9 Classes

Nothing can have value without being an object of utility.
—Karl Marx

Your public servants serve you right.
—Adlai E. Stevenson

Is it a world to hide virtues in?
—William Shakespeare

Objectives

In this chapter you'll learn:

- How to define a class and use it to create an object.
- To implement a class's behaviors as member functions.
- How to implement a class's attributes as data members.
- How to use a constructor to initialize an object's data when the object is created.
- How to separate a class's interface and implementation.
- To prevent multiple definition errors with preprocessor wrappers.
- To understand class scope and accessing class members.
- How destructors are used to perform "termination housekeeping" on an object.
- When constructors and destructors are called.
- The logic errors that may occur when a `public` member function returns a reference to `private` data.

Outline

9.1 Introduction
9.2 Classes, Objects, Member Functions and Data Members
9.3 `Time` Class
9.4 Class Scope and Accessing Class Members
9.5 Placing a Class in a Separate File for Reusability
9.6 `Time` Class: Separating Interface from Implementation
9.7 Access Functions and Utility Functions
9.8 `Time` Class: Constructors with Default Arguments
9.9 Destructors
9.10 When Constructors and Destructors Are Called
9.11 `Time` Class: Using *Set* and *Get* Functions
9.12 `Time` Class: A Subtle Trap—Returning a Reference to a `private` Data Member
9.13 Default Memberwise Assignment
9.14 Wrap-Up

Summary | *Terminology* | *Self-Review Exercises* | *Answers to Self-Review Exercises* | *Exercises* | *Making a Difference*

9.1 Introduction

In this chapter, you'll begin writing programs that employ the basic concepts of object-oriented programming that we introduced in Section 1.19. Most of the programs you've studied so far in this book consisted of a `main` function that performed statements, or a `main` function that called other functions to perform statements. Typically, programs consist of function `main` and one or more classes, each containing data members and member functions. If you become part of a development team in industry, you might work on software systems that contain hundreds, or even thousands, of classes.

In this chapter, we develop a simple, carefully engineered framework for organizing object-oriented programs. Among the examples, we use an integrated `Time` class case study to demonstrate several class construction capabilities.

Next, we discuss class scope and the relationships among class members. We demonstrate how client code can access a class's `public` members via three types of "handles"—the name of an object, a reference to an object or a pointer to an object. As you'll see, object names and references can be used with the dot (`.`) member selection operator to access a `public` member, and pointers can be used with the arrow (`->`) member selection operator.

We then revisit the `Time` class to demonstrates an important software engineering concept—separating interface from implementation. We discuss access functions that can read or display data in an object. A common use of access functions is to test the truth or falsity of conditions—such functions are known as predicate functions. We also demonstrate the notion of a utility function (also called a helper function)—a `private` member function that supports the operation of the class's `public` member functions, but is not intended for use by clients of the class.

In the second `Time` class case study example, we demonstrate how to pass arguments to constructors and show how default arguments can be used in a constructor to enable client code to initialize objects using a variety of arguments. We discuss a special member function called a destructor that's part of every class and is used to perform "termination housekeeping" on an object before it's destroyed. We demonstrate the order in which con-

structors and destructors are called, because your programs' correctness depends on using properly initialized objects that have not yet been destroyed. The next version of the `Time` class case study introduces *set* and *get* functions, which enable client code to modify and access a class's `private` data in a controlled manner. Such functions help ensure that the class's data remains valid.

Our last example of the `Time` class case study in this chapter shows a dangerous programming practice in which a member function returns a reference to `private` data. We discuss how this breaks the encapsulation of a class, allowing client code to directly access an object's data. This last example shows that objects of the same class can be assigned to one another using default memberwise assignment, which copies the data members in the object on the right side of the assignment into the corresponding data members of the object on the left side of the assignment. The chapter concludes with a discussion of software reusability, one of the key benefits of object-oriented programming.

9.2 Classes, Objects, Member Functions and Data Members

Let's begin with a simple analogy to help you reinforce your understanding from Section 1.19 of classes and their contents. Suppose you want to drive a car and make it go faster by pressing down on its accelerator pedal. What must happen before you can do this? Well, before you can drive a car, someone has to *design* it and *build* it. A car typically begins as engineering drawings, similar to the blueprints used to design a house. These drawings include the design for an accelerator pedal that the driver will use to make the car go faster. In a sense, the pedal "hides" the complex mechanisms that actually make the car go faster, just as the brake pedal "hides" the mechanisms that slow the car, the steering wheel "hides" the mechanisms that turn the car and so on. This enables people with little or no knowledge of how cars are engineered to drive a car easily, simply by using the accelerator pedal, the brake pedal, the steering wheel, the transmission shifting mechanism and other such simple and user-friendly "interfaces" to the car's complex internal mechanisms.

Of course, you cannot drive the engineering drawings of a car—so before you can drive a car, it must be built from the engineering drawings that describe it. A completed car will have an *actual* accelerator pedal to make the car go faster. But even that's not enough—the car will (hopefully) not accelerate on its own, so the driver must press the accelerator pedal to tell the car to go faster.

Now let's use our car example to introduce the key object-oriented programming concepts of this section. Performing a task in a program requires a function. The function describes the mechanisms that actually perform its tasks. The function hides from its user the complex tasks that it performs, just as the accelerator pedal of a car hides from the driver the complex mechanisms of making the car go faster. In C++, we begin by creating a program unit called a class to house a function, just as a car's engineering drawings house the design of an accelerator pedal. Recall from Section 1.19 that a function belonging to a class is called a member function. In a class, you provide one or more member functions that are designed to perform the class's tasks. For example, a class that represents a bank account might contain a member function to deposit money into the account, another to withdraw money from the account and a third to inquire what the current account balance is.

Just as you cannot drive an engineering drawing of a car, you cannot "drive" a class. Just as someone has to build a car from its engineering drawings before you can actually

drive the car, *you must create an object of a class before you can get a program to perform the tasks the class describes*. That's one reason C++ is known as an object-oriented programming language. Note also that just as *many* cars can be built from the same engineering drawing, *many* objects can be built from the same class.

When you drive a car, pressing its gas pedal sends a message to the car to perform a task—that is, make the car go faster. Similarly, you send **messages** to an object—each message is known as a **member-function call** and tells a member function of the object to perform its task. This also is called **requesting a service from an object**.

Thus far, we've used the car analogy to introduce classes, objects and member functions. In addition to the capabilities a car provides, it also has many attributes, such as its color, the number of doors, the amount of gas in its tank, its current speed and its total miles driven (i.e., its odometer reading). Like the car's capabilities, these attributes are represented as part of a car's design in its engineering diagrams. As you drive a car, these attributes are always associated with the car. Every car maintains its own attributes. For example, each car knows how much gas is in its *own* gas tank, but *not* how much is in the tanks of other cars. Similarly, an object has attributes that are carried with it as it's used in a program. These attributes are specified as part of the object's class. For example, a bank account object has a balance attribute that represents the amount of money in the account. Each bank account object knows the balance in the account it represents, but not the balances of the other accounts in the bank. Attributes are specified by the class's data members.

9.3 Time Class

We begin with an example (Fig. 9.1) that consists of class `Time` (lines 8–19), which represents the time of day in 24-hour clock format, the class's member functions (lines 23–50) and a `main` function (lines 52–79) that creates and manipulates a `Time` object. Function `main` uses this object and its member functions to set and display the time in both 24-hour and 12-hour formats.

```
1  // Fig. 9.1: fig09_01.cpp
2  // Time class.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  // Time class definition
8  class Time
9  {
10 public:
11    Time(); // constructor
12    void setTime( int, int, int ); // set hour, minute and second
13    void printUniversal(); // print time in universal-time format
14    void printStandard(); // print time in standard-time format
15 private:
16    int hour; // 0 - 23 (24-hour clock format)
17    int minute; // 0 - 59
18    int second; // 0 - 59
19 }; // end class Time
```

Fig. 9.1 | `Time` class definition. (Part 1 of 3.)

```
20
21 // Time constructor initializes each data member to zero.
22 // Ensures all Time objects start in a consistent state.
23 Time::Time()
24 {
25    hour = minute = second = 0;
26 } // end Time constructor
27
28 // set new Time value using universal time; ensure that
29 // the data remains consistent by setting invalid values to zero
30 void Time::setTime( int h, int m, int s )
31 {
32    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
33    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
34    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
35 } // end function setTime
36
37 // print Time in universal-time format (HH:MM:SS)
38 void Time::printUniversal()
39 {
40    cout << setfill( '0' ) << setw( 2 ) << hour << ":"
41       << setw( 2 ) << minute << ":" << setw( 2 ) << second;
42 } // end function printUniversal
43
44 // print Time in standard-time format (HH:MM:SS AM or PM)
45 void Time::printStandard()
46 {
47    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ) << ":"
48       << setfill( '0' ) << setw( 2 ) << minute << ":" << setw( 2 )
49       << second << ( hour < 12 ? " AM" : " PM" );
50 } // end function printStandard
51
52 int main()
53 {
54    Time t; // instantiate object t of class Time
55
56    // output Time object t's initial values
57    cout << "The initial universal time is ";
58    t.printUniversal(); // 00:00:00
59    cout << "\nThe initial standard time is ";
60    t.printStandard(); // 12:00:00 AM
61
62    t.setTime( 13, 27, 6 ); // change time
63
64    // output Time object t's new values
65    cout << "\n\nUniversal time after setTime is ";
66    t.printUniversal(); // 13:27:06
67    cout << "\nStandard time after setTime is ";
68    t.printStandard(); // 1:27:06 PM
69
70    t.setTime( 99, 99, 99 ); // attempt invalid settings
71
```

Fig. 9.1 | `Time` class definition. (Part 2 of 3.)

```
72      // output t's values after specifying invalid values
73      cout << "\n\nAfter attempting invalid settings:"
74         << "\nUniversal time: ";
75      t.printUniversal(); // 00:00:00
76      cout << "\nStandard time: ";
77      t.printStandard(); // 12:00:00 AM
78      cout << endl;
79   } // end main
```

```
The initial universal time is 00:00:00
The initial standard time is 12:00:00 AM

Universal time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

Fig. 9.1 | `Time` class definition. (Part 3 of 3.)

`Time` Class Definition

Before function `main` can create a `Time` object, we must tell the compiler what member functions and data members belong to the class—a process known as **defining the class**. The `Time` **class definition** (lines 8–19) begins with keyword **`class`** followed by the class name `Time` (line 8). By convention, the name of a class begins with a capital letter, and for readability, each subsequent word in the class name begins with a capital letter. This capitalization style is often referred to as **camel case**, because the pattern of uppercase and lowercase letters resembles the silhouette of a camel.

Every class's **body** is enclosed in a pair of left and right braces (`{` and `}`), as in lines 8 and 19. The class definition terminates with a semicolon (line 19).

Common Programming Error 9.1

Forgetting the semicolon at the end of a class definition is a syntax error.

Declaring the Class's `public` Services

Line 10 contains the **access-specifier label** **`public:`**. The keyword **`public`** is an **access specifier**. Lines 11–14 contain function prototypes that declare the class's member functions—`Time`, `setTime`, `printUniversal` and `printStandard`. These are the `public` member functions of the class (also known as the `public` services, `public` behaviors or interface of the class). They appear after access specifier `public:` to indicate that these functions are "available to the public"—that is, they can be called by other functions in the program (such as `main`), and by member functions of other classes (if there are any). Access specifiers are always followed by a colon (`:`). For the remainder of the text, when we refer to the access specifier `public`, we'll omit the colon as we did in this sentence.

The `public` member functions will be used by **clients** (i.e., portions of a program that are users) of the class to manipulate the class's data. These services allow the client code to

interact with an object of the class. We'll soon see that classes can have non-`public` member functions as well.

The member function with the same name as the class is called a **constructor**. This is a special member function that initializes the data members of a class object. A class's constructor is called automatically when a program creates an object of that class. If a class does not explicitly include a constructor, the compiler provides a **default constructor**—that is, a constructor with no parameters. Later, you'll see that it's common to have several constructors for a class, enabling objects to be initialized several ways. Constructors cannot specify a return type; otherwise, a compilation error occurs.

Declaring the Class's `private` Data Members

An object of class `Time` must provide a way to store the current time. Lines 16–18 declare three integer members to represent the hour, minute and second, respectively. These declarations appear after the access-specifier label **`private:`**. Like `public`, keyword `private` is an access specifier. Variables or functions declared after access specifier `private` (and before the next access specifier) are accessible only to member functions of the class for which they're declared. Thus, data members `hour`, `minute` and `second` can be used only in member functions `Time`, `setTime`, `printUniversal` and `printStandard` of (every object of) class `Time`. These data members, because they're `private`, cannot be accessed by functions outside the class (such as `main`) or by member functions of other classes in the program. Declaring data members with access specifier `private` is known as **data hiding**.

Normally, data members are listed in the `private` portion of a class and member functions are listed in the `public` portion. It's possible to have `private` member functions and `public` data, as we'll see later; using `public` data is uncommon and is considered poor software engineering.

Software Engineering Observation 9.1

Generally, data members should be declared `private` and member functions should be declared `public`. It's appropriate to declare certain member functions `private`, if they're to be accessed only by other member functions of the class.

Common Programming Error 9.2

An attempt by a function, which is not a member of a particular class (or a `friend` of that class, as we'll see in Chapter 10, Classes: A Deeper Look), to access a `private` member of that class is a compilation error.

Good Programming Practice 9.1

For clarity and readability, use each access specifier only once in a class definition. Place `public` members first, where they're easy to locate.

Software Engineering Observation 9.2

Each element of a class should be `private` unless it can be proven that the element needs to be `public`. This is another example of the principle of least privilege.

Class `Time`'s Constructor

The `Time` constructor (lines 23–26) initializes the data members to 0—the universal-time equivalent of 12 AM. This ensures that the object begins in a *consistent* state. Invalid values

cannot be stored in the data members of a Time object, because the constructor is called when the Time object is created, and all subsequent attempts by a client to modify the data members are scrutinized by function setTime (discussed shortly). In Chapter 16, Exception Handling, we use exceptions to indicate when a value is out of range, rather than simply assigning a default consistent value.

The data members of a class cannot be initialized where they're declared in the class body. It's strongly recommended that these data members be initialized by the class's constructor (as there's no default initialization for fundamental-type data members). Data members can also be assigned values by Time's other member functions. Chapter 10 demonstrates that only a class's static const data members of integral or enum types can be initialized in the class's body.

Class Time's setTime Member Function

Function setTime (lines 30–35) is a public function that declares three int parameters and uses them to set the time. A conditional expression tests each argument to determine whether the value is in a specified range. For example, the hour value (line 32) must be greater than or equal to 0 and less than 24, because the universal-time format represents hours as integers from 0 to 23 (e.g., 1 PM is hour 13 and 11 PM is hour 23; midnight is hour 0 and noon is hour 12). Similarly, both minute and second values (lines 33 and 34) must be greater than or equal to 0 and less than 60. Any values outside these ranges are set to zero to ensure that a Time object always contains consistent data—that is, the object's data values are always kept in range, even if the values provided as arguments to function setTime were incorrect. In this example, zero is a consistent value for hour, minute and second.

Class Time's printUniversal Member Function

Function printUniversal (lines 38–42) takes no arguments and outputs the time in universal-time format, consisting of three colon-separated pairs of digits for the hour, minute and second. For example, if the time were 1:30:07 PM, function printUniversal would display 13:30:07. Line 40 uses parameterized stream manipulator **setfill** to specify the **fill character** that's displayed when an integer is output in a field wider than the number of digits in the value. By default, the fill characters appear to the *left* of the digits in the number. In this example, if the minute value is 2, it will be displayed as 02, because the fill character is set to zero ('0'). If the number being output fills the specified field, the fill character will not be displayed. Once the fill character is specified with setfill, it applies for all subsequent values that are displayed in fields wider than the value being displayed—that is, setfill is a "sticky" setting. This is in contrast to setw, which applies only to the next value displayed—that is, setw is a "nonsticky" setting.

Error-Prevention Tip 9.1

Each sticky setting (such as a fill character or floating-point precision) should be restored to its previous setting when it's no longer needed. Failure to do so may result in incorrectly formatted output later in a program. Chapter 15, Stream Input/Output, discusses how to reset the fill character and precision.

Class Time's printStandard Member Function

Function printStandard (lines 45–50) takes no arguments and outputs the date in standard-time format, consisting of the hour, minute and second values separated by colons

and followed by an AM or PM indicator (e.g., `1:27:06 PM`). Like function `printUniversal`, function `printStandard` uses `setfill('0')` to format the `minute` and `second` as two digit values with leading zeros if necessary. Line 47 uses the conditional operator (`?:`) to determine the value of `hour` to be displayed—if the `hour` is `0` or `12` (AM or PM, respectively), it appears as 12; otherwise, the `hour` appears as a value from 1 to 11. The conditional operator in line 49 determines whether AM or PM will be displayed.

Defining Member Functions Outside the Class Definition; Class Scope

Each member-function name in the function headers (lines 23, 30, 38 and 45) is preceded by the class name and `::`, which is known as the **binary scope resolution operator**. This "ties" each member function to the `Time` class definition (lines 8–19), which declares the class's member functions and data members. Without "`Time::`" preceding each function name, these functions would not be recognized by the compiler as member functions of class `Time`—the compiler would consider them "free" or "loose" functions, like `main`. These are also called global functions. Such functions cannot access class `Time`'s `private` data or call its member functions, without specifying an object. So, the compiler would not be able to compile these functions. For example, line 25, which accesses variables `hour`, `minute` and `second`, would cause compilation errors because these variables are not declared as local variables in the constructor—the compiler wouldn't know that they are already declared as data members of class `Time`.

A member function that's declared in a class definition, but defined outside that class definition and "tied" to the class via the binary scope resolution operator is still within that **class's scope**—its name is known only to other members of the class unless referred to via an object of the class, a reference to an object of the class, a pointer to an object of the class or the binary scope resolution operator. We'll say more about class scope shortly.

Common Programming Error 9.3

When defining a class's member functions outside that class, omitting the class name and binary scope resolution operator (`::`) preceding the function names causes compilation errors.

Using Class `Time`

Next, we'd like to use class `Time` in a program. As you learned in Chapter 2, function `main` begins the execution of every program. In this program, we'd like to call class `Time`'s member functions to display and set the time. Typically, you cannot call a member function of a class until you create an object of that class. (As you'll learn in Section 10.6, `static` member functions are an exception.) Line 54 creates an object of class `Time` called `t`. The variable's type is `Time`. When we declare variables of type `int`, as we did in Chapter 2, the compiler knows what `int` is—it's a fundamental type. In line 54, however, the compiler does not automatically know what type `Time` is—it's a **user-defined type**. We tell the compiler what `Time` is by including the class definition (lines 8–19). If we omitted these lines, the compiler would issue an error message (such as "`'Time': undeclared identifier`" in Microsoft Visual C++ or "`'Time': undeclared`" in GNU C++). Each class you create becomes a new type that can be used to create objects. You can define new class types as needed; this is one reason why C++ is known as an **extensible language**. Once class `Time` has

been defined, it can be used as a type in object, array, pointer and reference declarations as follows:

```
Time sunset; // object of type Time
Time arrayOfTimes[ 5 ]; // array of 5 Time objects
Time &dinnerTime = sunset; // reference to a Time object
Time *timePtr = &dinnerTime; // pointer to a Time object
```

When the object is instantiated (line 54), the `Time` constructor is called to initialize each `private` data member to `0`. Lines 58 and 60 print the time in universal and standard formats, respectively, to confirm that the members were initialized properly. These two member-function calls each use variable `t` followed by the **dot operator** (`.`), the function name and an empty set of parentheses. These calls cause the `printUniversal` and `printStandard` functions to perform their tasks. At the beginning of line 58, "`t.`" indicates that `main` should use the `Time` object that was created in line 54. The empty parentheses indicate that member function `printUniversal` does not require any arguments.

Line 62 sets a new time by calling member function `setTime`, and lines 66 and 68 print the time again in both formats. Line 70 attempts to use `setTime` to set the data members to invalid values—function `setTime` recognizes this and sets the invalid values to 0 to maintain the object in a consistent state. Finally, lines 75 and 77 print the time again in both formats.

More on private Data Members

Note that the data members `hour`, `minute` and `second` (lines 16–18) are preceded by the `private` member access specifier (line 15). A class's `private` data members normally are not accessible outside the class. The philosophy here is that the data representation used within the class is of no concern to the class's clients. For example, it would be perfectly reasonable for the class to represent the time internally as the number of seconds since midnight. Clients could use the same `public` member functions and get the same results without being aware of this. In this sense, the implementation of a class is said to be *hidden* from its clients. Such **information hiding** promotes program modifiability and simplifies the client's perception of a class.

Classes simplify programming because the client (or user of the class object) need only be concerned with the operations encapsulated or embedded in the object. Such operations are usually designed to be client oriented rather than implementation oriented. Clients need not be concerned with a class's implementation (although the client, of course, wants a correct and efficient implementation). Interfaces do change, but less frequently than implementations. When an implementation changes, implementation-dependent code must change accordingly. Hiding the implementation eliminates the possibility of other program parts becoming dependent on the details of the class implementation.

Member Functions vs. Global Functions

The `printUniversal` and `printStandard` member functions take no arguments, because these member functions implicitly know that they're to print the data members of the particular `Time` object on which they're invoked. This can make member function calls more concise than conventional function calls in procedural programming.

Software Engineering Observation 9.3

Using an object-oriented programming approach often simplifies function calls by reducing the number of parameters. This benefit of object-oriented programming derives from the fact that encapsulating data members and member functions within an object gives the member functions the right to access the data members.

Software Engineering Observation 9.4

Member functions are usually shorter than functions in non-object-oriented programs, because the data stored in data members have ideally been validated by a constructor or by member functions that store new data. Because the data is already in the object, the member-function calls often have no arguments or fewer arguments than typical function calls in non-object-oriented languages. Thus, the calls are shorter, the function definitions are shorter and the function prototypes are shorter. This improves many aspects of program development.

Error-Prevention Tip 9.2

The fact that member function calls generally take either no arguments or substantially fewer arguments than conventional function calls in non-object-oriented languages reduces the likelihood of passing the wrong arguments, the wrong types of arguments or the wrong number of arguments.

Looking Ahead to Composition and Inheritance

Often, classes do not have to be created "from scratch." Rather, they can include objects of other classes as members or they may be **derived** from other classes that provide attributes and behaviors the new classes can use. Such software reuse can greatly enhance productivity and simplify code maintenance. *Including* class objects as members of other classes is called **composition** (or **aggregation**) and is discussed in Chapter 10. *Deriving* new classes from existing classes is called **inheritance** and is discussed in Chapter 12.

Object Size

People new to object-oriented programming often suppose that objects must be quite large because they contain data members and member functions. Logically, this is true—you may think of objects as containing data and functions (and our discussion has certainly encouraged this view); physically, however, this is not true.

Performance Tip 9.1

Objects contain only *data, so objects are much smaller than if they also contained member functions. Applying operator* `sizeof` *to a class name or to an object of that class will report only the size of the class's data members. The compiler creates one copy (only) of the member functions separate from all objects of the class. All objects of the class share this one copy. Each object, of course, needs its own copy of the class's data, because the data can vary among the objects. The function code is nonmodifiable and, hence, can be shared among all objects of one class.*

9.4 Class Scope and Accessing Class Members

A class's data members (variables declared in the class definition) and member functions (functions declared in the class definition) belong to that class's scope. Nonmember functions are defined at global namespace scope.

Within a class's scope, class members are immediately accessible by all of that class's member functions and can be referenced by name. Outside a class's scope, `public` class members are referenced through one of the **handles** on an object—an object name, a reference to an object or a pointer to an object. The *type* of the object, reference or pointer specifies the interface (i.e., the member functions) accessible to the client. [We'll see in Chapter 10 that an implicit handle is inserted by the compiler on every reference to a data member or member function from within an object.]

Member functions of a class can be overloaded, but *only* by other member functions of that class. To overload a member function, simply provide in the class definition a prototype for each version of the overloaded function, and provide a separate function definition for each version of the function.

Variables declared in a member function have local scope and are known only to that function. If a member function defines a variable with the same name as a variable with class scope, the class-scope variable is hidden by the block-scope variable in the local scope. Such a hidden variable can be accessed by preceding the variable name with the class name followed by the scope resolution operator (`::`). Hidden global variables can be accessed with the unary scope resolution operator (see Chapter 5).

The dot member selection operator (`.`) is preceded by an object's name or with a reference to an object to access the object's members. The **arrow member selection operator** (`->`) is preceded by a pointer to an object to access the object's members.

Figure 9.2 uses a simple class called `Count` (lines 7–24) with `private` data member `x` of type `int` (line 23), `public` member function `setX` (lines 11–14) and `public` member function `print` (lines 17–20) to illustrate accessing class members with the member-selection operators. For simplicity, we've included this small class in the same file as `main`. Lines 28–30 create three variables related to type `Count`—`counter` (a `Count` object), `counterPtr` (a pointer to a `Count` object) and `counterRef` (a reference to a `Count` object). Variable `counterRef` refers to `counter`, and variable `counterPtr` points to `counter`. In lines 33–34 and 37–38, note that the program can invoke member functions `setX` and `print` by using the dot (`.`) member selection operator preceded by either the name of the object (`counter`) or a reference to the object (`counterRef`, which is an alias for `counter`). Similarly, lines 41–42 demonstrate that the program can invoke member functions `setX` and `print` by using a pointer (`countPtr`) and the arrow (`->`) member-selection operator.

```
1  // Fig. 9.2: fig09_02.cpp
2  // Demonstrating the class member access operators . and ->
3  #include <iostream>
4  using namespace std;
5
6  // class Count definition
7  class Count
8  {
9  public: // public data is dangerous
10    // sets the value of private data member x
11    void setX( int value )
12    {
```

Fig. 9.2 | Accessing an object's member functions through each type of object handle—the object's name, a reference to the object and a pointer to the object. (Part 1 of 2.)

```
13          x = value;
14       } // end function setX
15
16       // prints the value of private data member x
17       void print()
18       {
19          cout << x << endl;
20       } // end function print
21
22    private:
23       int x;
24    }; // end class Count
25
26    int main()
27    {
28       Count counter; // create counter object
29       Count *counterPtr = &counter; // create pointer to counter
30       Count &counterRef = counter; // create reference to counter
31
32       cout << "Set x to 1 and print using the object's name: ";
33       counter.setX( 1 ); // set data member x to 1
34       counter.print(); // call member function print
35
36       cout << "Set x to 2 and print using a reference to an object: ";
37       counterRef.setX( 2 ); // set data member x to 2
38       counterRef.print(); // call member function print
39
40       cout << "Set x to 3 and print using a pointer to an object: ";
41       counterPtr->setX( 3 ); // set data member x to 3
42       counterPtr->print(); // call member function print
43    } // end main
```

```
Set x to 1 and print using the object's name: 1
Set x to 2 and print using a reference to an object: 2
Set x to 3 and print using a pointer to an object: 3
```

Fig. 9.2 | Accessing an object's member functions through each type of object handle—the object's name, a reference to the object and a pointer to the object. (Part 2 of 2.)

A Note About Inlining Member Functions

The member functions in Fig. 9.2 are defined completely in the body of class `Count`'s definition. In such cases, the compiler attempts to inline calls to the member function. Remember that the compiler reserves the right not to inline any function.

Performance Tip 9.2

Defining a member function inside the class definition inlines the member function (if the compiler chooses to do so). This can improve performance.

Software Engineering Observation 9.5

Defining a small member function inside the class definition does not promote the best software engineering, because clients of the class will be able to see the implementation of the function, and the client code must be recompiled if the function definition changes.

Software Engineering Observation 9.6

Only the simplest and most stable member functions—that is, the ones whose implementations are unlikely to change—should be defined in the class header.

9.5 Placing a Class in a Separate File for Reusability

One of the benefits of creating class definitions is that, when packaged properly, classes can be reused by programmers—potentially worldwide. For example, we can reuse C++ Standard Library type `string` in any C++ program by including the header file `<string>` (and, as we'll see, by being able to link to the library's object code).

Programmers who wish to use our `Time` class cannot simply include the file from Fig. 9.1 in another program. As you know, function `main` begins the execution of every program, and every program must have exactly one `main` function. If other programmers include the code from Fig. 9.1, they get extra baggage—our `main` function—and their programs will then have two `main` functions. Attempting to compile a program with two `main` functions in Microsoft Visual C++ produces an error such as

```
error C2084: function 'int main(void)' already has a body
```

when the compiler tries to compile the second `main` function it encounters. Similarly, the GNU C++ compiler produces the error

```
redefinition of 'int main()'
```

These errors indicate that a program already has a `main` function. So, placing `main` in the same file with a class definition prevents that class from being reused by other programs. In the next section, we demonstrate how to make class `Time` reusable by separating it into another file from the `main` function.

Header Files

The previous examples in the chapter consist of a single `.cpp` file, also known as a **source-code file**, that contains a class definition and a `main` function. When building an object-oriented C++ program, it's customary to define reusable source code (such as a class) in a file that by convention has a `.h` filename extension—known as a **header file**. Programs use `#include` preprocessor directives to include header files and take advantage of reusable software components, such as type `string` provided in the C++ Standard Library and user-defined types like class `Time`. In the next section, we'll separate class `Time` into its own header file.

9.6 `Time` Class: Separating Interface from Implementation

We now show how to promote software reusability by separating a class definition from the client code (e.g., function `main`) that uses the class. We also introduce another fundamental principle of good software engineering—**separating interface from implementation**.

Interface of a Class

Interfaces define and standardize the ways in which things such as people and systems interact with one another. For example, a radio's controls serve as an interface between the radio's users and its internal components. The controls allow users to perform a limited

set of operations (such as changing the station, adjusting the volume, and choosing between AM and FM stations). Various radios may implement these operations differently—some provide push buttons, some provide dials and some support voice commands. The interface specifies *what* operations a radio permits users to perform but does not specify *how* the operations are implemented inside the radio.

Similarly, the **interface of a class** describes *what* services a class's clients can use and how to *request* those services, but not *how* the class carries out the services. A class's `public` interface consists of the class's `public` member functions (also known as the class's **`public` services**). For example, class `Time`'s interface (Fig. 9.1) contains a constructor and member functions `setTime`, `printUniversal` and `printStandard`. `Time`'s clients, such as `main` in Fig. 9.1, use these functions to request the class's services.

Separating the Interface from the Implementation

In our prior examples, each class definition contained the complete definitions of the class's `public` member functions and the declarations of its `private` data members. However, it's better software engineering to define member functions *outside* the class definition, so that their implementation details can be hidden from the client code. This practice *ensures* that you do not write client code that depends on the class's implementation details. If you were to do so, the client code would be more likely to "break" if the class's implementation changed.

The program of Figs. 9.3–9.5 separates class `Time`'s interface from its implementation, and the class definition from the client code, by splitting the class definition of Fig. 9.1 into two files—the header file `Time.h` (Fig. 9.3) in which class `Time` is defined and the source-code file `Time.cpp` (Fig. 9.4) in which `Time`'s member functions are defined. By convention, member-function definitions are placed in a source-code file of the same base name (e.g., `Time`) as the class's header file but with a `.cpp` filename extension. The source-code file `fig09_05.cpp` (Fig. 9.5) defines function `main` (the client code). The code and output of Fig. 9.5 are identical to that of Fig. 9.1. Figure 9.6 shows how this three-file program is compiled from the perspectives of the `Time` class programmer and the client-code programmer—we'll explain this figure in detail.

`Time.h`: Defining a Class's Interface in a Header File

Header file `Time.h` (Fig. 9.3) contains the `Time`'s class definition (lines 10–21). Again, the compiler must know the data members of the class to determine how much memory to reserve for each object of the class. Including the header file `Time.h` in the client code (line 5 of Fig. 9.5) provides the compiler with the information it needs to ensure that the appropriate amount of memory is reserved for each `Time` object and that client code calls class `Time`'s member functions correctly.

```
1  // Fig. 9.3: Time.h
2  // Declaration of class Time.
3  // Member functions are defined in Time.cpp
4
5  // prevent multiple inclusions of header file
6  #ifndef TIME_H
7  #define TIME_H
```

Fig. 9.3 | `Time` class definition. (Part 1 of 2.)

```
 8
 9  // Time class definition
10  class Time
11  {
12  public:
13     Time(); // constructor
14     void setTime( int, int, int ); // set hour, minute and second
15     void printUniversal(); // print time in universal-time format
16     void printStandard(); // print time in standard-time format
17  private:
18     int hour; // 0 - 23 (24-hour clock format)
19     int minute; // 0 - 59
20     int second; // 0 - 59
21  }; // end class Time
22
23  #endif
```

Fig. 9.3 | `Time` class definition. (Part 2 of 2.)

Preprocessor Wrappers

In Fig. 9.3, the class definition is enclosed in the following **preprocessor wrapper** (lines 6, 7 and 23):

```
// prevent multiple inclusions of header file
#ifndef TIME_H
#define TIME_H
   ...
#endif
```

When we build larger programs, other definitions and declarations will also be placed in header files. The preceding preprocessor wrapper prevents the code between `#ifndef` (which means "if not defined") and `#endif` from being included if the name `TIME_H` has been defined. If the header has not been included previously in a file, the name `TIME_H` is defined by the `#define` directive and the header file statements are included. If the header has been included previously, `TIME_H` is defined already and the header file is not included again. Attempts to include a header file multiple times (inadvertently) typically occur in large programs with many header files that may themselves include other header files. The commonly used convention for the symbolic constant name in the preprocessor directives is simply the header file name in upper case with the underscore character replacing the period.

Error-Prevention Tip 9.3

Use `#ifndef`, `#define` and `#endif` preprocessor directives to form a preprocessor wrapper that prevents header files from being included more than once in a program.

`Time.cpp`: Defining Member Functions in a Separate Source-Code File

Source-code file `Time.cpp` (Fig. 9.4) defines class `Time`'s member functions, which were declared in lines 13–16 of Fig. 9.3. The definitions appear in lines 10–37 and are identical to the member-function definitions in lines 23–50 of Fig. 9.1.

```
1  // Fig. 9.4: Time.cpp
2  // Member-function definitions for class Time.
3  #include <iostream>
4  #include <iomanip>
5  #include "Time.h" // include definition of class Time from Time.h
6  using namespace std;
7
8  // Time constructor initializes each data member to zero.
9  // Ensures all Time objects start in a consistent state.
10 Time::Time()
11 {
12    hour = minute = second = 0;
13 } // end Time constructor
14
15 // set new Time value using universal time; ensure that
16 // the data remains consistent by setting invalid values to zero
17 void Time::setTime( int h, int m, int s )
18 {
19    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
20    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
21    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
22 } // end function setTime
23
24 // print Time in universal-time format (HH:MM:SS)
25 void Time::printUniversal()
26 {
27    cout << setfill( '0' ) << setw( 2 ) << hour << ":"
28       << setw( 2 ) << minute << ":" << setw( 2 ) << second;
29 } // end function printUniversal
30
31 // print Time in standard-time format (HH:MM:SS AM or PM)
32 void Time::printStandard()
33 {
34    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ) << ":"
35       << setfill( '0' ) << setw( 2 ) << minute << ":" << setw( 2 )
36       << second << ( hour < 12 ? " AM" : " PM" );
37 } // end function printStandard
```

Fig. 9.4 | `Time` class member-function definitions.

To indicate that the member functions in `Time.cpp` are part of class `Time`, we must first include the `Time.h` header file (line 5). This allows us to access the class name `Time` in the `Time.cpp` file. When compiling `Time.cpp`, the compiler uses the information in `Time.h` to ensure that

1. the first line of each member function (lines 10, 17, 25 and 32) matches its prototype in the `Time.h` file—for example, the compiler ensures that `printStandard` accepts no parameters and does not return a value, and that
2. each member function knows about the class's data members and other member functions—for example, lines 12, 19, 27 and 34 can access variable `hour` because it's declared in `Time.h` as a data member of class `Time`.

Testing Class `Time`

To help you prepare for the larger programs you'll encounter in this book and in industry, we often use a separate source-code file containing function `main` to test our classes (this is called a **driver program**). Figure 9.5 performs the same `Time` object manipulations as Fig. 9.1. Separating `Time`'s interface from its member-function implementation does not affect the way that this client code uses the class. It affects only how the program is compiled and linked, which we discuss in detail shortly.

As in Fig. 9.4, line 5 of Fig. 9.5 includes the `Time.h` header file so that the compiler can ensure that `Time` objects are created and manipulated correctly in the client code. Before executing this program, the source-code files in Figs. 9.4 and 9.5 must both be compiled, then linked together—that is, the member-function calls in the client code need to be tied to the implementations of the class's member functions—a job performed by the linker.

```
1  // Fig. 9.5: fig09_05.cpp
2  // Program to test class Time.
3  // NOTE: This file must be compiled with Time.cpp.
4  #include <iostream>
5  #include "Time.h" // include definition of class Time from Time.h
6  using namespace std;
7
8  int main()
9  {
10    Time t; // instantiate object t of class Time
11
12    // output Time object t's initial values
13    cout << "The initial universal time is ";
14    t.printUniversal(); // 00:00:00
15    cout << "\nThe initial standard time is ";
16    t.printStandard(); // 12:00:00 AM
17
18    t.setTime( 13, 27, 6 ); // change time
19
20    // output Time object t's new values
21    cout << "\n\nUniversal time after setTime is ";
22    t.printUniversal(); // 13:27:06
23    cout << "\nStandard time after setTime is ";
24    t.printStandard(); // 1:27:06 PM
25
26    t.setTime( 99, 99, 99 ); // attempt invalid settings
27
28    // output t's values after specifying invalid values
29    cout << "\n\nAfter attempting invalid settings:"
30       << "\nUniversal time: ";
31    t.printUniversal(); // 00:00:00
32    cout << "\nStandard time: ";
33    t.printStandard(); // 12:00:00 AM
34    cout << endl;
35 } // end main
```

Fig. 9.5 | Program to test class `Time`. (Part 1 of 2.)

```
The initial universal time is 00:00:00
The initial standard time is 12:00:00 AM

Universal time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

Fig. 9.5 | Program to test class `Time`. (Part 2 of 2.)

Software Engineering Observation 9.7

Clients of a class do not need access to the class's source code in order to use the class. The clients do, however, need to be able to link to the class's object code (i.e., the compiled version of the class). This encourages independent software vendors (ISVs) to provide class libraries for sale or license. The ISVs provide in their products only the header files and the object modules. No proprietary information is revealed—as would be the case if source code were provided. The C++ user community benefits by having more ISV-produced class libraries available.

Software Engineering Observation 9.8

Information important to the interface of a class should be included in the header file. Information that will be used only internally in the class and will not be needed by clients of the class should be included in the unpublished source file. This is yet another example of the principle of least privilege.

The Compilation and Linking Process

The diagram in Fig. 9.6 shows the compilation and linking process that results in an executable `Time` application. Often a class's interface and implementation will be created and compiled by one programmer and used by a separate programmer who implements the client code that uses the class. So, the diagram shows what's required by both the class-implementation programmer and the client-code programmer. The dashed lines in the diagram show the pieces required by the class-implementation programmer, the client-code programmer and the `Time` application user, respectively.

A class-implementation programmer responsible for creating a reusable `Time` class creates the header file `Time.h` and the source-code file `Time.cpp` that `#includes` the header file, then compiles the source-code file to create `Time`'s object code. To hide the class's member-function implementation details, the class-implementation programmer would provide the client-code programmer with the header file `Time.h` (which specifies the class's interface and data members) and the `Time` object code (i.e., the machine-language instructions that represent `Time`'s member functions). The client-code programmer is not given `Time.cpp`, so the client remains unaware of how `Time`'s member functions are implemented.

The client code needs to know only `Time`'s interface to use the class and must be able to link its object code. Since the interface of the class is part of the class definition in the `Time.h` header file, the client-code programmer must have access to this file and must

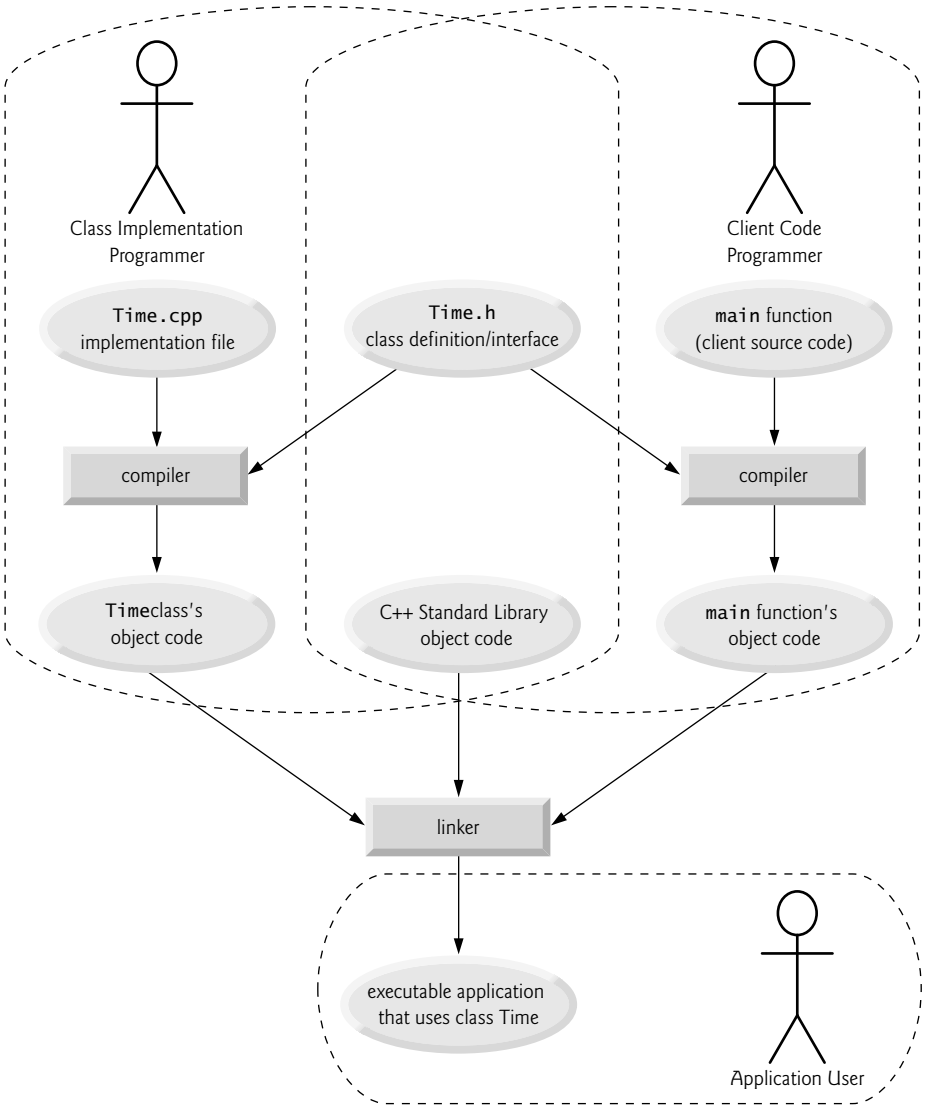

Fig. 9.6 | Compilation and linking process that produces an executable application.

`#include` it in the client's source-code file. When the client code is compiled, the compiler uses the class definition in `Time.h` to ensure that the `main` function creates and manipulates objects of class `Time` correctly.

To create the executable `Time` application, the last step is to link

1. the object code for the `main` function (i.e., the client code),
2. the object code for class `Time`'s member-function implementations and

3. the C++ Standard Library object code for the C++ library features (e.g., `cin` and `cout`) used by the class-implementation programmer and the client-code programmer.

The linker's output is the executable `Time` application. Compilers and IDEs typically invoke the linker for you after compiling your code.

For further information on compiling multiple-source-file programs, see your compiler's documentation. We provide links to various C++ compilers in our C++ Resource Center at `www.deitel.com/cplusplus/`.

Including a Header File That Contains a User-Defined Class

A header file such as `Time.h` (Fig. 9.3) cannot be used to begin program execution, because it does not contain a `main` function. If you try to compile and link `Time.h` by itself to create an executable application, Microsoft Visual C++ produces the linker error message:

```
error LNK2001: unresolved external symbol _mainCRTStartup
```

To compile and link with GNU C++ on Linux, you must first include the header file in a `.cpp` source-code file, then GNU C++ produces a linker error message containing:

```
undefined \reference to 'main'
```

This error indicates that the linker could not locate the program's `main` function. To test class `Time`, you must write a separate source-code file containing a `main` function (such as Fig. 9.5) that instantiates and uses objects of the class.

The compiler does not know what a `Time` is because it's a user-defined type. In fact, the compiler doesn't even know the classes in the C++ Standard Library. To help it understand how to use a class, we must explicitly provide the compiler with the class's definition. This enables the compiler to determine the amount of memory that it must reserve for each object of the class and ensure that a program calls the class's member functions correctly.

Line 5 in Fig. 9.5 instructs the C++ preprocessor to replace the directive with a copy of the contents of `Time.h` (i.e., the `Time` class definition) *before* the program is compiled. When the source-code file `fig09_05.cpp` is compiled, it now contains the `Time` class definition (because of the `#include`), and the compiler is able to determine how to create `Time` objects and see that their member functions are called correctly. Now that the class definition is in a header file (without a `main` function), we can include that header in *any* program that needs to reuse our `Time` class.

How Header Files Are Located

Notice that the name of the `Time.h` header file in line 5 of Fig. 9.5 is enclosed in quotes (" ") rather than angle brackets (< >). Normally, a program's source-code files and user-defined header files are placed in the same directory. When the preprocessor encounters a header file name in quotes, it attempts to locate the header file in the *same* directory as the file in which the `#include` directive appears. If the preprocessor cannot find the header file in that directory, it searches for it in the same location(s) as the C++ Standard Library header files. When the preprocessor encounters a header file name in angle brackets (e.g., `<iostream>`), it assumes that the header is part of the C++ Standard Library and does not look in the directory of the program that's being preprocessed.

Error-Prevention Tip 9.4

To ensure that the preprocessor can locate header files correctly, `#include` preprocessor directives should place the names of user-defined header files in quotes (e.g., `"Time.h"`) and place the names of C++ Standard Library header files in angle brackets (e.g., `<iostream>`).

9.7 Access Functions and Utility Functions

Access functions can read or display data. Another common use for access functions is to test the truth or falsity of conditions—such functions are often called **predicate functions.** An example of a predicate function would be an `isEmpty` function for any container class—a class capable of holding many objects, like a `vector`. A program might test `isEmpty` before attempting to read another item from the container object. An `isFull` predicate function might test a container-class object to determine whether it has no additional room. Useful predicate functions for our `Time` class might be `isAM` and `isPM`.

The program of Figs. 9.7–9.9 demonstrates the notion of a utility function (also called a **helper function**). A utility function is not part of a class's `public` interface; rather, it's a `private` member function that supports the operation of the class's `public` member functions. Utility functions are not intended to be used by clients of a class, but can be used by friends of a class, as we'll see in Chapter 10.

Class `SalesPerson` (Fig. 9.7) declares an array of 12 monthly sales figures (line 17) and the prototypes for the class's constructor and member functions that manipulate the array.

```
1  // Fig. 9.7: SalesPerson.h
2  // SalesPerson class definition.
3  // Member functions defined in SalesPerson.cpp.
4  #ifndef SALESP_H
5  #define SALESP_H
6
7  class SalesPerson
8  {
9  public:
10    static const int monthsPerYear = 12; // months in one year
11    SalesPerson(); // constructor
12    void getSalesFromUser(); // input sales from keyboard
13    void setSales( int, double ); // set sales for a specific month
14    void printAnnualSales(); // summarize and print sales
15 private:
16    double totalAnnualSales(); // prototype for utility function
17    double sales[ monthsPerYear ]; // 12 monthly sales figures
18 }; // end class SalesPerson
19
20 #endif
```

Fig. 9.7 | `SalesPerson` class definition.

In Fig. 9.8, the `SalesPerson` constructor (lines 9–13) initializes array `sales` to zero. The `public` member function `setSales` (lines 30–37) sets the sales figure for one month in array `sales`. The `public` member function `printAnnualSales` (lines 40–45) prints the total sales for the last 12 months. The `private` utility function `totalAnnualSales` (lines

48–56) totals the 12 monthly sales figures for the benefit of `printAnnualSales`. Member function `printAnnualSales` edits the sales figures into monetary format.

```
1  // Fig. 9.8: SalesPerson.cpp
2  // SalesPerson class member-function definitions.
3  #include <iostream>
4  #include <iomanip>
5  #include "SalesPerson.h" // include SalesPerson class definition
6  using namespace std;
7
8  // initialize elements of array sales to 0.0
9  SalesPerson::SalesPerson()
10 {
11    for ( int i = 0; i < monthsPerYear; i++ )
12       sales[ i ] = 0.0;
13 } // end SalesPerson constructor
14
15 // get 12 sales figures from the user at the keyboard
16 void SalesPerson::getSalesFromUser()
17 {
18    double salesFigure;
19
20    for ( int i = 1; i <= monthsPerYear; i++ )
21    {
22       cout << "Enter sales amount for month " << i << ": ";
23       cin >> salesFigure;
24       setSales( i, salesFigure );
25    } // end for
26 } // end function getSalesFromUser
27
28 // set one of the 12 monthly sales figures; function subtracts
29 // one from month value for proper subscript in sales array
30 void SalesPerson::setSales( int month, double amount )
31 {
32    // test for valid month and amount values
33    if ( month >= 1 && month <= monthsPerYear && amount > 0 )
34       sales[ month - 1 ] = amount; // adjust for subscripts 0-11
35    else // invalid month or amount value
36       cout << "Invalid month or sales figure" << endl;
37 } // end function setSales
38
39 // print total annual sales (with the help of utility function)
40 void SalesPerson::printAnnualSales()
41 {
42    cout << setprecision( 2 ) << fixed
43       << "\nThe total annual sales are: $"
44       << totalAnnualSales() << endl; // call utility function
45 } // end function printAnnualSales
46
47 // private utility function to total annual sales
48 double SalesPerson::totalAnnualSales()
49 {
```

Fig. 9.8 | `SalesPerson` class member-function definitions. (Part 1 of 2.)

```
50    double total = 0.0; // initialize total
51
52    for ( int i = 0; i < monthsPerYear; i++ ) // summarize sales results
53       total += sales[ i ]; // add month i sales to total
54
55    return total;
56 } // end function totalAnnualSales
```

Fig. 9.8 | `SalesPerson` class member-function definitions. (Part 2 of 2.)

In Fig. 9.9, notice that the application's `main` function includes only a simple sequence of member-function calls—there are no control statements. The logic of manipulating the `sales` array is completely encapsulated in class `SalesPerson`'s member functions.

Software Engineering Observation 9.9

A phenomenon of object-oriented programming is that once a class is defined, creating and manipulating objects of that class often involve issuing only a simple sequence of member-function calls—few, if any, control statements are needed. By contrast, it's common to have control statements in the implementation of a class's member functions.

```
1  // Fig. 9.9: fig09_09.cpp
2  // Utility function demonstration.
3  // Compile this program with SalesPerson.cpp
4
5  // include SalesPerson class definition from SalesPerson.h
6  #include "SalesPerson.h"
7
8  int main()
9  {
10    SalesPerson s; // create SalesPerson object s
11
12    s.getSalesFromUser(); // note simple sequential code; there are
13    s.printAnnualSales(); // no control statements in main
14 } // end main
```

```
Enter sales amount for month 1: 5314.76
Enter sales amount for month 2: 4292.38
Enter sales amount for month 3: 4589.83
Enter sales amount for month 4: 5534.03
Enter sales amount for month 5: 4376.34
Enter sales amount for month 6: 5698.45
Enter sales amount for month 7: 4439.22
Enter sales amount for month 8: 5893.57
Enter sales amount for month 9: 4909.67
Enter sales amount for month 10: 5123.45
Enter sales amount for month 11: 4024.97
Enter sales amount for month 12: 5923.92

The total annual sales are: $60120.59
```

Fig. 9.9 | Utility function demonstration.

9.8 Time Class: Constructors with Default Arguments

The program of Figs. 9.10–9.12 enhances class `Time` to demonstrate how arguments are implicitly passed to a constructor. The constructor defined in Fig. 9.1 initialized `hour`, `minute` and `second` to 0 (i.e., midnight in universal time). Like other functions, constructors can specify default arguments. Line 13 of Fig. 9.10 declares the `Time` constructor to include default arguments, specifying a default value of zero for each argument passed to the constructor. In Fig. 9.11, lines 10–13 define the new version of the `Time` constructor that receives values for parameters `hr`, `min` and `sec` that will be used to initialize `private` data members `hour`, `minute` and `second`, respectively. The `Time` constructor now calls `setTime` to validate and assign values to the data members. The default arguments to the constructor ensure that, even if no values are provided in a constructor call, the constructor still initializes the data members to maintain the `Time` object in a consistent state. A constructor that defaults all its arguments is also a default constructor—i.e., a constructor that can be invoked with no arguments. There can be at most one default constructor per class.

```
1  // Fig. 9.10: Time.h
2  // Time class containing a constructor with default arguments.
3  // Member functions defined in Time.cpp.
4
5  // prevent multiple inclusions of header file
6  #ifndef TIME_H
7  #define TIME_H
8
9  // Time abstract data type definition
10 class Time
11 {
12 public:
13    Time( int = 0, int = 0, int = 0 ); // default constructor
14    void setTime( int, int, int ); // set hour, minute, second
15    void printUniversal(); // output time in universal-time format
16    void printStandard(); // output time in standard-time format
17 private:
18    int hour; // 0 - 23 (24-hour clock format)
19    int minute; // 0 - 59
20    int second; // 0 - 59
21 }; // end class Time
22
23 #endif
```

Fig. 9.10 | `Time` class containing a constructor with default arguments.

In Fig. 9.11, line 12 of the constructor calls member function `setTime` with the values passed to the constructor (or the default values). Function `setTime` calls `setHour` to ensure that the value supplied for `hour` is in the range 0–23, then calls `setMinute` and `setSecond` to ensure that the values for `minute` and `second` are each in the range 0–59. If a value is out of range, that value is set to zero (to ensure that each data member remains in a consistent state). Again, in Chapter 16, we use exceptions to indicate when a value is out of range, rather than simply assigning a default consistent value.

```
1  // Fig. 9.11: Time.cpp
2  // Member-function definitions for class Time.
3  #include <iostream>
4  #include <iomanip>
5  #include "Time.h" // include definition of class Time from Time.h
6  using namespace std;
7
8  // Time constructor initializes each data member to zero;
9  // ensures that Time objects start in a consistent state
10 Time::Time( int hr, int min, int sec )
11 {
12    setTime( hr, min, sec ); // validate and set time
13 } // end Time constructor
14
15 // set new Time value using universal time; ensure that
16 // the data remains consistent by setting invalid values to zero
17 void Time::setTime( int h, int m, int s )
18 {
19    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
20    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
21    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
22 } // end function setTime
23
24 // print Time in universal-time format (HH:MM:SS)
25 void Time::printUniversal()
26 {
27    cout << setfill( '0' ) << setw( 2 ) << hour << ":"
28       << setw( 2 ) << minute << ":" << setw( 2 ) << second;
29 } // end function printUniversal
30
31 // print Time in standard-time format (HH:MM:SS AM or PM)
32 void Time::printStandard()
33 {
34    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ) << ":"
35       << setfill( '0' ) << setw( 2 ) << minute << ":" << setw( 2 )
36       << second << ( hour < 12 ? " AM" : " PM" );
37 } // end function printStandard
```

Fig. 9.11 | `Time` class member-function definitions including a constructor that takes arguments.

Function `main` in Fig. 9.12 initializes five `Time` objects—one with all three arguments defaulted in the implicit constructor call (line 9), one with one argument specified (line 10), one with two arguments specified (line 11), one with three arguments specified (line 12) and one with three invalid arguments specified (line 13). Then the program displays each object in universal-time and standard-time formats.

Software Engineering Observation 9.10

Any change to the default argument values of a function requires the client code to be recompiled (to ensure that the program still functions correctly).

```
1  // Fig. 9.12: fig09_12.cpp
2  // Demonstrating a default constructor for class Time.
3  #include <iostream>
4  #include "Time.h" // include definition of class Time from Time.h
5  using namespace std;
6
7  int main()
8  {
9     Time t1; // all arguments defaulted
10    Time t2( 2 ); // hour specified; minute and second defaulted
11    Time t3( 21, 34 ); // hour and minute specified; second defaulted
12    Time t4( 12, 25, 42 ); // hour, minute and second specified
13    Time t5( 27, 74, 99 ); // all bad values specified
14
15    cout << "Constructed with:\n\nt1: all arguments defaulted\n  ";
16    t1.printUniversal(); // 00:00:00
17    cout << "\n  ";
18    t1.printStandard(); // 12:00:00 AM
19
20    cout << "\n\nt2: hour specified; minute and second defaulted\n  ";
21    t2.printUniversal(); // 02:00:00
22    cout << "\n  ";
23    t2.printStandard(); // 2:00:00 AM
24
25    cout << "\n\nt3: hour and minute specified; second defaulted\n  ";
26    t3.printUniversal(); // 21:34:00
27    cout << "\n  ";
28    t3.printStandard(); // 9:34:00 PM
29
30    cout << "\n\nt4: hour, minute and second specified\n  ";
31    t4.printUniversal(); // 12:25:42
32    cout << "\n  ";
33    t4.printStandard(); // 12:25:42 PM
34
35    cout << "\n\nt5: all invalid values specified\n  ";
36    t5.printUniversal(); // 00:00:00
37    cout << "\n  ";
38    t5.printStandard(); // 12:00:00 AM
39    cout << endl;
40 } // end main
```

```
Constructed with:

t1: all arguments defaulted
  00:00:00
  12:00:00 AM

t2: hour specified; minute and second defaulted
  02:00:00
  2:00:00 AM
```

Fig. 9.12 | Constructor with default arguments. (Part 1 of 2.)

```
t3: hour and minute specified; second defaulted
  21:34:00
  9:34:00 PM

t4: hour, minute and second specified
  12:25:42
  12:25:42 PM

t5: all invalid values specified
  00:00:00
  12:00:00 AM
```

Fig. 9.12 | Constructor with default arguments. (Part 2 of 2.)

Two Ways to Provide a Default Constructor for a Class

Any constructor that takes no arguments is called a default constructor. A class gets a default constructor in one of two ways:

1. The compiler implicitly creates a default constructor in a class that does not define a constructor. Such a constructor does not initialize the class's data members, but does call the default constructor for each data member that's an object of another class. An uninitialized variable typically contains a "garbage" value.
2. You explicitly define a constructor that takes no arguments. Such a default constructor will call the default constructor for each data member that's an object of another class and will perform additional initialization specified by you.

If you define a constructor *with* arguments, C++ will not implicitly create a default constructor for that class.

Error-Prevention Tip 9.5

Unless no initialization of your class's data members is necessary (almost never), provide a constructor to ensure that your class's data members are initialized with meaningful values when each new object of your class is created.

Software Engineering Observation 9.11

Data members can be initialized in a constructor, or their values may be set *later after the object is created. However, it's a good software engineering practice to ensure that an object is fully initialized before the client code invokes the object's member functions. You should not rely on the client code to ensure that an object gets initialized properly.*

9.9 Destructors

A **destructor** is another type of special member function. The name of the destructor for a class is the **tilde character** (~) followed by the class name.

A class's destructor is called implicitly when an object is destroyed. This occurs, for example, as an automatic object is destroyed when program execution leaves the scope in which that object was instantiated. *The destructor itself does not actually release the object's memory*—it performs **termination housekeeping** before the object's memory is reclaimed, so the memory may be reused to hold new objects.

A destructor receives no parameters and returns no value. A destructor may not specify a return type—not even `void`. A class may have only one destructor—destructor overloading is not allowed. A destructor must be `public`.

Common Programming Error 9.4

It's a syntax error to attempt to pass arguments to a destructor, to specify a return type for a destructor (even `void` cannot be specified), to return values from a destructor or to overload a destructor.

Even though destructors have not been provided for the classes presented so far, *every class has a destructor*. If you do not explicitly provide a destructor, the compiler creates an "empty" destructor. [*Note:* We'll see that such an implicitly created destructor does, in fact, perform important operations on objects that are created through composition (Chapter 10) and inheritance (Chapter 12).] In Chapter 11, we'll build destructors appropriate for classes whose objects contain dynamically allocated memory (e.g., for arrays and strings) or use other system resources (e.g., files on disk, which we study in Chapter 17). We discuss how to dynamically allocate and deallocate memory in Chapter 10.

9.10 When Constructors and Destructors Are Called

Constructors and destructors are called implicitly by the compiler. The order in which these function calls occur depends on the order in which execution enters and leaves the scopes where the objects are instantiated. Generally, destructor calls are made in the reverse order of the corresponding constructor calls, but as we'll see in Figs. 9.13–9.15, the storage classes of objects can alter the order in which destructors are called.

Constructors are called for objects defined in global scope before any other function (including `main`) in that file begins execution (although the order of execution of global object constructors between files is not guaranteed). The corresponding destructors are called when `main` terminates. Function **`exit`** forces a program to terminate immediately and *does not execute the destructors of automatic objects*. The function often is used to terminate a program when an error is detected in the input or if a file to be processed by the program cannot be opened. Function **`abort`** performs similarly to function `exit` but forces the program to terminate immediately, without allowing the destructors of any objects to be called. Function `abort` is usually used to indicate an abnormal termination of the program. (See Appendix F, for more information on functions `exit` and `abort`.)

The constructor for an automatic local object is called when execution reaches the point where that object is defined—the corresponding destructor is called when execution leaves the object's scope (i.e., the block in which that object is defined has finished executing). Constructors and destructors for automatic objects are called each time execution enters and leaves the scope of the object. Destructors are not called for automatic objects if the program terminates with a call to function `exit` or function `abort`.

The constructor for a `static` local object is called only once, when execution first reaches the point where the object is defined—the corresponding destructor is called when `main` terminates or the program calls function `exit`. Global and `static` objects are destroyed in the reverse order of their creation. Destructors are not called for `static` objects if the program terminates with a call to function `abort`.

The program of Figs. 9.13–9.15 demonstrates the order in which constructors and destructors are called for objects of class `CreateAndDestroy` (Fig. 9.13 and Fig. 9.14) of various storage classes in several scopes. Each object of class `CreateAndDestroy` contains an integer (`objectID`) and a `string` (`message`) that are used in the program's output to identify the object (Fig. 9.13 lines 16–17). This mechanical example is purely for pedagogic purposes. For this reason, line 21 of the destructor in Fig. 9.14 determines whether the object being destroyed has an `objectID` value `1` or `6` and, if so, outputs a newline character. This line makes the program's output easier to follow.

```
1  // Fig. 9.13: CreateAndDestroy.h
2  // CreateAndDestroy class definition.
3  // Member functions defined in CreateAndDestroy.cpp.
4  #include <string>
5  using namespace std;
6
7  #ifndef CREATE_H
8  #define CREATE_H
9
10 class CreateAndDestroy
11 {
12 public:
13    CreateAndDestroy( int, string ); // constructor
14    ~CreateAndDestroy(); // destructor
15 private:
16    int objectID; // ID number for object
17    string message; // message describing object
18 }; // end class CreateAndDestroy
19
20 #endif
```

Fig. 9.13 | `CreateAndDestroy` class definition.

```
1  // Fig. 9.14: CreateAndDestroy.cpp
2  // CreateAndDestroy class member-function definitions.
3  #include <iostream>
4  #include "CreateAndDestroy.h"// include CreateAndDestroy class definition
5  using namespace std;
6
7  // constructor
8  CreateAndDestroy::CreateAndDestroy( int ID, string messageString )
9  {
10    objectID = ID; // set object's ID number
11    message = messageString; // set object's descriptive message
12
13    cout << "Object " << objectID << "   constructor runs   "
14       << message << endl;
15 } // end CreateAndDestroy constructor
```

Fig. 9.14 | `CreateAndDestroy` class member-function definitions. (Part 1 of 2.)

```
16
17 // destructor
18 CreateAndDestroy::~CreateAndDestroy()
19 {
20    // output newline for certain objects; helps readability
21    cout << ( objectID == 1 || objectID == 6 ? "\n" : "" );
22
23    cout << "Object " << objectID << "   destructor runs    "
24       << message << endl;
25 } // end ~CreateAndDestroy destructor
```

Fig. 9.14 | `CreateAndDestroy` class member-function definitions. (Part 2 of 2.)

Figure 9.15 defines object `first` (line 10) in global scope. Its constructor is actually called before any statements in `main` execute and its destructor is called at program termination after the destructors for all other objects have run.

```
1  // Fig. 9.15: fig09_15.cpp
2  // Demonstrating the order in which constructors and
3  // destructors are called.
4  #include <iostream>
5  #include "CreateAndDestroy.h" // include CreateAndDestroy class definition
6  using namespace std;
7
8  void create( void ); // prototype
9
10 CreateAndDestroy first( 1, "(global before main)" ); // global object
11
12 int main()
13 {
14    cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;
15    CreateAndDestroy second( 2, "(local automatic in main)" );
16    static CreateAndDestroy third( 3, "(local static in main)" );
17
18    create(); // call function to create objects
19
20    cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;
21    CreateAndDestroy fourth( 4, "(local automatic in main)" );
22    cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;
23 } // end main
24
25 // function to create objects
26 void create( void )
27 {
28    cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;
29    CreateAndDestroy fifth( 5, "(local automatic in create)" );
30    static CreateAndDestroy sixth( 6, "(local static in create)" );
31    CreateAndDestroy seventh( 7, "(local automatic in create)" );
32    cout << "\nCREATE FUNCTION: EXECUTION ENDS" << endl;
33 } // end function create
```

Fig. 9.15 | Order in which constructors and destructors are called. (Part 1 of 2.)

```
Object 1   constructor runs   (global before main)

MAIN FUNCTION: EXECUTION BEGINS
Object 2   constructor runs   (local automatic in main)
Object 3   constructor runs   (local static in main)

CREATE FUNCTION: EXECUTION BEGINS
Object 5   constructor runs   (local automatic in create)
Object 6   constructor runs   (local static in create)
Object 7   constructor runs   (local automatic in create)

CREATE FUNCTION: EXECUTION ENDS
Object 7   destructor runs    (local automatic in create)
Object 5   destructor runs    (local automatic in create)

MAIN FUNCTION: EXECUTION RESUMES
Object 4   constructor runs   (local automatic in main)

MAIN FUNCTION: EXECUTION ENDS
Object 4   destructor runs    (local automatic in main)
Object 2   destructor runs    (local automatic in main)

Object 6   destructor runs    (local static in create)
Object 3   destructor runs    (local static in main)

Object 1   destructor runs    (global before main)
```

Fig. 9.15 | Order in which constructors and destructors are called. (Part 2 of 2.)

Function `main` (lines 12–23) declares three objects. Objects `second` (line 15) and `fourth` (line 21) are local automatic objects, and object `third` (line 16) is a `static` local object. The constructor for each of these objects is called when execution reaches the point where that object is declared. The destructors for objects `fourth` then `second` are called (i.e., the reverse of the order in which their constructors were called) when execution reaches the end of `main`. Because object `third` is `static`, it exists until program termination. The destructor for object `third` is called before the destructor for global object `first`, but after all other objects are destroyed.

Function `create` (lines 26–33) declares three objects—`fifth` (line 29) and `seventh` (line 31) as local automatic objects, and `sixth` (line 30) as a `static` local object. The destructors for objects `seventh` then `fifth` are called (i.e., the reverse of the order in which their constructors were called) when `create` terminates. Because `sixth` is `static`, it exists until program termination. The destructor for `sixth` is called before the destructors for `third` and `first`, but after all other objects are destroyed.

9.11 `Time` Class: Using *Set* and *Get* Functions

The program of Figs. 9.16–9.18 enhances class `Time` (Figs. 9.16–9.17) to include public functions that allow the client code to *set* and *get* the values of the `private` data members `hour`, `minute` and `second`. The new member functions are declared in Fig. 9.16 at lines 17–19 and 22–24. Their definitions appear in Fig. 9.17. The *set* functions (defined at lines

25–28, 31–34 and 37–40) strictly control the setting of the data members. Attempts to *set* any data member to an incorrect value cause the data member to be set to zero (thus leaving the data member in a consistent state). Each *get* function (defined at lines 43–46, 49–52 and 55–58) simply returns the appropriate data member's value.

```
1  // Fig. 9.16: Time.h
2  // Time class containing a constructor with default arguments.
3  // Member functions defined in Time.cpp.
4
5  // prevent multiple inclusions of header file
6  #ifndef TIME_H
7  #define TIME_H
8
9  // Time abstract data type definition
10 class Time
11 {
12 public:
13    Time( int = 0, int = 0, int = 0 ); // default constructor
14
15    // set functions
16    void setTime( int, int, int ); // set hour, minute, second
17    void setHour( int ); // set hour (after validation)
18    void setMinute( int ); // set minute (after validation)
19    void setSecond( int ); // set second (after validation)
20
21    // get functions
22    int getHour(); // return hour
23    int getMinute(); // return minute
24    int getSecond(); // return second
25
26    void printUniversal(); // output time in universal-time format
27    void printStandard(); // output time in standard-time format
28 private:
29    int hour; // 0 - 23 (24-hour clock format)
30    int minute; // 0 - 59
31    int second; // 0 - 59
32 }; // end class Time
33
34 #endif
```

Fig. 9.16 | `Time` class containing a constructor with default arguments.

```
1  // Fig. 9.17: Time.cpp
2  // Member-function definitions for class Time.
3  #include <iostream>
4  #include <iomanip>
5  #include "Time.h" // include definition of class Time from Time.h
6  using namespace std;
7
```

Fig. 9.17 | `Time` class member-function definitions including a constructor that takes arguments. (Part 1 of 3.)

```
// Time constructor initializes each data member to zero;
// ensures that Time objects start in a consistent state
Time::Time( int hr, int min, int sec )
{
   setTime( hr, min, sec ); // validate and set time
} // end Time constructor

// set new Time value using universal time; ensure that
// the data remains consistent by setting invalid values to zero
void Time::setTime( int h, int m, int s )
{
   setHour( h ); // set private field hour
   setMinute( m ); // set private field minute
   setSecond( s ); // set private field second
} // end function setTime

// set hour value
void Time::setHour( int h )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
} // end function setHour

// set minute value
void Time::setMinute( int m )
{
   minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
} // end function setMinute

// set second value
void Time::setSecond( int s )
{
   second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
} // end function setSecond

// return hour value
int Time::getHour()
{
   return hour;
} // end function getHour

// return minute value
int Time::getMinute()
{
   return minute;
} // end function getMinute

// return second value
int Time::getSecond()
{
   return second;
} // end function getSecond

```

Fig. 9.17 | `Time` class member-function definitions including a constructor that takes arguments. (Part 2 of 3.)

```
60  // print Time in universal-time format (HH:MM:SS)
61  void Time::printUniversal()
62  {
63     cout << setfill( '0' ) << setw( 2 ) << getHour() << ":"
64        << setw( 2 ) << getMinute() << ":" << setw( 2 ) << getSecond();
65  } // end function printUniversal
66
67  // print Time in standard-time format (HH:MM:SS AM or PM)
68  void Time::printStandard()
69  {
70     cout << ( ( getHour() == 0 || getHour() == 12 ) ? 12 : getHour() % 12 )
71        << ":" << setfill( '0' ) << setw( 2 ) << getMinute()
72        << ":" << setw( 2 ) << getSecond() << ( hour < 12 ? " AM" : " PM" );
73  } // end function printStandard
```

Fig. 9.17 | `Time` class member-function definitions including a constructor that takes arguments. (Part 3 of 3.)

```
1  // Fig. 9.18: fig09_18.cpp
2  // Demonstrating the Time class set and get functions
3  #include <iostream>
4  #include "time.h"
5  using namespace std;
6
7  void incrementMinutes( Time &, const int ); // prototype
8
9  int main()
10 {
11    Time t; // create Time object
12
13    // set time using individual set functions
14    t.setHour( 17 );  // set hour to valid value
15    t.setMinute( 34 ); // set minute to valid value
16    t.setSecond( 25 ); // set second to valid value
17
18    // use get functions to obtain hour, minute and second
19    cout << "Result of setting all valid values:\n"
20         << "  Hour: " << t.getHour()
21         << "  Minute: " << t.getMinute()
22         << "  Second: " << t.getSecond();
23
24    // set time using individual set functions
25    t.setHour( 234 ); // invalid hour set to 0
26    t.setMinute( 43 ); // set minute to valid value
27    t.setSecond( 6373 ); // invalid second set to 0
28
29    // display hour, minute and second after setting
30    // invalid hour and second values
31    cout << "\n\nResult of attempting to set invalid hour and"
32         << " second:\n  Hour: " << t.getHour()
33         << "  Minute: " << t.getMinute()
34         << "  Second: " << t.getSecond() << "\n\n";
```

Fig. 9.18 | *Set* and *get* functions manipulating an object's `private` data. (Part 1 of 2.)

```
35
36     t.setTime( 11, 58, 0 ); // set time
37     incrementMinutes( t, 3 ); // increment t's minute by 3
38  } // end main
39
40  // add specified number of minutes to a Time object
41  void incrementMinutes( Time &tt, const int count )
42  {
43     cout << "Incrementing minute " << count
44          << " times:\nStart time: ";
45     tt.printStandard();
46
47     for ( int i = 0; i < count; i++ ) {
48        tt.setMinute( ( tt.getMinute() + 1 ) % 60 );
49
50        if ( tt.getMinute() == 0 )
51           tt.setHour( ( tt.getHour() + 1 ) % 24);
52
53        cout << "\nminute + 1: ";
54        tt.printStandard();
55     } // end for
56
57     cout << endl;
58  } // end function incrementMinutes
```

```
Result of setting all valid values:
  Hour: 17  Minute: 34  Second: 25

Result of attempting to set invalid hour and second:
  Hour: 0  Minute: 43  Second: 0

Incrementing minute 3 times:
Start time: 11:58:00 AM
minute + 1: 11:59:00 AM
minute + 1: 12:00:00 PM
minute + 1: 12:01:00 PM
```

Fig. 9.18 | *Set* and *get* functions manipulating an object's `private` data. (Part 2 of 2.)

Software Engineering with Set *and* Get *Functions*

A class's `private` data members can be manipulated only by member functions of that class (and by "friends" of the class, as we'll see in Chapter 10). So a client of an object calls the class's `public` member functions to request the class's services for particular objects of the class. This is why the statements in function `main` call member functions like `setHour` and `getTime` on a `Time` object. Classes often provide `public` member functions to allow clients of the class to ***set*** (i.e., assign values to) or ***get*** (i.e., obtain the values of) `private` data members. These member function names need not begin with `set` or `get`, but this naming convention is common. In this example, the new *set* member functions that *set* the `hour`, `minute` and `second` data members are called `setHour`, `setMinute` and `setSecond`, respectively. Similarly, the member functions that *get* the values of the `hour`, `minute` and `second` data members are called `getHour`, `getMinute` and `getSecond`, respectively. *Set* functions are also sometimes called **mutators** (because they mutate, or change, values), and *get* functions are also sometimes called **accessors** (because they access values).

Recall that declaring data members with access specifier `private` enforces data hiding. Providing `public` *set* and *get* functions allows clients of a class to access the hidden data, but only *indirectly*. The client knows that it's attempting to modify or obtain an object's data, but the client does not know how the object performs these operations. In some cases, a class may internally represent a piece of data one way, but expose that data to clients in a different way. For example, suppose a `Clock` class represents the time of day as a `private int` data member `time` that stores the number of seconds since midnight. However, when a client calls a `Clock` object's `getTime` member function, the object could return the time with hours, minutes and seconds in a `string` in the format `"HH:MM:SS"`. Similarly, suppose the `Clock` class provides a *set* function named `setTime` that takes a `string` parameter in the `"HH:MM:SS"` format. Using `string` capabilities presented in Chapter 18, the `setTime` function could convert this `string` to a number of seconds, which the function stores in its `private` data member. The *set* function could also check that the value it receives represents a valid time (e.g., `"12:30:45"` is valid but `"42:85:70"` is not). The *set* and *get* functions allow a client to interact with an object, but the object's `private` data remains safely encapsulated (i.e., hidden) in the object itself.

`Time`'s *set* and *get* functions are called throughout the class's body. In particular, function `setTime` (lines 17–22 of Fig. 9.11) calls functions `setHour`, `setMinute` and `setSecond`, and functions `printUniversal` and `printStandard` call functions `getHour`, `getMinute` and `getSecond` in line 63–64 and lines 70–72, respectively. In each case, these functions could have accessed the class's `private` data directly. However, consider changing the representation of the time from three `int` values (requiring 12 bytes of memory) to a single `int` value representing the total number of seconds that have elapsed since midnight (requiring only four bytes of memory). If we made such a change, only the bodies of the functions that access the `private` data directly would need to change—in particular, the individual *set* and *get* functions for the `hour`, `minute` and `second`. There would be no need to modify the bodies of functions `setTime`, `printUniversal` or `printStandard`, because they do not access the data directly. Designing the class in this manner reduces the likelihood of programming errors when altering the class's implementation.

Similarly, the `Time` constructor could be written to include a copy of the appropriate statements from function `setTime`. Doing so may be slightly more efficient, because the extra constructor call and call to `setTime` are eliminated. However, duplicating statements in multiple functions or constructors makes changing the class's internal data representation more difficult. Having the `Time` constructor call function `setTime` directly requires any changes to the implementation of `setTime` to be made only once.

Software Engineering Observation 9.12

If a member function of a class already provides all or part of the functionality required by a constructor (or other member function) of the class, call that member function from the constructor (or other member function). This simplifies the maintenance of the code and reduces the likelihood of an error if the implementation of the code is modified. As a general rule: Avoid repeating code.

Software Engineering Observation 9.13

Write programs that are understandable and easy to maintain. Change is the rule rather than the exception. You should anticipate that your code will be modified.

Good Programming Practice 9.2

Always try to localize the effects of changes to a class's data members by accessing and manipulating the data members through their get *and* set *functions. Changes to the name of a data member or the data type used to store a data member then affect only the corresponding* get *and* set *functions, but not the callers of those functions.*

Software Engineering Observation 9.14

Provide set *or* get *functions for each* private *data item only when appropriate. Services useful to the client should typically be provided in the class's* public *interface.*

Common Programming Error 9.5

A constructor can call other member functions of the class, such as set *or* get *functions, but because the constructor is initializing the object, the data members may not yet be in a consistent state. Using data members before they have been properly initialized can cause logic errors.*

9.12 Time Class: A Subtle Trap—Returning a Reference to a private Data Member

A reference to an object is an alias for the name of the object and, hence, may be used on the left side of an assignment statement. In this context, the reference makes a perfectly acceptable *lvalue* that can receive a value. One way to use this capability (unfortunately!) is to have a public member function of a class return a reference to a private data member of that class. If a function returns a const reference, that reference cannot be used as a modifiable *lvalue*.

The program of Figs. 9.19–9.21 uses a simplified Time class (Fig. 9.19 and Fig. 9.20) to demonstrate returning a reference to a private data member with member function badSetHour (declared in Fig. 9.19 in line 15 and defined in Fig. 9.20 in lines 27–31). Such a reference return actually makes a call to member function badSetHour an alias for private data member hour! The function call can be used in any way that the private data member can be used, including as an *lvalue* in an assignment statement, thus *enabling clients of the class to clobber the class's* private *data at will!* The same problem would occur if a pointer to the private data were to be returned by the function.

```
1  // Fig. 9.19: Time.h
2  // Time class declaration.
3  // Member functions defined in Time.cpp
4
5  // prevent multiple inclusions of header file
6  #ifndef TIME_H
7  #define TIME_H
8
9  class Time
10 {
11 public:
12    Time( int = 0, int = 0, int = 0 );
```

Fig. 9.19 | Time class declaration. (Part 1 of 2.)

```
13    void setTime( int, int, int );
14    int getHour();
15    int &badSetHour( int ); // DANGEROUS reference return
16 private:
17    int hour;
18    int minute;
19    int second;
20 }; // end class Time
21
22 #endif
```

Fig. 9.19 | `Time` class declaration. (Part 2 of 2.)

```
1  // Fig. 9.20: Time.cpp
2  // Time class member-function definitions.
3  #include "Time.h" // include definition of class Time
4
5  // constructor function to initialize private data; calls member function
6  // setTime to set variables; default values are 0 (see class definition)
7  Time::Time( int hr, int min, int sec )
8  {
9     setTime( hr, min, sec );
10 } // end Time constructor
11
12 // set values of hour, minute and second
13 void Time::setTime( int h, int m, int s )
14 {
15    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
16    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
17    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
18 } // end function setTime
19
20 // return hour value
21 int Time::getHour()
22 {
23    return hour;
24 } // end function getHour
25
26 // POOR PRACTICE: Returning a reference to a private data member.
27 int &Time::badSetHour( int hh )
28 {
29    hour = ( hh >= 0 && hh < 24 ) ? hh : 0;
30    return hour; // DANGEROUS reference return
31 } // end function badSetHour
```

Fig. 9.20 | `Time` class member-function definitions.

Figure 9.21 declares `Time` object `t` (line 10) and reference `hourRef` (line 13), which is initialized with the reference returned by the call `t.badSetHour(20)`. Line 15 displays the value of the alias `hourRef`. This shows how `hourRef` breaks the encapsulation of the class—statements in `main` should not have access to the `private` data of the class. Next, line 16

uses the alias to set the value of hour to 30 (an invalid value) and line 17 displays the value returned by function getHour to show that assigning a value to hourRef actually modifies the private data in the Time object t. Finally, line 21 uses the badSetHour function call itself as an *lvalue* and assigns 74 (another invalid value) to the reference returned by the function. Line 26 again displays the value returned by function getHour to show that assigning a value to the result of the function call in line 21 modifies the private data in the Time object t.

```
1  // Fig. 9.21: fig09_21.cpp
2  // Demonstrating a public membesr function that
3  // returns a reference to a private data member.
4  #include <iostream>
5  #include "Time.h" // include definition of class Time
6  using namespace std;
7
8  int main()
9  {
10    Time t; // create Time object
11
12    // initialize hourRef with the reference returned by badSetHour
13    int &hourRef = t.badSetHour( 20 ); // 20 is a valid hour
14
15    cout << "Valid hour before modification: " << hourRef;
16    hourRef = 30; // use hourRef to set invalid value in Time object t
17    cout << "\nInvalid hour after modification: " << t.getHour();
18
19    // Dangerous: Function call that returns
20    // a reference can be used as an lvalue!
21    t.badSetHour( 12 ) = 74; // assign another invalid value to hour
22
23    cout << "\n\n*************************************************\n"
24       << "POOR PROGRAMMING PRACTICE!!!!!!!!\n"
25       << "t.badSetHour( 12 ) as an lvalue, invalid hour: "
26       << t.getHour()
27       << "\n*************************************************" << endl;
28 } // end main
```

```
Valid hour before modification: 20
Invalid hour after modification: 30

*************************************************
POOR PROGRAMMING PRACTICE!!!!!!!!
t.badSetHour( 12 ) as an lvalue, invalid hour: 74
*************************************************
```

Fig. 9.21 | Returning a reference to a private data member.

Error-Prevention Tip 9.6

Returning a reference or a pointer to a private *data member breaks the encapsulation of the class and makes the client code dependent on the representation of the class's data; this is a dangerous practice that should be avoided.*

9.13 Default Memberwise Assignment

The assignment operator (=) can be used to assign an object to another object of the same type. By default, such assignment is performed by **memberwise assignment**—each data member of the object on the right of the assignment operator is assigned individually to the same data member in the object on the left of the assignment operator. Figures 9.22–9.23 define class `Date` for use in this example. Line 18 of Fig. 9.24 uses **default memberwise assignment** to assign the data members of `Date` object `date1` to the corresponding data members of `Date` object `date2`. In this case, the `month` member of `date1` is assigned to the `month` member of `date2`, the `day` member of `date1` is assigned to the `day` member of `date2` and the `year` member of `date1` is assigned to the `year` member of `date2`. [*Caution:* Memberwise assignment can cause serious problems when used with a class whose data members contain pointers to dynamically allocated memory; we discuss these problems in Chapter 11 and show how to deal with them.] The `Date` constructor does not contain any error checking; we leave this to the exercises.

```
1  // Fig. 9.19: Date.h
2  // Date class declaration.  Member functions are defined in Date.cpp.
3
4  // prevent multiple inclusions of header file
5  #ifndef DATE_H
6  #define DATE_H
7
8  // class Date definition
9  class Date
10 {
11 public:
12    Date( int = 1, int = 1, int = 2000 ); // default constructor
13    void print();
14 private:
15    int month;
16    int day;
17    int year;
18 }; // end class Date
19
20 #endif
```

Fig. 9.22 | `Date` class declaration.

```
1  // Fig. 9.20: Date.cpp
2  // Date class member-function definitions.
3  #include <iostream>
4  #include "Date.h" // include definition of class Date from Date.h
5  using namespace std;
6
7  // Date constructor (should do range checking)
8  Date::Date( int m, int d, int y )
9  {
10    month = m;
```

Fig. 9.23 | `Date` class member-function definitions. (Part 1 of 2.)

```
11      day = d;
12      year = y;
13   } // end constructor Date
14
15   // print Date in the format mm/dd/yyyy
16   void Date::print()
17   {
18      cout << month << '/' << day << '/' << year;
19   } // end function print
```

Fig. 9.23 | `Date` class member-function definitions. (Part 2 of 2.)

```
1  // Fig. 9.21: fig09_21.cpp
2  // Demonstrating that class objects can be assigned
3  // to each other using default memberwise assignment.
4  #include <iostream>
5  #include "Date.h" // include definition of class Date from Date.h
6  using namespace std;
7
8  int main()
9  {
10    Date date1( 7, 4, 2004 );
11    Date date2; // date2 defaults to 1/1/2000
12
13    cout << "date1 = ";
14    date1.print();
15    cout << "\ndate2 = ";
16    date2.print();
17
18    date2 = date1; // default memberwise assignment
19
20    cout << "\n\nAfter default memberwise assignment, date2 = ";
21    date2.print();
22    cout << endl;
23 } // end main
```

```
date1 = 7/4/2004
date2 = 1/1/2000

After default memberwise assignment, date2 = 7/4/2004
```

Fig. 9.24 | Default memberwise assignment.

Objects may be passed as function arguments and may be returned from functions. Such passing and returning is performed using pass-by-value by default—a copy of the object is passed or returned. In such cases, C++ creates a new object and uses a **copy constructor** to copy the original object's values into the new object. For each class, the compiler provides a default copy constructor that copies each member of the original object into the corresponding member of the new object. Like memberwise assignment, copy constructors can cause serious problems when used with a class whose data members contain pointers to dynamically allocated memory. Chapter 11 discusses how to define cus-

tomized copy constructors that properly copy objects containing pointers to dynamically allocated memory.

Performance Tip 9.3

Passing an object by value is good from a security standpoint, because the called function has no access to the original object in the caller, but pass-by-value can degrade performance when making a copy of a large object. An object can be passed by reference by passing either a pointer or a reference to the object. Pass-by-reference offers good performance but is weaker from a security standpoint, because the called function is given access to the original object. Pass-by-`const`*-reference is a safe, good-performing alternative (this can be implemented with a* `const` *reference parameter or with a pointer-to-*`const`*-data parameter).*

9.14 Wrap-Up

This chapter began with an introduction to the concepts of classes, objects, member functions and data members. We then introduced classes, using a rich `Time` class case study to present many of these concepts. You saw that member functions are usually shorter than global functions because member functions can directly access an object's data members, so the member functions can receive fewer arguments than functions in procedural programming languages. You learned how to use the arrow operator to access an object's members via a pointer of the object's class type.

You learned that member functions have class scope—the member function's name is known only to the class's other members unless referred to via an object of the class, a reference to an object of the class, a pointer to an object of the class or the binary scope resolution operator. We also discussed access functions (commonly used to retrieve the values of data members or to test the truth or falsity of conditions) and utility functions (`private` member functions that support the operation of the class's `public` member functions).

You learned that a constructor can specify default arguments that enable it to be called in a variety of ways. You also learned that any constructor that can be called with no arguments is a default constructor and that there can be at most one default constructor per class. We discussed destructors and their purpose of performing termination housekeeping on an object of a class before that object is destroyed. We also demonstrated the order in which an object's constructors and destructors are called. You learned how to use *set* and *get* functions to control access to a class's `private` data and to allow client code to access that data indirectly.

We demonstrated the problems that can occur when a member function returns a reference to a `private` data member, which breaks the encapsulation of the class. We also showed that objects of the same type can be assigned to one another using default memberwise assignment. We also discussed the benefits of using class libraries to enhance the speed with which code can be created and to increase the quality of software.

Chapter 10 presents additional class features. We'll demonstrate how `const` can be used to indicate that a member function does not modify an object of a class. You'll build classes with composition, which allows a class to contain objects of other classes as members. We'll show how a class can allow so-called "friend" functions to access the class's non-`public` members. We'll also show how a class's non-`static` member functions can use a special pointer named `this` to access an object's members.

Summary

Section 9.2 Classes, Objects, Member Functions and Data Members

- Performing a task in a program requires a function. The function hides from its user the complex tasks that it performs.
- A function in a class is known as a member function and performs one of the class's tasks.
- You must create an object of a class before a program can perform the tasks the class describes.
- Each message sent to an object is a member-function call that tells the object to perform a task.
- An object has attributes that are carried with the object as it's used in a program. These attributes are specified as data members in the object's class.

Section 9.3 `Time` Class

- Data members cannot be initialized where they're declared in the class body (except for a class's `static const` data members of integral or `enum` types). Initialize these data members in the class's constructor (as there is no default initialization for data members of fundamental types).
- Stream manipulator `setfill` specifies the fill character that's displayed when an integer is output in a field that's wider than the number of digits in the value.
- By default, the fill characters appear before the digits in the number.
- Stream manipulator `setfill` is a "sticky" setting, meaning that once the fill character is set, it applies for all subsequent fields being printed.
- Even though a member function declared in a class definition may be defined outside that class definition (and "tied" to the class via the binary scope resolution operator), that member function is still within that class's scope.
- Classes can include objects of other classes as members or they may be derived from other classes that provide attributes and behaviors the new classes can use.

Section 9.4 Class Scope and Accessing Class Members

- A class's data members and member functions belong to that class's scope.
- Nonmember functions are defined at global namespace scope.
- Within a class's scope, class members are immediately accessible by all of that class's member functions and can be referenced by name.
- Outside a class's scope, class members are referenced through one of the handles on an object—an object name, a reference to an object or a pointer to an object.
- Member functions of a class can be overloaded, but only by other member functions of that class.
- Variables declared in a member function have local scope and are known only to that function.
- If a member function defines a variable with the same name as a variable with class scope, the class-scope variable is hidden by the block-scope variable in the local scope.
- The dot member selection operator (`.`) is preceded by an object's name or by a reference to an object to access the object's `public` members.
- The arrow member selection operator (`->`) is preceded by a pointer to an object to access that object's `public` members.
- If a member function is defined in the body of a class definition, the C++ compiler attempts to inline calls to the member function.

Section 9.5 Placing a Class in a Separate File for Reusability

- Class definitions, when packaged properly, can be reused by programmers worldwide.

- It's customary to define a class in a header file that has a `.h` filename extension.
- If the class's implementation changes, the class's clients should not be required to change.
- Interfaces define and standardize the ways in which things such as people and systems interact.
- A class's `public` interface describes the `public` member functions that are made available to the class's clients. The interface describes *what* services clients can use and how to *request* those services, but does not specify *how* the class carries out the services.

Section 9.6 `Time` Class: Separating Interface from Implementation

- Header files contain some portions of a class's implementation and hints about others. Inline member functions, for example, should be in a header file, so that when the compiler compiles a client, the client can include the `inline` function definition in place.
- Preprocessor directives `#ifndef` (which means "if not defined") and `#endif` are used to prevent multiple inclusions of a header file. If the code between these directives has not previously been included in an application, `#define` defines a name that can be used to prevent future inclusions, and the code is included in the source code file.
- A class's `private` members that are listed in the class definition in the header file are visible to clients, even though the clients may not access the `private` members.

Section 9.7 Access Functions and Utility Functions

- A utility function is a `private` member function that supports the operation of the class's `public` member functions. Utility functions are not intended to be used by clients of a class.

Section 9.8 `Time` Class: Constructors with Default Arguments

- Like other functions, constructors can specify default arguments.

Section 9.9 Destructors

- A class's destructor is called implicitly when an object of the class is destroyed.
- The name of the destructor for a class is the tilde (~) character followed by the class name.
- A destructor does not release an object's storage—it performs termination housekeeping before the system reclaims an object's memory, so the memory may be reused to hold new objects.
- A destructor receives no parameters and returns no value. A class may have only one destructor.
- If you do not explicitly provide a destructor, the compiler creates an "empty" destructor, so every class has exactly one destructor.

Section 9.10 When Constructors and Destructors Are Called

- The order in which constructors and destructors are called depends on the order in which execution enters and leaves the scopes where the objects are instantiated.
- Generally, destructor calls are made in the reverse order of the corresponding constructor calls, but the storage classes of objects can alter the order in which destructors are called.

Section 9.11 `Time` Class: Using Set *and* Get *Functions*

- Classes often provide `public` member functions to allow clients of the class to *set* or *get* `private` data members. The names of these member functions normally begin with *set* or *get*.
- *Set* and *get* functions allow clients of a class to indirectly access the hidden data. The client does not know how the object performs these operations.
- A class's *set* and *get* functions should be used by other member functions of the class to manipulate the class's `private` data. If the class's data representation is changed, member functions that access the data only via the *set* and *get* functions will not require modification.

- A `public` *set* function should carefully scrutinize any attempt to modify the value of a data member to ensure that the new value is appropriate for that data item.

Section 9.12 `Time` Class: A Subtle Trap—Returning a Reference to a `private` Data Member

- A reference to an object is an alias for the name of the object and, hence, may be used on the left side of an assignment statement. In this context, the reference makes a perfectly acceptable *lvalue* that can receive a value.
- If the function returns a `const` reference, then the reference cannot be used as a modifiable *lvalue*.

Section 9.13 Default Memberwise Assignment

- The assignment operator (=) can be used to assign an object to another object of the same type. By default, such assignment is performed by memberwise assignment.
- Objects may be passed by value to or returned by value from functions. C++ creates a new object and uses a copy constructor to copy the original object's values into the new object.
- For each class, the compiler provides a default copy constructor that copies each member of the original object into the corresponding member of the new object.

Terminology

`abort` function 390
access function 383
access specifier 367
accessor 397
aggregation 372
arrow member selection operator (`->`) 373
assigning class objects 364
binary scope resolution operator (`::`) 370
body of a class definition 367
class definition 367
class-implementation programmer 380
class library 380
class scope 370
client-code programmer 380
composition 372
copy constructor 403
data hiding 368
default constructor
default memberwise assignment 402
`#define` preprocessor directive 377
defining a class 367
derive one class from another 372
derived 372
destructor 389
driver program 379
`#endif` preprocessor directive 377
`exit` function 390
extensible language 370
file scope 372
fill character 369
get function 397
`.h` file name extension for a header file 375
handle on an object 373
header file 375
helper function 383
`#ifndef` preprocessor directive 377
implicit handle on an object 373
inheritance 372
initializer 368
interface 375
interface of a class 376
memberwise assignment 402
member function 364
member-function call 365
message (sent to an object) 365
mutator 397
name handle on an object 373
object code 380
object handle 373
overloaded member function 373
pointer handle on an object 373
predicate function 383
preprocessor wrapper 377
`private` access specifier 368
`public` access specifier 367
`public` service of a class 376
reference handle on an object 373
requesting a service from an object 365
reusable componentry 372
separating interface from implementation 375

set function 397
`setfill` parameterized stream manipulator 369
source-code file 375
termination housekeeping 389
tilde character (~) in a destructor name 389

Self-Review Exercises

9.1 Fill in the blanks in each of the following:

a) A house is to a blueprint as a(n) _____ is to a class.
b) Every class definition contains the keyword _____ followed immediately by the class's name.
c) A class definition is typically stored in a file with the _____ filename extension.
d) When each object of a class maintains its own copy of an attribute, the variable that represents the attribute is also known as a(n) _____.
e) Keyword `public` is a(n) _____.
f) When a member function is defined outside the class definition, the function header must include the class name and the _____, followed by the function name to "tie" the member function to the class definition.
g) The source-code file and any other files that use a class can include the class's header file via a(n) _____ preprocessor directive.
h) Class members are accessed via the _____ operator in conjunction with the name of an object (or reference to an object) of the class or via the _____ operator in conjunction with a pointer to an object of the class.
i) Class members specified as _____ are accessible only to member functions of the class and friends of the class.
j) Class members specified as _____ are accessible anywhere an object of the class is in scope.
k) _____ can be used to assign an object of a class to another object of the same class.

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

a) Empty parentheses following a function name in a function prototype indicate that the function does not require any parameters to perform its task.
b) Data members or member functions declared with access specifier `private` are accessible to member functions of the class in which they're declared.
c) Variables declared in the body of a particular member function are known as data members and can be used in all member functions of the class.

9.3 Find the error(s) in each of the following and explain how to correct it (them):

a) Assume the following prototype is declared in class `Time`:

```
void ~Time( int );
```

b) The following is a partial definition of class `Time`:

```
class Time
{
public:
   // function prototypes

private:
   int hour = 0;
   int minute = 0;
   int second = 0;
}; // end class Time
```

c) Assume the following prototype is declared in class `Time`:

```
int Time( int, int, int );
```

9.4 What is the difference between a local variable and a data member?

Answers to Self-Review Exercises

9.1 a) object. b) `class`. c) `.h`. d) data member. e) access specifier. f) binary scope resolution operator (`::`). g) `#include`. h) dot (`.`), arrow (`->`). i) `private`. j) `public`. k) Default memberwise assignment (performed by the assignment operator).

9.2 a) True. b) True. c) False. Such variables are local variables and can be used only in the member function in which they're declared. Variables declared in the body of the class are called data members.

9.3 a) *Error:* Destructors are not allowed to return values (or even specify a return type) or take arguments.
Correction: Remove the return type `void` and the parameter `int` from the declaration.
b) *Error:* Members cannot be explicitly initialized in the class definition.
Correction: Remove the explicit initialization from the class definition and initialize the data members in a constructor.
c) *Error:* Constructors are not allowed to return values.
Correction: Remove the return type `int` from the declaration.

9.4 A local variable is declared in the body of a function and can be used only from the point at which it's declared to the closing brace of the block in which it's declared. A data member is declared in a class, but not in the body of any of the class's member functions. Every object of a class has a separate copy of the class's data members. Data members are accessible to all member functions of the class.

Exercises

9.5 What's the purpose of the scope resolution operator?

9.6 What's a default constructor? How are an object's data members initialized if a class has only an implicitly defined default constructor?

9.7 What's a header file? What's a source-code file? Discuss the purpose of each.

9.8 Explain why a class might provide a *set* function and a *get* function for a data member.

9.9 ***(`Account` Class)*** Create an `Account` class that a bank might use to represent customers' bank accounts. Include a data member of type `int` to represent the account balance. [*Note:* In subsequent chapters, we'll use numbers that contain decimal points (e.g., 2.75)—called floating-point values—to represent dollar amounts.] Provide a constructor that receives an initial balance and uses it to initialize the data member. The constructor should validate the initial balance to ensure that it's greater than or equal to 0. If not, set the balance to 0 and display an error message indicating that the initial balance was invalid. Provide three member functions. Member function `credit` should add an amount to the current balance. Member function `debit` should withdraw money from the `Account` and ensure that the debit amount does not exceed the `Account`'s balance. If it does, the balance should be left unchanged and the function should print a message indicating `"Debit amount exceeded account balance."` Member function `getBalance` should return the current balance. Create a program that creates two `Account` objects and tests the member functions of class `Account`.

9.10 ***(`Invoice` Class)*** Create a class called `Invoice` that a hardware store might use to represent an invoice for an item sold at the store. An `Invoice` should include four data members—a part num-

ber (type `string`), a part description (type `string`), a quantity of the item being purchased (type `int`) and a price per item (type `int`). [*Note:* In subsequent chapters, we'll use numbers that contain decimal points (e.g., 2.75)—called floating-point values—to represent dollar amounts.] Your class should have a constructor that initializes the four data members. Provide a *set* and a *get* function for each data member. In addition, provide a member function named `getInvoiceAmount` that calculates the invoice amount (i.e., multiplies the quantity by the price per item), then returns the amount as an `int` value. If the quantity is not positive, it should be set to `0`. If the price per item is not positive, it should be set to `0`. Write a test program that demonstrates class `Invoice`'s capabilities.

9.11 ***(`Employee` Class)*** Create a class called `Employee` that includes three pieces of information as data members—a first name (type `string`), a last name (type `string`) and a monthly salary (type `int`). [*Note:* In subsequent chapters, we'll use numbers that contain decimal points (e.g., 2.75)—called floating-point values—to represent dollar amounts.] Your class should have a constructor that initializes the three data members. Provide a *set* and a *get* function for each data member. If the monthly salary is not positive, set it to `0`. Write a test program that demonstrates class `Employee`'s capabilities. Create two `Employee` objects and display each object's *yearly* salary. Then give each `Employee` a 10 percent raise and display each `Employee`'s yearly salary again.

9.12 ***(`Date` Class)*** Create a class called `Date` that includes three pieces of information as data members—a month (type `int`), a day (type `int`) and a year (type `int`). Your class should have a constructor with three parameters that uses the parameters to initialize the three data members. For the purpose of this exercise, assume that the values provided for the year and day are correct, but ensure that the month value is in the range 1–12; if it isn't, set the month to 1. Provide a *set* and a *get* function for each data member. Provide a member function `displayDate` that displays the month, day and year separated by forward slashes (`/`). Write a test program that demonstrates class `Date`'s capabilities.

9.13 ***(Enhancing Class `Time`)*** Provide a constructor that's capable of using the current time from the `time` and `localtime` functions—declared in the C++ Standard Library header `<ctime>`—to initialize an object of the `Time` class.

9.14 ***(`Complex` Class)*** Create a class called `Complex` for performing complex-number arithmetic. Write a program to test your class. Complex numbers have the form `realPart + imaginaryPart * i` where i is $\sqrt{-1}$. Use `double` variables to represent the `private` data of the class. Provide a constructor that enables an object of this class to be initialized when it's declared. The constructor should contain default values in case no initializers are provided. Provide `public` member functions that perform the following tasks:

a) Adding two `Complex` numbers: The real parts are added together and the imaginary parts are added together.
b) Subtracting two `Complex` numbers: The real part of the right operand is subtracted from the real part of the left operand, and the imaginary part of the right operand is subtracted from the imaginary part of the left operand.
c) Printing `Complex` numbers in the form `(a, b)`, where `a` is the real part and `b` is the imaginary part.

9.15 ***(`Rational` Class)*** Create a class called `Rational` for performing arithmetic with fractions. Write a program to test your class.

Use integer variables to represent the `private` data of the class—the `numerator` and the `denominator`. Provide a constructor that enables an object of this class to be initialized when it's declared. The constructor should contain default values in case no initializers are provided and should store the fraction in reduced form. For example, the fraction

$$\frac{2}{4}$$

would be stored in the object as 1 in the `numerator` and 2 in the `denominator`. Provide `public` member functions that perform each of the following tasks:

a) Adding two `Rational` numbers. The result should be stored in reduced form.
b) Subtracting two `Rational` numbers. The result should be stored in reduced form.
c) Multiplying two `Rational` numbers. The result should be stored in reduced form.
d) Dividing two `Rational` numbers. The result should be stored in reduced form.
e) Printing `Rational` numbers in the form `a/b`, where `a` is the numerator and `b` is the denominator.
f) Printing `Rational` numbers in floating-point format.

9.16 *(Enhancing Class `Time`)* Modify the `Time` class of Figs. 9.10–9.11 to include a `tick` member function that increments the time stored in a `Time` object by one second. The `Time` object should always remain in a consistent state. Write a program that tests the `tick` member function in a loop that prints the time in standard format during each iteration of the loop to illustrate that the `tick` member function works correctly. Be sure to test the following cases:

a) Incrementing into the next minute.
b) Incrementing into the next hour.
c) Incrementing into the next day (i.e., 11:59:59 PM to 12:00:00 AM).

9.17 *(Enhancing Class `Date`)* Modify the `Date` class of Figs. 9.22–9.23 to perform error checking on the initializer values for data members `month`, `day` and `year`. Also, provide a member function `nextDay` to increment the day by one. The `Date` object should always remain in a consistent state. Write a program that tests function `nextDay` in a loop that prints the date during each iteration to illustrate that `nextDay` works correctly. Be sure to test the following cases:

a) Incrementing into the next month.
b) Incrementing into the next year.

9.18 *(Combining Class `Time` and Class `Date`)* Combine the modified `Time` class of Exercise 9.16 and the modified `Date` class of Exercise 9.17 into one class called `DateAndTime`. (In Chapter 12, we'll discuss inheritance, which will enable us to accomplish this task quickly without modifying the existing class definitions.) Modify the `tick` function to call the `nextDay` function if the time increments into the next day. Modify functions `printStandard` and `printUniversal` to output the date and time. Write a program to test the new class `DateAndTime`. Specifically, test incrementing the time into the next day.

9.19 *(Returning Error Indicators from Class `Time`'s* **set** *Functions)* Modify the *set* functions in the `Time` class of Figs. 9.10–9.11 to return appropriate error values if an attempt is made to *set* a data member of an object of class `Time` to an invalid value. Write a program that tests your new version of class `Time`. Display error messages when *set* functions return error values.

9.20 *(`Rectangle` Class)* Create a class `Rectangle` with attributes `length` and `width`, each of which defaults to 1. Provide member functions that calculate the `perimeter` and the `area` of the rectangle. Also, provide *set* and *get* functions for the `length` and `width` attributes. The *set* functions should verify that `length` and `width` are each floating-point numbers larger than 0.0 and less than 20.0.

9.21 *(Enhancing Class `Rectangle`)* Create a more sophisticated `Rectangle` class than the one you created in Exercise 9.20. This class stores only the Cartesian coordinates of the four corners of the rectangle. The constructor calls a *set* function that accepts four sets of coordinates and verifies that each of these is in the first quadrant with no single x- or y-coordinate larger than 20.0. The *set* function also verifies that the supplied coordinates do, in fact, specify a rectangle. Provide member functions that calculate the `length`, `width`, `perimeter` and `area`. The length is the larger of the two dimensions. Include a predicate function `square` that determines whether the rectangle is a square.

9.22 *(Enhancing Class `Rectangle`)* Modify class `Rectangle` from Exercise 9.21 to include a `draw` function that displays the rectangle inside a 25-by-25 box enclosing the portion of the first quadrant in which the rectangle resides. Include a `setFillCharacter` function to specify the character out of which the body of the rectangle will be drawn. Include a `setPerimeterCharacter` function to specify

the character that will be used to draw the border of the rectangle. If you feel ambitious, you might include functions to scale the size of the rectangle, rotate it, and move it around within the designated portion of the first quadrant.

9.23 ***(HugeInteger Class)*** Create a class `HugeInteger` that uses a 40-element array of digits to store integers as large as 40 digits each. Provide member functions `input`, `output`, `add` and `subtract`. For comparing `HugeInteger` objects, provide functions `isEqualTo`, `isNotEqualTo`, `isGreaterThan`, `isLessThan`, `isGreaterThanOrEqualTo` and `isLessThanOrEqualTo`—each of these is a "predicate" function that simply returns `true` if the relationship holds between the two `HugeIntegers` and returns `false` if the relationship does not hold. Also, provide a predicate function `isZero`. If you feel ambitious, provide member functions `multiply`, `divide` and `modulus`.

9.24 ***(TicTacToe Class)*** Create a class `TicTacToe` that will enable you to write a complete program to play the game of tic-tac-toe. The class contains as `private` data a 3-by-3 two-dimensional array of integers. The constructor should initialize the empty board to all zeros. Allow two human players. Wherever the first player moves, place a 1 in the specified square. Place a 2 wherever the second player moves. Each move must be to an empty square. After each move, determine whether the game has been won or is a draw. If you feel ambitious, modify your program so that the computer makes the moves for one of the players. Also, allow the player to specify whether he or she wants to go first or second. If you feel exceptionally ambitious, develop a program that will play three-dimensional tic-tac-toe on a 4-by-4-by-4 board. [*Caution:* This is an extremely challenging project that could take many weeks of effort!]

Making a Difference

9.25 ***(Target-Heart-Rate Calculator)*** While exercising, you can use a heart-rate monitor to see that your heart rate stays within a safe range suggested by your trainers and doctors. According to the American Heart Association (AHA) (`www.americanheart.org/presenter.jhtml?identifier=4736`), the formula for calculating your *maximum heart rate* in beats per minute is 220 minus your age in years. Your *target heart rate* is a range that's 50–85% of your maximum heart rate. *[*Note: *These formulas are estimates provided by the AHA. Maximum and target heart rates may vary based on the health, fitness and gender of the individual. Always consult a physician or qualified health care professional before beginning or modifying an exercise program.]* Create a class called `HeartRates`. The class attributes should include the person's first name, last name and date of birth (consisting of separate attributes for the month, day and year of birth). Your class should have a constructor that receives this data as parameters. For each attribute provide *set* and *get* functions. The class also should include a function `getAge` that calculates and returns the person's age (in years), a function `getMaxiumumHeartRate` that calculates and returns the person's maximum heart rate and a function `getTargetHeartRate` that calculates and returns the person's target heart rate. Since you do not yet know how to obtain the current date from the computer, function `getAge` should prompt the user to enter the current month, day and year before calculating the person's age. Write an application that prompts for the person's information, instantiates an object of class `HeartRates` and prints the information from that object—including the person's first name, last name and date of birth—then calculates and prints the person's age in (years), maximum heart rate and target-heart-rate range.

9.26 ***(Computerization of Health Records)*** A health care issue that has been in the news lately is the computerization of health records. This possibility is being approached cautiously because of sensitive privacy and security concerns, among others. [We address such concerns in later exercises.] Computerizing health records could make it easier for patients to share their health profiles and histories among their various health care professionals. This could improve the quality of health care, help avoid drug conflicts and erroneous drug prescriptions, reduce costs and in emergencies, could save lives. In this exercise, you'll design a "starter" `HealthProfile` class for a person. The class attributes should include the person's first name, last name, gender, date of birth (consisting of separate

attributes for the month, day and year of birth), height (in inches) and weight (in pounds). Your class should have a constructor that receives this data. For each attribute, provide *set* and *get* functions. The class also should include functions that calculate and return the user's age in years, maximum heart rate and target-heart-rate range (see Exercise 9.25), and body mass index (BMI; see Exercise 2.30). Write an application that prompts for the person's information, instantiates an object of class `HealthProfile` for that person and prints the information from that object—including the person's first name, last name, gender, date of birth, height and weight—then calculates and prints the person's age in years, BMI, maximum heart rate and target-heart-rate range. It should also display the "BMI values" chart from Exercise 2.30. Use the same technique as Exercise 9.25 to calculate the person's age.