Chapter 10 Operator Overloading; Class `string`

C++ How to Program, 9/e

OBJECTIVES

In this chapter you'll:

- Learn how operator overloading can help you craft valuable classes.
- Overload unary and binary operators.
- Convert objects from one class to another class.
- Use overloaded operators and additional features of the `string` class.
- Create `PhoneNumber`, `Date` and `Array` classes that provide overloaded operators.
- Perform dynamic memory allocation with `new` and `delete`.
- Use keyword `explicit` to indicate that a constructor cannot be used for implicit conversions.
- Experience a "light-bulb moment" when you'll truly appreciate the elegance and beauty of the class concept.

10.1 Introduction

- This chapter shows how to enable C++'s operators to work with objects—a process called operator overloading.
- One example of an overloaded operator built into C++ is <<, which is used *both* as the stream insertion operator and as the bitwise left-shift operator..
- C++ overloads the addition operator (+) and the subtraction operator (-) to perform differently, depending on their context in integer, floating-point and pointer arithmetic with data of fundamental types.
- You can overload most operators to be used with class objects—the compiler generates the appropriate code based on the types of the operands.

10.2 Using the Overloaded Operators of Standard Library Class `string`

- Figure 10.1 demonstrates many of class `string`'s overloaded operators and several other useful member functions, including `empty`, `substr` and `at`.
- Function `empty` determines whether a `string` is empty, function `substr` returns a `string` that represents a portion of an existing `string` and function `at` returns the character at a specific index in a `string` (after checking that the index is in range).
- Chapter 21 presents class string in detail.

```
1  // Fig. 10.1: fig10_01.cpp
2  // Standard Library string class test program.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string s1( "happy" );
10    string s2( " birthday" );
11    string s3;
12
13    // test overloaded equality and relational operators
14    cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
15       << "\"; s3 is \"" << s3 << '\"'
16       << "\n\nThe results of comparing s2 and s1:"
17       << "\ns2 == s1 yields " << ( s2 == s1 ? "true" : "false" )
18       << "\ns2 != s1 yields " << ( s2 != s1 ? "true" : "false" )
19       << "\ns2 >  s1 yields " << ( s2 > s1 ? "true" : "false" )
20       << "\ns2 <  s1 yields " << ( s2 < s1 ? "true" : "false" )
21       << "\ns2 >= s1 yields " << ( s2 >= s1 ? "true" : "false" )
22       << "\ns2 <= s1 yields " << ( s2 <= s1 ? "true" : "false" );
23
```

Fig. 10.1 | Standard Library `string` class test program. (Part 1 of 6.)

```
24      // test string member-function empty
25      cout << "\n\nTesting s3.empty():" << endl;
26
27      if ( s3.empty() )
28      {
29         cout << "s3 is empty; assigning s1 to s3;" << endl;
30         s3 = s1; // assign s1 to s3
31         cout << "s3 is \"" << s3 << "\"";
32      } // end if
33
34      // test overloaded string concatenation operator
35      cout << "\n\ns1 += s2 yields s1 = ";
36      s1 += s2; // test overloaded concatenation
37      cout << s1;
38
39      // test overloaded string concatenation operator with a C string
40      cout << "\n\ns1 += \" to you\" yields" << endl;
41      s1 += " to you";
42      cout << "s1 = " << s1 << "\n\n";
43
44      // test string member function substr
45      cout << "The substring of s1 starting at location 0 for\n"
46         << "14 characters, s1.substr(0, 14), is:\n"
47         << s1.substr( 0, 14 ) << "\n\n";
```

Fig. 10.1 | Standard Library `string` class test program. (Part 2 of 6.)

```
48
49      // test substr "to-end-of-string" option
50      cout << "The substring of s1 starting at\n"
51         << "location 15, s1.substr(15), is:\n"
52         << s1.substr( 15 ) << endl;
53
54      // test copy constructor
55      string s4( s1 );
56      cout << "\ns4 = " << s4 << "\n\n";
57
58      // test overloaded copy assignment (=) operator with self-assignment
59      cout << "assigning s4 to s4" << endl;
60      s4 = s4;
61      cout << "s4 = " << s4 << endl;
62
63      // test using overloaded subscript operator to create lvalue
64      s1[ 0 ] = 'H';
65      s1[ 6 ] = 'B';
66      cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
67         << s1 << "\n\n";
68
```

Fig. 10.1 | Standard Library `string` class test program. (Part 3 of 6.)

```
69      // test subscript out of range with string member function "at"
70      try
71      {
72         cout << "Attempt to assign 'd' to s1.at( 30 ) yields:" << endl;
73         s1.at( 30 ) = 'd'; // ERROR: subscript out of range
74      } // end try
75      catch ( out_of_range &ex )
76      {
77         cout << "An exception occurred: " << ex.what() << endl;
78      } // end catch
79   } // end main
```

```
s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
```

Fig. 10.1 | Standard Library `string` class test program. (Part 4 of 6.)

```
s2 >= s1 yields false
s2 <= s1 yields true

Testing s3.empty():
s3 is empty; assigning s1 to s3;
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
s1 = happy birthday to you

The substring of s1 starting at location 0 for
14 characters, s1.substr(0, 14), is:
happy birthday

The substring of s1 starting at
location 15, s1.substr(15), is:
to you
```

Fig. 10.1 | Standard Library `string` class test program. (Part 5 of 6.)

```
s4 = happy birthday to you

assigning s4 to s4
s4 = happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1.at( 30 ) yields:
An exception occurred: invalid string position
```

Fig. 10.1 | Standard Library `string` class test program. (Part 6 of 6.)

10.2 Using the Overloaded Operators of Standard Library Class `string` (cont.)

- Class `string`'s overloaded equality and relational operators perform lexicographical comparisons (i.e., like a dictionary ordering) using the numerical values of the characters (see Appendix B, ASCII Character Set) in each `string`.
- Class `string` provides member function `empty` to determine whether a `string` is empty, which we demonstrate in line 27.
  - Returns `true` if the `string` is empty; otherwise, it returns `false`.
- Line 36 demonstrates class `string`'s overloaded `+=` operator for string concatenation.
  - Line 41 demonstrates that a string literal can be appended to a `string` object by using operator `+=`. Line 42 displays the result.

10.2 Using the Overloaded Operators of Standard Library Class `string` (cont.)

- Class `string` provides member function `substr` (lines 47 and 52) to return a *portion* of a string as a `string` object.
  - The call to `substr` in line 47 obtains a 14-character substring (specified by the second argument) of `s1` starting at position 0 (specified by the first argument).
  - The call to `substr` in line 52 obtains a substring starting from position 15 of `s1`.
  - When the second argument is not specified, `substr` returns the *remainder* of the `string` on which it's called.
- Lines 64-65 use class `string`'s overloaded `[]` operator can create *lvalues* that enable new characters to replace existing characters in `s1`.
  - *Class `string`'s overloaded `[]` operator does not perform any bounds checking.*

10.2 Using the Overloaded Operators of Standard Library Class `string` (cont.)

- Class `string` *does* provide bounds checking in its member function `at`, which throws an exception if its argument is an invalid subscript.
  - If the subscript is valid, function `at` returns the character at the specified location as a modifiable *lvalue* or a nonmodifiable *lvalue* (e.g., a `const` reference), depending on the context in which the call appears.

10.3 Fundamentals of Operator Overloading

- As you saw in Fig. 10.1, operators provide a concise notation for manipulating string objects.
- You can use operators with your own user-defined types as well.
- Although C++ does not allow new operators to be created, it does allow most existing operators to be overloaded so that, when they're used with objects, they have meaning appropriate to those objects.

10.3 Fundamentals of Operator Overloading (cont.)

- Operator overloading is not automatic—you must write operator-overloading functions to perform the desired operations.
- An operator is overloaded by writing a non-`static` member function definition or non-member function definition as you normally would, except that the function name starts with the keyword `operator` followed by the symbol for the operator being overloaded.
  - For example, the function name `operator+` would be used to overload the addition operator (+) for use with objects of a particular class (or `enum`).

10.3 Fundamentals of Operator Overloading (cont.)

- When operators are overloaded as member functions, they must be non-`static`, because *they must be called on an object of the class* and operate on that object.
- To use an operator on class objects, you must define overloaded operator functions for that class—with three exceptions.
  - The *assignment operator (=)* may be used with *most* classes to perform *memberwise assignment* of the data members—each data member is assigned from the assignment's "source" object (on the right) to the "target" object (on the left).
    - *Memberwise assignment is dangerous for classes with pointer members*, so we'll explicitly overload the assignment operator for such classes.
  - The *address operator (&)* returns a pointer to the object; this operator also can be overloaded.
  - The *comma operator* evaluates the expression to its left then the expression to its right, and returns the value of the latter expression.

10.3 Fundamentals of Operator Overloading (cont.)

- Most of C++'s operators can be overloaded.
- Figure 10.2 shows the operators that cannot be overloaded.

Operators that cannot be overloaded			
.	.* (pointer to member)	::	?:

Fig. 10.2 | Operators that cannot be overloaded.

10.3 Fundamentals of Operator Overloading (cont.)

- The *precedence of an operator cannot be changed by overloading*.
  - However, parentheses can be used to force the order of evaluation of overloaded operators in an expression.
- *The associativity of an operator cannot be changed by overloading*
  - if an operator normally associates from left to right, then so do all of its overloaded versions.
- *You cannot change the "arity" of an operator* (that is, the number of operands an operator takes)
  - overloaded unary operators remain unary operators; overloaded binary operators remain binary operators. Operators &, *, + and – all have both unary and binary versions; these unary and binary versions can be separately overloaded.

10.3 Fundamentals of Operator Overloading (cont.)

- *You cannot create new operators; only existing operators can be overloaded.*
- The meaning of how an operator works on values of fundamental types *cannot* be changed by operator overloading.
  - For example, you cannot make the + operator subtract two `int`s. Operator overloading works only with *objects of user-defined types or with a mixture of an object of a user-defined type and an object of a fundamental type.*

10.3 Fundamentals of Operator Overloading (cont.)

- Related operators, like + and +=, must be overloaded separately.
- When overloading `()`, `[]`, `->` or any of the assignment operators, the operator overloading function must be declared as a class member.
  - For all other overloadable operators, the operator overloading functions can be member functions or non-member functions.

Software Engineering Observation 10.1

Overload operators for class types so they work as closely as possible to the way built-in operators work on fundamental types.

10.4 Overloading Binary Operators

- *A binary operator can be overloaded as a non-`static` member function with one parameter or as a non-member function with two parameters (one of those parameters must be either a class object or a reference to a class object).*
- As a non-member function, binary operator < must take two arguments—one of which must be an object (or a reference to an object) of the class.

10.5 Overloading the Binary Stream Insertion and Stream Extraction Operators

- You can input and output fundamental-type data using the stream extraction operator >> and the stream insertion operator <<.
- The C++ class libraries overload these binary operators for each fundamental type, including pointers and `char *` strings.
- You can also overload these operators to perform input and output for your own types.
- The program of Figs. 10.3–10.5 overloads these operators to input and output `PhoneNumber` objects in the format "`(000) 000-0000`." The program assumes telephone numbers are input correctly.

```
1  // Fig. 10.3: PhoneNumber.h
2  // PhoneNumber class definition
3  #ifndef PHONENUMBER_H
4  #define PHONENUMBER_H
5
6  #include <iostream>
7  #include <string>
8
9  class PhoneNumber
10 {
11    friend std::ostream &operator<<( std::ostream &, const PhoneNumber & );
12    friend std::istream &operator>>( std::istream &, PhoneNumber & );
13 private:
14    std::string areaCode; // 3-digit area code
15    std::string exchange; // 3-digit exchange
16    std::string line; // 4-digit line
17 }; // end class PhoneNumber
18
19 #endif
```

Fig. 10.3 | PhoneNumber class with overloaded stream insertion and stream extraction operators as `friend` functions.

```
1  // Fig. 10.4: PhoneNumber.cpp
2  // Overloaded stream insertion and stream extraction operators
3  // for class PhoneNumber.
4  #include <iomanip>
5  #include "PhoneNumber.h"
6  using namespace std;
7
8  // overloaded stream insertion operator; cannot be
9  // a member function if we would like to invoke it with
10 // cout << somePhoneNumber;
11 ostream &operator<<( ostream &output, const PhoneNumber &number )
12 {
13    output << "(" << number.areaCode << ") "
14       << number.exchange << "-" << number.line;
15    return output; // enables cout << a << b << c;
16 } // end function operator<<
17
```

Fig. 10.4 | Overloaded stream insertion and stream extraction operators for class PhoneNumber. (Part 1 of 2.)

```
18 // overloaded stream extraction operator; cannot be
19 // a member function if we would like to invoke it with
20 // cin >> somePhoneNumber;
21 istream &operator>>( istream &input, PhoneNumber &number )
22 {
23    input.ignore(); // skip (
24    input >> setw( 3 ) >> number.areaCode; // input area code
25    input.ignore( 2 ); // skip ) and space
26    input >> setw( 3 ) >> number.exchange; // input exchange
27    input.ignore(); // skip dash (-)
28    input >> setw( 4 ) >> number.line; // input line
29    return input; // enables cin >> a >> b >> c;
30 } // end function operator>>
```

Fig. 10.4 | Overloaded stream insertion and stream extraction operators for class PhoneNumber. (Part 2 of 2.)

```
1  // Fig. 10.5: fig10_05.cpp
2  // Demonstrating class PhoneNumber's overloaded stream insertion
3  // and stream extraction operators.
4  #include <iostream>
5  #include "PhoneNumber.h"
6  using namespace std;
7
8  int main()
9  {
10    PhoneNumber phone; // create object phone
11
12    cout << "Enter phone number in the form (123) 456-7890:" << endl;
13
14    // cin >> phone invokes operator>> by implicitly issuing
15    // the non-member function call operator>>( cin, phone )
16    cin >> phone;
17
18    cout << "The phone number entered was: ";
19
20    // cout << phone invokes operator<< by implicitly issuing
21    // the non-member function call operator<<( cout, phone )
22    cout << phone << endl;
23 } // end main
```

Fig. 10.5 | Overloaded stream insertion and stream extraction operators. (Part 1 of 2.)

```
Enter phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was: (800) 555-1212
```

Fig. 10.5 | Overloaded stream insertion and stream extraction operators. (Part 2 of 2.)

10.5 Overloading the Binary Stream Insertion and Stream Extraction Operators (cont.)

Overloading the Stream Extraction (>>) Operator

- The stream extraction operator function `operator>>` (Fig. 10.4, lines 21–30) takes `istream` reference `input` and `PhoneNumber` reference `number` as arguments and returns an `istream` reference.
- Operator function `operator>>` inputs phone numbers of the form
  - `(800) 555-1212`
- When the compiler sees the expression
  - `cin >> phone`
- In line 16 of Fig. 10.5, the compiler generates the *non-member function call*
  - `operator>>( cin, phone );`
- When this call executes, reference parameter `input` (Fig. 10.4, line 21) becomes an alias for `cin` and reference parameter `number` becomes an alias for `phone`.

10.5 Overloading the Binary Stream Insertion and Stream Extraction Operators (cont.)

- The operator function reads as `string`s the three parts of the telephone number into the `areaCode` (line 24), `exchange` (Line 26) and `line` (line 28) members of the `PhoneNumber` object referenced by parameter `Number`.
- Stream manipulator `setw` limits the number of characters read into each `string`.
- The parentheses, space and dash characters are skipped by calling `istream` member function `ignore` (Fig. 10.4, lines 23, 25 and 27), which discards the specified number of characters in the input stream (one character by default).

10.5 Overloading Stream Insertion and Stream Extraction Operators (cont.)

- Function `operator>>` returns `istream` reference `input` (i.e., `cin`).
- This enables input operations on `PhoneNumber` objects to be cascaded with input operations on other `PhoneNumber` objects or on objects of other data types.

Good Programming Practice 10.1

Overloaded operators should mimic the functionality of their built-in counterparts—e.g., the + operator should perform addition, not subtraction. Avoid excessive or inconsistent use of operator overloading, as this can make a program cryptic and difficult to read.

10.5 Overloading Stream Insertion and Stream Extraction Operators (cont.)

Overloading the Stream Insertion (<<) Operator

- The stream insertion operator function (Fig. 10.4, lines 11-16) takes an `ostream` reference (`output`) and a `const PhoneNumber` reference (`number`) as arguments and returns an `ostream` reference.
- Function `operator<<` displays objects of type `PhoneNumber`.
- When the compiler sees the expression
  - `cout << phone`

  in line 22 if Fig. 10.5, the compiler generates the non-member function call
  - `operator<<( cout, phone );`
- Function `operator<<` displays the parts of the telephone number as `string`s, because they're stored as `string` objects.

10.5 Overloading the Binary Stream Insertion and Stream Extraction Operators (cont.)

Overloaded Operators as Non-Member `friend` Functions

- The functions `operator>>` and `operator<<` are declared in `PhoneNumber` as non-member, `friend` functions.
- They're *non-member functions* because the object of class `PhoneNumber` is the operator's *right* operand.

Software Engineering Observation 10.2

New input/output capabilities for user-defined types are added to C++ without modifying standard input/output library classes. This is another example of C++'s extensibility.

10.5 Overloading the Binary Stream Insertion and Stream Extraction Operators (cont.)

Why Overloaded Stream Insertion and Stream Extraction Operators Are Overloaded as Non-Member Functions

- The overloaded stream insertion operator (<<) is used in an expression in which the left operand has type `ostream&`, as in `cout << classObject`.
- To use the operator in this manner where the *right* operand is an object of a user-defined class, it must be overloaded as a *non-member function*.

10.5 Overloading the Binary Stream Insertion and Stream Extraction Operators (cont.)

- Similarly, the overloaded stream extraction operator (>>) is used in an expression in which the left operand has type `istream &`, as in `cin >> classObject`, and the *right* operand is an object of a user-defined class, so it, too, must be a non-member function.
- Each of these overloaded operator functions may require access to the `private` data members of the class object being output or input, so these overloaded operator functions can be made `friend` functions of the class for performance reasons.

10.6 Overloading Unary Operators

- *A unary operator for a class can be overloaded as a non-`static` member function with no arguments or as a non-member function with one argument that must be an object (or a reference to an object) of the class.*
- A unary operator such as `!` may be overloaded as a *non-member function* with one parameter.

10.7 Overloading the Unary Prefix and Postfix ++ and -- Operators

- The prefix and postfix versions of the increment and decrement operators can all be overloaded.
- *To overload the increment operator to allow both prefix and postfix increment usage, each overloaded operator function must have a distinct signature, so that the compiler will be able to determine which version of ++ is intended.*
- The prefix versions are overloaded exactly as any other prefix unary operator would be.

10.7 Overloading the Unary Prefix and Postfix ++ and -- Operators (cont.)

- Suppose that we want to add 1 to the day in `Date` object `d1`.
- When the compiler sees the preincrementing expression `++d1`, the compiler generates the *member-function call*
  - `d1.operator++()`
- The prototype for this operator function would be
  - `Date &operator++();`
- If the prefix increment operator is implemented as a non-member function, then, when the compiler sees the expression `++d1`, the compiler generates the function call
  - `operator++( d1 )`
- The prototype for this operator function would be declared in the `Date` class as
  - `Date &operator++( Date & );`

10.7 Overloading the Unary Prefix and Postfix ++ and -- Operators (cont.)

Overloading the Postfix Increment Operator

- Overloading the postfix increment operator presents a challenge, because the compiler must be able to distinguish between the signatures of the overloaded prefix and postfix increment operator functions.
- The *convention* that has been adopted in C++ is that, when the compiler sees the postincrementing expression `d1++`, it generates the *member-function call*
  - `d1.operator++( 0 )`
- The prototype for this function is
  - `Date operator++( int )`
- The argument `0` is strictly a "dummy value" that enables the compiler to distinguish between the prefix and postfix increment operator functions.
- The same syntax is used to differentiate between the prefix and postfix decrement operator functions.

10.7 Overloading the Unary Prefix and Postfix ++ and -- Operators (cont.)

- If the postfix increment is implemented as a non-member function, then, when the compiler sees the expression `d1++`, the compiler generates the function call
  - `operator++( d1, 0 )`
- The prototype for this function would be
  - `Date operator++( Date &, int );`
- Once again, the `0` argument is used by the compiler to distinguish between the prefix and postfix increment operators implemented as non-member functions.
- The postfix increment operator returns `Date` objects *by value*, whereas the prefix increment operator returns `Date` objects *by reference*—the postfix increment operator typically returns a temporary object that contains the original value of the object before the increment occurred.

Performance Tip 10.1

The extra object that's created by the postfix increment (or decrement) operator can result in a performance problem—especially when the operator is used in a loop. For this reason, you should prefer the overloaded prefix increment and decrement operators.

10.8 Case Study: A `Date` Class

- The program of Figs. 10.6–10.8 demonstrates a `Date` class, which uses overloaded prefix and postfix increment operators to add 1 to the day in a `Date` object, while causing appropriate increments to the month and year if necessary.

```
1  // Fig. 10.6: Date.h
2  // Date class definition with overloaded increment operators.
3  #ifndef DATE_H
4  #define DATE_H
5
6  #include <array>
7  #include <iostream>
8
9  class Date
10 {
11    friend std::ostream &operator<<( std::ostream &, const Date & );
12 public:
13    Date( int m = 1, int d = 1, int y = 1900 ); // default constructor
14    void setDate( int, int, int ); // set month, day, year
15    Date &operator++(); // prefix increment operator
16    Date operator++( int ); // postfix increment operator
17    Date &operator+=( unsigned int ); // add days, modify object
18    static bool leapYear( int ); // is date in a leap year?
19    bool endOfMonth( int ) const; // is date at the end of month?
20 private:
21    unsigned int month;
22    unsigned int day;
23    unsigned int year;
```

Fig. 10.6 | Date class definition with overloaded increment operators. (Part 1 of 2.)

```
24
25      static const std::array< unsigned int, 13 > days; // days per month
26      void helpIncrement(); // utility function for incrementing date
27   }; // end class Date
28
29   #endif
```

Fig. 10.6 | `Date` class definition with overloaded increment operators. (Part 2 of 2.)

```
1  // Fig. 10.7: Date.cpp
2  // Date class member- and friend-function definitions.
3  #include <iostream>
4  #include <string>
5  #include "Date.h"
6  using namespace std;
7
8  // initialize static member; one classwide copy
9  const array< unsigned int, 13 > Date::days =
10    { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
11
12 // Date constructor
13 Date::Date( int month, int day, int year )
14 {
15    setDate( month, day, year );
16 } // end Date constructor
17
```

Fig. 10.7 | Date class member- and friend-function definitions. (Part 1 of 6.)

```
18  // set month, day and year
19  void Date::setDate( int mm, int dd, int yy )
20  {
21     if ( mm >= 1 && mm <= 12 )
22        month = mm;
23     else
24        throw invalid_argument( "Month must be 1-12" );
25
26     if ( yy >= 1900 && yy <= 2100 )
27        year = yy;
28     else
29        throw invalid_argument( "Year must be >= 1900 and <= 2100" );
30
31     // test for a leap year
32     if ( ( month == 2 && leapYear( year ) && dd >= 1 && dd <= 29 ) ||
33          ( dd >= 1 && dd <= days[ month ] ) )
34        day = dd;
35     else
36        throw invalid_argument(
37           "Day is out of range for current month and year" );
38  } // end function setDate
39
```

Fig. 10.7 | `Date` class member- and `friend`-function definitions. (Part 2 of 6.)

```
40 // overloaded prefix increment operator
41 Date &Date::operator++()
42 {
43    helpIncrement(); // increment date
44    return *this; // reference return to create an lvalue
45 } // end function operator++
46
47 // overloaded postfix increment operator; note that the
48 // dummy integer parameter does not have a parameter name
49 Date Date::operator++( int )
50 {
51    Date temp = *this; // hold current state of object
52    helpIncrement();
53
54    // return unincremented, saved, temporary object
55    return temp; // value return; not a reference return
56 } // end function operator++
57
```

Fig. 10.7 | `Date` class member- and `friend`-function definitions. (Part 3 of 6.)

```
58 // add specified number of days to date
59 Date &Date::operator+=( unsigned int additionalDays )
60 {
61    for ( int i = 0; i < additionalDays; ++i )
62       helpIncrement();
63
64    return *this; // enables cascading
65 } // end function operator+=
66
67 // if the year is a leap year, return true; otherwise, return false
68 bool Date::leapYear( int testYear )
69 {
70    if ( testYear % 400 == 0 ||
71       ( testYear % 100 != 0 && testYear % 4 == 0 ) )
72       return true; // a leap year
73    else
74       return false; // not a leap year
75 } // end function leapYear
76
```

Fig. 10.7 | `Date` class member- and `friend`-function definitions. (Part 4 of 6.)

```
77  // determine whether the day is the last day of the month
78  bool Date::endOfMonth( int testDay ) const
79  {
80     if ( month == 2 && leapYear( year ) )
81        return testDay == 29; // last day of Feb. in leap year
82     else
83        return testDay == days[ month ];
84  } // end function endOfMonth
85
86  // function to help increment the date
87  void Date::helpIncrement()
88  {
89     // day is not end of month
90     if ( !endOfMonth( day ) )
91        ++day; // increment day
92     else
93        if ( month < 12 ) // day is end of month and month < 12
94        {
95           ++month; // increment month
96           day = 1; // first day of new month
97        } // end if
```

Fig. 10.7 | `Date` class member- and `friend`-function definitions. (Part 5 of 6.)

```
98          else // last day of year
99          {
100            ++year; // increment year
101            month = 1; // first month of new year
102            day = 1; // first day of new month
103         } // end else
104   } // end function helpIncrement
105
106   // overloaded output operator
107   ostream &operator<<( ostream &output, const Date &d )
108   {
109      static string monthName[ 13 ] = { "", "January", "February",
110         "March", "April", "May", "June", "July", "August",
111         "September", "October", "November", "December" };
112      output << monthName[ d.month ] << ' ' << d.day << ", " << d.year;
113      return output; // enables cascading
114   } // end function operator<<
```

Fig. 10.7 | `Date` class member- and `friend`-function definitions. (Part 6 of 6.)

```
1  // Fig. 10.8: fig10_08.cpp
2  // Date class test program.
3  #include <iostream>
4  #include "Date.h" // Date class definition
5  using namespace std;
6
7  int main()
8  {
9     Date d1( 12, 27, 2010 ); // December 27, 2010
10    Date d2; // defaults to January 1, 1900
11
12    cout << "d1 is " << d1 << "\nd2 is " << d2;
13    cout << "\n\nd1 += 7 is " << ( d1 += 7 );
14
15    d2.setDate( 2, 28, 2008 );
16    cout << "\n\n  d2 is " << d2;
17    cout << "\n++d2 is " << ++d2 << " (leap year allows 29th)";
18
19    Date d3( 7, 13, 2010 );
20
21    cout << "\n\nTesting the prefix increment operator:\n"
22       << "  d3 is " << d3 << endl;
23    cout << "++d3 is " << ++d3 << endl;
24    cout << "  d3 is " << d3;
```

Fig. 10.8 | Date class test program. (Part 1 of 2.)

```
25
26      cout << "\n\nTesting the postfix increment operator:\n"
27         << "  d3 is " << d3 << endl;
28      cout << "d3++ is " << d3++ << endl;
29      cout << "  d3 is " << d3 << endl;
30   } // end main
```

```
d1 is December 27, 2010
d2 is January 1, 1900

d1 += 7 is January 3, 2011

  d2 is February 28, 2008
++d2 is February 29, 2008 (leap year allows 29th)

Testing the prefix increment operator:
  d3 is July 13, 2010
++d3 is July 14, 2010
  d3 is July 14, 2010

Testing the postfix increment operator:
  d3 is July 14, 2010
d3++ is July 14, 2010
  d3 is July 15, 2010
```

Fig. 10.8 | Date class test program. (Part 2 of 2.)

10.8 Case Study: A `Date` Class (cont.)

- The `Date` constructor (defined in Fig. 10.7, lines 13–16) calls `setDate` to validate the month, day and year specified.
  - Invalid values for the month, day or year result in `invalid_argument` exceptions.

10.8 Case Study: A `Date` Class (cont.)

Date Class Prefix Increment Operator

- Overloading the prefix increment operator is straightforward.
  - The prefix increment operator (defined in Fig. 10.7, lines 41–45) calls utility function `helpIncrement` (defined in Fig. 10.7, lines 87–104) to increment the date.
  - This function deals with "wraparounds" or "carries" that occur when we increment the last day of the month.
  - These carries require incrementing the month.
  - If the month is already 12, then the year must also be incremented and the month must be set to 1.
  - Function `helpIncrement` uses function `endOfMonth` to determine whether the end of a month has been reached and increment the day correctly.

10.8 Case Study: A `Date` Class (cont.)

- The overloaded prefix increment operator returns a reference to the current `Date` object (i.e., the one that was just incremented).
- This occurs because the current object, `*this`, is returned as a `Date &`.
  - Enables a preincremented `Date` object to be used as an *lvalue*, which is how the built-in prefix increment operator works for fundamental types.

10.8 Case Study: A `Date` Class (cont.)

Date Class Postfix Increment Operator

- Overloading the postfix increment operator (defined in Fig. 10.7, lines 49–56) is trickier.
- To emulate the effect of the postincrement, we must return an unincremented copy of the `Date` object.
- So we'd like our postfix increment operator to operate the same way on a `Date` object.
- On entry to `operator++`, we save the current object (`*this`) in `temp` (line 51).
- Next, we call `helpIncrement` to increment the current `Date` object.
- Then, line 55 returns the unincremented copy of the object previously stored in `temp`.
- This function cannot return a reference to the local `Date` object `temp`, because a local variable is destroyed when the function in which it's declared exits.