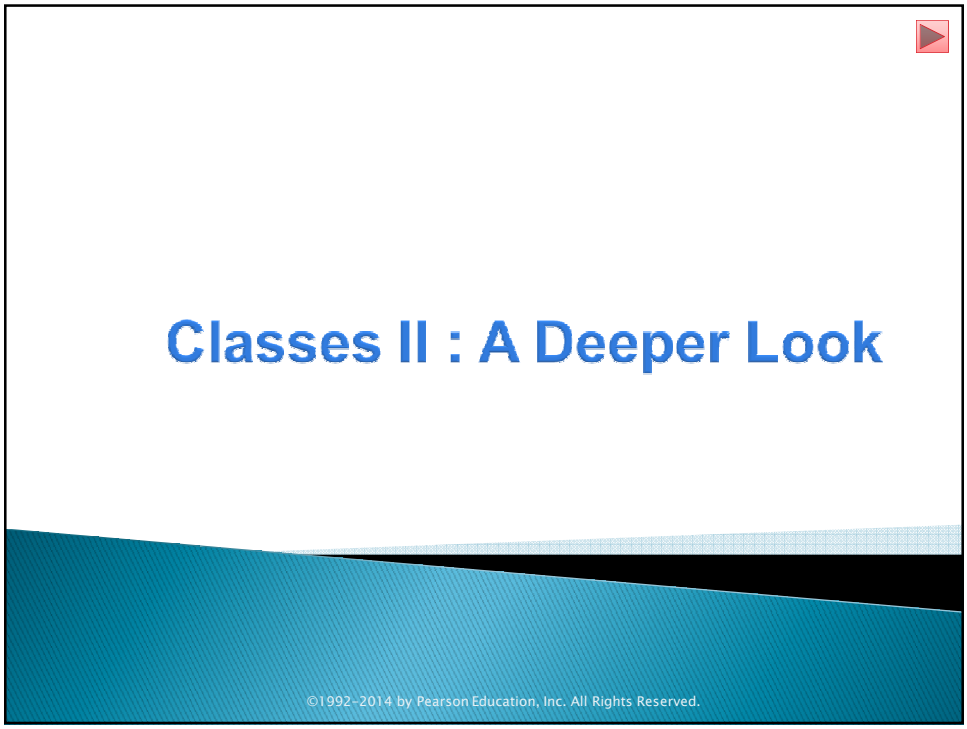
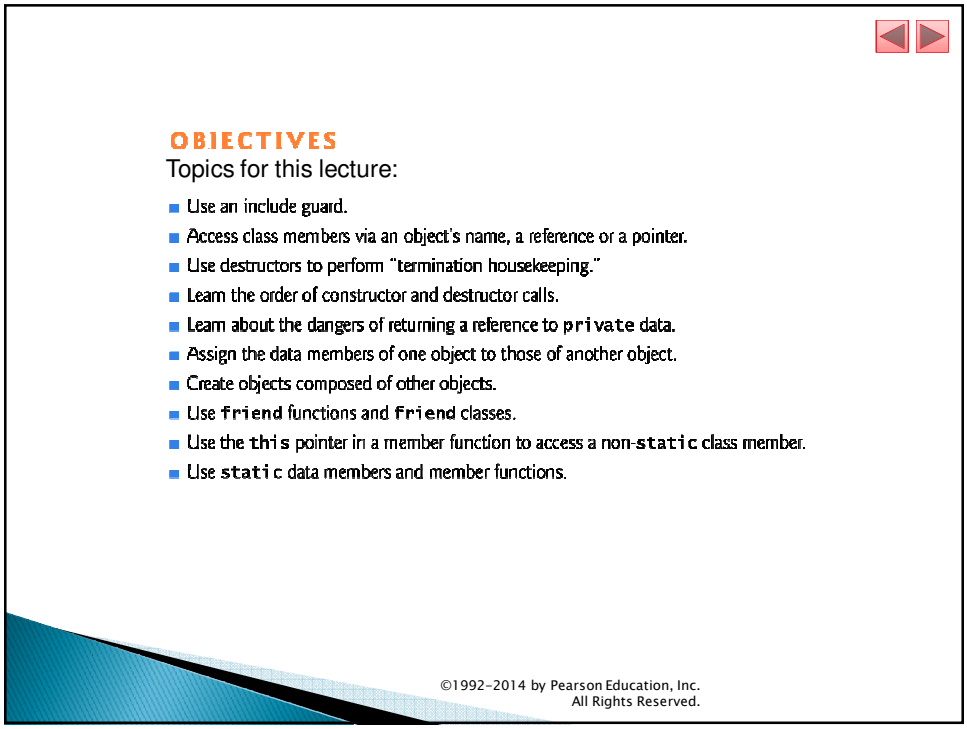

Classes II : A Deeper Look
©1992-2014 by Pearson Education, Inc. All Rights Reserved.


OBJECTIVES
Topics for this lecture:
Use an include guard.
Access class members via an object's name, a reference or a pointer.
Use destructors to perform "termination housekeeping."
Learn the order of constructor and destructor calls.
Learn about the dangers of returning a reference to private data.
Assign the data members of one object to those of another object.
Create objects composed of other objects.
Use friend functions and friend classes.
Use the this pointer in a member function to access a non-static class member.
Use static data members and member functions.
©1992-2014 by Pearson Education, Inc.
All Rights Reserved.

1 Introduction
2 `Time` Class Case Study
3 Class Scope and Accessing Class Members
4 Access Functions and Utility Functions
5 `Time` Class Case Study: Constructors with Default Arguments
6 Destructors
7 When Constructors and Destructors Are Called
8 `Time` Class Case Study: A Subtle Trap—Returning a Reference or a Pointer to a `private` Data Member
9 Default Memberwise Assignment
10 `const` Objects and `const` Member Functions
11 Composition: Objects as Members of Classes
12 `friend` Functions and `friend` Classes
13 Using the `this` Pointer
14 `static` Class Members
15 Wrap-Up

1 Introduction

- Let's take a deeper look at classes.
- Coverage includes:
  - Use of an integrated Time class example to demonstrate several class construction capabilities.
  - Demonstrate how client code can access a class's `public` members via the name of an object, a reference to an object or a pointer to an object.
  - Discuss access functions that can read or write an object's data members.
  - Demonstrate utility functions—`private` member functions that support the operation of the class's `public` member functions.

1 Introduction (cont.)

- Coverage includes (cont.):
  - How default arguments can be used in constructors.
  - Destructors that perform "termination housekeeping" on objects before they're destroyed.
  - The *order* in which constructors and destructors are called.
  - We show that returning a reference or pointer to `private` data *breaks the encapsulation* of a class, allowing client code to directly access an object's data.
  - We use default memberwise assignment to assign an object of a class to another object of the same class.

1 Introduction (cont.)

- Coverage includes (cont.):
  - `const` objects and `const` member functions to prevent modifications of objects and enforce the principle of least privilege.
  - *Composition*—a form of reuse in which a class can have objects of other classes as members.
  - *Friendship* to specify that a nonmember function can also access a class's non-public members—a technique that's often used in operator overloading for performance reasons.
  - `this` pointer, which is an implicit argument in all calls to a class's non-static member functions, allowing them to access the correct object's data members and non-static member functions.

2 `Time` Class Case Study

- Our first example (Fig. 9.1) creates class `Time` and tests the class.

Good Programming Practice 9.1

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

Software Engineering Observation 9.1

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


```
1  // Fig. 9.1: Time.h
2  // Time class definition.
3  // Member functions are defined in Time.cpp
4
5  // prevent multiple inclusions of header
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() const; // print time in universal-time format
16    void printStandard() const; // print time in standard-time format
17 private:
18    unsigned int hour; // 0 - 23 (24-hour clock format)
19    unsigned int minute; // 0 - 59
20    unsigned int second; // 0 - 59
21 }; // end class Time
22
23 #endif
```

Fig. 9.1 | Time class definition.

2 Time Class Case Study (cont.)

- In Fig. 9.1, the class definition is enclosed in the following include guard:

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

  - Prevents the code between `#ifndef` and `#endif` from being included if the name `TIME_H` has been defined.
  - If the header has *not* been included previously in a file, the name `TIME_H` is *defined* by the `#define` directive and the header file statements are included.
  - If the header has been included previously, `TIME_H` is defined already and the header file is not included again.

Error-Prevention Tip 9.1

Use `#ifndef`, `#define` and `#endif` preprocessing directives to form an include guard that prevents headers from being included more than once in a source-code file.

Good Programming Practice 9.2

By convention, use the name of the header in uppercase with the period replaced by an underscore in the `#ifndef` and `#define` preprocessing directives of a header.

2 `Time` Class Case Study (cont.)

Time Class Member Functions

- In Fig. 9.2, the `Time` constructor (lines 11–14) initializes the data members to 0—the universal-time equivalent of 12 AM.
- 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).
- You can define *overloaded constructors* for a class.

```
1  // Fig. 9.2: Time.cpp
2  // Time class member-function definitions.
3  #include <iostream>
4  #include <iomanip>
5  #include <stdexcept> // for invalid_argument exception class
6  #include "Time.h" // include definition of class Time from Time.h
7
8  using namespace std;
9
10 // Time constructor initializes each data member to zero.
11 Time::Time()
12    : hour( 0 ), minute( 0 ), second( 0 )
13 {
14 } // end Time constructor
15
```

Fig. 9.2 | Time class member-function definitions. (Part 1 of 3.)

```
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.)

2 Time Class Case Study (cont.)

- Before C++11, only `static const int` data members 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.

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.

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.

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.)

2 Time Class Case Study (cont.)

Object Size

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

Performance Tip 9.2

Objects contain only data, so objects are much smaller than if they also contained member functions. 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.

3 Class Scope and Accessing Class Members

- A class's data members and member functions belong to that class's scope.
- Nonmember functions are defined at *global namespace scope*, by default.
- 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.

3 Class Scope and Accessing Class Members (cont.)

Class Scope and Block Scope

- If a member function defines a variable with the same name as a variable with class scope, the class-scope variable is *hidden* in the function by the block-scope variable.
  - Such a hidden variable can be accessed by preceding the variable name with the class name followed by the scope resolution operator (`::`).

3 Class Scope and Accessing Class Members (cont.)

Dot (.) and Arrow (->) Member Selection Operators

- 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.

3 Class Scope and Accessing Class Members (cont.)

Accessing public Class Members Through Objects, References and Pointers

- Consider an Account class that has a public `setBalance` member function. Given the following declarations:

```
Account account; // an Account object

// accountRef refers to an Account object
Account &accountRef = account;

// accountPtr points to an Account object
Account *accountPtr = &account;
```

3 Class Scope and Accessing Class Members (cont.)

You can invoke member function setBalance using the dot (.) and arrow (->) member selection operators as follows:

```
// call setBalance via the Account object
account.setBalance( 123.45 );

// call setBalance via a reference to the
Account object
accountRef.setBalance( 123.45 );

// call setBalance via a pointer to the Account
object
accountPtr->setBalance( 123.45 );
```


4 Access Functions and Utility Functions

Access Functions

- Access functions can read or display data.
- A common use for access functions is to test the truth or falsity of conditions—such functions are often called predicate functions.

Utility Functions

- A utility function (also called a helper function) is a `private` member function that supports the operation of the class's other member functions.

5 `Time` Class Case Study: Constructors with Default Arguments

- The program of Figs. 9.4–9.6 enhances class `Time` to demonstrate how arguments are implicitly passed to a constructor.
- The constructor defined in Fig. 9.2 initialized `hour`, `minute` and `second` to 0 (i.e., midnight in universal time).
- Like other functions, constructors can specify *default arguments*.


```
1  // Fig. 9.4: 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
6  #ifndef TIME_H
7  #define TIME_H
8
9  // Time class definition
10 class Time
11 {
12 public:
13    explicit 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. 9.4 | `Time` class containing a constructor with default arguments. (Part 1 of 2.)

```
21     // get functions
22     unsigned int getHour() const; // return hour
23     unsigned int getMinute() const; // return minute
24     unsigned int getSecond() const; // return second
25
26     void printUniversal() const; // output time in universal-time format
27     void printStandard() const; // output time in standard-time format
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. 9.4 | Time class containing a constructor with default arguments. (Part 2 of 2.)

Software Engineering Observation 9.5

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

```
1  // Fig. 9.5: Time.cpp
2  // Member-function definitions for class Time.
3  #include <iostream>
4  #include <iomanip>
5  #include <stdexcept>
6  #include "Time.h" // include definition of class Time from Time.h
7  using namespace std;
8
9  // Time constructor initializes each data member
10 Time::Time( int hour, int minute, int second )
11 {
12    setTime( hour, minute, second ); // validate and set time
13 } // end Time constructor
14
15 // set new Time value using universal time
16 void Time::setTime( int h, int m, int s )
17 {
18    setHour( h ); // set private field hour
19    setMinute( m ); // set private field minute
20    setSecond( s ); // set private field second
21 } // end function setTime
22
```

Fig. 9.5 | Member-function definitions for class `Time`. (Part 1 of 4.)

```
23 // set hour value
24 void Time::setHour( int h )
25 {
26    if ( h >= 0 && h < 24 )
27       hour = h;
28    else
29       throw invalid_argument( "hour must be 0-23" );
30 } // end function setHour
31
32 // set minute value
33 void Time::setMinute( int m )
34 {
35    if ( m >= 0 && m < 60 )
36       minute = m;
37    else
38       throw invalid_argument( "minute must be 0-59" );
39 } // end function setMinute
40
```

Fig. 9.5 | Member-function definitions for class `Time`. (Part 2 of 4.)

```
41 // set second value
42 void Time::setSecond( int s )
43 {
44    if ( s >= 0 && s < 60 )
45       second = s;
46    else
47       throw invalid_argument( "second must be 0-59" );
48 } // end function setSecond
49
50 // return hour value
51 unsigned int Time::getHour() const
52 {
53    return hour;
54 } // end function getHour
55
56 // return minute value
57 unsigned Time::getMinute() const
58 {
59    return minute;
60 } // end function getMinute
61
```

Fig. 9.5 | Member-function definitions for class Time. (Part 3 of 4.)

```
62 // return second value
63 unsigned Time::getSecond() const
64 {
65    return second;
66 } // end function getSecond
67
68 // print Time in universal-time format (HH:MM:SS)
69 void Time::printUniversal() const
70 {
71    cout << setfill( '0' ) << setw( 2 ) << getHour() << ":"
72       << setw( 2 ) << getMinute() << ":" << setw( 2 ) << getSecond();
73 } // end function printUniversal
74
75 // print Time in standard-time format (HH:MM:SS AM or PM)
76 void Time::printStandard() const
77 {
78    cout << ( ( getHour() == 0 || getHour() == 12 ) ? 12 : getHour() % 12 )
79       << ":" << setfill( '0' ) << setw( 2 ) << getMinute()
80       << ":" << setw( 2 ) << getSecond() << ( hour < 12 ? " AM" : " PM" );
81 } // end function printStandard
```

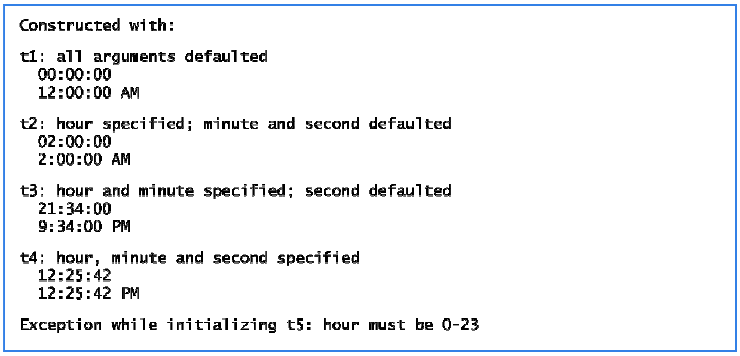
Fig. 9.5 | Member-function definitions for class Time. (Part 4 of 4.)

```
1  // Fig. 9.6: fig09_06.cpp
2  // Constructor with default arguments.
3  #include <iostream>
4  #include <stdexcept>
5  #include "Time.h" // include definition of class Time from Time.h
6  using namespace std;
7
8  int main()
9  {
10    Time t1; // all arguments defaulted
11    Time t2( 2 ); // hour specified; minute and second defaulted
12    Time t3( 21, 34 ); // hour and minute specified; second defaulted
13    Time t4( 12, 25, 42 ); // hour, minute and second specified
14
15    cout << "Constructed with:\n\nt1: all arguments defaulted\n  ";
16    t1.printUniversal(); // 00:00:00
17    cout << "\n  ";
18    t1.printStandard(); // 12:00:00 AM
19
20    cout << "\n\nt2: hour specified; minute and second defaulted\n  ";
21    t2.printUniversal(); // 02:00:00
22    cout << "\n  ";
23    t2.printStandard(); // 2:00:00 AM
24
```

Fig. 9.6 | Constructor with default arguments. (Part 1 of 3.)

```
25    cout << "\n\nt3: hour and minute specified; second defaulted\n  ";
26    t3.printUniversal(); // 21:34:00
27    cout << "\n  ";
28    t3.printStandard(); // 9:34:00 PM
29
30    cout << "\n\nt4: hour, minute and second specified\n  ";
31    t4.printUniversal(); // 12:25:42
32    cout << "\n  ";
33    t4.printStandard(); // 12:25:42 PM
34
35    // attempt to initialize t6 with invalid values
36    try
37    {
38       Time t5( 27, 74, 99 ); // all bad values specified
39    } // end try
40    catch ( invalid_argument &e )
41    {
42       cerr << "\n\nException while initializing t5: " << e.what() << endl;
43    } // end catch
44 } // end main
```

Fig. 9.6 | 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

Exception while initializing t5: hour must be 0-23
```

Fig. 9.6 | Constructor with default arguments. (Part 3 of 3.)

5 `Time` Class Case Study: Constructors with Default Arguments (cont.)

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

- `Time`'s *set* and *get* functions are called throughout the class's body.
- In each case, these functions could have accessed the class's `private` data directly.

5 `Time` Class Case Study: Constructors with Default Arguments (cont.)

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

- Consider changing the representation of the time from three `int` values (requiring 12 bytes of memory on systems with four-byte `int`s) 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.
  - No need to modify the bodies of the other functions.

5 `Time` Class Case Study: Constructors with Default Arguments (cont.)

- Designing the class in this manner reduces the likelihood of programming errors when altering the class's implementation.
- Duplicating statements in multiple functions or constructors makes changing the class's internal data representation more difficult.

Software Engineering Observation 9.6

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.

Common Programming Error 9.1

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 initialized. Using data members before they have been properly initialized can cause logic errors.

5 Time Class Case Study: Constructors with Default Arguments (cont.)

C++11: Using List Initializers to Call Constructors

- C++11 now provides a uniform initialization syntax called list initializers that can be used to initialize any variable. Lines 11–13 of Fig. 9.6 can be written using list initializers as follows:

```
Time t2{ 2 }; // hour specified; minute and second defaulted
Time t3{ 21, 34 }; // hour and minute specified; second defaulted
Time t4{ 12, 25, 42 }; // hour, minute and second specified
```

or

```
Time t2 = { 2 }; // hour specified; minute and second defaulted
Time t3 = { 21, 34 }; // hour and minute specified; second defaulted
Time t4 = { 12, 25, 42 }; // hour, minute and second specified
```

The form without the = is preferred.

5 Time Class Case Study: Constructors with Default Arguments (cont.)

C++11: Overloaded Constructors and Delegating Constructors

- A class's constructors and member functions can also be overloaded.
- Overloaded constructors typically allow objects to be initialized with different types and/or numbers of arguments.
- To overload a constructor, provide in the class definition a prototype for each version of the constructor, and provide a separate constructor definition for each overloaded version.
  - This also applies to the class's member functions.

5 Time Class Case Study: Constructors with Default Arguments (cont.)

- In Figs. 9.4–9.6, the Time constructor with three parameters had a default argument for each parameter. We could have defined that constructor instead as four overloaded constructors with the following prototypes:

```
Time(); // default hour, minute and second to 0
Time( int ); // initialize hour; default minute and second to 0
Time( int, int); // initialize hour and minute; default second to 0
Time( int, int, int ); // initialize hour, minute and second
```

- C++11 now allows constructors to call other constructors in the same class.
- The calling constructor is known as a delegating constructor—it *delegates* its work to another constructor.

5 Time Class Case Study: Constructors with Default Arguments (cont.)

- The first three of the four `Time` constructors declared on the previous slide can delegate work to one with three `int` arguments, passing 0 as the default value for the extra parameters.
- Use a member initializer with the name of the class as follows:

```
Time::Time()
   Time( 0, 0, 0 ) //delegate to Time( int, int, int )
{
} // end constructor with no arguments

Time::Time( int hour )
   Time( hour, 0, 0 ) //delegate to Time( int, int, int )
{
} // end constructor with one argument
Time::Time( int hour, int minute )
   Time( hour, minute, 0 ) //delegate to Time( int, int, int )
{
} // end constructor with two arguments
```