getLength() const
24       { return length; }
25
26    double getArea() const
27       { return width * length; }
28 };
29 #endif
```

This class is designed to hold data about a rectangle. It specifies two constructors. The default constructor, in lines 11 through 13, simply initializes the `width` and `length` member variables to 0.0. The second constructor, in lines 16 through 18, takes two arguments, which are assigned to the `width` and `length` member variables. Now let's look at a class that is derived from the `Rectangle` class:

Contents of `Cube.h`

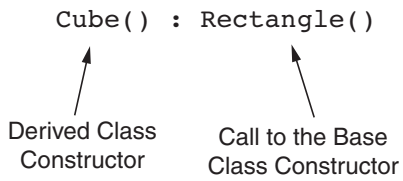
```
1  #ifndef CUBE_H
2  #define CUBE_H
3  #include "Rectangle.h"
4
5  class Cube : public Rectangle
6  {
7  protected:
8     double height;
9     double volume;
10 public:
11    // Default constructor
12    Cube() : Rectangle()
13       { height = 0.0; volume = 0.0; }
14
```

```
15      // Constructor #2
16      Cube(double w, double len, double h) : Rectangle(w, len)
17         { height = h;
18           volume = getArea() * h; }
19
20      double getHeight() const
21         { return height; }
22
23      double getVolume() const
24         { return volume; }
25  };
26  #endif
```

The `Cube` class is designed to hold data about cubes, which not only have a length and width, but a height and volume as well. Look at line 12, which is the first line of the `Cube` class's default constructor:

```
Cube() : Rectangle()
```

Notice the added notation in the header of the constructor. A colon is placed after the derived class constructor's parentheses, followed by a function call to a base class constructor. In this case, the base class's default constructor is being called. When this `Cube` class constructor executes, it will first call the `Rectangle` class's default constructor. This is illustrated here:



The general format of this type of constructor declaration is

```
ClassName::ClassName(ParameterList) : BaseClassName(ArgumentList)
```

You can also pass arguments to the base class constructor, as shown in the `Cube` class's second constructor. Look at line 16:

```
Cube(double w, double len, double h) : Rectangle(w, len)
```

This `Cube` class constructor has three parameters: `w`, `len`, and `h`. Notice that the `Rectangle` class's constructor is called, and the `w` and `len` parameters are passed as arguments. This causes the `Rectangle` class's second constructor to be called.

You only write this notation in the definition of a constructor, not in a prototype. In this example, the derived class constructor is written inline (inside the class declaration), so the notation that contains the call to the base class constructor appears there. If the constructor were defined outside the class, the notation would appear in the function header. For example, the `Cube` class could appear as follows.

```
class Cube : public Rectangle
{
protected:
   double height;
   double volume;
public:
    // Default constructor
    Cube() : Rectangle()
       { height = 0.0; volume = 0.0; }

    // Constructor #2
    Cube(double, double, double);

    double getHeight() const
       { return height; }

    double getVolume() const
       { return volume; }
};

// Cube class constructor #2
Cube::Cube(double w, double len, double h) : Rectangle(w, len)
{
   height = h;
   volume = getArea() * h;
}
```

The base class constructor is always executed before the derived class constructor. When the `Rectangle` constructor finishes, the `Cube` constructor is then executed.

Any literal value or variable that is in scope may be used as an argument to the derived class constructor. Usually, one or more of the arguments passed to the derived class constructor are, in turn, passed to the base class constructor. The values that may be used as base class constructor arguments are

- Derived class constructor parameters
- Literal values
- Global variables that are accessible to the file containing the derived class constructor definition
- Expressions involving any of these items

Program 15-5 shows the `Rectangle` and `Cube` classes in use.

Program 15-5

```
1  // This program demonstrates passing arguments to a base
2  // class constructor.
3  #include <iostream>
4  #include "Cube.h"
5  using namespace std;
6
```

```
int main()
{
   double cubeWidth;  // To hold the cube's width
   double cubeLength; // To hold the cube's length
   double cubeHeight; // To hold the cube's height

   // Get the width, length, and height of
   // the cube from the user.
   cout << "Enter the dimensions of a cube:\n";
   cout << "Width: ";
   cin >> cubeWidth;
   cout << "Length: ";
   cin >> cubeLength;
   cout << "Height: ";
   cin >> cubeHeight;

   // Define a Cube object and use the dimensions
   // entered by the user.
   Cube myCube(cubeWidth, cubeLength, cubeHeight);

   // Display the Cube object's properties.
   cout << "Here are the cube's properties:\n";
   cout << "Width: " << myCube.getWidth() << endl;
   cout << "Length: " << myCube.getLength() << endl;
   cout << "Height: " << myCube.getHeight() << endl;
   cout << "Base area: " << myCube.getArea() << endl;
   cout << "Volume: " << myCube.getVolume() << endl;

   return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter the dimensions of a cube:
Width: 10 [Enter]
Length: 15 [Enter]
Height: 12 [Enter]
Here are the cube's properties:
Width: 10
Length: 15
Height: 12
Base area: 150
Volume: 1800
```

NOTE: If the base class has no default constructor, then the derived class must have a constructor that calls one of the base class constructors.

In the Spotlight:

The `Automobile`, `Car`, `Truck`, and `SUV` classes

Suppose we are developing a program that a car dealership can use to manage its inventory of used cars. The dealership's inventory includes three types of automobiles: cars, pickup trucks, and sport-utility vehicles (SUVs). Regardless of the type, the dealership keeps the following data about each automobile:

- Make
- Year model
- Mileage
- Price

Each type of vehicle that is kept in inventory has these general characteristics, plus its own specialized characteristics. For cars, the dealership keeps the following additional data:

- Number of doors (2 or 4)

For pickup trucks, the dealership keeps the following additional data:

- Drive type (two-wheel drive or four-wheel drive)

And, for SUVs, the dealership keeps the following additional data:

- Passenger capacity

In designing this program, one approach would be to write the following three classes:

- A `Car` class with attributes for the make, year model, mileage, price, and number of doors.
- A `Truck` class with attributes for the make, year model, mileage, price, and drive type.
- An `SUV` class with attributes for the make, year model, mileage, price, and passenger capacity.

This would be an inefficient approach, however, because all three classes have a large number of common data attributes. As a result, the classes would contain a lot of duplicated code. In addition, if we discover later that we need to add more common attributes, we would have to modify all three classes.

A better approach would be to write an `Automobile` base class to hold all the general data about an automobile, and then write derived classes for each specific type of automobile. The following code shows the `Automobile` class. (This file is stored in the Student Source Code Folder `Chapter 15\Automobile`.)

Contents of `Automobile.h`

```
1  #ifndef AUTOMOBILE_H
2  #define AUTOMOBILE_H
3  #include <string>
4  using namespace std;
5
6  // The Automobile class holds general data
7  // about an automobile in inventory.
8  class Automobile
9  {
```

```
10  private:
11     string make;   // The auto's make
12     int model;     // The auto's year model
13     int mileage;   // The auto's mileage
14     double price;  // The auto's price
15
16  public:
17     // Default constructor
18     Automobile()
19     {  make = "";
20        model = 0;
21        mileage = 0;
22        price = 0.0; }
23
24     // Constructor
25     Automobile(string autoMake, int autoModel,
26                int autoMileage, double autoPrice)
27     {  make = autoMake;
28        model = autoModel;
29        mileage = autoMileage;
30        price = autoPrice; }
31
32     // Accessors
33     string getMake() const
34     { return make; }
35
36     int getModel() const
37     { return model; }
38
39     int getMileage() const
40     { return mileage; }
41
42     double getPrice() const
43     { return price; }
44  };
45  #endif
```

Notice that the class has a default constructor in lines 18 through 22, and a constructor that accepts arguments for all of the class's attributes in lines 25 through 30. The `Automobile` class is a complete class that we can create objects from. If we wish, we can write a program that creates instances of the `Automobile` class. However, the `Automobile` class holds only general data about an automobile. It does not hold any of the specific pieces of data that the dealership wants to keep about cars, pickup trucks, and SUVs. To hold data about those specific types of automobiles we will write derived classes that inherit from the `Automobile` class. The following shows the code for the `Car` class. (This file is also stored in the Student Source Code Folder `Chapter 15\Automobile`.)

Contents of `Car.h`

```
1  #ifndef CAR_H
2  #define CAR_H
3  #include "Automobile.h"
4  #include <string>
5  using namespace std;
```

```
 6
 7  // The Car class represents a car.
 8  class Car : public Automobile
 9  {
10  private:
11     int doors;
12
13  public:
14     // Default constructor
15     Car() : Automobile()
16     { doors = 0;}
17
18     // Constructor #2
19     Car(string carMake, int carModel, int carMileage,
20         double carPrice, int carDoors) :
21         Automobile(carMake, carModel, carMileage, carPrice)
22     { doors = carDoors; }
23
24     // Accessor for doors attribute
25     int getDoors()
26     {return doors;}
27  };
28  #endif
```

The `Car` class defines a `doors` attribute in line 11 to hold the car's number of doors. The class has a default constructor in lines 15 through 16 that sets the `doors` attribute to 0. Notice in line 15 that the default constructor calls the `Automobile` class's default constructor, which initializes all of the inherited attributes to their default values.

The `Car` class also has an overloaded constructor, in lines 19 through 22, that accepts arguments for the car's make, model, mileage, price, and number of doors. Line 21 calls the `Automobile` class's constructor, passing the make, model, mileage, and price as arguments. Line 22 sets the value of the `doors` attribute.

Now let's look at the `Truck` class, which also inherits from the `Automobile` class. (This file is also stored in the Student Source Code Folder `Chapter 15\Automobile`.)

Contents of `Truck.h`

```
 1  #ifndef TRUCK_H
 2  #define TRUCK_H
 3  #include "Automobile.h"
 4  #include <string>
 5  using namespace std;
 6
 7  // The Truck class represents a truck.
 8  class Truck : public Automobile
 9  {
10  private:
11     string driveType;
12
13  public:
14     // Default constructor
15     Truck() : Automobile()
16     { driveType = ""; }
17
```

```
18      // Constructor #2
19      Truck(string truckMake, int truckModel, int truckMileage,
20         double truckPrice, string truckDriveType) :
21         Automobile(truckMake, truckModel, truckMileage, truckPrice)
22      { driveType = truckDriveType; }
23
24      // Accessor for driveType attribute
25      string getDriveType()
26      { return driveType; }
27  };
28  #endif
```

The `Truck` class defines a `driveType` attribute in line 11 to hold a string describing the truck's drive type. The class has a default constructor in lines 15 through 16 that sets the `driveType` attribute to an empty string. Notice in line 15 that the default constructor calls the `Automobile` class's default constructor, which initializes all of the inherited attributes to their default values.

The `Truck` class also has an overloaded constructor, in lines 19 through 22, that accepts arguments for the truck's make, model, mileage, price, and drive type. Line 21 calls the `Automobile` class's constructor, passing the make, model, mileage, and price as arguments. Line 22 sets the value of the `driveType` attribute.

Now let's look at the `SUV` class, which also inherits from the `Automobile` class. (This file is also stored in the Student Source Code Folder `Chapter 15\Automobile`.)

Contents of `SUV.h`

```
1   #ifndef SUV_H
2   #define SUV_H
3   #include "Automobile.h"
4   #include <string>
5   using namespace std;
6
7   // The SUV class represents a SUV.
8   class SUV : public Automobile
9   {
10  private:
11      int passengers;
12
13  public:
14      // Default constructor
15      SUV() : Automobile()
16      { passengers = 0; }
17
18      // Constructor #2
19      SUV(string SUVMake, int SUVModel, int SUVMileage,
20          double SUVPrice, int SUVPassengers) :
21          Automobile(SUVMake, SUVModel, SUVMileage, SUVPrice)
22      { passengers = SUVPassengers; }
23
24      // Accessor for passengers attribute
25      int getPassengers()
26      {return passengers;}
27  };
28  #endif
```

The SUV class defines a passengers attribute in line 11 to hold the number of passengers that the vehicle can accommodate. The class has a default constructor in lines 15 through 16 that sets the passengers attribute to 0. Notice in line 15 that the default constructor calls the Automobile class's default constructor, which initializes all of the inherited attributes to their default values.

The SUV class also has an overloaded constructor, in lines 19 through 22, that accepts arguments for the SUV's make, model, mileage, price, and number of passengers. Line 21 calls the Automobile class's constructor, passing the make, model, mileage, and price as arguments. Line 22 sets the value of the passengers attribute.

Program 15-6 demonstrates each of the derived classes. It creates a Car object, a Truck object, and an SUV object. (This file is also stored in the Student Source Code Folder Chapter 15\Automobile.)

Program 15-6

```
1  // This program demonstrates the Car, Truck, and SUV
2  // classes that are derived from the Automobile class.
3  #include <iostream>
4  #include <iomanip>
5  #include "Car.h"
6  #include "Truck.h"
7  #include "SUV.h"
8  using namespace std;
9
10 int main()
11 {
12     // Create a Car object for a used 2007 BMW with
13     // 50,000 miles, priced at $15,000, with 4 doors.
14     Car car("BMW", 2007, 50000, 15000.0, 4);
15
16     // Create a Truck object for a used 2006 Toyota
17     // pickup with 40,000 miles, priced at $12,000,
18     // with 4-wheel drive.
19     Truck truck("Toyota", 2006, 40000, 12000.0, "4WD");
20
21     // Create an SUV object for a used 2005 Volvo
22     // with 30,000 miles, priced at $18,000, with
23     // 5 passenger capacity.
24     SUV suv("Volvo", 2005, 30000, 18000.0, 5);
25
26     // Display the automobiles we have in inventory.
27     cout << fixed << showpoint << setprecision(2);
28     cout << "We have the following car in inventory:\n"
29          << car.getModel() << " " << car.getMake()
30          << " with " << car.getDoors() << " doors and "
31          << car.getMileage() << " miles.\nPrice: $"
32          << car.getPrice() << endl << endl;
33
```

```
34      cout << "We have the following truck in inventory:\n"
35           << truck.getModel() << " " << truck.getMake()
36           << " with " << truck.getDriveType()
37           << " drive type and " << truck.getMileage()
38           << " miles.\nPrice: $" << truck.getPrice()
39           << endl << endl;
40
41      cout << "We have the following SUV in inventory:\n"
42           << suv.getModel() << " " << suv.getMake()
43           << " with " << suv.getMileage() << " miles and "
44           << suv.getPassengers() << " passenger capacity.\n"
45           << "Price: $" << suv.getPrice() << endl;
46
47      return 0;
48  }
```

Program Output

```
We have the following car in inventory:
2007 BMW with 4 doors and 50000 miles.
Price: $15000.00

We have the following truck in inventory:
2006 Toyota with 4WD drive type and 40000 miles.
Price: $12000.00

We have the following SUV in inventory:
2005 Volvo with 30000 miles and 5 passenger capacity.
Price: $18000.00
```

Checkpoint

15.7 What will the following program display?

```
#include <iostream>
using namespace std;

class Sky
{
public:
   Sky()
      { cout << "Entering the sky.\n"; }
   ~Sky()
      { cout << "Leaving the sky.\n"; }
};

class Ground : public Sky
{
public:
   Ground()
      { cout << "Entering the Ground.\n"; }
   ~Ground()
      { cout << "Leaving the Ground.\n"; }
};
```

```
int main()
{
   Ground object;
   return 0;
}
```

15.8 What will the following program display?

```
#include <iostream>
using namespace std;

class Sky
{
public:
   Sky()
      { cout << "Entering the sky.\n"; }
   Sky(string color)
      { cout << "The sky is " << color << endl; }
   ~Sky()
      { cout << "Leaving the sky.\n"; }
};

class Ground : public Sky
{
public:
   Ground()
      { cout << "Entering the Ground.\n"; }
   Ground(string c1, string c2) : Sky(c1)
      { cout << "The ground is " << c2 << endl; }
   ~Ground()
      { cout << "Leaving the Ground.\n"; }
};

int main()
{
   Ground object;
   return 0;
}
```

15.4 Redefining Base Class Functions

CONCEPT: A base class member function may be redefined in a derived class.

VideoNote
Redefining a Base Class Function in a Derived Class

Inheritance is commonly used to extend a class or give it additional capabilities. Sometimes it may be helpful to overload a base class function with a function of the same name in the derived class. For example, recall the `GradedActivity` class that was presented earlier in this chapter:

```
class GradedActivity
{
protected:
   char letter;              // To hold the letter grade
   double score;             // To hold the numeric score
   void determineGrade();    // Determines the letter grade
```

```
public:
   // Default constructor
   GradedActivity()
      { letter = ' '; score = 0.0; }

   // Mutator function
   void setScore(double s)
      { score = s;
        determineGrade(); }

   // Accessor functions
   double getScore() const
      { return score; }

   char getLetterGrade() const
      { return letter; }
};
```

This class holds a numeric score and determines a letter grade based on that score. The `setScore` member function stores a value in score, then calls the `determineGrade` member function to determine the letter grade.

Suppose a teacher wants to "curve" a numeric score before the letter grade is determined. For example, Dr. Harrison determines that in order to curve the grades in her class she must multiply each student's score by a certain percentage. This gives an adjusted score, which is used to determine the letter grade.

The following `CurvedActivity` class is derived from the `GradedActivity` class. It multiplies the numeric score by a percentage, and passes that value as an argument to the base class's `setScore` function. (This file is stored in the Student Source Code Folder `Chapter 15\CurvedActivity`.)

Contents of `CurvedActivity.h`

```
 1  #ifndef CURVEDACTIVITY_H
 2  #define CURVEDACTIVITY_H
 3  #include "GradedActivity.h"
 4
 5  class CurvedActivity : public GradedActivity
 6  {
 7  protected:
 8       double rawScore;     // Unadjusted score
 9       double percentage;   // Curve percentage
10  public:
11       // Default constructor
12       CurvedActivity() : GradedActivity()
13          { rawScore = 0.0; percentage = 0.0; }
14
15       // Mutator functions
16       void setScore(double s)
17          { rawScore = s;
18            GradedActivity::setScore(rawScore * percentage); }
19
```

```
20       void setPercentage(double c)
21          { percentage = c; }
22
23       // Accessor functions
24       double getPercentage() const
25          { return percentage; }
26
27       double getRawScore() const
28          { return rawScore; }
29    };
30    #endif
```

This `CurvedActivity` class has the following member variables:

- `rawScore` This variable holds the student's unadjusted score.
- `percentage` This variable holds the value that the unadjusted score must be multiplied by to get the curved score.

It also has the following member functions:

- A default constructor that calls the `GradedActivity` default constructor, then sets `rawScore` and `percentage` to 0.0.
- `setScore` This function accepts an argument that is the student's unadjusted score. The function stores the argument in the `rawScore` variable, then passes `rawScore * percentage` as an argument to the base class's `setScore` function.
- `setPercentage` This function stores a value in the `percentage` variable.
- `getPercentage` This function returns the value in the `percentage` variable.
- `getRawScore` This function returns the value in the `rawScore` variable.

NOTE: Although we are not using the `CurvedActivity` class as a base class, it still has a protected member section. This is because we might want to use the `CurvedActivity` class itself as a base class, as you will see in the next section.

Notice that the `CurvedActivity` class has a `setScore` member function. This function has the same name as one of the base class member functions. When a derived class's member function has the same name as a base class member function, it is said that the derived class function *redefines* the base class function. When an object of the derived class calls the function, it calls the derived class's version of the function.

There is a distinction between redefining a function and overloading a function. An overloaded function is one with the same name as one or more other functions, but with a different parameter list. The compiler uses the arguments passed to the function to tell which version to call. Overloading can take place with regular functions that are not members of a class. Overloading can also take place inside a class when two or more member functions *of the same class* have the same name. These member functions must have different parameter lists for the compiler to tell them apart in function calls.

Redefining happens when a derived class has a function with the same name as a base class function. The parameter lists of the two functions can be the same because the derived class function is always called by objects of the derived class type.

Let's continue our look at the `CurvedActivity` class. Here is the `setScore` member function:

```
void setScore(double s)
   { rawScore = s;
     GradedActivity::setScore(rawScore * percentage); }
```

This function accepts an argument that should be the student's unadjusted numeric score, into the parameter `s`. This value is stored in the `rawScore` variable. Then the following statement is executed:

```
GradedActivity::setScore(rawScore * percentage);
```

This statement calls the base class's version of the `setScore` function with the expression `rawScore * percentage` passed as an argument. Notice that the name of the base class and the scope resolution operator precede the name of the function. This specifies that the base class's version of the `setScore` function is being called. A derived class function may call a base class function of the same name using this notation, which takes this form:

```
BaseClassName::functionName(ArgumentList);
```

Program 15-7 shows the `GradedActivity` and `CurvedActivity` classes used in a complete program. (This file is stored in the Student Source Code Folder `Chapter 15\CurvedActivity`.)

Program 15-7

```
1  // This program demonstrates a class that redefines
2  // a base class function.
3  #include <iostream>
4  #include <iomanip>
5  #include "CurvedActivity.h"
6  using namespace std;
7
8  int main()
9  {
10     double numericScore;  // To hold the numeric score
11     double percentage;    // To hold curve percentage
12
13     // Define a CurvedActivity object.
14     CurvedActivity exam;
15
16     // Get the unadjusted score.
17     cout << "Enter the student's raw numeric score: ";
18     cin >> numericScore;
19
20     // Get the curve percentage.
21     cout << "Enter the curve percentage for this student: ";
22     cin >> percentage;
23
24     // Send the values to the exam object.
25     exam.setPercentage(percentage);
26     exam.setScore(numericScore);
27
```

(program continues)

Program 15-7 *(continued)*

```
28       // Display the grade data.
29       cout << fixed << setprecision(2);
30       cout << "The raw score is "
31            << exam.getRawScore() << endl;
32       cout << "The curved score is "
33            << exam.getScore() << endl;
34       cout << "The curved grade is "
35            << exam.getLetterGrade() << endl;
36
37       return 0;
38   }
```

Program Output with Example Input Shown in Bold

```
Enter the student's raw numeric score: 87 [Enter]
Enter the curve percentage for this student: 1.06 [Enter]
The raw score is 87.00
The curved score is 92.22
The curved grade is A
```

It is important to note that even though a derived class may redefine a function in the base class, objects that are defined of the base class type still call the base class version of the function. This is demonstrated in Program 15-8.

Program 15-8

```
1   // This program demonstrates that when a derived class function
2   // overrides a   class function, objects of the base class
3   // still call the base class version of the function.
4   #include <iostream>
5   using namespace std;
6
7   class BaseClass
8   {
9   public:
10      void showMessage()
11         { cout << "This is the Base class.\n";}
12  };
13
14  class DerivedClass : public BaseClass
15  {
16  public:
17      void showMessage()
18         { cout << "This is the Derived class.\n"; }
19  };
20
21  int main()
22  {
23      BaseClass b;
24      DerivedClass d;
```

```
25
26       b.showMessage();
27       d.showMessage();
28
29       return 0;
30   }
```

Program Output

```
This is the Base class.
This is the Derived class.
```

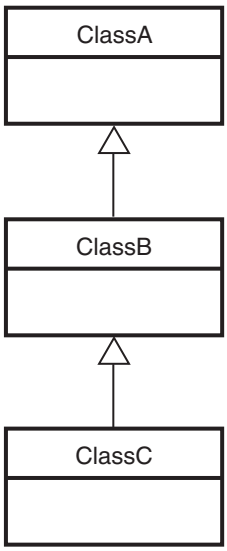
In Program 15-8, a class named `BaseClass` is declared with a member function named `showMessage`. A class named `DerivedClass` is then declared, also with a `showMessage` member function. As their names imply, `DerivedClass` is derived from `BaseClass`. Two objects, `b` and `d`, are defined in function `main`. The object `b` is a `BaseClass` object, and `d` is a `DerivedClass` object. When `b` is used to call the `showMessage` function, it is the `BaseClass` version that is executed. Likewise, when `d` is used to call `showMessage`, the `DerivedClass` version is used.

15.5 Class Hierarchies

CONCEPT: **A base class can also be derived from another class.**

Sometimes it is desirable to establish a hierarchy of classes in which one class inherits from a second class, which in turn inherits from a third class, as illustrated by Figure 15-4. In some cases, the inheritance of classes goes on for many layers.

Figure 15-4



In Figure 15-4, `ClassC` inherits `ClassB`'s members, including the ones that `ClassB` inherited from `ClassA`. Let's look at an example of such a chain of inheritance. Consider the following `PassFailActivity` class, which inherits from the `GradedActivity` class. The class is intended to determine a letter grade of 'P' for passing, or 'F' for failing. (This file is stored in the Student Source Code Folder `Chapter 15\PassFailActivity`.)

Contents of `PassFailActivity.h`

```
1  #ifndef PASSFAILACTIVITY_H
2  #define PASSFAILACTIVITY_H
3  #include "GradedActivity.h"
4
5  class PassFailActivity : public GradedActivity
6  {
7  protected:
8       double minPassingScore;  // Minimum passing score.
9  public:
10      // Default constructor
11      PassFailActivity() : GradedActivity()
12         { minPassingScore = 0.0; }
13
14      // Constructor
15      PassFailActivity(double mps) : GradedActivity()
16           { minPassingScore = mps; }
17
18      // Mutator
19      void setMinPassingScore(double mps)
20           { minPassingScore = mps; }
21
22      // Accessors
23      double getMinPassingScore() const
24           { return minPassingScore; }
25
26      char getLetterGrade() const;
27 };
28 #endif
```

The `PassFailActivity` class has a private member variable named `minPassingScore`. This variable holds the minimum passing score for an activity. The default constructor, in lines 11 through 12, sets `minPassingScore` to 0.0. An overloaded constructor in lines 15 through 16 accepts a `double` argument that is the minimum passing grade for the activity. This value is stored in the `minPassingScore` variable. The `getLetterGrade` member function is defined in the following `PassFailActivity.cpp` file. (This file is also stored in the Student Source Code Folder `Chapter 15\PassFailActivity`.)

Contents of `PassFailActivity.cpp`

```
1  #include "PassFailActivity.h"
2
3  //*****************************************************
4  // Member function PassFailActivity::getLetterGrade    *
5  // This function returns 'P' if the score is passing, *
6  // otherwise it returns 'F'.                           *
7  //*****************************************************
```

```
 8
 9  char PassFailActivity::getLetterGrade() const
10  {
11      char letterGrade;
12
13      if (score >= minPassingScore)
14          letterGrade = 'P';
15      else
16          letterGrade = 'F';
17
18      return letterGrade;
19  }
```

This `getLetterGrade` member function redefines the `getLetterGrade` member function of `GradedActivity` class. This version of the function returns a grade of `'P'` if the numeric score is greater than or equal to `minPassingScore`. Otherwise, the function returns a grade of `'F'`.

The `PassFailActivity` class represents the general characteristics of a student's pass-or-fail activity. There might be numerous types of pass-or-fail activities, however. Suppose we need a more specialized class, such as one that determines a student's grade for a pass-or-fail exam. The following `PassFailExam` class is an example. This class is derived from the `PassFailActivity` class. It inherits all of the members of `PassFailActivity`, including the ones that `PassFailActivity` inherits from `GradedActivity`. The `PassFailExam` class calculates the number of points that each question on the exam is worth, as well as the student's numeric score. (These files are stored in the Student Source Code Folder `Chapter 15\PassFailActivity`.)

Contents of `PassFailExam.h`

```
 1  #ifndef PASSFAILEXAM_H
 2  #define PASSFAILEXAM_H
 3  #include "PassFailActivity.h"
 4
 5  class PassFailExam : public PassFailActivity
 6  {
 7  private:
 8      int numQuestions;     // Number of questions
 9      double pointsEach;    // Points for each question
10      int numMissed;        // Number of questions missed
11  public:
12      // Default constructor
13      PassFailExam() : PassFailActivity()
14          { numQuestions = 0;
15            pointsEach = 0.0;
16            numMissed = 0; }
17
18      // Constructor
19      PassFailExam(int questions, int missed, double mps) :
20          PassFailActivity(mps)
21          { set(questions, missed); }
22
```

```
23      // Mutator function
24      void set(int, int);  // Defined in PassFailExam.cpp
25
26      // Accessor functions
27      double getNumQuestions() const
28          { return numQuestions; }
29
30      double getPointsEach() const
31          { return pointsEach; }
32
33      int getNumMissed() const
34          { return numMissed; }
35  };
36  #endif
```

Contents of `PassFailExam.cpp`

```
1   #include "PassFailExam.h"
2
3   //*******************************************************
4   // set function                                         *
5   // The parameters are the number of questions and the   *
6   // number of questions missed.                          *
7   //*******************************************************
8
9   void PassFailExam::set(int questions, int missed)
10  {
11      double numericScore; // To hold the numeric score
12
13      // Set the number of questions and number missed.
14      numQuestions = questions;
15      numMissed = missed;
16
17      // Calculate the points for each question.
18      pointsEach = 100.0 / numQuestions;
19
20      // Calculate the numeric score for this exam.
21      numericScore = 100.0 - (missed * pointsEach);
22
23      // Call the inherited setScore function to set
24      // the numeric score.
25      setScore(numericScore);
26  }
```

The `PassFailExam` class inherits all of the `PassFailActivity` class's members, including the ones that `PassFailActivity` inherited from `GradedActivity`. Because the `public` base class access specification is used, all of the protected members of `PassFailActivity` become protected members of `PassFailExam`, and all of the `public` members of `PassFailActivity` become `public` members of `PassFailExam`. Table 15-2 lists all of the member variables of the `PassFailExam` class, and Table 15-3 lists all the member functions. These include the members that were inherited from the base classes.

Table 15-2

Member Variable of the `PassFailExam` Class	Access	Inherited?
`numQuestions`	protected	No
`pointsEach`	protected	No
`numMissed`	protected	No
`minPassingScore`	protected	Yes, from `PassFailActivity`
`score`	protected	Yes, from `PassFailActivity`, which inherited it from `GradedActivity`

Table 15-3

Member Function of the `PassFailExam` Class	Access	Inherited?
`set`	public	No
`getNumQuestions`	public	No
`getPointsEach`	public	No
`getNumMissed`	public	No
`setMinPassingScore`	public	Yes, from `PassFailActivity`
`getMinPassingScore`	public	Yes, from `PassFailActivity`
`getLetterGrade`	public	Yes, from `PassFailActivity`
`setScore`	public	Yes, from `PassFailActivity`, which inherited it from `GradedActivity`
`getScore`	public	Yes, from `PassFailActivity`, which inherited it from `GradedActivity`

Program 15-9 demonstrates the `PassFailExam` class. This file is also stored in the student source code folder `Chapter 15\PassFailActivity`.

Program 15-9

```
1  // This program demonstrates the PassFailExam class.
2  #include <iostream>
3  #include <iomanip>
4  #include "PassFailExam.h"
5  using namespace std;
6
7  int main()
8  {
9       int questions;        // Number of questions
10      int missed;           // Number of questions missed
11      double minPassing;    // The minimum passing score
12
13      // Get the number of questions on the exam.
```

(program continues)

Program 15-9 *(continued)*

```
14       cout << "How many questions are on the exam? ";
15       cin >> questions;
16
17       // Get the number of questions the student missed.
18       cout << "How many questions did the student miss? ";
19       cin >> missed;
20
21       // Get the minimum passing score.
22       cout << "Enter the minimum passing score for this test: ";
23       cin >> minPassing;
24
25       // Define a PassFailExam object.
26       PassFailExam exam(questions, missed, minPassing);
27
28       // Display the test results.
29       cout << fixed << setprecision(1);
30       cout << "\nEach question counts "
31            << exam.getPointsEach() << " points.\n";
32       cout << "The minimum passing score is "
33            << exam.getMinPassingScore() << endl;
34       cout << "The student's exam score is "
35            << exam.getScore() << endl;
36       cout << "The student's grade is "
37            << exam.getLetterGrade() << endl;
38       return 0;
39   }
```

Program Output with Example Input Shown in Bold

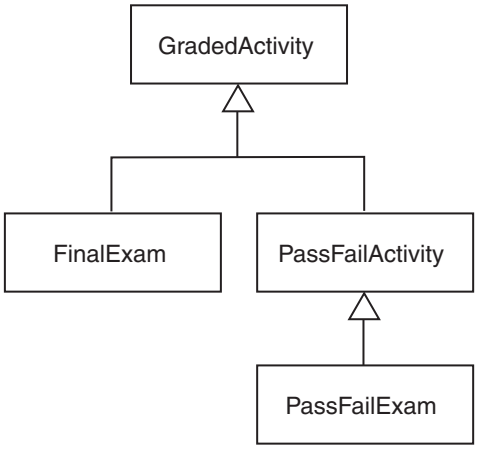
```
How many questions are on the exam? 100 [Enter]
How many questions did the student miss? 25 [Enter]
Enter the minimum passing score for this test: 60 [Enter]

Each question counts 1.0 points.
The minimum passing score is 60.0
The student's exam score is 75.0
The student's grade is P
```

This program uses the `PassFailExam` object to call the `getLetterGrade` member function in line 37. Recall that the `PassFailActivity` class redefines the `getLetterGrade` function to report only grades of 'P' or 'F'. Because the `PassFailExam` class is derived from the `PassFailActivity` class, it inherits the redefined `getLetterGrade` function.

Software designers often use class hierarchy diagrams. Like a family tree, a class hierarchy diagram shows the inheritance relationships between classes. Figure 15-5 shows a class hierarchy for the `GradedActivity`, `FinalExam`, `PassFailActivity`, and `PassFailExam` classes. The more general classes are toward the top of the tree and the more specialized classes are toward the bottom.

Figure 15-5



15.6 Polymorphism and Virtual Member Functions

CONCEPT: **Polymorphism allows an object reference variable or an object pointer to reference objects of different types and to call the correct member functions, depending upon the type of object being referenced.**

VideoNote
Polymorphism

Look at the following code for a function named `displayGrade`:

```
void displayGrade(const GradedActivity &activity)
{
   cout << setprecision(1) << fixed;
   cout << "The activity's numeric score is "
        << activity.getScore() << endl;
   cout << "The activity's letter grade is "
        << activity.getLetterGrade() << endl;
}
```

This function uses a `const GradedActivity` reference variable as its parameter. When a `GradedActivity` object is passed as an argument to this function, the function calls the object's `getScore` and `getLetterGrade` member functions to display the numeric score and letter grade. The following code shows how we might call the function.

```
GradedActivity test(88.0);   // The score is 88
displayGrade(test);          // Pass test to displayGrade
```

This code will produce the following output:

```
The activity's numeric score is 88.0
The activity's letter grade is B
```

Recall that the `GradedActivity` class is also the base class for the `FinalExam` class. Because of the "is-a" relationship between a base class and a derived class, an object of the `FinalExam` class is not just a `FinalExam` object. It is also a `GradedActivity` object.

(A final exam *is a* graded activity.) Because of this relationship, we can also pass a `FinalExam` object to the `displayGrade` function. For example, look at the following code:

```
// There are 100 questions. The student missed 25.
FinalExam test2(100, 25);
displayGrade(test2);
```

This code will produce the following output:

```
The activity's numeric score is 75.0
The activity's letter grade is C
```

Because the parameter in the `displayGrade` function is a `GradedActivity` reference variable, it can reference any object that is derived from `GradedActivity`. A problem can occur with this type of code, however, when redefined member functions are involved. For example, recall that the `PassFailActivity` class is derived from the `GradedActivity` class. The `PassFailActivity` class redefines the `getLetterGrade` function. Although we can pass a `PassFailActivity` object as an argument to the `displayGrade` function, we will not get the results we wish. This is demonstrated in Program 15-10. (This file is stored in the Student Source Code Folder `Chapter 15\PassFailActivity`.)

Program 15-10

```
1  #include <iostream>
2  #include <iomanip>
3  #include "PassFailActivity.h"
4  using namespace std;
5
6  // Function prototype
7  void displayGrade(const GradedActivity &);
8
9  int main()
10 {
11     // Create a PassFailActivity object. Minimum passing
12     // score is 70.
13     PassFailActivity test(70);
14
15     // Set the score to 72.
16     test.setScore(72);
17
18     // Display the object's grade data. The letter grade
19     // should be 'P'. What will be displayed?
20     displayGrade(test);
21     return 0;
22 }
23
24 //************************************************************
25 // The displayGrade function displays a GradedActivity object's *
26 // numeric score and letter grade.                             *
27 //************************************************************
28
29 void displayGrade(const GradedActivity &activity)
```

```
30  {
31       cout << setprecision(1) << fixed;
32       cout << "The activity's numeric score is "
33            << activity.getScore() << endl;
34       cout << "The activity's letter grade is "
35            << activity.getLetterGrade() << endl;
36  }
```

Program Output

```
The activity's numeric score is 72.0
The activity's letter grade is C
```

As you can see from the example output, the `getLetterGrade` member function returned 'C' instead of 'P'. This is because the `GradedActivity` class's `getLetterGrade` function was executed instead of the `PassFailActivity` class's version of the function.

This behavior happens because of the way C++ matches function calls with the correct function. This process is known as *binding*. In Program 15-10, C++ decides at compile time which version of the `getLetterGrade` function to execute when it encounters the call to the function in line 35. Even though we passed a `PassFailActivity` object to the `displayGrade` function, the `activity` parameter in the `displayGrade` function is a `GradedActivity` reference variable. Because it is of the `GradedActivity` type, the compiler binds the function call in line 35 with the `GradedActivity` class's `getLetterGrade` function. When the program executes, it has already been determined by the compiler that the `GradedActivity` class's `getLetterGrade` function will be called. The process of matching a function call with a function at compile time is called *static binding*.

To remedy this, the `getLetterGrade` function can be made *virtual*. A *virtual function* is a member function that is dynamically bound to function calls. In *dynamic binding*, C++ determines which function to call at runtime, depending on the type of the object responsible for the call. If a `GradedActivity` object is responsible for the call, C++ will execute the `GradedActivity::getLetterGrade` function. If a `PassFailActivity` object is responsible for the call, C++ will execute the `PassFailActivity::getLetterGrade` function.

Virtual functions are declared by placing the key word `virtual` before the return type in the base class's function declaration, such as

```
virtual char getLetterGrade() const;
```

This declaration tells the compiler to expect `getLetterGrade` to be redefined in a derived class. The compiler does not bind calls to the function with the actual function. Instead, it allows the program to bind calls, at runtime, to the version of the function that belongs to the same class as the object responsible for the call.

NOTE: You place the `virtual` key word only in the function's declaration or prototype. If the function is defined outside the class, you do not place the `virtual` key word in the function header.

The following code shows an updated version of the `GradedActivity` class, with the `getLetterGrade` function declared `virtual`. This file is stored in the Student Source Code Folder `Chapter 15\GradedActivity Version 3`. The `GradedActivity.cpp` file has not changed, so it is not shown again.

Contents of `GradedActivity.h` (Version 3)

```
1  #ifndef GRADEDACTIVITY_H
2  #define GRADEDACTIVITY_H
3
4  // GradedActivity class declaration
5
6  class GradedActivity
7  {
8  protected:
9      double score;  // To hold the numeric score
10 public:
11     // Default constructor
12     GradedActivity()
13        { score = 0.0; }
14
15     // Constructor
16     GradedActivity(double s)
17        { score = s; }
18
19     // Mutator function
20     void setScore(double s)
21        { score = s; }
22
23     // Accessor functions
24     double getScore() const
25        { return score; }
26
27     virtual char getLetterGrade() const;
28 };
29 #endif
```

The only change we have made to this class is to declare `getLetterGrade` as `virtual` in line 27. This tells the compiler not to bind calls to `getLetterGrade` with the function at compile time. Instead, calls to the function will be bound dynamically to the function at runtime.

When a member function is declared `virtual` in a base class, any redefined versions of the function that appear in derived classes automatically become `virtual`. So, it is not necessary to declare the `getLetterGrade` function in the `PassFailActivity` class as `virtual`. It is still a good idea to declare the function `virtual` in the `PassFailActivity` class for documentation purposes. A new version of the `PassFailActivity` class is shown here. This file is stored in the Student Source Code Folder `Chapter 15\GradedActivity Version 3`. The `PassFailActivity.cpp` file has not changed, so it is not shown again.

Contents of `PassFailActivity.h`

```
1  #ifndef PASSFAILACTIVITY_H
2  #define PASSFAILACTIVITY_H
3  #include "GradedActivity.h"
4
5  class PassFailActivity : public GradedActivity
```

```
 6  {
 7  protected:
 8       double minPassingScore;  // Minimum passing score
 9   public:
10       // Default constructor
11       PassFailActivity() : GradedActivity()
12          { minPassingScore = 0.0; }
13
14       // Constructor
15       PassFailActivity(double mps) : GradedActivity()
16          { minPassingScore = mps; }
17
18       // Mutator
19       void setMinPassingScore(double mps)
20          { minPassingScore = mps; }
21
22       // Accessors
23       double getMinPassingScore() const
24          { return minPassingScore; }
25
26       virtual char getLetterGrade() const;
27  };
28  #endif
```

The only change we have made to this class is to declare `getLetterGrade` as `virtual` in line 26. Program 15-11 is identical to Program 15-10, except it uses the corrected version of the `GradedActivity` and `PassFailActivity` classes. This file is also stored in the student source code folder `Chapter 15\GradedActivity Version 3`.

Program 15-11

```
 1  #include <iostream>
 2  #include <iomanip>
 3  #include "PassFailActivity.h"
 4  using namespace std;
 5
 6  // Function prototype
 7  void displayGrade(const GradedActivity &);
 8
 9  int main()
10  {
11       // Create a PassFailActivity object. Minimum passing
12       // score is 70.
13       PassFailActivity test(70);
14
15       // Set the score to 72.
16       test.setScore(72);
17
18       // Display the object's grade data. The letter grade
19       // should be 'P'. What will be displayed?
20       displayGrade(test);
21       return 0;
22  }
```

(program continues)

Program 15-11 *(continued)*

```
23
24  //************************************************************
25  // The displayGrade function displays a GradedActivity object's *
26  // numeric score and letter grade.                             *
27  //************************************************************
28
29  void displayGrade(const GradedActivity &activity)
30  {
31       cout << setprecision(1) << fixed;
32       cout << "The activity's numeric score is "
33            << activity.getScore() << endl;
34       cout << "The activity's letter grade is "
35            << activity.getLetterGrade() << endl;
36  }
```

Program Output

```
The activity's numeric score is 72.0
The activity's letter grade is P
```

Now that the `getLetterGrade` function is declared `virtual`, the program works properly. This type of behavior is known as polymorphism. The term *polymorphism* means the ability to take many forms. Program 15-12 demonstrates polymorphism by passing objects of the `GradedActivity` and `PassFailExam` classes to the `displayGrade` function. This file is stored in the Student Source Code Folder `Chapter 15\GradedActivity Version 3`.

Program 15-12

```
1   #include <iostream>
2   #include <iomanip>
3   #include "PassFailExam.h"
4   using namespace std;
5
6   // Function prototype
7   void displayGrade(const GradedActivity &);
8
9   int main()
10  {
11       // Create a GradedActivity object. The score is 88.
12       GradedActivity test1(88.0);
13
14       // Create a PassFailExam object. There are 100 questions,
15       // the student missed 25 of them, and the minimum passing
16       // score is 70.
17       PassFailExam test2(100, 25, 70.0);
18
19       // Display the grade data for both objects.
20       cout << "Test 1:\n";
21       displayGrade(test1);     // GradedActivity object
22       cout << "\nTest 2:\n";
```

```
23      displayGrade(test2);  // PassFailExam object
24      return 0;
25  }
26
27  //***************************************************************
28  // The displayGrade function displays a GradedActivity object's *
29  // numeric score and letter grade.                               *
30  //***************************************************************
31
32  void displayGrade(const GradedActivity &activity)
33  {
34      cout << setprecision(1) << fixed;
35      cout << "The activity's numeric score is "
36           << activity.getScore() << endl;
37      cout << "The activity's letter grade is "
38           << activity.getLetterGrade() << endl;
39  }
```

Program Output

```
Test 1:
The activity's numeric score is 88.0
The activity's letter grade is B

Test 2:
The activity's numeric score is 75.0
The activity's letter grade is P
```

Polymorphism Requires References or Pointers

The `displayGrade` function in Programs 15-11 and 15-12 uses a `GradedActivity` reference variable as its parameter. When we call the function, we pass an object by reference. Polymorphic behavior is not possible when an object is passed by value, however. For example, suppose the `displayGrade` function had been written as shown here:

```
// Polymorphic behavior is not possible with this function.
void displayGrade(const GradedActivity activity)
{
   cout << setprecision(1) << fixed;
   cout << "The activity's numeric score is "
        << activity.getScore() << endl;
   cout << "The activity's letter grade is "
        << activity.getLetterGrade() << endl;
}
```

In this version of the function the `activity` parameter is an object variable, not a reference variable. Suppose we call this version of the function with the following code:

```
// Create a GradedActivity object. The score is 88.
GradedActivity test1(88.0);

// Create a PassFailExam object. There are 100 questions,
// the student missed 25 of them, and the minimum passing
// score is 70.
PassFailExam test2(100, 25, 70.0);
```

```
// Display the grade data for both objects.
cout << "Test 1:\n";
displayGrade(test1);  // Pass the GradedActivity object
cout << "\nTest 2:\n";
displayGrade(&test2); // Pass the PassFailExam object
```

This code will produce the following output:

```
Test 1:
The activity's numeric score is 88.0
The activity's letter grade is B

Test 2:
The activity's numeric score is 75.0
The activity's letter grade is C
```

Even though the `getLetterGrade` function is declared `virtual`, static binding still takes place because `activity` is not a reference variable or a pointer.

Alternatively we could have used a `GradedActivity` pointer in the `displayGrade` function, as shown in Program 15-13. This file is also stored in the Student Source Code Folder `Chapter 15\GradedActivity Version 3`.

Program 15-13

```
1  #include <iostream>
2  #include <iomanip>
3  #include "PassFailExam.h"
4  using namespace std;
5
6  // Function prototype
7  void displayGrade(const GradedActivity *);
8
9  int main()
10 {
11     // Create a GradedActivity object. The score is 88.
12     GradedActivity test1(88.0);
13
14     // Create a PassFailExam object. There are 100 questions,
15     // the student missed 25 of them, and the minimum passing
16     // score is 70.
17     PassFailExam test2(100, 25, 70.0);
18
19     // Display the grade data for both objects.
20     cout << "Test 1:\n";
21     displayGrade(&test1);  // Address of the GradedActivity object
22     cout << "\nTest 2:\n";
23     displayGrade(&test2);  // Address of the PassFailExam object
24     return 0;
25 }
26
27 //*************************************************************
28 // The displayGrade function displays a GradedActivity object's *
29 // numeric score and letter grade. This version of the function *
30 // uses a GradedActivity pointer as its parameter.              *
31 //*************************************************************
```

```
32
33  void displayGrade(const GradedActivity *activity)
34  {
35       cout << setprecision(1) << fixed;
36       cout << "The activity's numeric score is "
37            << activity->getScore() << endl;
38       cout << "The activity's letter grade is "
39            << activity->getLetterGrade() << endl;
40  }
```

Program Output

```
Test 1:
The activity's numeric score is 88.0
The activity's letter grade is B

Test 2:
The activity's numeric score is 75.0
The activity's letter grade is P
```

Base Class Pointers

Pointers to a base class may be assigned the address of a derived class object. For example, look at the following code:

```
GradedActivity *exam = new PassFailExam(100, 25, 70.0);
```

This statement dynamically allocates a `PassFailExam` object and assigns its address to `exam`, which is a `GradedActivity` pointer. We can then use the `exam` pointer to call member functions, as shown here:

```
cout << exam->getScore() << endl;
cout << exam->getLetterGrade() << endl;
```

Program 15-14 is an example that uses base class pointers to reference derived class objects. This file is also stored in the Student Source Code Folder `Chapter 15\ GradedActivity Version 3`.

Program 15-14

```
1  #include <iostream>
2  #include <iomanip>
3  #include "PassFailExam.h"
4  using namespace std;
5
6  // Function prototype
7  void displayGrade(const GradedActivity *);
8
9  int main()
10 {
11      // Constant for the size of an array.
12      const int NUM_TESTS = 4;
13
```

(program continues)

Program 15-14 *(continued)*

```
14       // tests is an array of GradedActivity pointers.
15       // Each element of tests is initialized with the
16       // address of a dynamically allocated object.
17       GradedActivity *tests[NUM_TESTS] =
18            { new GradedActivity(88.0),
19              new PassFailExam(100, 25, 70.0),
20              new GradedActivity(67.0),
21              new PassFailExam(50, 12, 60.0)
22            };
23
24       // Display the grade data for each element in the array.
25       for (int count = 0; count < NUM_TESTS; count++)
26       {
27            cout << "Test #" << (count + 1) << ":\n";
28            displayGrade(tests[count]);
29            cout << endl;
30       }
31       return 0;
32  }
33
34  //***************************************************************
35  // The displayGrade function displays a GradedActivity object's *
36  // numeric score and letter grade. This version of the function *
37  // uses a GradedActivity pointer as its parameter.              *
38  //***************************************************************
39
40  void displayGrade(const GradedActivity *activity)
41  {
42       cout << setprecision(1) << fixed;
43       cout << "The activity's numeric score is "
44            << activity->getScore() << endl;
45       cout << "The activity's letter grade is "
46            << activity->getLetterGrade() << endl;
47  }
```

Program Output

```
Test #1:
The activity's numeric score is 88.0
The activity's letter grade is B

Test #2:
The activity's numeric score is 75.0
The activity's letter grade is P

Test #3:
The activity's numeric score is 67.0
The activity's letter grade is D

Test #4:
The activity's numeric score is 76.0
The activity's letter grade is P
```

Let's take a closer look at this program. An array named `tests` is defined in lines 17 through 22. This is an array of `GradedActivity` pointers. The array elements are initialized with the addresses of dynamically allocated objects. The `tests[0]` element is initialized with the address of the `GradedActivity` object returned from this expression:

```
new GradedActivity(88.0)
```

The `tests[1]` element is initialized with the address of the `GradedActivity` object returned from this expression:

```
new PassFailExam(100, 25, 70.0)
```

The `tests[2]` element is initialized with the address of the `GradedActivity` object returned from this expression:

```
new GradedActivity(67.0)
```

Finally, the `tests[3]` element is initialized with the address of the `GradedActivity` object returned from this expression:

```
new PassFailExam(50, 12, 60.0)
```

Although each element in the array is a `GradedActivity` pointer, some of the elements point to `GradedActivity` objects and some point to `PassFailExam` objects. The loop in lines 25 through 30 steps through the array, passing each pointer element to the `displayGrade` function.

Base Class Pointers and References Know Only About Base Class Members

Although a base class pointer can reference objects of any class that derives from the base class, there are limits to what the pointer can do with those objects. Recall that the `GradedActivity` class has, other than its constructors, only three member functions: `setScore`, `getScore`, and `getLetterGrade`. So, a `GradedActivity` pointer can be used to call only those functions, regardless of the type of object it points to. For example, look at the following code.

```
GradedActivity *exam = new PassFailExam(100, 25, 70.0);
cout << exam->getScore() << endl;         // This works.
cout << exam->getLetterGrade() << endl;   // This works.
cout << exam->getPointsEach() << endl;    // ERROR! Won't work!
```

In this code, `exam` is a `GradedActivity` pointer, and is assigned the address of a `PassFailExam` object. The `GradedActivity` class has only the `setScore`, `getScore`, and `getLetterGrade` member functions, so those are the only member functions that the `exam` variable knows how to execute. The last statement in this code is a call to the `getPointsEach` member function, which is defined in the `PassFailExam` class. Because the `exam` variable only knows about member functions in the `GradedActivity` class, it cannot execute this function.

The "Is-a" Relationship Does Not Work in Reverse

It is important to note that the "is-a" relationship does not work in reverse. Although the statement "a final exam is a graded activity" is true, the statement "a graded activity is a

final exam" is not true. This is because not all graded activities are final exams. Likewise, not all `GradedActivity` objects are `FinalExam` objects. So, the following code will not work.

```
// Create a GradedActivity object.
GradedActivity *gaPointer = new GradedActivity(88.0);

// Error! This will not work.
FinalExam *fePointer = gaPointer;
```

You cannot assign the address of a `GradedActivity` object to a `FinalExam` pointer. This makes sense because `FinalExam` objects have capabilities that go beyond those of a `GradedActivity` object. Interestingly, the C++ compiler will let you make such an assignment if you use a type cast, as shown here:

```
// Create a GradedActivity object.
GradedActivity *gaPointer = new GradedActivity(88.0);

// This will work, but with limitations.
FinalExam *fePointer = static_cast<FinalExam *>(gaPointer);
```

After this code executes, the derived class pointer `fePointer` will be pointing to a base class object. We can use the pointer to access members of the object, but only the members that exist. The following code demonstrates:

```
// This will work. The object has a getScore function.
cout << fePointer->getScore() << endl;

// This will work. The object has a getLetterGrade function.
cout << fePointer->getLetterGrade() << endl;

// This will compile, but an error will occur at runtime.
// The object does not have a getPointsEach function.
cout << fePointer->getPointsEach() << endl;
```

In this code `fePointer` is a `FinalExam` pointer, and it points to a `GradedActivity` object. The first two `cout` statements work because the `GradedActivity` object has `getScore` and a `getLetterGrade` member functions. The last `cout` statement will cause an error, however, because it calls the `getPointsEach` member function. The `GradedActivity` object does not have a `getPointsEach` member function.

Redefining vs. Overriding

Earlier in this chapter you learned how a derived class can redefine a base class member function. When a class redefines a virtual function, it is said that the class *overrides* the function. In C++, the difference between overriding and redefining base class functions is that overridden functions are dynamically bound, and redefined functions are statically bound. Only virtual functions can be overridden.

Virtual Destructors

When you write a class with a destructor, and that class could potentially become a base class, you should always declare the destructor `virtual`. This is because the compiler will perform static binding on the destructor if it is not declared `virtual`. This can lead to problems when a base class pointer or reference variable references a derived class object. If

the derived class has its own destructor, it will not execute when the object is destroyed or goes out of scope. Only the base class destructor will execute. Program 15-15 demonstrates.

Program 15-15

```
#include <iostream>
using namespace std;

// Animal is a base class.
class Animal
{
public:
     // Constructor
     Animal()
          { cout << "Animal constructor executing.\n"; }

     // Destructor
     ~Animal()
          { cout << "Animal destructor executing.\n"; }
};

// The Dog class is derived from Animal
class Dog : public Animal
{
public:
     // Constructor
     Dog() : Animal()
          { cout << "Dog constructor executing.\n"; }

     // Destructor
     ~Dog()
          { cout << "Dog destructor executing.\n"; }
};

//************************************************
// main function                                 *
//************************************************

int main()
{
     // Create a Dog object, referenced by an
     // Animal pointer.
     Animal *myAnimal = new Dog;

     // Delete the dog object.
     delete myAnimal;
     return 0;
}
```

Program Output

```
Animal constructor executing.
Dog constructor executing.
Animal destructor executing.
```

This program declares two classes: `Animal` and `Dog`. `Animal` is the base class and `Dog` is the derived class. Each class has its own constructor and destructor. In line 38, a `Dog` object is created, and its address is stored in an `Animal` pointer. Both the `Animal` and the `Dog` constructors execute. In line 41 the object is deleted. When this statement executes, however, only the `Animal` destructor executes. The `Dog` destructor does not execute because the object is referenced by an `Animal` pointer. We can fix this problem by declaring the `Animal` class destructor `virtual`, as shown in Program 15-16.

Program 15-16

```
1  #include <iostream>
2  using namespace std;
3
4  // Animal is a base class.
5  class Animal
6  {
7  public:
8       // Constructor
9       Animal()
10           { cout << "Animal constructor executing.\n"; }
11
12      // Destructor
13      virtual ~Animal()
14           { cout << "Animal destructor executing.\n"; }
15 };
16
17 // The Dog class is derived from Animal
18 class Dog : public Animal
19 {
20 public:
21      // Constructor
22      Dog() : Animal()
23           { cout << "Dog constructor executing.\n"; }
24
25      // Destructor
26      ~Dog()
27           { cout << "Dog destructor executing.\n"; }
28 };
29
30 //************************************************
31 // main function                                 *
32 //************************************************
33
34 int main()
35 {
36      // Create a Dog object, referenced by an
37      // Animal pointer.
38      Animal *myAnimal = new Dog;
39
40      // Delete the dog object.
41      delete myAnimal;
42      return 0;
43 }
```

Program Output

```
Animal constructor executing.
Dog constructor executing.
Dog destructor executing.
Animal destructor executing.
```

The only thing that has changed in this program is that the `Animal` class destructor is declared virtual in line 13. As a result, the destructor is dynamically bound at runtime. When the `Dog` object is destroyed, both the `Animal` and `Dog` destructors execute.

A good programming practice to follow is that any class that has a virtual member function should also have a virtual destructor. If the class doesn't require a destructor, it should have a virtual destructor that performs no statements. Remember, when a base class function is declared `virtual`, all overridden versions of the function in derived classes automatically become virtual. Including a virtual destructor in a base class, even one that does nothing, will ensure that any derived class destructors will also be virtual.

C++ 11's `override` and `final` Key Words

C++ 11 introduces the `override` key word to help prevent subtle errors when overriding virtual functions. For example, can you find the mistake in Program 15-17?

Program 15-17

```
1  // This program has a subtle error in the virtual functions.
2  #include <iostream>
3  using namespace std;
4
5  class Base
6  {
7  public:
8      virtual void functionA(int arg) const
9          { cout << "This is Base::functionA" << endl; }
10 };
11
12 class Derived : public Base
13 {
14 public:
15     virtual void functionA(long arg) const
16         { cout << "This is Derived::functionA" << endl; }
17 };
18
19 int main()
20 {
21     // Allocate instances of the Derived class.
22     Base *b = new Derived();
23     Derived *d = new Derived();
24
25     // Call functionA with the two pointers.
26     b->functionA(99);
```

(program continues)

Program 15-17 *(continued)*

```
27      d->functionA(99);
28
29      return 0;
30  }
```

Program Output

```
This is Base::functionA
This is Derived::functionA
```

Both the `Base` class and the `Derived` class have a virtual member function named `functionA`.

Notice that in lines 22 and 23 in the `main` function, we allocate two instances of the `Derived` class. We reference one of the instances with a `Base` class pointer (`b`), and we reference the other instance with a `Derived` class pointer (`d`). When we call `functionA` in lines 26 and 27, we might expect that the `Derived` class's `functionA` would be called in both lines. This is not the case, however, as you can see from the program's output.

The `functionA` in the `Derived` class does not override the `functionA` in the `Base` class because the function signatures are different. The `functionA` in the `Base` class takes an `int` argument, but the one in the `Derived` class takes a `long` argument. So, `functionA` in the `Derived` class merely overloads `functionA` in the `Base` class.

To make sure that a member function in a derived class overrides a virtual member function in a base class, you can use the `override` key word in the derived class's function prototype (or the function header, if the function is written inline). The `override` key word tells the compiler that the function is supposed to override a function in the base class. It will cause a compiler error if the function does not actually override any functions. Program 15-18 demonstrates how Program 15-17 can be fixed so that the `Derived` class function does, in fact, override the `Base` class function. Notice in line 15 that we have changed the parameter in the `Derived` class function to an `int`, and we have added the `override` key word to the function header.

Program 15-18

```
1   // This program demonstrates the override key word.
2   #include <iostream>
3   using namespace std;
4
5   class Base
6   {
7   public:
8       virtual void functionA(int arg) const
9       { cout << "This is Base::functionA" << endl; }
10  };
11
12  class Derived : public Base
13  {
14  public:
```

```
15      virtual void functionA(int arg) const override
16      { cout << "This is Derived::functionA" << endl; }
17  };
18
19  int main()
20  {
21      // Allocate instances of the Derived class.
22      Base *b = new Derived();
23      Derived *d = new Derived();
24
25      // Call functionA with the two pointers.
26      b->functionA(99);
27      d->functionA(99);
28
29      return 0;
30  }
```

Program Output

```
This is Derived::functionA
This is Derived::functionA
```

Preventing a Member Function from Being Overridden

In some derived classes, you might want to make sure that a virtual member function cannot be overridden any further down the class hierarchy. When a member function is declared with the `final` key word, it cannot be overridden in a derived class. The following member function prototype is an example that uses the `final` key word:

```
virtual void message() const final;
```

If a derived class attempts to override a `final` member function, the compiler generates an error.

15.7 Abstract Base Classes and Pure Virtual Functions

CONCEPT: **An abstract base class cannot be instantiated, but other classes are derived from it. A pure virtual function is a virtual member function of a base class that must be overridden. When a class contains a pure virtual function as a member, that class becomes an abstract base class.**

Sometimes it is helpful to begin a class hierarchy with an *abstract base class*. An abstract base class is not instantiated itself, but serves as a base class for other classes. The abstract base class represents the generic, or abstract, form of all the classes that are derived from it.

For example, consider a factory that manufactures airplanes. The factory does not make a generic airplane, but makes three specific types of planes: two different models of prop-driven planes, and one commuter jet model. The computer software that catalogs the planes might use an abstract base class called `Airplane`. That class has members representing the common characteristics of all airplanes. In addition, it has classes for each of the three specific airplane models the factory manufactures. These classes have members representing

the unique characteristics of each type of plane. The base class, `Airplane`, is never instantiated, but is used to derive the other classes.

A class becomes an abstract base class when one or more of its member functions is a *pure virtual function*. A pure virtual function is a virtual member function declared in a manner similar to the following:

```
virtual void showInfo() = 0;
```

The `= 0` notation indicates that `showInfo` is a pure virtual function. Pure virtual functions have no body, or definition, in the base class. They must be overridden in derived classes. Additionally, the presence of a pure virtual function in a class prevents a program from instantiating the class. The compiler will generate an error if you attempt to define an object of an abstract base class.

For example, look at the following abstract base class `Student`. It holds data common to all students, but does not hold all the data needed for students of specific majors.

Contents of `Student.h`

```
1  // Specification file for the Student class
2  #ifndef STUDENT_H
3  #define STUDENT_H
4  #include <string>
5  using namespace std;
6
7  class Student
8  {
9  protected:
10     string name;         // Student name
11     string idNumber;     // Student ID
12     int yearAdmitted;    // Year student was admitted
13 public:
14     // Default constructor
15     Student()
16          { name = "";
17            idNumber = "";
18            yearAdmitted = 0; }
19
20     // Constructor
21     Student(string n, string id, int year)
22     { set(n, id, year); }
23
24     // The set function sets the attribute data.
25     void set(string n, string id, int year)
26          { name = n;              // Assign the name
27            idNumber = id;         // Assign the ID number
28            yearAdmitted = year; } // Assign the year admitted
29
30     // Accessor functions
31     const string getName() const
32          { return name; }
33
34     const string getIdNum() const
35          { return idNumber; }
```

```
36
37      int getYearAdmitted() const
38          { return yearAdmitted; }
39
40      // Pure virtual function
41      virtual int getRemainingHours() const = 0;
42  };
43  #endif
```

The `Student` class contains members for storing a student's name, ID number, and year admitted. It also has constructors and a mutator function for setting values in the `name`, `idNumber`, and `yearAdmitted` members. Accessor functions are provided that return the values in the `name`, `idNumber`, and `yearAdmitted` members. A pure virtual function named `getRemainingHours` is also declared.

The pure virtual function must be overridden in classes derived from the `Student` class. It was made a pure virtual function because this class is intended to be the base for classes that represent students of specific majors. For example, a `CsStudent` class might hold the data for a computer science student, and a `BiologyStudent` class might hold the data for a biology student. Computer science students must take courses in different disciplines than those taken by biology students. It stands to reason that the `CsStudent` class will calculate the number of hours taken in a different manner than the `BiologyStudent` class.

Let's look at an example of the `CsStudent` class.

Contents of `CsStudent.h`

```
1   // Specification file for the CsStudent class
2   #ifndef CSSTUDENT_H
3   #define CSSTUDENT_H
4   #include "Student.h"
5
6   // Constants for required hours
7   const int MATH_HOURS = 20;     // Math hours
8   const int CS_HOURS = 40;       // Computer science hours
9   const int GEN_ED_HOURS = 60;   // General Ed hours
10
11  class CsStudent : public Student
12  {
13  private:
14      int mathHours;     // Hours of math taken
15      int csHours;       // Hours of Computer Science taken
16      int genEdHours;    // Hours of general education taken
17
18  public:
19      // Default constructor
20      CsStudent() : Student()
21          { mathHours = 0;
22            csHours = 0;
23            genEdHours = 0; }
24
25      // Constructor
26      CsStudent(string n, string id, int year) :
```

```
27          Student(n, id, year)
28          { mathHours = 0;
29            csHours = 0;
30            genEdHours = 0; }
31
32       // Mutator functions
33       void setMathHours(int mh)
34          { mathHours = mh; }
35
36       void setCsHours(int csh)
37          { csHours = csh; }
38
39       void setGenEdHours(int geh)
40          { genEdHours = geh; }
41
42       // Overridden getRemainingHours function,
43       // defined in CsStudent.cpp
44       virtual int getRemainingHours() const;
45  };
46  #endif
```

This file declares the following `const int` member variables in lines 7 through 9: `MATH_HOURS`, `CS_HOURS`, and `GEN_ED_HOURS`. These variables hold the required number of math, computer science, and general education hours for a computer science student. The `CsStudent` class, which derives from the `Student` class, declares the following member variables in lines 14 through 16: `mathHours`, `csHours`, and `genEdHours`. These variables hold the number of math, computer science, and general education hours taken by the student. Mutator functions are provided to store values in these variables. In addition, the class overrides the pure virtual `getRemainingHours` function in the `CsStudent.cpp` file.

Contents of `CsStudent.cpp`

```
1   #include <iostream>
2   #include "CsStudent.h"
3   using namespace std;
4
5   //*************************************************
6   // The CsStudent::getRemainingHours function returns *
7   // the number of hours remaining to be taken.       *
8   //*************************************************
9
10  int CsStudent::getRemainingHours() const
11  {
12       int reqHours,   // Total required hours
13       remainingHours; // Remaining hours
14
15       // Calculate the required hours.
16       reqHours = MATH_HOURS + CS_HOURS + GEN_ED_HOURS;
17
18       // Calculate the remaining hours.
19       remainingHours = reqHours - (mathHours + csHours +
20                        genEdHours);
```

```
21
22       // Return the remaining hours.
23       return remainingHours;
24   }
```

Program 15-19 provides a simple demonstration of the class.

Program 15-19

```
1  // This program demonstrates the CsStudent class, which is
2  // derived from the abstract base class, Student.
3  #include <iostream>
4  #include "CsStudent.h"
5  using namespace std;
6
7  int main()
8  {
9       // Create a CsStudent object for a student.
10     CsStudent student("Jennifer Haynes", "167W98337", 2006);
11
12     // Store values for Math, Computer Science, and General
13     // Ed hours.
14     student.setMathHours(12);   // Student has taken 12 Math hours
15     student.setCsHours(20);     // Student has taken 20 CS hours
16     student.setGenEdHours(40);  // Student has taken 40 Gen Ed hours
17
18     // Display the number of remaining hours.
19     cout << "The student " << student.getName()
20          << " needs to take " << student.getRemainingHours()
21          << " more hours to graduate.\n";
22
23     return 0;
24 }
```

Program Output

```
The student Jennifer Haynes needs to take 48 more hours to graduate.
```

Remember the following points about abstract base classes and pure virtual functions:

- When a class contains a pure virtual function, it is an abstract base class.
- Pure virtual functions are declared with the `= 0` notation.
- Abstract base classes cannot be instantiated.
- Pure virtual functions have no body, or definition, in the base class.
- A pure virtual function *must* be overridden at some point in a derived class in order for it to become nonabstract.

Checkpoint

15.9 Explain the difference between overloading a function and redefining a function.

15.10 Explain the difference between static binding and dynamic binding.

15.11 Are virtual functions statically bound or dynamically bound?

15.12 What will the following program display?

```
#include <iostream.>
using namespace std;

class First
{
protected:
   int a;
public:
   First(int x = 1)
      { a = x; }

   int getVal()
      { return a; }
};

class Second : public First
{
private:
   int b;

public:
   Second(int y = 5)
      { b = y; }
   int getVal()
      { return b; }
};

int main()
{
   First object1;
   Second object2;

   cout << object1.getVal() << endl;
   cout << object2.getVal() << endl;
   return 0;
}
```

15.13 What will the following program display?

```
#include <iostream>
using namespace std;

class First
{
protected:
   int a;
public:
   First(int x = 1)
      { a = x; }

   void twist()
      { a *= 2; }
   int getVal()
      { twist(); return a; }
};
```

```
class Second : public First
{
private:
   int b;
public:
   Second(int y = 5)
      { b = y; }

   void twist()
      { b *= 10; }
};

int main()
{
   First object1;
   Second object2;

   cout << object1.getVal() << endl;
   cout << object2.getVal() << endl;
   return 0;
}
```

15.14 What will the following program display?

```
#include <iostream>
using namespace std;

class First
{
protected:
   int a;
public:
   First(int x = 1)
      { a = x; }

   virtual void twist()
      { a *= 2; }

   int getVal()
      { twist(); return a; }
};

class Second : public First
{
private:
   int b;
public:
   Second(int y = 5)
      { b = y; }
   virtual void twist()
      { b *= 10; }
};

int main()
{
   First object1;
   Second object2;
```

```
    cout << object1.getVal() << endl;
    cout << object2.getVal() << endl;
    return 0;
}
```

15.15 What will the following program display?

```
#include <iostream>
using namespace std;

class Base
{
protected:
    int baseVar;
public:
    Base(int val = 2)
        { baseVar = val; }

    int getVar()
        { return baseVar; }
};

class Derived : public Base
{
private:
    int derivedVar;

public:
    Derived(int val = 100)
        { derivedVar = val; }
    int getVar()
        { return derivedVar; }
};

int main()
{
    Base *optr = nullptr;
    Derived object;

    optr = &object;
    cout << optr->getVar() << endl;
    return 0;
}
```

15.8 Multiple Inheritance

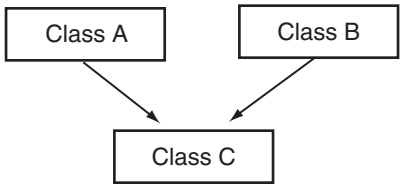
CONCEPT: Multiple inheritance is when a derived class has two or more base classes.

Previously we discussed how a class may be derived from a second class that is itself derived from a third class. The series of classes establishes a chain of inheritance. In such a scheme, you might be tempted to think of the lowest class in the chain as having multiple base classes. A base class, however, should be thought of as the class that another class is directly

derived from. Even though there may be several classes in a chain, each class (below the topmost class) only has one base class.

Another way of combining classes is through multiple inheritance. *Multiple inheritance* is when a class has two or more base classes. This is illustrated in Figure 15-6.

Figure 15-6



In Figure 15-6, class C is directly derived from classes A and B and inherits the members of both. Neither class A nor B, however, inherits members from the other. Their members are only passed down to class C. Let's look at an example of multiple inheritance. Consider the two classes declared here:

Contents of `Date.h`

```
1  // Specification file for the Date class
2  #ifndef DATE_H
3  #define DATE_H
4
5  class Date
6  {
7  protected:
8       int day;
9       int month;
10      int year;
11 public:
12      // Default constructor
13      Date(int d, int m, int y)
14         { day = 1; month = 1; year = 1900; }
15
16      // Constructor
17      Date(int d, int m, int y)
18         { day = d; month = m; year = y; }
19
20      // Accessors
21      int getDay() const
22         { return day; }
23
24      int getMonth() const
25         { return month; }
26
27      int getYear() const
28         { return year; }
29 };
30 #endif
```

Contents of `Time.h`

```
// Specification file for the Time class
#ifndef TIME_H
#define TIME_H

class Time
{
protected:
     int hour;
     int min;
     int sec;
public:
     // Default constructor
     Time()
        { hour = 0; min = 0; sec = 0; }

     // Constructor
     Time(int h, int m, int s)
        { hour = h; min = m; sec = s; }

     // Accessor functions
     int getHour() const
        { return hour; }

     int getMin() const
        { return min; }

     int getSec() const
        { return sec; }
};
#endif
```

These classes are designed to hold integers that represent the date and time. They both can be used as base classes for a third class we will call `DateTime`:

Contents of `DateTime.h`

```
// Specification file for the DateTime class
#ifndef DATETIME_H
#define DATETIME_H
#include <string>
#include "Date.h"
#include "Time.h"
using namespace std;

class DateTime : public Date, public Time
{
public:
     // Default constructor
     DateTime();

     // Constructor
     DateTime(int, int, int, int, int, int);
```

```
17
18      // The showDateTime function displays the
19      // date and the time.
20      void showDateTime() const;
21  };
22  #endif
```

In line 9, the first line in the `DateTime` declaration reads

```
class DateTime : public Date, public Time
```

Notice there are two base classes listed, separated by a *comma*. Each base class has its own access specification. The general format of the first line of a class declaration with multiple base classes is

```
class DerivedClassName : AccessSpecification BaseClassName,
                         AccessSpecification BaseClassName [, ...]
```

The notation in the square brackets indicates that the list of base classes with their access specifications may be repeated. (It is possible to have several base classes.)

Contents of `DateTime.cpp`

```
1   // Implementation file for the DateTime class
2   #include <iostream>
3   #include <string>
4   #include "DateTime.h"
5   using namespace std;
6
7   //***********************************************
8   // Default constructor                          *
9   // Note that this constructor does nothing other *
10  // than call default base class constructors.    *
11  //***********************************************
12  DateTime::DateTime() : Date(), Time()
13  {}
14
15  //***********************************************
16  // Constructor                                  *
17  // Note that this constructor does nothing other *
18  // than call base class constructors.           *
19  //***********************************************
20  DateTime::DateTime(int dy, int mon, int yr, int hr, int mt, int sc) :
21      Date(dy, mon, yr), Time(hr, mt, sc)
22  {}
23
24  //***********************************************
25  // The showDateTime member function displays the *
26  // date and the time.                           *
27  //***********************************************
28  void DateTime::showDateTime() const
29  {
30      // Display the date in the form MM/DD/YYYY.
31      cout << getMonth() << "/" << getDay() << "/" << getYear() << " ";
```

```
32
33       // Display the time in the form HH:MM:SS.
34       cout << getHour() << ":" << getMin() << ":" << getSec() << endl;
35   }
```

The class has two constructors: a default constructor and a constructor that accepts arguments for each component of a date and time. Let's look at the function header for the default constructor, in line 12:

```
DateTime::DateTime() : Date(), Time()
```

After the `DateTime` constructor's parentheses is a colon, followed by calls to the `Date` constructor and the `Time` constructor. The calls are separated by a comma. When using multiple inheritance, the general format of a derived class's constructor header is

```
DerivedClassName(ParameterList) : BaseClassName(ArgumentList),
                                  BaseClassName(ArgumentList)[, ...]
```

Look at the function header for the second constructor, which appears in lines 20 and 21:

```
DateTime::DateTime(int dy, int mon, int yr, int hr, int mt, int sc) :
    Date(dy, mon, yr), Time(hr, mt, sc)
```

This `DateTime` constructor accepts arguments for the day (`dy`), month (`mon`), year (`yr`), hour (`hr`), minute (`mt`), and second (`sc`). The `dy`, `mon`, and `yr` parameters are passed as arguments to the `Date` constructor. The `hr`, `mt`, and `sc` parameters are passed as arguments to the `Time` constructor.

The order that the base class constructor calls appear in the list does not matter. They are always called in the order of inheritance. That is, they are always called in the order they are listed in the first line of the class declaration. Here is line 9 from the `DateTime.h` file:

```
class DateTime : public Date, public Time
```

Because `Date` is listed before `Time` in the `DateTime` class declaration, the `Date` constructor will always be called first. If the classes use destructors, they are always called in reverse order of inheritance. Program 15-20 shows these classes in use.

Program 15-20

```
1  // This program demonstrates a class with multiple inheritance.
2  #include "DateTime.h"
3  using namespace std;
4
5  int main()
6  {
7       // Define a DateTime object and use the default
8       // constructor to initialize it.
9       DateTime emptyDay;
10
11      // Display the object's date and time.
12      emptyDay.showDateTime();
13
14      // Define a DateTime object and initialize it
15      // with the date 2/4/1960 and the time 5:32:27.
```

```
16      DateTime pastDay(2, 4, 1960, 5, 32, 27);
17
18      // Display the object's date and time.
19      pastDay.showDateTime();
20      return 0;
21  }
```

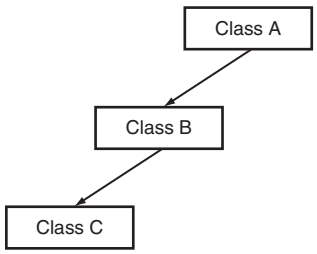
Program Output

```
1/1/1900 0:0:0
4/2/1960 5:32:27
```

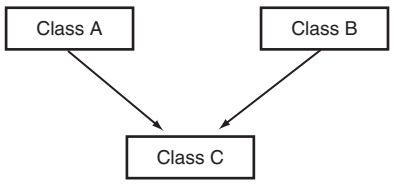
NOTE: It should be noted that multiple inheritance opens the opportunity for a derived class to have ambiguous members. That is, two base classes may have member variables or functions of the same name. In situations like these, the derived class should always redefine or override the member functions. Calls to the member functions of the appropriate base class can be performed within the derived class using the scope resolution operator (::). The derived class can also access the ambiguously named member variables of the correct base class using the scope resolution operator. If these steps aren't taken, the compiler will generate an error when it can't tell which member is being accessed.

Checkpoint

15.16 Does the following diagram depict multiple inheritance or a chain of inheritance?



15.17 Does the following diagram depict multiple inheritance or a chain of inheritance?



15.18 Examine the following classes. The table lists the variables that are members of the `Third` class (some are inherited). Complete the table by filling in the access specification each member will have in the `Third` class. Write "inaccessible" if a member is inaccessible to the `Third` class.

```
class First
{
   private:
```

```
      int a;
   protected:
      double b;
   public:
      long c;
};

class Second : protected First
{
   private:
      int d;
   protected:
      double e;
   public:
      long f;
};

class Third : public Second
{
   private:
      int g;
   protected:
      double h;
   public:
      long i;
};
```

Member Variable	Access Specification in Third Class
a	
b	
c	
d	
e	
f	
g	
h	
i	

15.19 Examine the following class declarations:

```
class Van
{
protected:
   int passengers;
public:
   Van(int p)
   { passengers = p; }
};
```

```
class FourByFour
{
protected:
   double cargoWeight;
public:
   FourByFour(float w)
      { cargoWeight = w; }
};
```

Write the declaration of a class named `SportUtility`. The class should be derived from both the `Van` and `FourByFour` classes above. (This should be a case of multiple inheritance, where both `Van` and `FourByFour` are base classes.)

Review Questions and Exercises

Short Answer

1. What is an "is a" relationship?
2. A program uses two classes: `Dog` and `Poodle`. Which class is the base class and which is the derived class?
3. How does base class access specification differ from class member access specification?
4. What is the difference between a protected class member and a private class member?
5. Can a derived class ever directly access the private members of its base class?
6. Which constructor is called first, that of the derived class or the base class?
7. What is the difference between redefining a base class function and overriding a base class function?
8. When does static binding take place? When does dynamic binding take place?
9. What is an abstract base class?
10. A program has a class `Potato`, which is derived from the class `Vegetable`, which is derived from the class `Food`. Is this an example of multiple inheritance? Why or why not?
11. What base class is named in the line below?

    ```
    class Pet : public Dog
    ```

12. What derived class is named in the line below?

    ```
    class Pet : public Dog
    ```

13. What is the class access specification of the base class named below?

    ```
    class Pet : public Dog
    ```

14. What is the class access specification of the base class named below?

    ```
    class Pet : Fish
    ```

15. Protected members of a base class are like _____ members, except they may be accessed by derived classes.
16. Complete the table on the next page by filling in private, protected, public, or inaccessible in the right-hand column:

In a private base class, this base class MEMBER access specification…	…becomes this access specification in the derived class.
`private`	
`protected`	
`public`	

17. Complete the table below by filling in private, protected, public, or inaccessible in the right-hand column:

In a protected base class, this base class MEMBER access specification…	…becomes this access specification in the derived class.
`private`	
`protected`	
`public`	

18. Complete the table below by filling in private, protected, public, or inaccessible in the right-hand column:

In a public base class, this base class MEMBER access specification…	…becomes this access specification in the derived class.
`private`	
`protected`	
`public`	

Fill-in-the-Blank

19. A derived class inherits the _____ of its base class.
20. When both a base class and a derived class have constructors, the base class's constructor is called _____ (first/last).
21. When both a base class and a derived class have destructors, the base class's constructor is called _____ (first/last).
22. An overridden base class function may be called by a function in a derived class by using the _____ operator.
23. When a derived class redefines a function in a base class, which version of the function do objects that are defined of the base class call? _____
24. A(n) _____ member function in a base class expects to be overridden in a derived class.
25. _____ binding is when the compiler binds member function calls at compile time.
26. _____ binding is when a function call is bound at runtime.
27. _____ is when member functions in a class hierarchy behave differently, depending upon which object performs the call.
28. When a pointer to a base class is made to point to a derived class, the pointer ignores any _____ the derived class performs, unless the function is _____.

29. A(n) _____ class cannot be instantiated.
30. A(n) _____ function has no body, or definition, in the class in which it is declared.
31. A(n) _____ of inheritance is where one class is derived from a second class, which in turn is derived from a third class.
32. _____ is where a derived class has two or more base classes.
33. In multiple inheritance, the derived class should always _____ a function that has the same name in more than one base class.

Algorithm Workbench

34. Write the first line of the declaration for a `Poodle` class. The class should be derived from the `Dog` class with public base class access.
35. Write the first line of the declaration for a `SoundSystem` class. Use multiple inheritance to base the class on the `CDplayer` class, the `Tuner` class, and the `CassettePlayer` class. Use public base class access in all cases.
36. Suppose a class named `Tiger` is derived from both the `Felis` class and the `Carnivore` class. Here is the first line of the `Tiger` class declaration:

```
class Tiger : public Felis, public Carnivore
```

Here is the function header for the Tiger constructor:

```
Tiger(int x, int y) : Carnivore(x), Felis(y)
```

Which base class constructor is called first, `Carnivore` or `Felis`?

37. Write the declaration for class `B`. The class's members should be
    - `m`, an integer. This variable should not be accessible to code outside the class or to member functions in any class derived from class `B`.
    - `n`, an integer. This variable should not be accessible to code outside the class, but should be accessible to member functions in any class derived from class `B`.
    - `setM`, `getM`, `setN`, and `getN`. These are the set and get functions for the member variables `m` and `n`. These functions should be accessible to code outside the class.
    - `calc`, a public virtual member function that returns the value of `m` times `n`.

    Next write the declaration for class `D`, which is derived from class `B`. The class's members should be
    - `q`, a `float`. This variable should not be accessible to code outside the class but should be accessible to member functions in any class derived from class `D`.
    - `r`, a `float`. This variable should not be accessible to code outside the class, but should be accessible to member functions in any class derived from class `D`.
    - `setQ`, `getQ`, `setR`, and `getR`. These are the set and get functions for the member variables `q` and `r`. These functions should be accessible to code outside the class.
    - `calc`, a public member function that overrides the base class `calc` function. This function should return the value of `q` times `r`.

True or False

38. T F The base class's access specification affects the way base class member functions may access base class member variables.
39. T F The base class's access specification affects the way the derived class inherits members of the base class.

40. T F Private members of a private base class become inaccessible to the derived class.
41. T F Public members of a private base class become private members of the derived class.
42. T F Protected members of a private base class become public members of the derived class.
43. T F Public members of a protected base class become private members of the derived class.
44. T F Private members of a protected base class become inaccessible to the derived class.
45. T F Protected members of a public base class become public members of the derived class.
46. T F The base class constructor is called after the derived class constructor.
47. T F The base class destructor is called after the derived class destructor.
48. T F It isn't possible for a base class to have more than one constructor.
49. T F Arguments are passed to the base class constructor by the derived class constructor.
50. T F A member function of a derived class may not have the same name as a member function of the base class.
51. T F Pointers to a base class may be assigned the address of a derived class object.
52. T F A base class may not be derived from another class.

Find the Errors

Each of the class declarations and/or member function definitions below has errors. Find as many as you can.

53.
```
class Car, public Vehicle
{
   public:
      Car();
      ~Car();
   protected:
      int passengers;
}
```

54.
```
class Truck, public : Vehicle, protected
{
   private:
      double cargoWeight;
   public:
      Truck();
      ~Truck();
};
```

55.
```
class SnowMobile : Vehicle
{
   protected:
```

```
        int horsePower;
        double weight;
    public:
        SnowMobile(int h, double w), Vehicle(h)
            { horsePower = h; }
        ~SnowMobile();
};
```

56.
```
class Table : public Furniture
{
   protected:
      int numSeats;
   public:
      Table(int n) : Furniture(numSeats)
         { numSeats = n; }
      ~Table();
};
```

57.
```
class Tank : public Cylinder
{
   private:
      int fuelType;
      double gallons;
   public:
      Tank();
      ~Tank();
      void setContents(double);
      void setContents(double);
};
```

58.
```
class Three : public Two : public One
{
   protected:
      int x;
   public:
      Three(int a, int b, int c), Two(b), Three(c)
         { x = a; }
      ~Three();
};
```

Programming Challenges

VideoNote
Solving the `Employee` and `ProductionWorker` Classes Problem

1. **`Employee` and `ProductionWorker` Classes**

   Design a class named `Employee`. The class should keep the following information in

   - Employee name
   - Employee number
   - Hire date

   Write one or more constructors and the appropriate accessor and mutator functions for the class.

Next, write a class named `ProductionWorker` that is derived from the `Employee` class. The `ProductionWorker` class should have member variables to hold the following information:

- Shift (an integer)
- Hourly pay rate (a `double`)

The workday is divided into two shifts: day and night. The shift variable will hold an integer value representing the shift that the employee works. The day shift is shift 1, and the night shift is shift 2. Write one or more constructors and the appropriate accessor and mutator functions for the class. Demonstrate the classes by writing a program that uses a `ProductionWorker` object.

2. **`ShiftSupervisor` Class**

   In a particular factory a shift supervisor is a salaried employee who supervises a shift. In addition to a salary, the shift supervisor earns a yearly bonus when his or her shift meets production goals. Design a `ShiftSupervisor` class that is derived from the `Employee` class you created in Programming Challenge 1. The `ShiftSupervisor` class should have a member variable that holds the annual salary and a member variable that holds the annual production bonus that a shift supervisor has earned. Write one or more constructors and the appropriate accessor and mutator functions for the class. Demonstrate the class by writing a program that uses a `ShiftSupervisor` object.

3. **`TeamLeader` Class**

   In a particular factory, a team leader is an hourly paid production worker who leads a small team. In addition to hourly pay, team leaders earn a fixed monthly bonus. Team leaders are required to attend a minimum number of hours of training per year. Design a `TeamLeader` class that extends the `ProductionWorker` class you designed in Programming Challenge 1. The `TeamLeader` class should have member variables for the monthly bonus amount, the required number of training hours, and the number of training hours that the team leader has attended. Write one or more constructors and the appropriate accessor and mutator functions for the class. Demonstrate the class by writing a program that uses a `TeamLeader` object.

4. **Time Format**

   In Program 15-20, the file `Time.h` contains a `Time` class. Design a class called `MilTime` that is derived from the `Time` class. The `MilTime` class should convert time in military (24-hour) format to the standard time format used by the `Time` class. The class should have the following member variables:

   `milHours`: Contains the hour in 24-hour format. For example, 1:00 pm would be stored as 1300 hours, and 4:30 pm would be stored as 1630 hours.

   `milSeconds`: Contains the seconds in standard format.

   The class should have the following member functions:

   `Constructor`: The constructor should accept arguments for the hour and seconds, in military format. The time should then be converted to standard time and stored in the `hours`, `min`, and `sec` variables of the `Time` class.

   `setTime`: Accepts arguments to be stored in the `milHours` and `milSeconds` variables. The time should then be converted to standard time and stored in the `hours`, `min`, and `sec` variables of the `Time` class.

`getHour`: Returns the hour in military format.

`getStandHr`: Returns the hour in standard format.

Demonstrate the class in a program that asks the user to enter the time in military format. The program should then display the time in both military and standard format.

Input Validation: The `MilTime` *class should not accept hours greater than 2359, or less than 0. It should not accept seconds greater than 59 or less than 0.*

5. Time Clock

Design a class named `TimeClock`. The class should be derived from the `MilTime` class you designed in Programming Challenge 4. The class should allow the programmer to pass two times to it: starting time and ending time. The class should have a member function that returns the amount of time elapsed between the two times. For example, if the starting time is 900 hours (9:00 am), and the ending time is 1300 hours (1:00 pm), the elapsed time is 4 hours.

Input Validation: The class should not accept hours greater than 2359 or less than 0.

6. `Essay` class

Design an `Essay` class that is derived from the `GradedActivity` class presented in this chapter. The `Essay` class should determine the grade a student receives on an essay. The student's essay score can be up to 100, and is determined in the following manner:

- Grammar: 30 points
- Spelling: 20 points
- Correct length: 20 points
- Content: 30 points

Demonstrate the class in a simple program.

7. `PersonData` and `CustomerData` classes

Design a class named `PersonData` with the following member variables:

- `lastName`
- `firstName`
- `address`
- `city`
- state
- `zip`
- `phone`

Write the appropriate accessor and mutator functions for these member variables.

Next, design a class named `CustomerData`, which is derived from the `PersonData` class. The `CustomerData` class should have the following member variables:

- `customerNumber`
- `mailingList`

The `customerNumber` variable will be used to hold a unique integer for each customer. The `mailingList` variable should be a `bool`. It will be set to `true` if the customer wishes to be on a mailing list, or `false` if the customer does not wish to be on a mailing list. Write appropriate accessor and mutator functions for these member variables. Demonstrate an object of the `CustomerData` class in a simple program.

8. **`PreferredCustomer` Class**

   A retail store has a preferred customer plan where customers may earn discounts on all their purchases. The amount of a customer's discount is determined by the amount of the customer's cumulative purchases in the store.

   - When a preferred customer spends $500, he or she gets a 5% discount on all future purchases.
   - When a preferred customer spends $1,000, he or she gets a 6% discount on all future purchases.
   - When a preferred customer spends $1,500, he or she gets a 7% discount on all future purchases.
   - When a preferred customer spends $2,000 or more, he or she gets a 10% discount on all future purchases.

   Design a class named `PreferredCustomer`, which is derived from the `CustomerData` class you created in Programming Challenge 7. The `PreferredCustomer` class should have the following member variables:

   - `purchasesAmount` (a `double`)
   - `discountLevel` (a `double`)

   The `purchasesAmount` variable holds the total of a customer's purchases to date. The `discountLevel` variable should be set to the correct discount percentage, according to the store's preferred customer plan. Write appropriate member functions for this class and demonstrate it in a simple program.

   Input Validation: Do not accept negative values for any sales figures.

9. **File Filter**

   A file filter reads an input file, transforms it in some way, and writes the results to an output file. Write an abstract file filter class that defines a pure virtual function for transforming a character. Create one derived class of your file filter class that performs encryption, another that transforms a file to all uppercase, and another that creates an unchanged copy of the original file. The class should have the following member function:

   ```
   void doFilter(ifstream &in, ofstream &out)
   ```

   This function should be called to perform the actual filtering. The member function for transforming a single character should have the prototype:

   ```
   char transform(char ch)
   ```

   The encryption class should have a constructor that takes an integer as an argument and uses it as the encryption key.

10. **File Double-Spacer**

    Create a derived class of the abstract filter class of Programming Challenge 9 that double-spaces a file: that is, it inserts a blank line between any two lines of the file.

11. **Course Grades**

    In a course, a teacher gives the following tests and assignments:

    - A **lab activity** that is observed by the teacher and assigned a numeric score.
    - A **pass/fail exam** that has 10 questions. The minimum passing score is 70.
    - An **essay** that is assigned a numeric score.
    - A **final exam** that has 50 questions.

Write a class named `CourseGrades`. The class should have a member named `grades` that is an array of `GradedActivity` pointers. The `grades` array should have four elements, one for each of the assignments previously described. The class should have the following member functions:

`setLab`: This function should accept the address of a `GradedActivity` object as its argument. This object should already hold the student's score for the lab activity. Element 0 of the `grades` array should reference this object.

`setPassFailExam`: This function should accept the address of a `PassFailExam` object as its argument. This object should already hold the student's score for the pass/fail exam. Element 1 of the `grades` array should reference this object.

`setEssay`: This function should accept the address of an `Essay` object as its argument. (See Programming Challenge 6 for the `Essay` class. If you have not completed Programming Challenge 6, use a `GradedActivity` object instead.) This object should already hold the student's score for the essay. Element 2 of the `grades` array should reference this object.

`setPassFailExam`: This function should accept the address of a `FinalExam` object as its argument. This object should already hold the student's score for the final exam. Element 3 of the `grades` array should reference this object.

`print`: This function should display the numeric scores and grades for each element in the `grades` array.

Demonstrate the class in a program.

12. `Ship`, `CruiseShip`, and `CargoShip` Classes

Design a `Ship` class that has the following members:

- A member variable for the name of the ship (a string)
- A member variable for the year that the ship was built (a string)
- A constructor and appropriate accessors and mutators
- A virtual `print` function that displays the ship's name and the year it was built.

Design a `CruiseShip` class that is derived from the `Ship` class. The `CruiseShip` class should have the following members:

- A member variable for the maximum number of passengers (an `int`)
- A constructor and appropriate accessors and mutators
- A `print` function that overrides the `print` function in the base class. The `CruiseShip` class's print function should display only the ship's name and the maximum number of passengers.

Design a `CargoShip` class that is derived from the `Ship` class. The `CargoShip` class should have the following members:

- A member variable for the cargo capacity in tonnage (an `int`).
- A constructor and appropriate accessors and mutators.
- A `print` function that overrides the `print` function in the base class. The `CargoShip` class's print function should display only the ship's name and the ship's cargo capacity.

Demonstrate the classes in a program that has an array of `Ship` pointers. The array elements should be initialized with the addresses of dynamically allocated `Ship`, `CruiseShip`, and `CargoShip` objects. (See Program 15-14, lines 17 through 22, for an example of how to do this.) The program should then step through the array, calling each object's `print` function.

13. Pure Abstract Base Class Project

Define a pure abstract base class called `BasicShape`. The `BasicShape` class should have the following members:

Private Member Variable:

`area, a double used to hold the shape's area.`

Public Member Functions:

`getArea.` This function should return the value in the member variable area.

`calcArea.` This function should be a pure virtual function.

Next, define a class named `Circle`. It should be derived from the `BasicShape` class. It should have the following members:

Private Member Variables:

`centerX,` a long integer used to hold the x coordinate of the circle's center.

`centerY`, a long integer used to hold the y coordinate of the circle's center.

`radius`, a double used to hold the circle's radius.

Public Member Functions:

constructor—accepts values for `centerX, centerY`, and `radius.` Should call the overridden `calcArea` function described below.

`getCenterX`—returns the value in `centerX.`

`getCenterY`—returns the value in `centerY.`

`calcArea`—calculates the area of the circle (area = 3.14159 * radius * radius) and stores the result in the inherited member area.

Next, define a class named `Rectangle`. It should be derived from the `BasicShape` class. It should have the following members:

Private Member Variables:

`width`, a long integer used to hold the width of the rectangle.

`length`, a long integer used to hold the length of the rectangle.

Public Member Functions:

constructor—accepts values for width and length. Should call the overridden `calcArea` function described below.

`getWidth`—returns the value in width.

`getLength`—returns the value in length.

`calcArea`—calculates the area of the rectangle (area = length * width) and stores the result in the inherited member area.

After you have created these classes, create a driver program that defines a `Circle` object and a `Rectangle` object. Demonstrate that each object properly calculates and reports its area.

Group Project

14. Bank Accounts

This program should be designed and written by a team of students. Here are some suggestions:

- One or more students may work on a single class.
- The requirements of the program should be analyzed so each student is given about the same work load.
- The parameters and return types of each function and class member function should be decided in advance.
- The program will be best implemented as a multi-file program.

Design a generic class to hold the following information about a bank account:

Balance

Number of deposits this month

Number of withdrawals

Annual interest rate

Monthly service charges

The class should have the following member functions:

Constructor: Accepts arguments for the balance and annual interest rate.

`deposit`: A virtual function that accepts an argument for the amount of the deposit. The function should add the argument to the account balance. It should also increment the variable holding the number of deposits.

`withdraw`: A virtual function that accepts an argument for the amount of the withdrawal. The function should subtract the argument from the balance. It should also increment the variable holding the number of withdrawals.

`calcInt`: A virtual function that updates the balance by calculating the monthly interest earned by the account, and adding this interest to the balance. This is performed by the following formulas:

Monthly Interest Rate = (Annual Interest Rate / 12)
Monthly Interest = Balance * Monthly Interest Rate
Balance = Balance + Monthly Interest

`monthlyProc`: A virtual function that subtracts the monthly service charges from the balance, calls the `calcInt` function, and then sets the variables that hold the number of withdrawals, number of deposits, and monthly service charges to zero.

Next, design a savings account class, derived from the generic account class. The savings account class should have the following additional member:

```
status (to represent an active or inactive account)
```

If the balance of a savings account falls below $25, it becomes inactive. (The `status` member could be a flag variable.) No more withdrawals may be made until the balance is raised above $25, at which time the account becomes active again. The savings account class should have the following member functions:

`withdraw`: A function that checks to see if the account is inactive before a withdrawal is made. (No withdrawal will be allowed if the account is not active.) A withdrawal is then made by calling the base class version of the function.

`deposit`: A function that checks to see if the account is inactive before a deposit is made. If the account is inactive and the deposit brings the balance above $25, the account becomes active again. The deposit is then made by calling the base class version of the function.

`monthlyProc`: Before the base class function is called, this function checks the number of withdrawals. If the number of withdrawals for the month is more than 4, a service charge of $1 for each withdrawal above 4 is added to the base class variable that holds the monthly service charges. (Don't forget to check the account balance after the service charge is taken. If the balance falls below $25, the account becomes inactive.)

Next, design a checking account class, also derived from the generic account class. It should have the following member functions:

`withdraw`: Before the base class function is called, this function will determine if a withdrawal (a check written) will cause the balance to go below $0. If the balance goes below $0, a service charge of $15 will be taken from the account. (The withdrawal will not be made.) If there isn't enough in the account to pay the service charge, the balance will become negative and the customer will owe the negative amount to the bank.

`monthlyProc`: Before the base class function is called, this function adds the monthly fee of $5 plus $0.10 per withdrawal (check written) to the base class variable that holds the monthly service charges.

Write a complete program that demonstrates these classes by asking the user to enter the amounts of deposits and withdrawals for a savings account and checking account. The program should display statistics for the month, including beginning balance, total amount of deposits, total amount of withdrawals, service charges, and ending balance.

NOTE: You may need to add more member variables and functions to the classes than those listed above.