



# Chapter 9

## Classes

C++ How to Program,  
Late Objects Version, 7/e



## 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.



- 
- 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
-



## 9.1 Introduction

- ▶ In this chapter, you'll begin writing programs that employ the basic concepts of object-oriented programming.
- ▶ Typically, programs consist of function `main` and one or more classes, each containing data members and member functions.
- ▶ In this chapter, we develop a simple, well-engineered framework for organizing object-oriented programs.



## 9.2 Classes, Objects, Member Functions and Data Members

- ▶ Let's begin with a simple analogy to help you reinforce your understanding 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*.



## 9.2 Classes, Objects, Member Functions and Data Members (cont.)

- ▶ 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.
- ▶ People with little or no knowledge of how cars are engineered can drive a car easily, simply by using the accelerator pedal, the brake pedal, the steering wheel, the transmission shifting mechanism and other user-friendly “interfaces” to the car’s complex internal mechanisms.



## 9.2 Classes, Objects, Member Functions and Data Members (cont.)

- ▶ Unfortunately, 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 not accelerate on its own, so the driver must press the accelerator pedal to tell the car to go faster.



## 9.2 Classes, Objects, Member Functions and Data Members (cont.)

- ▶ 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.



## 9.2 Classes, Objects, Member Functions and Data Members (cont.)

- ▶ 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 9.1 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.



## 9.2 Classes, Objects, Member Functions and Data Members (cont.)

- ▶ 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. many cars can be built from the same engineering drawing, many objects can be built from the same class.
- ▶ Note also that just as



## 9.2 Classes, Objects, Member Functions and Data Members (cont.)

- ▶ 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**.



## 9.2 Classes, Objects, Member Functions and Data Members (cont.)

- ▶ 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.



## 9.2 Classes, Objects, Member Functions and Data Members (cont.)

- ▶ 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
20
```

**Fig. 9.1** | Time class definition. (Part I of 4.)



```
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
```

**Fig. 9.1** | Time class definition. (Part 2 of 4.)



```
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
```

**Fig. 9.1** | Time class definition. (Part 3 of 4.)



```
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
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 4 of 4.)



## 9.3 Time Class (cont.)

- ▶ 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.*



## 9.3 Time Class (cont.)

- ▶ 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.
- ▶ These 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 (`:`).



## 9.3 Time Class (cont.)

- ▶ We provide prototypes in class `Time` for four member functions after the `public` member access specifier—`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).
- ▶ These 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.



## 9.3 Time Class (cont.)

- ▶ 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 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.
- ▶ 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.



## 9.3 Time Class (cont.)

- ▶ 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**.



## 9.3 Time Class (cont.)

- ▶ **private** data members 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.*



## 9.3 Time Class (cont.)

- ▶ 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).
- ▶ 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.



## 9.3 Time Class (cont.)

- ▶ 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.
- ▶ In class `Time`, invalid values are set to zero to ensure that a `Time` object always contains consistent data
  - 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.



## 9.3 Time Class (cont.)

- ▶ 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.
- ▶ Parameterized stream manipulator `setfill` specifies 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.
- ▶ 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 9.1

*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.*



## 9.3 Time Class (cont.)

- ▶ 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.



## 9.3 Time Class (cont.)

- ▶ 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.
  - The compiler would not be able to compile these functions.



## 9.3 Time Class (cont.)

- ▶ A member function declared in a class definition, 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.



## 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.*



## 9.3 Time Class (cont.)

- ▶ Next, we'd like to use class `Time` in a program.
- ▶ In this program, we'd like to call class `Time`'s members 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.
- ▶ Line 54 creates an object of class `Time` called `t`.
  - The variable's type is `Time`.
- ▶ 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++).



## 9.3 Time Class (cont.)

- ▶ 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.
- ▶ When the object is instantiated (line 54), the **Time** constructor is called to initialize each **private** data member to 0.



## 9.3 Time Class (cont.)

- ▶ 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 additional data to perform its task.



## 9.3 Time Class (cont.)

- ▶ 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*.
- ▶ **Information hiding** promotes program modifiability and simplifies the client's perception of a class.



## 9.3 Time Class (cont.)

- ▶ 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.



## 9.3 Time Class (cont.)

- ▶ 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 9.3

*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.*



## 9.3 Time Class (cont.)

- ▶ 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.



## 9.3 Time Class (cont.)

- ▶ 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.



## 9.4 Class Scope and Accessing Class Members (cont.)

- ▶ 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 (::).



## 9.4 Class Scope and Accessing Class Members (cont.)

- ▶ 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.



---

```
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    {
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
```

---

**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 I of 3.)



```
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
```

**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 3.)



```
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 3 of 3.)



## 9.4 Class Scope and Accessing Class Members (cont.)

- ▶ 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  
(i.e., 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, our 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).



## 9.5 Placing a Class in a Separate File for Reusability (cont.)

- ▶ Programmers who wish to use our `Time` class cannot simply include the file from Fig. 9.1 in another program.
- ▶ 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.



## 9.5 Placing a Class in a Separate File for Reusability (cont.)

- ▶ 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`.



## 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**.



## 9.6 Time Class: Separating Interface from Implementation (cont.)

- ▶ Interfaces define and standardize the ways in which things such as people and systems interact with one another.
- ▶ The interface specifies *what* operations a radio permits users to perform but does not specify *how* the operations are implemented inside the radio.
- ▶ 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**).



## 9.6 Time Class: Separating Interface from Implementation (cont.)

- ▶ 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.
- ▶ 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.



## 9.6 Time Class: Separating Interface from Implementation (cont.)

- ▶ 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.



## 9.6 Time Class: Separating Interface from Implementation (cont.)

- ▶ 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
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.



## 9.6 Time Class: Separating Interface from Implementation (cont.)

- 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.



## 9.6 Time Class: Separating Interface from Implementation (cont.)

- ▶ 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.*



## 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.*



## 9.6 Time Class: Separating Interface from Implementation (cont.)

- ▶ 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
```

**Fig. 9.4** | Time class member-function definitions. (Part I of 2.)



---

```
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. (Part 2 of 2.)



## 9.6 Time Class: Separating Interface from Implementation (cont.)

- ▶ 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).
- ▶ When compiling `Time.cpp`, the compiler uses the information in `Time.h` to ensure that
  - the first line of each member function (lines 10, 17, 25 and 32) matches its prototype in the `Time.h`
  - each member function knows about the class's data members and other member.



## 9.6 Time Class: Separating Interface from Implementation (cont.)

- ▶ To help you prepare for the larger programs you'll encounter going forward 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.



## 9.6 Time Class: Separating Interface from Implementation (cont.)

- ▶ 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
```

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



```
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. 9.5** | 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.*



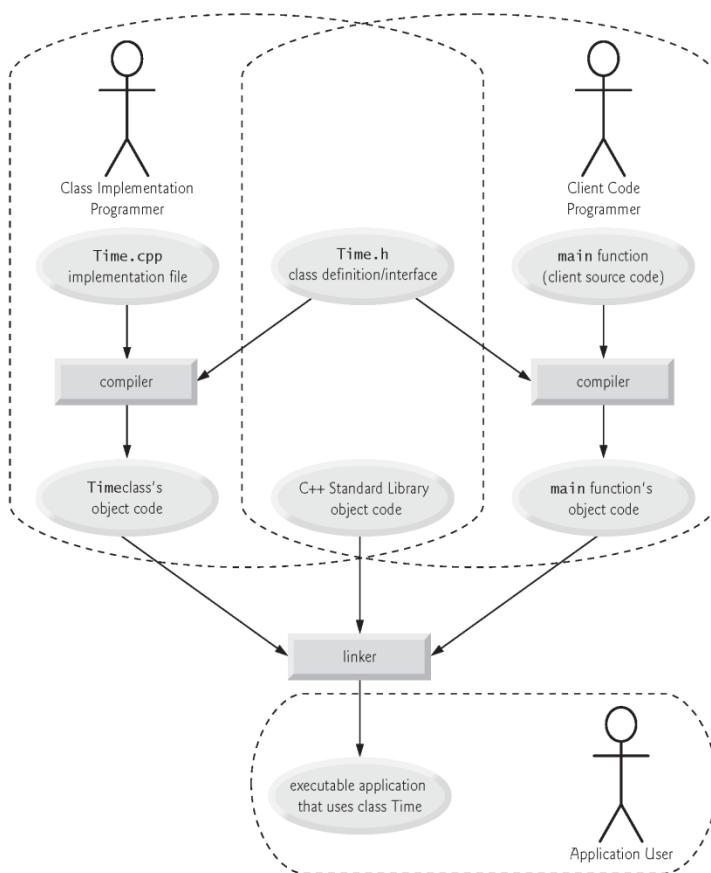
## Software Engineering Observation 9.8

*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.*



## 9.6 Time Class: Separating Interface from Implementation (cont.)

- ▶ 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.



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



## 9.6 Time Class: Separating Interface from Implementation (cont.)

- ▶ To create the executable `Time` application, the last step is to link
  - the object code for the `main` function (i.e., the client code),
  - the object code for class `Time`'s member-function implementations
  - 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.



## 9.6 Time Class: Separating Interface from Implementation (cont.)

- ▶ 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++ 2008 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.



## 9.6 Time Class: Separating Interface from Implementation (cont.)

- ▶ 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`.
- ▶ Useful predicate functions for our `Time` class might be `isAM` and `isPM`.



## 9.7 Access Functions and Utility Functions (cont.)

- ▶ 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).



```
1 // Fig. 9.7: SalesPerson.h
2 // SalesPerson class definition.
3 // Member functions defined in SalesPerson.cpp.
4 #ifndef SALES_P_H
5 #define SALES_P_H
6
7 class SalesPerson
{
8 public:
9     static const int monthsPerYear = 12; // months in one year
10    SalesPerson(); // constructor
11    void getSalesFromUser(); // input sales from keyboard
12    void setSales( int, double ); // set sales for a specific month
13    void printAnnualSales(); // summarize and print sales
14
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.



## 9.7 Access Functions and Utility Functions (cont.)

- ▶ 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.
- ▶ 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
```

---

**Fig. 9.8** | SalesPerson class member-function definitions. (Part I of 3.)



```
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
```

**Fig. 9.8** | SalesPerson class member-function definitions. (Part 2 of 3.)



---

```
38 // print total annual sales (with the help of utility function)
39 void SalesPerson::printAnnualSales()
40 {
41     cout << setprecision( 2 ) << fixed
42     << "\nThe total annual sales are: $"
43     << totalAnnualSales() << endl; // call utility function
44 } // end function printAnnualSales
45
46 // private utility function to total annual sales
47 double SalesPerson::totalAnnualSales()
48 {
49     double total = 0.0; // initialize total
50
51     for ( int i = 0; i < monthsPerYear; i++ ) // summarize sales results
52         total += sales[ i ]; // add month i sales to total
53
54     return total;
55 } // end function totalAnnualSales
```

---

**Fig. 9.8** | SalesPerson class member-function definitions. (Part 3 of 3.)



## 9.7 Access Functions and Utility Functions (cont.)

- ▶ 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
```

---

**Fig. 9.9** | Utility function demonstration. (Part I of 2.)



```
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. (Part 2 of 2.)



## 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.



```
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.



## 9.8 Time Class: Constructors with Default Arguments (cont.)

- ▶ 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).
- ▶ 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.9.11: 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
```

**Fig. 9.11** | Time class member-function definitions including a constructor that takes arguments. (Part I of 2.)



---

```
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. (Part 2 of 2.)



## 9.8 Time Class: Constructors with Default Arguments (cont.)

- ▶ 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.



```
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     Time t1; // all arguments defaulted
9     Time t2( 2 ); // hour specified; minute and second defaulted
10    Time t3( 21, 34 ); // hour and minute specified; second defaulted
11    Time t4( 12, 25, 42 ); // hour, minute and second specified
12    Time t5( 27, 74, 99 ); // all bad values specified
13
14
15    cout << "Constructed with:\n\tt1: all arguments defaulted\n\t";
16    t1.printUniversal(); // 00:00:00
17    cout << "\n\t";
18    t1.printStandard(); // 12:00:00 AM
19
20    cout << "\n\tt2: hour specified; minute and second defaulted\n\t";
21    t2.printUniversal(); // 02:00:00
22    cout << "\n\t";
23    t2.printStandard(); // 2:00:00 AM
```

**Fig. 9.12** | Constructor with default arguments. (Part I of 3.)



---

```
24
25     cout << "\n\n";
26     t3.printUniversal(); // 21:34:00
27     cout << "\n ";
28     t3.printStandard(); // 9:34:00 PM
29
30     cout << "\n\n";
31     t4.printUniversal(); // 12:25:42
32     cout << "\n ";
33     t4.printStandard(); // 12:25:42 PM
34
35     cout << "\n\n";
36     t5.printUniversal(); // 00:00:00
37     cout << "\n ";
38     t5.printStandard(); // 12:00:00 AM
39     cout << endl;
40 } // end main
```

---

**Fig. 9.12** | Constructor with default arguments. (Part 2 of 3.)



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. 9.12** | Constructor with default arguments. (Part 3 of 3.)



## 9.8 Time Class: Constructors with Default Arguments (cont.)

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



## 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).*



## 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 destructor is often referred to with the abbreviation “**dtr**” in the literature.
- ▶ 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.



## 9.9 Destructors (cont.)

- ▶ 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**.
- ▶ *Every class has a destructor.*
- ▶ If you do not explicitly provide a destructor, the compiler creates an “empty” destructor.



## 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.*



## 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
- ▶ The storage classes of objects can alter the order in which destructors are called.



## 9.10 When Constructors and Destructors Are Called (cont.)

- ▶ 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.
  - 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.
  - Usually used to indicate an abnormal termination of the program.



## 9.10 When Constructors and Destructors Are Called (cont.)

- ▶ 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`.



## 9.10 When Constructors and Destructors Are Called (cont.)

- ▶ 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.



```
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
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.



```
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
```

**Fig. 9.15** | Order in which constructors and destructors are called. (Part I of 3.)



---

```
24 // function to create objects
25 void create( void )
26 {
27     cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;
28     CreateAndDestroy fifth( 5, "(local automatic in create)" );
29     static CreateAndDestroy sixth( 6, "(local static in create)" );
30     CreateAndDestroy seventh( 7, "(local automatic in create)" );
31     cout << "\nCREATE FUNCTION: EXECUTION ENDS" << endl;
32 } // end function create
```

---

**Fig. 9.15** | Order in which constructors and destructors are called. (Part 2 of 3.)



```
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 3 of 3.)



## 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 }
```

---

**Fig. 9.16** | Time class containing a constructor with default arguments. (Part I of 2.)



---

```
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. (Part 2 of 2.)



```
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
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
```

**Fig. 9.17** | Time class member-function definitions including a constructor that takes arguments. (Part I of 4.)



---

```
23
24 // set hour value
25 void Time::setHour( int h )
26 {
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
```

---

**Fig. 9.17** | Time class member-function definitions including a constructor that takes arguments. (Part 2 of 4.)



---

```
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
```

---

**Fig. 9.17** | Time class member-function definitions including a constructor that takes arguments. (Part 3 of 4.)



---

```
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 4 of 4.)



```
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();
```

**Fig. 9.18** | Set and get functions manipulating an object's private data. (Part 1 of 4.)



---

```
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";
35
36 t.setTime( 11, 58, 0 ); // set time
37 incrementMinutes( t, 3 ); // increment t's minute by 3
38 } // end main
39
```

---

**Fig. 9.18** | Set and get functions manipulating an object's private data. (Part 2 of 4.)



---

```
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
```

---

**Fig. 9.18** | Set and get functions manipulating an object's private data. (Part 3 of 4.)



```
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 4 of 4.)



## 9.11 Time Class: Using Set and Get Functions (cont.)

- ▶ 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.
- ▶ 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.
- ▶ **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).



## 9.11 Time Class: Using Set and Get Functions (cont.)

- ▶ 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.
- ▶ 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.



## 9.11 Time Class: Using Set and Get Functions (cont.)

- ▶ Time's *set* and *get* functions are called throughout the class's body.
- ▶ 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.



## 9.11 Time Class: Using Set and Get Functions (cont.)

- ▶ 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 9.3

*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*.



## 9.12 Time Class: A Subtle Trap—Returning a Reference to a private Data Member (cont.)

- ▶ 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 );
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.



```
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
```

**Fig. 9.20** | Time class member-function definitions. (Part 1 of 2.)



---

```
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. (Part 2 of 2.)



```
1 // Fig. 9.21: fig09_21.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
```

**Fig. 9.21** | Returning a reference to a private data member. (Part 1 of 2.)



```
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. (Part 2 of 2.)



## 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**.
- ▶ 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.*



---

```
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;
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.



```
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 << "\n";
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
```

**Fig. 9.24** | Default memberwise assignment. (Part 1 of 2.)



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

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

**Fig. 9.24** | Default memberwise assignment. (Part 2 of 2.)



## 9.13 Default Memberwise Assignment (cont.)

- ▶ 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).*