```
16  // set new Time value using universal time
17  void Time::setTime( int h, int m, int s )
18  {
19     // validate hour, minute and second
20     if ( ( h >= 0 && h < 24 ) && ( m >= 0 && m < 60 ) &&
21        ( s >= 0 && s < 60 ) )
22     {
23        hour = h;
24        minute = m;
25        second = s;
26     } // end if
27     else
28        throw invalid_argument(
29           "hour, minute and/or second was out of range" );
30  } // end function setTime
31
32  // print Time in universal-time format (HH:MM:SS)
33  void Time::printUniversal() const
34  {
35     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
36        << setw( 2 ) << minute << ":" << setw( 2 ) << second;
37  } // end function printUniversal
38
```

Fig. 9.2 | Time class member-function definitions. (Part 2 of 3.)

```
39  // print Time in standard-time format (HH:MM:SS AM or PM)
40  void Time::printStandard() const
41  {
42     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ) << ":"
43        << setfill( '0' ) << setw( 2 ) << minute << ":" << setw( 2 )
44        << second << ( hour < 12 ? " AM" : " PM" );
45  } // end function printStandard
```

Fig. 9.2 | `Time` class member-function definitions. (Part 3 of 3.)

9.2 `Time` Class Case Study (cont.)

- Before C++11, only `static const int` data members (which you saw in Chapter 7) could be initialized where they were declared in the class body.
- For this reason, data members typically should be initialized by the class's constructor as *there is no default initialization for fundamental-type data members.*
- As of C++11, you can now use an *in-class initializer* to initialize any data member where it's declared in the class definition.

9.2 `Time` Class Case Study (cont.)

- Parameterized stream manipulator `setfill` specifies the fill character that is displayed when an integer is output in a field wider than the number of digits in the value.
- The fill characters appear to the *left* of the digits in the number, because the number is *right aligned* by default—for *left aligned* values, the fill characters would appear to the right.
- 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.

Error-Prevention Tip 9.2

Each sticky setting (such as a fill character or floating-point precision) should be restored to its previous setting when it's no longer needed. Failure to do so may result in incorrectly formatted output later in a program. Chapter 13, Stream Input/Output: A Deeper Look, discusses how to reset the fill character and precision.

9.2 `Time` Class Case Study (cont.)

Defining Member Functions Outside the Class Definition; Class Scope

- Even though a member function declared in a class definition may be defined outside that class definition, that member function is still within that class's scope.
- If a member function is defined in the class's body, the compiler attempts to inline calls to the member function.

Performance Tip 9.1

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

Software Engineering Observation 9.2

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

Software Engineering Observation 9.3

Using an object-oriented programming approach often simplifies function calls by reducing the number of parameters. This benefit derives from the fact that encapsulating data members and member functions within a class 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 function calls in non-object-oriented languages. Thus, the calls, the function definitions and the function prototypes are shorter. This improves many aspects of program development.

Error-Prevention Tip 9.3

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

9.2 `Time` Class Case Study (cont.)

Using Class `Time`

- Once class `Time` has been defined, it can be used as a type in object, array, pointer and reference declarations as follows:

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

- Figure 9.3 uses class `Time`.

```
1  // Fig. 9.3: fig09_03.cpp
2  // Program to test class Time.
3  // NOTE: This file must be compiled with Time.cpp.
4  #include <iostream>
5  #include <stdexcept> // for invalid_argument exception class
6  #include "Time.h" // include definition of class Time from Time.h
7  using namespace std;
8
9  int main()
10 {
11    Time t; // instantiate object t of class Time
12
13    // output Time object t's initial values
14    cout << "The initial universal time is ";
15    t.printUniversal(); // 00:00:00
16    cout << "\nThe initial standard time is ";
17    t.printStandard(); // 12:00:00 AM
18
19    t.setTime( 13, 27, 6 ); // change time
20
```

Fig. 9.3 | Program to test class `Time`. (Part 1 of 3.)

```
21      // output Time object t's new values
22      cout << "\n\nUniversal time after setTime is ";
23      t.printUniversal(); // 13:27:06
24      cout << "\nStandard time after setTime is ";
25      t.printStandard(); // 1:27:06 PM
26
27      // attempt to set the time with invalid values
28      try
29      {
30         t.setTime( 99, 99, 99 ); // all values out of range
31      } // end try
32      catch ( invalid_argument &e )
33      {
34         cout << "Exception: " << e.what() << endl;
35      } // end catch
36
37      // output t's values after specifying invalid values
38      cout << "\n\nAfter attempting invalid settings:"
39         << "\nUniversal time: ";
40      t.printUniversal(); // 13:27:06
41      cout << "\nStandard time: ";
42      t.printStandard(); // 1:27:06 PM
43      cout << endl;
44   } // end main
```

Fig. 9.3 | Program to test class `Time`. (Part 2 of 3.)

```
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

Exception: hour, minute and/or second was out of range
```

```
After attempting invalid settings:
Universal time: 13:27:06
Standard time: 1:27:06 PM
```

Fig. 9.3 | Program to test class `Time`. (Part 3 of 3.)