```
// return second value
unsigned Time::getSecond() const
{
   return second;
} // end function getSecond

// print Time in universal-time format (HH:MM:SS)
void Time::printUniversal() const
{
   cout << setfill( '0' ) << setw( 2 ) << getHour() << ":"
      << setw( 2 ) << getMinute() << ":" << setw( 2 ) << getSecond();
} // end function printUniversal

// print Time in standard-time format (HH:MM:SS AM or PM)
void Time::printStandard() const
{
   cout << ( ( getHour() == 0 || getHour() == 12 ) ? 12 : getHour() % 12 )
      << ":" << setfill( '0' ) << setw( 2 ) << getMinute()
      << ":" << setw( 2 ) << getSecond() << ( hour < 12 ? " AM" : " PM" );
} // end function printStandard
```

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

```cpp
// Fig. 9.6: fig09_06.cpp
// Constructor with default arguments.
#include <iostream>
#include <stdexcept>
#include "Time.h" // include definition of class Time from Time.h
using namespace std;

int main()
{
   Time t1; // all arguments defaulted
   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

   cout << "Constructed with:\n\nt1: all arguments defaulted\n  ";
   t1.printUniversal(); // 00:00:00
   cout << "\n  ";
   t1.printStandard(); // 12:00:00 AM

   cout << "\n\nt2: hour specified; minute and second defaulted\n  ";
   t2.printUniversal(); // 02:00:00
   cout << "\n  ";
   t2.printStandard(); // 2:00:00 AM

```

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

```
    cout << "\n\nt3: hour and minute specified; second defaulted\n  ";
    t3.printUniversal(); // 21:34:00
    cout << "\n  ";
    t3.printStandard(); // 9:34:00 PM

    cout << "\n\nt4: hour, minute and second specified\n  ";
    t4.printUniversal(); // 12:25:42
    cout << "\n  ";
    t4.printStandard(); // 12:25:42 PM

    // attempt to initialize t6 with invalid values
    try
    {
       Time t5( 27, 74, 99 ); // all bad values specified
    } // end try
    catch ( invalid_argument &e )
    {
       cerr << "\n\nException while initializing t5: " << e.what() << endl;
    } // end catch
} // 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.)

9.5 `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.
- 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.

9.5 `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.

9.5 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.

9.5 `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.

9.6 `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.

9.5 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
```

9.6 Destructors

- The name of the destructor for a class is the tilde character (~) followed by the class name.
- Called *implicitly* when an object is destroyed.
- *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.
- Receives no parameters and returns no value.
- May not specify a return type—not even `void`.
- A class has *one destructor*.
- A destructor must be `public`.
- If you do not *explicitly* define a destructor, the compiler defines an "empty" destructor.

9.7 When Constructors and Destructors Are Called

- Constructors and destructors are called implicitly.
- 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.

9.7 When Constructors and Destructors Are Called

Constructors and Destructors for Objects in Global Scope

- Constructors are called for objects defined in global scope (also called global namespace scope) *before* any other function (including `main`) in that program 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 local objects.
- Function `abort` performs similarly to function `exit` but forces the program to terminate *immediately*, without allowing the destructors of any objects to be called.