9.13 Using the `this` Pointer (cont.)

Using the `this` Pointer to Enable Cascaded Function Calls

- Another use of the `this` pointer is to enable cascaded member-function calls—that is, invoking multiple functions in the same statement (as in line 12 of Fig. 9.26).
- The program of Figs. 9.24–9.26 modifies class `Time`'s `set` functions `setTime`, `setHour`, `setMinute` and `setSecond` such that each returns a reference to a `Time` object to enable cascaded member-function calls.
- Notice in Fig. 9.25 that the last statement in the body of each of these member functions returns `*this` (lines 23, 34, 45 and 56) into a return type of `Time &`.
- The program of Fig. 9.26 creates `Time` object `t` (line 9), then uses it in *cascaded member-function calls* (lines 12 and 24).

```
1  // Fig. 9.24: Time.h
2  // Cascading member function calls.
3
4  // Time class definition.
5  // Member functions defined in Time.cpp.
6  #ifndef TIME_H
7  #define TIME_H
8
9  class Time
10 {
11 public:
12    explicit Time( int = 0, int = 0, int = 0 ); // default constructor
13
14    // set functions (the Time & return types enable cascading)
15    Time &setTime( int, int, int ); // set hour, minute, second
16    Time &setHour( int ); // set hour
17    Time &setMinute( int ); // set minute
18    Time &setSecond( int ); // set second
19
20    // get functions (normally declared const)
21    unsigned int getHour() const; // return hour
22    unsigned int getMinute() const; // return minute
23    unsigned int getSecond() const; // return second
```

Fig. 9.24 | `Time` class modified to enable cascaded member-function calls. (Part 1 of 2.)

```
24
25      // print functions (normally declared const)
26      void printUniversal() const; // print universal time
27      void printStandard() const; // print standard time
28   private:
29      unsigned int hour; // 0 - 23 (24-hour clock format)
30      unsigned int minute; // 0 - 59
31      unsigned int second; // 0 - 59
32   }; // end class Time
33
34   #endif
```

Fig. 9.24 | `Time` class modified to enable cascaded member-function calls. (Part 2 of 2.)

```
1  // Fig. 9.25: Time.cpp
2  // Time class member-function definitions.
3  #include <iostream>
4  #include <iomanip>
5  #include <stdexcept>
6  #include "Time.h" // Time class definition
7  using namespace std;
8
9  // constructor function to initialize private data;
10 // calls member function setTime to set variables;
11 // default values are 0 (see class definition)
12 Time::Time( int hr, int min, int sec )
13 {
14    setTime( hr, min, sec );
15 } // end Time constructor
16
```

Fig. 9.25 | `Time` class member-function definitions modified to enable cascaded member-function calls. (Part 1 of 5.)

```
17 // set values of hour, minute, and second
18 Time &Time::setTime( int h, int m, int s ) // note Time & return
19 {
20    setHour( h );
21    setMinute( m );
22    setSecond( s );
23    return *this; // enables cascading
24 } // end function setTime
25
26 // set hour value
27 Time &Time::setHour( int h ) // note Time & return
28 {
29    if ( h >= 0 && h < 24 )
30       hour = h;
31    else
32       throw invalid_argument( "hour must be 0-23" );
33
34    return *this; // enables cascading
35 } // end function setHour
36
```

Fig. 9.25 | `Time` class member-function definitions modified to enable cascaded member-function calls. (Part 2 of 5.)

```
37  // set minute value
38  Time &Time::setMinute( int m ) // note Time & return
39  {
40     if ( m >= 0 && m < 60 )
41        minute = m;
42     else
43        throw invalid_argument( "minute must be 0-59" );
44
45     return *this; // enables cascading
46  } // end function setMinute
47
48  // set second value
49  Time &Time::setSecond( int s ) // note Time & return
50  {
51     if ( s >= 0 && s < 60 )
52        second = s;
53     else
54        throw invalid_argument( "second must be 0-59" );
55
56     return *this; // enables cascading
57  } // end function setSecond
```

Fig. 9.25 | `Time` class member-function definitions modified to enable cascaded member-function calls. (Part 3 of 5.)

```
58
59  // get hour value
60  unsigned int Time::getHour() const
61  {
62     return hour;
63  } // end function getHour
64
65  // get minute value
66  unsigned int Time::getMinute() const
67  {
68     return minute;
69  } // end function getMinute
70
71  // get second value
72  unsigned int Time::getSecond() const
73  {
74     return second;
75  } // end function getSecond
76
```

Fig. 9.25 | `Time` class member-function definitions modified to enable cascaded member-function calls. (Part 4 of 5.)

```
77  // print Time in universal-time format (HH:MM:SS)
78  void Time::printUniversal() const
79  {
80     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
81        << setw( 2 ) << minute << ":" << setw( 2 ) << second;
82  } // end function printUniversal
83
84  // print Time in standard-time format (HH:MM:SS AM or PM)
85  void Time::printStandard() const
86  {
87     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
88        << ":" << setfill( '0' ) << setw( 2 ) << minute
89        << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
90  } // end function printStandard
```

Fig. 9.25 | `Time` class member-function definitions modified to enable cascaded member-function calls. (Part 5 of 5.)

```
1  // Fig. 9.26: fig09_26.cpp
2  // Cascading member-function calls with the this pointer.
3  #include <iostream>
4  #include "Time.h" // Time class definition
5  using namespace std;
6
7  int main()
8  {
9     Time t; // create Time object
10
11    // cascaded function calls
12    t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
13
14    // output time in universal and standard formats
15    cout << "Universal time: ";
16    t.printUniversal();
17
18    cout << "\nStandard time: ";
19    t.printStandard();
20
21    cout << "\n\nNew standard time: ";
22
```

Fig. 9.26 | Cascading member-function calls with the `this` pointer. (Part 1 of 2.)

```
23      // cascaded function calls
24      t.setTime( 20, 20, 20 ).printStandard();
25      cout << endl;
26   } // end main
```

```
Universal time: 18:30:22
Standard time: 6:30:22 PM

New standard time: 8:20:20 PM
```

Fig. 9.26 | Cascading member-function calls with the `this` pointer. (Part 2 of 2.)

9.14 `static` Class Members

- In certain cases, only one copy of a variable should be *shared* by *all* objects of a class.
- A `static` data member is used for these and other reasons.
- Such a variable represents "class-wide" information, i.e., data that is shared by all instances and is not specific to any one object of the class.

Performance Tip 9.5

Use `static` data members to save storage when a single copy of the data for all objects of a class will suffice.

9.14 `static` Class Members (cont.)

Scope and Initialization of `static` Data Members

- static data members have *class scope.*
- A static data member must be initialized *exactly* once.
- Fundamental-type `static` data members are initialized by default to `0`.
- Prior to C++11, a `static const` data member of `int` or `enum` type could be initialized in its declaration in the class definition and all other `static` data members had to be defined and initialized *at global namespace scope* (i.e., outside the body of the class definition).
- Again, C++11's in-class initializers also allow you to initialize these variables where they're declared in the class definition.

9.14 `static` Class Members (cont.)

Accessing `static` Data Members

- A class's `private` and `protected` `static` members are normally accessed through the class's `public` member functions or friends.
- *A class's `static` members exist even when no objects of that class exist.*
- To access a `public static` class member when no objects of the class exist, simply prefix the class name and the scope resolution operator (`::`) to the name of the data member.
- To access a private or protected static class member when no objects of the class exist, provide a public `static` member function and call the function by prefixing its name with the class name and scope resolution operator.
- A `static` member function is a service of the *class*, *not* of a specific *object* of the class.

Software Engineering Observation 9.14

A class's `static` data members and `static` member functions exist and can be used even if no objects of that class have been instantiated.