10 Classes: A Deeper Look, Part 2

*But what, to serve our private ends,
Forbids the cheating of our friends?*
—Charles Churchill

Instead of this absurd division into sexes they ought to class people as static and dynamic.
—Evelyn Waugh

Have no friends not equal to yourself.
—Confucius

Objectives

In this chapter you'll learn:

- To specify `const` (constant) objects and `const` member functions.
- To create objects composed of other objects.
- To use `friend` functions and `friend` classes.
- To use the `this` pointer.
- To use `static` data members and member functions.
- The concept of a container class.
- The notion of iterator classes that walk through the elements of container classes.
- To use proxy classes to hide implementation details from a class's clients.

Outline

10.1 Introduction

In this chapter, we continue our study of classes and data abstraction with several more advanced topics. We use `const` objects and `const` member functions to prevent modifications of objects and enforce the principle of least privilege. We discuss composition—a form of reuse in which a class can have objects of other classes as members. Next, we introduce friendship, which enables a class designer to specify nonmember functions that can access a class's non-`public` members—a technique that is often used in operator overloading (Chapter 11) for performance reasons. We discuss a special pointer (called `this`), which is an implicit argument to each of a class's non-`static` member functions. It allows those member functions to access the correct object's data members and other non-`static` member functions. Finally, we motivate the need for `static` class members and show how to use `static` data members and member functions in your own classes.

10.2 `const` (Constant) Objects and `const` Member Functions

Let's see how the principle of least privilege applies to objects. Some objects need to be modifiable and some do not. You may use keyword `const` to specify that an object is not modifiable and that any attempt to modify the object should result in a compilation error. The statement

```
    const Time noon( 12, 0, 0 );
```

declares a `const` object `noon` of class `Time` and initializes it to 12 noon.

Software Engineering Observation 10.1

Attempts to modify a `const` *object are caught at compile time rather than causing execution-time errors.*

Performance Tip 10.1

Declaring variables and objects `const` *when appropriate can improve performance—compilers can perform certain optimizations on constants that cannot be performed on variables.*

C++ disallows member function calls for `const` objects unless the member functions themselves are also declared `const`. This is true even for *get* member functions that do not modify the object.

A member function is specified as `const` *both* in its prototype (Fig. 10.1; lines 19–24) and in its definition (Fig. 10.2; lines 43, 49, 55 and 61) by inserting the keyword `const` after the function's parameter list and, in the case of the function definition, before the left brace that begins the function body.

Common Programming Error 10.1

Defining as `const` a member function that modifies a data member of the object is a compilation error.

Common Programming Error 10.2

Defining as `const` a member function that calls a non-`const` member function of the class on the same object is a compilation error.

Common Programming Error 10.3

Invoking a non-`const` member function on a `const` object is a compilation error.

Software Engineering Observation 10.2

A `const` member function can be overloaded with a non-`const` version. The compiler chooses which overloaded member function to use based on the object on which the function is invoked. If the object is `const`, the compiler uses the `const` version. If the object is not `const`, the compiler uses the non-`const` version.

An interesting problem arises for constructors and destructors, each of which typically modifies objects. A constructor must be allowed to modify an object so that the object can be initialized properly. A destructor must be able to perform its termination housekeeping chores before an object's memory is reclaimed by the system.

Common Programming Error 10.4

Attempting to declare a constructor or destructor `const` is a compilation error.

Defining and Using const *Member Functions*

The program of Figs. 10.1–10.3 modifies class `Time` of Figs. 9.8–9.9 by making its *get* functions and `printUniversal` function `const`. In the header file `Time.h` (Fig. 10.1), lines 19–21 and 24 now include keyword `const` after each function's parameter list. The corresponding definition of each function in Fig. 10.2 (lines 43, 49, 55 and 61, respectively) also specifies keyword `const` after each function's parameter list.

```
1  // Fig. 10.1: Time.h
2  // Time class definition with const member functions.
3  // Member functions defined in Time.cpp.
4  #ifndef TIME_H
5  #define TIME_H
```

Fig. 10.1 | `Time` class definition with `const` member functions. (Part 1 of 2.)

```cpp

class Time
{
public:
   Time( int = 0, int = 0, int = 0 ); // default constructor

   // set functions
   void setTime( int, int, int ); // set time
   void setHour( int ); // set hour
   void setMinute( int ); // set minute
   void setSecond( int ); // set second

   // get functions (normally declared const)
   int getHour() const; // return hour
   int getMinute() const; // return minute
   int getSecond() const; // return second

   // print functions (normally declared const)
   void printUniversal() const; // print universal time
   void printStandard(); // print standard time (should be const)
private:
   int hour; // 0 - 23 (24-hour clock format)
   int minute; // 0 - 59
   int second; // 0 - 59
}; // end class Time

#endif
```

Fig. 10.1 | Time class definition with const member functions. (Part 2 of 2.)

```cpp
// Fig. 10.2: Time.cpp
// Time class member-function definitions.
#include <iostream>
#include <iomanip>
#include "Time.h" // include definition of class Time
using namespace std;

// constructor function to initialize private data;
// calls member function setTime to set variables;
// default values are 0 (see class definition)
Time::Time( int hour, int minute, int second )
{
   setTime( hour, minute, second );
} // end Time constructor

// set hour, minute and second values
void Time::setTime( int hour, int minute, int second )
{
   setHour( hour );
   setMinute( minute );
   setSecond( second );
} // end function setTime
```

Fig. 10.2 | Time class member-function definitions. (Part 1 of 2.)

```

// set hour value
void Time::setHour( int h )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
} // end function setHour

// set minute value
void Time::setMinute( int m )
{
   minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
} // end function setMinute

// set second value
void Time::setSecond( int s )
{
   second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
} // end function setSecond

// return hour value
int Time::getHour() const // get functions should be const
{
   return hour;
} // end function getHour

// return minute value
int Time::getMinute() const
{
   return minute;
} // end function getMinute

// return second value
int 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 ) << hour << ":"
      << setw( 2 ) << minute << ":" << setw( 2 ) << second;
} // end function printUniversal

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

Fig. 10.2 | Time class member-function definitions. (Part 2 of 2.)

Figure 10.3 instantiates two `Time` objects—non-`const` object `wakeUp` (line 7) and `const` object `noon` (line 8). The program attempts to invoke non-`const` member functions `setHour` (line 13) and `printStandard` (line 20) on the `const` object `noon`. In each case, the compiler generates an error message. The program also illustrates the three other member-function-call combinations on objects—a non-`const` member function on a non-`const` object (line 11), a `const` member function on a non-`const` object (line 15) and a `const` member function on a `const` object (lines 17–18). The error messages generated for non-`const` member functions called on a `const` object are shown in the output window.

```
 1  // Fig. 10.3: fig10_03.cpp
 2  // Attempting to access a const object with non-const member functions.
 3  #include "Time.h" // include Time class definition
 4
 5  int main()
 6  {
 7     Time wakeUp( 6, 45, 0 ); // non-constant object
 8     const Time noon( 12, 0, 0 ); // constant object
 9
10                             // OBJECT      MEMBER FUNCTION
11     wakeUp.setHour( 18 );   // non-const   non-const
12
13     noon.setHour( 12 );     // const       non-const
14
15     wakeUp.getHour();       // non-const   const
16
17     noon.getMinute();       // const       const
18     noon.printUniversal(); // const       const
19
20     noon.printStandard();  // const       non-const
21  } // end main
```

Microsoft Visual C++ compiler error messages:

```
C:\cpphtp7_examples\ch10\Fig10_01_03\fig10_03.cpp(13) : error C2662:
   'Time::setHour' : cannot convert 'this' pointer from 'const Time' to
   'Time &'
         Conversion loses qualifiers
C:\cpphtp7_examples\ch10\Fig10_01_03\fig10_03.cpp(20) : error C2662:
   'Time::printStandard' : cannot convert 'this' pointer from 'const Time' to
   'Time &'
         Conversion loses qualifiers
```

GNU C++ compiler error messages:

```
fig10_03.cpp:13: error: passing 'const Time' as 'this' argument of
   'void Time::setHour(int)' discards qualifiers
fig10_03.cpp:20: error: passing 'const Time' as 'this' argument of
   'void Time::printStandard()' discards qualifiers
```

Fig. 10.3 | `const` objects and `const` member functions.

A constructor must be a non-`const` member function (Fig. 10.2, lines 11–14), but it can still be used to initialize a `const` object (Fig. 10.3, line 8). The `Time` constructor's definition (Fig. 10.2, lines 11–14) shows that it calls another non-`const` member function—`setTime` (lines 17–22)—to perform the initialization of a `Time` object. Invoking a non-`const` member function from the constructor call as part of the initialization of a `const` object is allowed. The "`const`ness" of a `const` object is enforced from the time the constructor completes initialization of the object until that object's destructor is called.

Also, line 20 in Fig. 10.3 generates a compilation error even though member function `printStandard` of class `Time` does not modify the object on which it's invoked. The fact that a member function does not modify an object is not sufficient to indicate that the function is constant function—the function must *explicitly* be declared `const`.

Initializing a `const` Data Member with a Member Initializer

The program of Figs. 10.4–10.6 introduces using **member initializer syntax**. All data members *can* be initialized using member initializer syntax, but `const` data members and data members that are references *must* be initialized using member initializers. Later in this chapter, we'll see that member objects must be initialized this way as well.

```
// Fig. 10.4: Increment.h
// Definition of class Increment.
#ifndef INCREMENT_H
#define INCREMENT_H

class Increment
{
public:
   Increment( int c = 0, int i = 1 ); // default constructor

   // function addIncrement definition
   void addIncrement()
   {
      count += increment;
   } // end function addIncrement

   void print() const; // prints count and increment
private:
   int count;
   const int increment; // const data member
}; // end class Increment

#endif
```

Fig. 10.4 | `Increment` class definition containing non-`const` data member `count` and `const` data member `increment`.

```
// Fig. 10.5: Increment.cpp
// Member-function definitions for class Increment demonstrate using a
// member initializer to initialize a constant of a built-in data type.
```

Fig. 10.5 | Member initializer used to initialize a constant of a built-in data type. (Part 1 of 2.)

```
 4  #include <iostream>
 5  #include "Increment.h" // include definition of class Increment
 6  using namespace std;
 7
 8  // constructor
 9  Increment::Increment( int c, int i )
10     : count( c ), // initializer for non-const member
11       increment( i ) // required initializer for const member
12  {
13     // empty body
14  } // end constructor Increment
15
16  // print count and increment values
17  void Increment::print() const
18  {
19     cout << "count = " << count << ", increment = " << increment << endl;
20  } // end function print
```

Fig. 10.5 | Member initializer used to initialize a constant of a built-in data type. (Part 2 of 2.)

```
// Fig. 10.6: fig10_06.cpp
// Program to test class Increment.
#include <iostream>
#include "Increment.h" // include definition of class Increment
using namespace std;

int main()
{
   Increment value( 10, 5 );

   cout << "Before incrementing: ";
   value.print();

   for ( int j = 1; j <= 3; j++ )
   {
      value.addIncrement();
      cout << "After increment " << j << ": ";
      value.print();
   } // end for
} // end main
```

```
Before incrementing: count = 10, increment = 5
After increment 1: count = 15, increment = 5
After increment 2: count = 20, increment = 5
After increment 3: count = 25, increment = 5
```

Fig. 10.6 | Invoking an `Increment` object's `print` and `addIncrement` member functions.

The constructor definition (Fig. 10.5, lines 9–14) uses a **member initializer list** to initialize class `Increment`'s data members—non-`const` integer `count` and `const` integer `increment` (declared in lines 19–20 of Fig. 10.4). Member initializers appear between a constructor's parameter list and the left brace that begins the constructor's body. The member initializer list (Fig. 10.5, lines 10–11) is separated from the parameter list with a colon (`:`). Each member initializer consists of the data member name followed by parentheses containing the member's initial value. In this example, `count` is initialized with the value of constructor parameter `c` and `increment` is initialized with the value of constructor parameter `i`. Multiple member initializers are separated by commas. Also, the member initializer list executes before the body of the constructor executes.

Software Engineering Observation 10.3

A `const` object cannot be modified by assignment, so it must be initialized. When a data member of a class is declared `const`, a member initializer must be used to provide the constructor with the initial value of the data member for an object of the class. The same is true for references.

Erroneously Attempting to Initialize a `const` Data Member with an Assignment

The program of Figs. 10.7–10.9 illustrates the compilation errors caused by attempting to initialize `const` data member `increment` with an assignment statement (Fig. 10.8, line 12) in the `Increment` constructor's body rather than with a member initializer. Line 11 of Fig. 10.8 does not generate a compilation error, because `count` is not declared `const`.

Common Programming Error 10.5

Not providing a member initializer for a `const` data member is a compilation error.

Software Engineering Observation 10.4

Constant data members (`const` objects and `const` variables) and data members declared as references must be initialized with member initializer syntax; assignments for these types of data in the constructor body are not allowed.

```cpp
// Fig. 10.7: Increment.h
// Definition of class Increment.
#ifndef INCREMENT_H
#define INCREMENT_H

class Increment
{
public:
   Increment( int c = 0, int i = 1 ); // default constructor

   // function addIncrement definition
   void addIncrement()
   {
      count += increment;
   } // end function addIncrement

   void print() const; // prints count and increment
private:
   int count;
```

Fig. 10.7 | `Increment` class definition containing non-`const` data member `count` and `const` data member `increment`. (Part 1 of 2.)

```
   const int increment; // const data member
}; // end class Increment

#endif
```

Fig. 10.7 | Increment class definition containing non-const data member count and const data member increment. (Part 2 of 2.)

```
// Fig. 10.8: Increment.cpp
// Erroneous attempt to initialize a constant of a built-in data
// type by assignment.
#include <iostream>
#include "Increment.h" // include definition of class Increment
using namespace std;

// constructor; constant member 'increment' is not initialized
Increment::Increment( int c, int i )
{
   count = c; // allowed because count is not constant
   increment = i; // ERROR: Cannot modify a const object
} // end constructor Increment

// print count and increment values
void Increment::print() const
{
   cout << "count = " << count << ", increment = " << increment << endl;
} // end function print
```

Fig. 10.8 | Erroneous attempt to initialize a constant of a built-in data type by assignment.

```
// Fig. 10.9: fig10_09.cpp
// Program to test class Increment.
#include <iostream>
#include "Increment.h" // include definition of class Increment
using namespace std;

int main()
{
   Increment value( 10, 5 );

   cout << "Before incrementing: ";
   value.print();

   for ( int j = 1; j <= 3; j++ )
   {
      value.addIncrement();
      cout << "After increment " << j << ": ";
      value.print();
   } // end for
} // end main
```

Fig. 10.9 | Program to test class Increment generates compilation errors. (Part 1 of 2.)

Microsoft Visual C++ compiler error messages:

```
C:\cpphtp7_examples\ch10\Fig10_07_09\Increment.cpp(10) : error C2758:
   'Increment::increment' : must be initialized in constructor base/member
   initializer list
       C:\cpphtp7_examples\ch10\Fig10_07_09\increment.h(20) : see
          declaration of 'Increment::increment'
C:\cpphtp7_examples\ch10\Fig10_07_09\Increment.cpp(12) : error C2166:
   l-value specifies const object
```

GNU C++ compiler error messages:

```
Increment.cpp:9: error: uninitialized member 'Increment::increment' with
   'const' type 'const int'
Increment.cpp:12: error: assignment of read-only data-member
   'Increment::increment'
```

Fig. 10.9 | Program to test class `Increment` generates compilation errors. (Part 2 of 2.)

Function `print` (Fig. 10.8, lines 16–19) is declared `const`. It might seem strange to label this function `const`, because a program probably will never have a `const Increment` object. However, it's possible that a program will have a `const` reference to an `Increment` object or a pointer to `const` that points to an `Increment` object. Typically, this occurs when objects of class `Increment` are passed to functions or returned from functions. In these cases, only class `Increment`'s `const` member functions can be called through the reference or pointer. Thus, it's reasonable to declare function `print` as `const`—doing so prevents errors in these situations where an `Increment` object is treated as a `const` object.

Error-Prevention Tip 10.1

Declare as `const` all of a class's member functions that do not modify the object in which they operate. Occasionally this may seem inappropriate, because you'll have no intention of creating `const` objects of that class or accessing objects of that class through `const` references or pointers to `const`. Declaring such member functions `const` does offer a benefit, though. If the member function is inadvertently written to modify the object, the compiler will issue an error message.

10.3 Composition: Objects as Members of Classes

An `AlarmClock` object needs to know when it's supposed to sound its alarm, so why not include a `Time` object as a member of the `AlarmClock` class? Such a capability is called **composition** and is sometimes referred to as a ***has-a* relationship**—a class can have objects of other classes as members.

Software Engineering Observation 10.5

A common form of software reusability is composition, in which a class has objects of other classes as members.

When an object is created, its constructor is called automatically. Previously, we saw how to pass arguments to the constructor of an object we created in `main`. This section shows how an object's constructor can pass arguments to member-object constructors via member initializers.

Software Engineering Observation 10.6

Member objects are constructed in the order in which they're declared in the class definition (not in the order they're listed in the constructor's member initializer list) and before their enclosing class objects (sometimes called ***host objects****) are constructed.*

The next program uses classes `Date` (Figs. 10.10–10.11) and `Employee` (Figs. 10.12–10.13) to demonstrate composition. Class `Employee`'s definition (Fig. 10.12) contains `private` data members `firstName`, `lastName`, `birthDate` and `hireDate`. Members `birthDate` and `hireDate` are `const` objects of class `Date`, which contains `private` data members `month`, `day` and `year`. The `Employee` constructor's header (Fig. 10.13, lines 10–11) specifies that the constructor has four parameters (`first`, `last`, `dateOfBirth` and `dateOfHire`). The first two parameters are passed via member initializers to the `string` class constructor. The last two are passed via member initializers to the `Date` class constructor.

```
// Fig. 10.10: Date.h
// Date class definition; Member functions defined in Date.cpp
#ifndef DATE_H
#define DATE_H

class Date
{
public:
   static const int monthsPerYear = 12; // number of months in a year
   Date( int = 1, int = 1, int = 1900 ); // default constructor
   void print() const; // print date in month/day/year format
   ~Date(); // provided to confirm destruction order
private:
   int month; // 1-12 (January-December)
   int day; // 1-31 based on month
   int year; // any year

   // utility function to check if day is proper for month and year
   int checkDay( int ) const;
}; // end class Date

#endif
```

Fig. 10.10 | `Date` class definition.

```
// Fig. 10.11: Date.cpp
// Date class member-function definitions.
#include <iostream>
#include "Date.h" // include Date class definition
using namespace std;

// constructor confirms proper value for month; calls
// utility function checkDay to confirm proper value for day
```

Fig. 10.11 | `Date` class member-function definitions. (Part 1 of 2.)

```
Date::Date( int mn, int dy, int yr )
{
   if ( mn > 0 && mn <= monthsPerYear ) // validate the month
      month = mn;
   else
   {
      month = 1; // invalid month set to 1
      cout << "Invalid month (" << mn << ") set to 1.\n";
   } // end else

   year = yr; // could validate yr
   day = checkDay( dy ); // validate the day

   // output Date object to show when its constructor is called
   cout << "Date object constructor for date ";
   print();
   cout << endl;
} // end Date constructor

// print Date object in form month/day/year
void Date::print() const
{
   cout << month << '/' << day << '/' << year;
} // end function print

// output Date object to show when its destructor is called
Date::~Date()
{
   cout << "Date object destructor for date ";
   print();
   cout << endl;
} // end ~Date destructor

// utility function to confirm proper day value based on
// month and year; handles leap years, too
int Date::checkDay( int testDay ) const
{
   static const int daysPerMonth[ monthsPerYear + 1 ] =
      { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

   // determine whether testDay is valid for specified month
   if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
      return testDay;

   // February 29 check for leap year
   if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
      ( year % 4 == 0 && year % 100 != 0 ) ) )
      return testDay;

   cout << "Invalid day (" << testDay << ") set to 1.\n";
   return 1; // leave object in consistent state if bad value
} // end function checkDay
```

Fig. 10.11 | `Date` class member-function definitions. (Part 2 of 2.)

```cpp
// Fig. 10.12: Employee.h
// Employee class definition showing composition.
// Member functions defined in Employee.cpp.
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>
#include "Date.h" // include Date class definition
using namespace std;

class Employee
{
public:
   Employee( const string &, const string &,
      const Date &, const Date & );
   void print() const;
   ~Employee(); // provided to confirm destruction order
private:
   string firstName; // composition: member object
   string lastName; // composition: member object
   const Date birthDate; // composition: member object
   const Date hireDate; // composition: member object
}; // end class Employee

#endif
```

Fig. 10.12 | Employee class definition showing composition.

```cpp
// Fig. 10.13: Employee.cpp
// Employee class member-function definitions.
#include <iostream>
#include "Employee.h" // Employee class definition
#include "Date.h" // Date class definition
using namespace std;

// constructor uses member initializer list to pass initializer
// values to constructors of member objects
Employee::Employee( const string &first, const string &last,
   const Date &dateOfBirth, const Date &dateOfHire )
   : firstName( first ), // initialize firstName
     lastName( last ), // initialize lastName
     birthDate( dateOfBirth ), // initialize birthDate
     hireDate( dateOfHire ) // initialize hireDate
{
   // output Employee object to show when constructor is called
   cout << "Employee object constructor: "
      << firstName << ' ' << lastName << endl;
} // end Employee constructor

```

Fig. 10.13 | Employee class member-function definitions, including constructor with a member initializer list. (Part 1 of 2.)

```
// print Employee object
void Employee::print() const
{
   cout << lastName << ", " << firstName << "  Hired: ";
   hireDate.print();
   cout << "  Birthday: ";
   birthDate.print();
   cout << endl;
} // end function print

// output Employee object to show when its destructor is called
Employee::~Employee()
{
   cout << "Employee object destructor: "
      << lastName << ", " << firstName << endl;
} // end ~Employee destructor
```

Fig. 10.13 | `Employee` class member-function definitions, including constructor with a member initializer list. (Part 2 of 2.)

`Employee` Constructor's Member Initializer List

The colon (`:`) following the constructor's header (Fig. 10.13, line 12) begins the member initializer list. The member initializers specify the `Employee` constructor parameters being passed to the constructors of the `string` and `Date` data members. Parameters `first`, `last`, `dateOfBirth` and `dateOfHire` are passed to the constructors for objects `firstName`'s (Fig. 10.13, line 12), `lastName` (Fig. 10.13, line 13), `birthDate` (Fig. 10.13, line 14) and `hireDate` (Fig. 10.13, line 15), respectively. Again, member initializers are separated by commas.

`Date` Class's Default Copy Constructor

As you study class `Date` (Fig. 10.10), notice that the class does not provide a constructor that receives a parameter of type `Date`. So, why can the `Employee` constructor's member initializer list initialize the `birthDate` and `hireDate` objects by passing `Date` object's to their `Date` constructors? As we mentioned in Chapter 9, the compiler provides each class with a default copy constructor that copies each data member of the constructor's argument object into the corresponding member of the object being initialized. Chapter 11 discusses how you can define customized copy constructors.

Testing Classes `Date` and `Employee`

Figure 10.14 creates two `Date` objects (lines 9–10) and passes them as arguments to the constructor of the `Employee` object created in line 11. Line 14 outputs the `Employee` object's data. When each `Date` object is created in lines 9–10, the `Date` constructor defined in lines 9–26 of Fig. 10.11 displays a line of output to show that the constructor was called (see the first two lines of the sample output). [*Note:* Line 11 of Fig. 10.14 causes two additional `Date` constructor calls that do not appear in the program's output. When each of the `Employee`'s `Date` member object's is initialized in the `Employee` constructor's member initializer list (Fig. 10.13, lines 14–15), the default copy constructor for class `Date` is called. Since this constructor is defined implicitly by the compiler, it does not contain any output statements to demonstrate when it's called.]

```cpp
 1  // Fig. 10.14: fig10_14.cpp
 2  // Demonstrating composition--an object with member objects.
 3  #include <iostream>
 4  #include "Employee.h" // Employee class definition
 5  using namespace std;
 6
 7  int main()
 8  {
 9     Date birth( 7, 24, 1949 );
10     Date hire( 3, 12, 1988 );
11     Employee manager( "Bob", "Blue", birth, hire );
12
13     cout << endl;
14     manager.print();
15
16     cout << "\nTest Date constructor with invalid values:\n";
17     Date lastDayOff( 14, 35, 1994 ); // invalid month and day
18     cout << endl;
19  } // end main
```

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Employee object constructor: Bob Blue

Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949

Test Date constructor with invalid values:
Invalid month (14) set to 1.
Invalid day (35) set to 1.
Date object constructor for date 1/1/1994

Date object destructor for date 1/1/1994
Employee object destructor: Blue, Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
```

There are actually five constructor calls when an `Employee` is constructed—two calls to the `string` class's constructor (lines 12–13 of Fig. 10.13), two calls to the `Date` class's default copy constructor (lines 14–15 of Fig. 10.13) and the call to the `Employee` class's constructor.

Fig. 10.14 | Demonstrating composition—an object with member objects.

Class `Date` and class `Employee` each include a destructor (lines 35–40 of Fig. 10.11 and lines 33–37 of Fig. 10.13, respectively) that prints a message when an object of its class is destructed. This enables us to confirm in the program output that objects are constructed from the *inside out* and destroyed in the reverse order, from the *outside in* (i.e., the `Date` member objects are destroyed after the `Employee` object that contains them). Notice the last four lines in the output of Fig. 10.14. The last two lines are the outputs of the `Date` destructor running on `Date` objects `hire` (line 10) and `birth` (line 9), respectively. These outputs confirm that the three objects created in `main` are destructed in the *reverse* of the order in which they were constructed. The `Employee` destructor output is five lines from the bottom. The fourth and third lines from the bottom of the output window show the destructors running for the `Employee`'s member objects `hireDate` (Fig. 10.12, line 22) and `birthDate` (Fig. 10.12, line 21). These outputs confirm that the `Employee` object is destructed from the *outside in*—i.e., the `Employee` destructor runs first (output shown five lines from the bottom of the output window), then the member objects are destructed in the *reverse order* from which they were constructed. Class `string`'s destructor does not contain output statements, so we do not see the `firstName` and `lastName` objects being destructed. Again, Fig. 10.14's output did not show the constructors running for member objects `birthDate` and `hireDate`, because these objects were initialized with the default `Date` class copy constructors provided by the compiler.

What Happens When I Do Not Use the Member Initializer List?

If a member object is not initialized through a member initializer, the member object's default constructor will be called implicitly. Values, if any, established by the default constructor can be overridden by *set* functions. However, for complex initialization, this approach may require significant additional work and time.

Common Programming Error 10.6

A compilation error occurs if a member object is not initialized with a member initializer and the member object's class does not provide a default constructor (i.e., the member object's class defines one or more constructors, but none is a default constructor).

Performance Tip 10.2

Initialize member objects explicitly through member initializers. This eliminates the overhead of "doubly initializing" member objects—once when the member object's default constructor is called and again when set *functions are called in the constructor body (or later) to initialize the member object.*

Software Engineering Observation 10.7

If a class member is an object of another class, making that member object `public` does not violate the encapsulation and hiding of that member object's `private` members. But, it does violate the encapsulation and hiding of the containing class's implementation, so member objects of class types should still be `private`, like all other data members.

10.4 `friend` Functions and `friend` Classes

A **`friend` function** of a class is defined outside that class's scope, yet has the right to access the non-`public` (and `public`) members of the class. Standalone functions, entire classes or member functions of other classes may be declared to be friends of another class.

Using `friend` functions can enhance performance. This section presents a mechanical example of how a `friend` function works. Later in the book, `friend` functions are used to overload operators for use with class objects (Chapter 11) and to create iterator classes (Chapter 20, Data Structures). Objects of an iterator class can successively select items or perform an operation on items in a container class object. Objects of container classes can store items. Using friends is often appropriate when a member function cannot be used for certain operations, as we'll see in Chapter 11.

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword `friend`. To declare all member functions of class `ClassTwo` as friends of class `ClassOne`, place a declaration of the form

```
    friend class ClassTwo;
```

in the definition of class `ClassOne`.

Software Engineering Observation 10.8

Even though the prototypes for `friend` functions appear in the class definition, friends are not member functions.

Software Engineering Observation 10.9

Member access notions of `private`, `protected` and `public` are not relevant to `friend` declarations, so `friend` declarations can be placed anywhere in a class definition.

Good Programming Practice 10.1

Place all friendship declarations first inside the class definition's body and do not precede them with any access specifier.

Friendship is granted, not taken—i.e., for class B to be a `friend` of class A, class A must explicitly declare that class B is its `friend`. Also, the friendship relation is neither symmetric nor transitive; i.e., if class A is a `friend` of class B, and class B is a `friend` of class C, you cannot infer that class B is a `friend` of class A (again, friendship is not symmetric), that class C is a `friend` of class B (also because friendship is not symmetric), or that class A is a `friend` of class C (friendship is not transitive).

Software Engineering Observation 10.10

Some people in the OOP community feel that "friendship" corrupts information hiding and weakens the value of the object-oriented design approach. In this text, we identify several examples of the responsible use of friendship.

Modifying a Class's `private` Data with a Friend Function

Figure 10.15 is a mechanical example in which we define `friend` function `setX` to set the `private` data member `x` of class `Count`. The `friend` declaration (line 9) appears first (by convention) in the class definition, even before `public` member functions are declared. Again, this `friend` declaration can appear anywhere in the class.

```
 1  // Fig. 10.15: fig10_15.cpp
 2  // Friends can access private members of a class.
 3  #include <iostream>
 4  using namespace std;
 5
 6  // Count class definition
 7  class Count
 8  {
 9     friend void setX( Count &, int ); // friend declaration
10  public:
11     // constructor
12     Count()
13        : x( 0 ) // initialize x to 0
14     {
15        // empty body
16     } // end constructor Count
17
```

Fig. 10.15 | Friends can access `private` members of a class. (Part 1 of 2.)

```
18       // output x
19       void print() const
20       {
21          cout << x << endl;
22       } // end function print
23    private:
24       int x; // data member
25    }; // end class Count
26
27    // function setX can modify private data of Count
28    // because setX is declared as a friend of Count (line 9)
29    void setX( Count &c, int val )
30    {
31       c.x = val; // allowed because setX is a friend of Count
32    } // end function setX
33
34    int main()
35    {
36       Count counter; // create Count object
37
38       cout << "counter.x after instantiation: ";
39       counter.print();
40
41       setX( counter, 8 ); // set x using a friend function
42       cout << "counter.x after call to setX friend function: ";
43       counter.print();
44    } // end main
```

```
counter.x after instantiation: 0
counter.x after call to setX friend function: 8
```

Fig. 10.15 | Friends can access private members of a class. (Part 2 of 2.)

Function setX (lines 29–32) is a C-style, stand-alone function—it isn't a member function of class Count. For this reason, when setX is invoked for object counter, line 41 passes counter as an argument to setX rather than using a handle (such as the name of the object) to call the function, as in

```
counter.setX( 8 );
```

If you remove the friend declaration in line 9, you'll receive error messages indicating that function setX cannot modify class Count's private data member x.

As we mentioned, Fig. 10.15 is a mechanical example of using the friend construct. It would normally be appropriate to define function setX as a member function of class Count. It would also normally be appropriate to separate the program of Fig. 10.15 into three files:

1. A header file (e.g., Count.h) containing the Count class definition, which in turn contains the prototype of friend function setX
2. An implementation file (e.g., Count.cpp) containing the definitions of class Count's member functions and the definition of friend function setX
3. A test program (e.g., fig10_15.cpp) with main.

Overloaded `friend` Functions

It's possible to specify overloaded functions as `friend`s of a class. Each function intended to be a `friend` must be explicitly declared in the class definition as a `friend` of the class.

10.5 Using the this Pointer

We've seen that an object's member functions can manipulate the object's data. How do member functions know *which* object's data members to manipulate? Every object has access to its own address through a pointer called **`this`** (a C++ keyword). The `this` pointer is *not* part of the object itself—i.e., the memory occupied by the `this` pointer is not reflected in the result of a `sizeof` operation on the object. Rather, the `this` pointer is passed (by the compiler) as an implicit argument to each of the object's non-`static` member functions. Section 10.6 introduces `static` class members and explains why the `this` pointer is *not* implicitly passed to `static` member functions.

Objects use the `this` pointer implicitly (as we've done to this point) or explicitly to reference their data members and member functions. The type of the `this` pointer depends on the type of the object and whether the member function in which `this` is used is declared `const`. For example, in a nonconstant member function of class `Employee`, the `this` pointer has type `Employee * const` (a constant pointer to a nonconstant `Employee` object). In a constant member function of the class `Employee`, the `this` pointer has the data type `const Employee * const` (a constant pointer to a constant `Employee` object).

The next example shows implicit and explicit use of the `this` pointer; later in this chapter and in Chapter 11, we show some substantial and subtle examples of using `this`.

Implicitly and Explicitly Using the `this` Pointer to Access an Object's Data Members

Figure 10.16 demonstrates the implicit and explicit use of the `this` pointer to enable a member function of class `Test` to print the `private` data `x` of a `Test` object.

```
// Fig. 10.16: fig10_16.cpp
// Using the this pointer to refer to object members.
#include <iostream>
using namespace std;

class Test
{
public:
   Test( int = 0 ); // default constructor
   void print() const;
private:
   int x;
}; // end class Test

// constructor
Test::Test( int value )
   : x( value ) // initialize x to value
{
   // empty body
} // end constructor Test
```

Fig. 10.16 | `this` pointer implicitly and explicitly accessing an object's members. (Part 1 of 2.)

```
21
22 // print x using implicit and explicit this pointers;
23 // the parentheses around *this are required
24 void Test::print() const
25 {
26    // implicitly use the this pointer to access the member x
27    cout << "        x = " << x;
28
29    // explicitly use the this pointer and the arrow operator
30    // to access the member x
31    cout << "\n  this->x = " << this->x;
32
33    // explicitly use the dereferenced this pointer and
34    // the dot operator to access the member x
35    cout << "\n(*this).x = " << ( *this ).x << endl;
36 } // end function print
37
38 int main()
39 {
40    Test testObject( 12 ); // instantiate and initialize testObject
41
42    testObject.print();
43 } // end main
```

```
        x = 12
  this->x = 12
(*this).x = 12
```

Fig. 10.16 | `this` pointer implicitly and explicitly accessing an object's members. (Part 2 of 2.)

For illustration purposes, member function `print` (lines 24–36) first prints `x` by using the `this` pointer implicitly (line 27)—only the name of the data member is specified. Then `print` uses two different notations to access `x` through the `this` pointer—the arrow operator (`->`) off the `this` pointer (line 31) and the dot operator (`.`) off the dereferenced `this` pointer (line 35). Note the parentheses around `*this` (line 35) when used with the dot member selection operator (`.`). The parentheses are required because the dot operator has higher precedence than the `*` operator. Without the parentheses, the expression `*this.x` would be evaluated as if it were parenthesized as `*( this.x )`, which is a compilation error, because the dot operator cannot be used with a pointer.

Common Programming Error 10.7

Attempting to use the member selection operator (.) with a pointer to an object is a compilation error—the dot member selection operator may be used only with an lvalue *such as an object's name, a reference to an object or a dereferenced pointer to an object.*

One interesting use of the `this` pointer is to prevent an object from being assigned to itself. As we'll see in Chapter 11, self-assignment can cause serious errors when the object contains pointers to dynamically allocated storage.

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. 10.19). The program of Figs. 10.17–10.19 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. 10.18 that the last statement in the body of each of these member functions returns `*this` (lines 22, 29, 36 and 43) into a return type of `Time &`.

```cpp
// Fig. 10.17: Time.h
// Cascading member function calls.

// Time class definition.
// Member functions defined in Time.cpp.
#ifndef TIME_H
#define TIME_H

class Time
{
public:
   Time( int = 0, int = 0, int = 0 ); // default constructor

   // set functions (the Time & return types enable cascading)
   Time &setTime( int, int, int ); // set hour, minute, second
   Time &setHour( int ); // set hour
   Time &setMinute( int ); // set minute
   Time &setSecond( int ); // set second

   // get functions (normally declared const)
   int getHour() const; // return hour
   int getMinute() const; // return minute
   int getSecond() const; // return second

   // print functions (normally declared const)
   void printUniversal() const; // print universal time
   void printStandard() const; // print standard time
private:
   int hour; // 0 - 23 (24-hour clock format)
   int minute; // 0 - 59
   int second; // 0 - 59
}; // end class Time

#endif
```

Fig. 10.17 | `Time` class definition modified to enable cascaded member-function calls.

```cpp
1  // Fig. 10.18: Time.cpp
2  // Time class member-function definitions.
3  #include <iostream>
4  #include <iomanip>
5  #include "Time.h" // Time class definition
6  using namespace std;
7
```

Fig. 10.18 | `Time` class member-function definitions modified to enable cascaded member-function calls. (Part 1 of 3.)

```
// constructor function to initialize private data;
// calls member function setTime to set variables;
// default values are 0 (see class definition)
Time::Time( int hr, int min, int sec )
{
   setTime( hr, min, sec );
} // end Time constructor

// set values of hour, minute, and second
Time &Time::setTime( int h, int m, int s ) // note Time & return
{
   setHour( h );
   setMinute( m );
   setSecond( s );
   return *this; // enables cascading
} // end function setTime

// set hour value
Time &Time::setHour( int h ) // note Time & return
{
   hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
   return *this; // enables cascading
} // end function setHour

// set minute value
Time &Time::setMinute( int m ) // note Time & return
{
   minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
   return *this; // enables cascading
} // end function setMinute

// set second value
Time &Time::setSecond( int s ) // note Time & return
{
   second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
   return *this; // enables cascading
} // end function setSecond

// get hour value
int Time::getHour() const
{
   return hour;
} // end function getHour

// get minute value
int Time::getMinute() const
{
   return minute;
} // end function getMinute

```

Fig. 10.18 | Time class member-function definitions modified to enable cascaded member-function calls. (Part 2 of 3.)

```
// get second value
int 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 ) << hour << ":"
      << setw( 2 ) << minute << ":" << setw( 2 ) << second;
} // end function printUniversal

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

Fig. 10.18 | `Time` class member-function definitions modified to enable cascaded member-function calls. (Part 3 of 3.)

```
// Fig. 10.19: fig10_19.cpp
// Cascading member-function calls with the this pointer.
#include <iostream>
#include "Time.h" // Time class definition
using namespace std;

int main()
{
   Time t; // create Time object

   // cascaded function calls
   t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );

   // output time in universal and standard formats
   cout << "Universal time: ";
   t.printUniversal();

   cout << "\nStandard time: ";
   t.printStandard();

   cout << "\n\nNew standard time: ";

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

Fig. 10.19 | Cascading member-function calls with the `this` pointer. (Part 1 of 2.)

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

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

Fig. 10.19 | Cascading member-function calls with the `this` pointer. (Part 2 of 2.)

The program of Fig. 10.19 creates `Time` object `t` (line 9), then uses it in cascaded member-function calls (lines 12 and 24). Why does the technique of returning `*this` as a reference work? The dot operator (`.`) associates from left to right, so line 12 first evaluates `t.setHour(18)`, then returns a reference to object `t` as the value of this function call. The remaining expression is then interpreted as

```
      t.setMinute( 30 ).setSecond( 22 );
```

The `t.setMinute( 30 )` call executes and returns a reference to the object `t`. The remaining expression is interpreted as

```
      t.setSecond( 22 );
```

Line 24 also uses cascading. The calls must appear in the order shown in line 24, because `printStandard` as defined in the class does not return a reference to `t`. Placing the call to `printStandard` before the call to `setTime` in line 24 results in a compilation error. Chapter 11 presents several practical examples of using cascaded function calls. One such example uses multiple `<<` operators with `cout` to output multiple values in a single statement.

10.6 static Class Members

There is an important exception to the rule that each object of a class has its own copy of all the data members of the class. 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., a property that is shared by all instances and is not specific to any one object of the class). Recall that the versions of class `GradeBook` in Chapter 7 use `static` data members to store constants representing the number of grades that all `GradeBook` objects can hold.

Motivating Class-Wide Data

Let's further motivate the need for `static` class-wide data with an example. Suppose that we have a video game with `Martian`s and other space creatures. Each `Martian` tends to be brave and willing to attack other space creatures when the `Martian` is aware that there are at least five `Martian`s present. If fewer than five are present, each `Martian` becomes cowardly. So each `Martian` needs to know the `martianCount`. We could endow each instance of class `Martian` with `martianCount` as a data member. If we do, every `Martian` will have a separate copy of the data member. Every time we create a new `Martian`, we'll have to update the data member `martianCount` in all `Martian` objects. Doing this would require every `Martian` object to have, or have access to, handles to all other `Martian` objects in memory. This wastes space with the redundant copies and wastes time in updating the separate copies. Instead, we declare `martianCount` to be `static`. This makes `martianCount` class-wide data. Every `Martian` can access `martianCount` as if it were a data member of the

Martian, but only one copy of the static variable martianCount is maintained by C++. This saves space. We save time by having the Martian constructor increment static variable martianCount and having the Martian destructor decrement martianCount. Because there is only one copy, we do not have to increment or decrement separate copies of martianCount for each Martian object.

Performance Tip 10.3

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

Scope and Initialization of **static** *Data Members*

Although they may seem like global variables, a class's static data members have class scope. Also, static members can be declared public, private or protected. A fundamental-type static data member is initialized by default to 0. If you want a different initial value, a static data member can be initialized *once*. A static const data member of int or enum type can be initialized in its declaration in the class definition. However, all other static data members must be defined *at global namespace scope* (i.e., outside the body of the class definition) and can be initialized only in those definitions. If a static data member is an object of a class that provides a default constructor, the static data member need not be initialized because its default constructor will be called.

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 binary scope resolution operator (::) to the name of the data member. For example, if our preceding variable martianCount is public, it can be accessed with the expression Martian::martianCount when there are no Martian objects. (Of course, using public data is discouraged.)

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 binary scope resolution operator. A static member function is a service of the *class*, not of a specific object of the class.

Software Engineering Observation 10.11

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.

Demonstrating **static** *Data Members*

The program of Figs. 10.20–10.22 demonstrates a private static data member called count (Fig. 10.20, line 25) and a public static member function called getCount (Fig. 10.20, line 19). In Fig. 10.21, line 8 defines and initializes the data member count to zero *at global namespace scope* and lines 12–15 define static member function getCount. Notice that neither line 8 nor line 12 includes keyword static, yet both lines refer to static class members. When static is applied to an item at global namespace scope, that item becomes known only in that file. The static class members need to be available to any client code that uses the class, so we declare them `static` only in the `.h` file. Data member `count` maintains a count of the number of objects of class `Employee` that have been instantiated. When objects of class `Employee` exist, member `count` can be referenced through any member function of an `Employee` object—in Fig. 10.21, `count` is referenced by both line 22 in the constructor and line 32 in the destructor.

Common Programming Error 10.8

It's a compilation error to include keyword `static` in the definition of a `static` data member at global namespace scope.

```cpp
// Fig. 10.20: Employee.h
// Employee class definition with a static data member to
// track the number of Employee objects in memory
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>
using namespace std;

class Employee
{
public:
   Employee( const string &, const string & ); // constructor
   ~Employee(); // destructor
   string getFirstName() const; // return first name
   string getLastName() const; // return last name

   // static member function
   static int getCount(); // return number of objects instantiated
private:
   string firstName;
   string lastName;

   // static data
   static int count; // number of objects instantiated
}; // end class Employee

#endif
```

Fig. 10.20 | `Employee` class definition with a `static` data member to track the number of `Employee` objects in memory.

```cpp
1  // Fig. 10.21: Employee.cpp
2  // Employee class member-function definitions.
3  #include <iostream>
4  #include "Employee.h" // Employee class definition
5  using namespace std;
6
```

Fig. 10.21 | `Employee` class member-function definitions. (Part 1 of 2.)

```cpp
// define and initialize static data member at global namespace scope
int Employee::count = 0; // cannot include keyword static

// define static member function that returns number of
// Employee objects instantiated (declared static in Employee.h)
int Employee::getCount()
{
   return count;
} // end static function getCount

// constructor initializes non-static data members and
// increments static data member count
Employee::Employee( const string &first, const string &last )
   : firstName( first ), lastName( last )
{
   ++count; // increment static count of employees
   cout << "Employee constructor for " << firstName
      << ' ' << lastName << " called." << endl;
} // end Employee constructor

// destructor deallocates dynamically allocated memory
Employee::~Employee()
{
   cout << "~Employee() called for " << firstName
      << ' ' << lastName << endl;
   --count; // decrement static count of employees
} // end ~Employee destructor

// return first name of employee
string Employee::getFirstName() const
{
   return firstName; // return copy of first name
} // end function getFirstName

// return last name of employee
string Employee::getLastName() const
{
   return lastName; // return copy of last name
} // end function getLastName
```

Fig. 10.21 | `Employee` class member-function definitions. (Part 2 of 2.)

Figure 10.22 uses `static` member function `getCount` to determine the number of `Employee` objects in memory at various points in the program. The program calls `Employee::getCount()` before any `Employee` objects have been created (line 12), after two `Employee` objects have been created (line 23) and after those `Employee` objects have been destroyed (line 34). Lines 16–29 in `main` define a nested scope. Recall that local variables exist until the scope in which they are defined terminates. In this example, we create two `Employee` objects in lines 17–18 inside the nested scope. As each constructor executes, it increments class `Employee`'s `static` data member `count`. These `Employee` objects are destroyed when the program reaches line 29. At that point, each object's destructor executes and decrements class `Employee`'s `static` data member count.

```
// Fig. 10.22: fig10_22.cpp
// static data member tracking the number of objects of a class.
#include <iostream>
#include "Employee.h" // Employee class definition
using namespace std;

int main()
{
   // no objects exist; use class name and binary scope resolution
   // operator to access static member function getCount
   cout << "Number of employees before instantiation of any objects is "
      << Employee::getCount() << endl; // use class name

   // the following scope creates and destroys
   // Employee objects before main terminates
   {
      Employee e1( "Susan", "Baker" );
      Employee e2( "Robert", "Jones" );

      // two objects exist; call static member function getCount again
      // using the class name and the binary scope resolution operator
      cout << "Number of employees after objects are instantiated is "
         << Employee::getCount();

      cout << "\n\nEmployee 1: "
         << e1.getFirstName() << " " << e1.getLastName()
         << "\nEmployee 2: "
         << e2.getFirstName() << " " << e2.getLastName() << "\n\n";
   } // end nested scope in main

   // no objects exist, so call static member function getCount again
   // using the class name and the binary scope resolution operator
   cout << "\nNumber of employees after objects are deleted is "
      << Employee::getCount() << endl;
} // end main
```

```
Number of employees before instantiation of any objects is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after objects are instantiated is 2

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Robert Jones
~Employee() called for Susan Baker

Number of employees after objects are deleted is 0
```

Fig. 10.22 | static data member tracking the number of objects of a class.

A member function should be declared static if it does not access non-static data members or non-static member functions of the class. Unlike non-static member functions, a static member function does not have a this pointer, because static data mem-