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.