9

Classes: A Deeper Look, Part 1

My object all sublime
I shall achieve in time.
—W. S. Gilbert

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

Don't be "consistent," but be simply true.
—Oliver Wendell Holmes, Jr.

Objectives

In this chapter you'll learn:

- How to use a preprocessor wrapper to prevent multiple definition errors.
- To understand class scope and accessing class members via the name of an object, a reference to an object or a pointer to an object.
- To define constructors with default arguments.
- How destructors are used to perform "termination housekeeping" on an object before it's destroyed.
- When constructors and destructors are called and the order in which they're called.
- The logic errors that may occur when a `public` member function returns a reference to `private` data.
- To assign the data members of one object to those of another object by default memberwise assignment.

9.1 Introduction

In the preceding chapters, we introduced many basic terms and concepts of C++ object-oriented programming. We also discussed our program development methodology: We selected appropriate attributes and behaviors for each class and specified the manner in which objects of our classes collaborated with objects of C++ Standard Library classes to accomplish each program's overall goals.

In this chapter, we take a deeper look at classes. We use an integrated `Time` class case study in both this chapter and Chapter 10, Classes: A Deeper Look, Part 2 to demonstrate several class construction capabilities. We begin with a `Time` class that reviews several of the features presented in the preceding chapters. The example also demonstrates an important C++ software engineering concept—using a "preprocessor wrapper" in header files to prevent the code in the header from being included into the same source code file more than once. Since a class can be defined only once, using such preprocessor directives prevents multiple definition errors.

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 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. Next, we discuss a special member function called a destructor that is part of every class and is used to perform "termination housekeeping" on an object before the object is destroyed. We then demonstrate the order in which constructors and destructors are called, because your programs' correctness depends on using properly initialized objects that have not yet been destroyed.

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 and allows 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.

9.2 Time Class Case Study

Our first example (Figs. 9.1–9.3) creates class Time and a driver program that tests the class. You've already created many classes in this book. In this section, we review many of the concepts covered in Chapter 3 and demonstrate an important C++ software engineering concept—using a "preprocessor wrapper" in header files to prevent the code in the header from being included into the same source code file more than once. Since a class can be defined only once, using such preprocessor directives prevents multiple-definition errors.

```
1  // Fig. 9.1: 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
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. 9.1 | Time class definition.

Time Class Definition

The class definition (Fig. 9.1) contains prototypes (lines 13–16) for member functions Time, setTime, printUniversal and printStandard, and includes private integer members hour, minute and second (lines 18–20). Class Time's private data members can be accessed only by its four member functions. Chapter 12 introduces a third access specifier, protected, as we study inheritance and the part it plays in object-oriented programming.

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 9.1

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

In Fig. 9.1, 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. [*Note:* 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 9.1

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.*

Good Programming Practice 9.2

Use the name of the header file in upper case with the period replaced by an underscore in the `#ifndef` *and* `#define` *preprocessor directives of a header file.*

`Time` Class Member Functions

In Fig. 9.2, the `Time` constructor (lines 10–13) 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). Finally, it's important to note that you can define several overloaded constructors for a class.

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 is no default initialization for fundamental-type data members). Data members can also be assigned values by `Time`'s *set* functions. [*Note:* Chapter 10 demon-

```
1  // Fig. 9.2: 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. 9.2 | `Time` class member-function definitions.

strates that only a class's `static const` data members of integral or `enum` types can be initialized in the class's body.]

Common Programming Error 9.1

Attempting to initialize a non-`static` data member of a class explicitly in the class definition is a syntax error.

Function `setTime` (lines 17–22) 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 19) 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 20 and 21)

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`.

A value passed to `setTime` is a correct value if it's in the allowed range for the member it's initializing. So, any number in the range 0–23 would be a correct value for the `hour`. A correct value is always a consistent value. However, a consistent value is not necessarily a correct value. If `setTime` sets `hour` to `0` because the argument received was out of range, then `hour` is correct only if the current time is coincidentally midnight.

Function `printUniversal` (lines 25–29 of Fig. 9.2) 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 return `13:30:07`. Line 27 uses parameterized stream manipulator **`setfill`** to specify the **fill character** that is 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 (i.e., `setfill` is a "sticky" setting). This is in contrast to `setw`, which applies only to the next value displayed (`setw` is a "nonsticky" setting).

Error-Prevention Tip 9.2

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.

Function `printStandard` (lines 32–37) 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 34 uses the conditional operator (`?:`) to determine the value of `hour` to be displayed—if the `hour` is `0` or `12` (AM or PM), it appears as 12; otherwise, the `hour` appears as a value from 1 to 11. The conditional operator in line 36 determines whether AM or PM will be displayed.

Defining Member Functions Outside the Class Definition; Class Scope

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**; i.e., 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.

If a member function is defined in the body of a class definition, the compiler attempts to inline calls to the member function. Remember that the compiler reserves the right not to inline any function.

Performance Tip 9.1

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 9.2

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 9.3

Only the simplest and most stable member functions (i.e., whose implementations are unlikely to change) should be defined in the class header.

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 for which they're invoked. This can make member function calls more concise than conventional function calls in procedural programming.

Software Engineering Observation 9.4

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 9.5

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 9.3

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.

Using Class `Time`

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
```

Figure 9.3 uses class `Time`. Line 10 instantiates a single object of class `Time` called `t`. When the object is instantiated, the `Time` constructor is called to initialize each `private` data member to 0. Then, lines 14 and 16 print the time in universal and standard formats, respectively, to confirm that the members were initialized properly. Line 18 sets a new time by calling member function `setTime`, and lines 22 and 24 print the time again in

```
1  // Fig. 9.3: fig09_03.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
```

```
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. 9.3 | Program to test class `Time`.

both formats. Line 26 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 31 and 33 print the time again in both formats.

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 9.2

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.*

9.3 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 6).

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 9.4 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. 9.4: fig09_04.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    {
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
```

Fig. 9.4 | 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.)

```
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. 9.4 | 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.)

9.4 Separating Interface from Implementation

In Chapter 3, we began by including a class's definition and member-function definitions in one file. We then demonstrated separating this code into two files—a header file for the class definition (i.e., the class's interface) and a source code file for the class's member-function definitions (i.e., the class's implementation). Recall that this makes it easier to modify programs—as far as clients of a class are concerned, changes in the class's implementation do not affect the client as long as the class's interface originally provided to the client remains unchanged.

Software Engineering Observation 9.6

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.

Actually, things are not quite this rosy. Header files do contain some portions of the 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. A class's `private` members are listed in the class definition in the header file, so these members are visible to clients even though the clients may not access the `private` members. In Chapter 10, we show how to use a "proxy class" to hide even the `private` data of a class from clients of the class.

Software Engineering Observation 9.7

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.

9.5 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. 9.5–9.7 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. 9.5) 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. 9.5: 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. 9.5 | `SalesPerson` class definition.

In Fig. 9.6, 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. 9.6: 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
```

Fig. 9.6 | `SalesPerson` class member-function definitions. (Part 1 of 2.)

```
47 // private utility function to total annual sales
48 double SalesPerson::totalAnnualSales()
49 {
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. 9.6 | `SalesPerson` class member-function definitions. (Part 2 of 2.)

In Fig. 9.7, 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 9.8

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. 9.7: fig09_07.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
```

Fig. 9.7 | Utility function demonstration. (Part 1 of 2.)

```
Enter sales amount for month 11: 4024.97
Enter sales amount for month 12: 5923.92

The total annual sales are: $60120.59
```

Fig. 9.7 | Utility function demonstration. (Part 2 of 2.)

9.6 Time Class Case Study: Constructors with Default Arguments

The program of Figs. 9.8–9.10 enhances class `Time` to demonstrate how arguments are implicitly passed to a constructor. The constructor defined in Fig. 9.2 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. 9.8 declares the `Time` constructor to include default arguments, specifying a default value of zero for each argument passed to the constructor. In Fig. 9.9, 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. Class `Time` provides *set* and *get* functions for each data member. The `Time` constructor now calls `setTime`, which calls the `setHour`, `setMinute` and `setSecond` functions 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. 9.8: 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
```

Fig. 9.8 | `Time` class containing a constructor with default arguments. (Part 1 of 2.)

```
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. 9.8 | `Time` class containing a constructor with default arguments. (Part 2 of 2.)

In Fig. 9.9, 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). 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.

```
1  // Fig. 9.9: 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    setHour( h ); // set private field hour
20    setMinute( m ); // set private field minute
21    setSecond( s ); // set private field second
22 } // end function setTime
23
24 // set hour value
25 void Time::setHour( int h )
26 {
```

Fig. 9.9 | `Time` class member-function definitions including a constructor that takes arguments. (Part 1 of 2.)

```
27    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
28 } // end function setHour
29
30 // set minute value
31 void Time::setMinute( int m )
32 {
33    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
34 } // end function setMinute
35
36 // set second value
37 void Time::setSecond( int s )
38 {
39    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
40 } // end function setSecond
41
42 // return hour value
43 int Time::getHour()
44 {
45    return hour;
46 } // end function getHour
47
48 // return minute value
49 int Time::getMinute()
50 {
51    return minute;
52 } // end function getMinute
53
54 // return second value
55 int Time::getSecond()
56 {
57    return second;
58 } // end function getSecond
59
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. 9.9 | `Time` class member-function definitions including a constructor that takes arguments. (Part 2 of 2.)

The `Time` constructor could be written to include the same statements as member function `setTime`, or even the individual statements in the `setHour`, `setMinute` and `setSecond` functions. Calling `setHour`, `setMinute` and `setSecond` from the constructor may

be slightly more efficient because the extra call to `setTime` would be eliminated. Similarly, copying the code from lines 27, 33 and 39 into constructor would eliminate the overhead of calling `setTime`, `setHour`, `setMinute` and `setSecond`. Coding the `Time` constructor or member function `setTime` as a copy of the code in lines 27, 33 and 39 would make maintenance of this class more difficult. If the implementations of `setHour`, `setMinute` and `setSecond` were to change, the implementation of any member function that duplicates lines 27, 33 and 39 would have to change accordingly. Having the `Time` constructor call `setTime` and having `setTime` call `setHour`, `setMinute` and `setSecond` enables us to limit the changes to code that validates the `hour`, `minute` or `second` to the corresponding *set* function. This reduces the likelihood of errors when altering the class's implementation. Also, the performance of the `Time` constructor and `setTime` can be enhanced by explicitly declaring them `inline` or by defining them in the class definition (which implicitly inlines the function definition).

Software Engineering Observation 9.9

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 9.10

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).

Function `main` in Fig. 9.10 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.

```
1  // Fig. 9.10: fig09_10.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  ";
```

Fig. 9.10 | Constructor with default arguments. (Part 1 of 2.)

```
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

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. 9.10 | Constructor with default arguments. (Part 2 of 2.)

Notes Regarding Class `Time`*'s* Set *and* Get *Functions and Constructor*

`Time`'s *set* and *get* functions are called throughout the class's body. In particular, function `setTime` (lines 17–22 of Fig. 9.9) 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 represen-

tation 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.

Common Programming Error 9.2

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.*

9.7 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. This naming convention has intuitive appeal, because as we'll see in a later chapter, the tilde operator is the bitwise complement operator, and, in a sense, the destructor is the complement of the constructor. A destructor is often referred to with the abbreviation "dtor" in the literature. We prefer not to use this abbreviation.

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 9.3

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.

Software Engineering Observation 9.11

As we'll see in the remainder of the book, constructors and destructors have much greater prominence in C++ and object-oriented programming than is possible to convey after only our brief introduction here.

9.8 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. 9.11–9.13, 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. 9.11–9.13 demonstrates the order in which constructors and destructors are called for objects of class `CreateAndDestroy` (Fig. 9.11 and Fig. 9.12) 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. 9.11 lines 16–17). This mechanical example is purely for pedagogic purposes. For this reason, line 21 of the destructor in Fig. 9.12 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. 9.11: 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. 9.11 | `CreateAndDestroy` class definition.

```
1  // Fig. 9.12: 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
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. 9.12 | `CreateAndDestroy` class member-function definitions.

Figure 9.13 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.

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.

```
1  // Fig. 9.13: fig09_13.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. 9.13 | 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. 9.13 | Order in which constructors and destructors are called. (Part 2 of 2.)

9.9 `Time` Class Case Study: 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. 9.14–9.16 uses a simplified `Time` class (Fig. 9.14 and Fig. 9.15) to demonstrate returning a reference to a `private` data member with member function `badSetHour` (declared in Fig. 9.14 in line 15 and defined in Fig. 9.15 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. 9.14: Time.h
2  // Time class declaration.
3  // Member functions defined in Time.cpp
4
```

Fig. 9.14 | `Time` class declaration. (Part 1 of 2.)

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

class Time
{
public:
   Time( int = 0, int = 0, int = 0 );
   void setTime( int, int, int );
   int getHour();
   int &badSetHour( int ); // DANGEROUS reference return
private:
   int hour;
   int minute;
   int second;
}; // end class Time

#endif
```

Fig. 9.14 | Time class declaration. (Part 2 of 2.)

```
// Fig. 9.15: Time.cpp
// Time class member-function definitions.
#include "Time.h" // include definition of class Time

// constructor function to initialize private data; calls member function
// setTime to set variables; default values are 0 (see class definition)
Time::Time( int hr, int min, int sec )
{
   setTime( hr, min, sec );
} // end Time constructor

// set values of hour, minute and second
void Time::setTime( int h, int m, int s )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
   minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
   second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
} // end function setTime

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

// POOR PRACTICE: Returning a reference to a private data member.
int &Time::badSetHour( int hh )
{
   hour = ( hh >= 0 && hh < 24 ) ? hh : 0;
   return hour; // DANGEROUS reference return
} // end function badSetHour
```

Fig. 9.15 | Time class member-function definitions.

Figure 9.16 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. 9.16: fig09_16.cpp
2  // Demonstrating a public member 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. 9.16 | Returning a reference to a `private` data member.

Error-Prevention Tip 9.4

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.*

9.10 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 9.17–9.18 define class `Date` for use in this example. Line 18 of Fig. 9.19 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. 9.17: 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. 9.17 | `Date` class declaration.

```
1  // Fig. 9.18: Date.cpp
2  // Date class member-function definitions.
3  #include <iostream>
4  #include "Date.h" // include definition of class Date from Date.h
```

Fig. 9.18 | `Date` class member-function definitions. (Part 1 of 2.)

```
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;
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. 9.18 | `Date` class member-function definitions. (Part 2 of 2.)

```
1  // Fig. 9.19: fig09_19.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. 9.19 | 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 customized 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).*

9.11 Wrap-Up

This chapter deepened our coverage of classes, using a rich `Time` class case study to introduce several new features. 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.

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.