Chapter 18 Class **string** and String Stream Processing

C++ How to Program, Late Objects Version, 7/e

OBJECTIVES

In this chapter you'll learn:

- To assign, concatenate, compare, search and swap `string`s.
- To determine `string` characteristics.
- To find, replace and insert characters in `string`s.
- To convert `string`s to C-style strings and vice versa.
- To use `string` iterators.
- To perform input from and output to `string`s in memory.

18.1 Introduction

- The class template `basic_string` provides typical string-manipulation operations such as copying, searching, etc.
- The template definition and all support facilities are defined in `namespace std`; these include the `typedef` statement
  - `typedef basic_string< char > string;`
- A `typedef` is also provided for the `wchar_t` type (`wstring`).
  - Type `wchar_t` stores characters (e.g., two-byte characters, four-byte characters, etc.) for supporting other character sets.
- To use `string`s, include header file `<string>`.

18.1 Introduction (cont.)

- A `string` object can be initialized with a constructor argument such as
  - `// creates a string from a const char *`
    `string text( "Hello" );`
- which creates a `string` containing the characters in `"Hello"`, or with two constructor arguments as in
  - `string name( 8, 'x' ); // string of 8 'x' characters`
- which creates a `string` containing eight `'x'` characters.
- Class `string` also provides a default constructor (which creates an empty string) and a copy constructor.
- An empty `string` is a `string` that does not contain any characters.

18.1 Introduction (cont.)

- A `string` also can be initialized via the alternate constructor syntax in the definition of a `string` as in
  - ```
    // same as: string month( "March" );
    string month = "March";
    ```
- Remember that operator = in the preceding declaration is not an assignment; rather it's an implicit call to the `string` class constructor, which does the conversion.
- Class `string` provides no conversions from `int` or `char` to `string`.
    ```

Common Programming Error 18.1

Attempting to convert an `int` *or* `char` *to a* `string` *via an initialization in a declaration or via a constructor argument is a compilation error.*

18.1 Introduction (cont.)

- Unlike C-style `char *` strings, `string`s are not necessarily null terminated.
- The length of a `string` can be retrieved with member function `length` and with member function `size`.
- The subscript operator, `[]`, can be used with `string`s to access and modify individual characters.
- Like C-style strings, `string`s have a first subscript of `0` and a last subscript of `length() - 1`.
- Most `string` member functions take as arguments a starting subscript location and the number of characters on which to operate.

18.1 Introduction (cont.)

- The stream extraction operator (>>) is overloaded to support `string`s.
  - Input is delimited by white-space characters.
  - When a delimiter is encountered, the input operation is terminated.
- Function `getline` also is overloaded for `string`s.
- Assuming `string1` is a `string`, the statement
  - `getline( cin, string1 );`
- reads a `string` from the keyboard into `string1`.
- Input is delimited by a newline (`'\n'`), so `getLine` can read a line of text into a `string` object.

18.2 `string` Assignment and Concatenation

- Figure 18.1 demonstrates `string` assignment and concatenation.
- Line 4 includes header `<string>` for class `string`.
- Line 13 assigns the value of `string1` to `string2`.
  - After the assignment takes place, `string2` is a copy of `string1`.
- Line 14 uses member function `assign` to copy `string1` into `string3`.
  - A separate copy is made (i.e., `string1` and `string3` are independent objects).

18.2 `string` Assignment and Concatenation (cont.)

- Class `string` also provides an overloaded version of member function `assign` that copies a specified number of characters, as in
  - `targetString.assign( sourceString, start, numberOfCharacters);`
  - `sourceString` is the `string` to be copied,
  - `start` is the starting subscript and
  - `numberOfCharacters` is the number of characters to copy.

```
1  // Fig. 18.1: Fig18_01.cpp
2  // Demonstrating string assignment and concatenation.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string string1( "cat" );
10    string string2; // initialized to the empty string
11    string string3; // initialized to the empty string
12
13    string2 = string1; // assign string1 to string2
14    string3.assign( string1 ); // assign string1 to string3
15    cout << "string1: " << string1 << "\nstring2: " << string2
16       << "\nstring3: " << string3 << "\n\n";
17
18    // modify string2 and string3
19    string2[ 0 ] = string3[ 2 ] = 'r';
20
21    cout << "After modification of string2 and string3:\n" << "string1: "
22       << string1 << "\nstring2: " << string2 << "\nstring3: ";
23
```

Fig. 18.1 | Demonstrating `string` assignment and concatenation. (Part 1 of 3.)

```
24      // demonstrating member function at
25      for ( int i = 0; i < string3.length(); i++ )
26         cout << string3.at( i );
27
28      // declare string4 and string5
29      string string4( string1 + "apult" ); // concatenation
30      string string5;
31
32      // overloaded +=
33      string3 += "pet"; // create "carpet"
34      string1.append( "acomb" ); // create "catacomb"
35
36      // append subscript locations 4 through end of string1 to
37      // create string "comb" (string5 was initially empty)
38      string5.append( string1, 4, string1.length() - 4 );
39
40      cout << "\n\nAfter concatenation:\nstring1: " << string1
41         << "\nstring2: " << string2 << "\nstring3: " << string3
42         << "\nstring4: " << string4 << "\nstring5: " << string5 << endl;
43   } // end main
```

Fig. 18.1 | Demonstrating `string` assignment and concatenation. (Part 2 of 3.)

```
string1: cat
string2: cat
string3: cat

After modification of string2 and string3:
string1: cat
string2: rat
string3: car

After concatenation:
string1: catacomb
string2: rat
string3: carpet
string4: catapult
string5: comb
```

Fig. 18.1 | Demonstrating `string` assignment and concatenation. (Part 3 of 3.)

18.2 `string` Assignment and Concatenation (cont.)

- Line 19 uses the subscript operator to assign `'r'` to `string3[ 2 ]` (forming `"car"`) and to assign `'r'` to `string2[ 0 ]` (forming `"rat"`).
- Lines 25–26 output the contents of `string3` one character at a time using member function `at`, which provides checked access (or range checking)
  - going past the end of the `string` throws an `out_of_range` exception.
- The subscript operator, `[]`, does not provide checked access.
- This is consistent with its use on arrays.

Common Programming Error 18.2

Accessing a `string` *subscript outside the bounds of the* `string` *using function* `at` *is a logic error that causes an* `out_of_range` *exception.*

Common Programming Error 18.3

Accessing an element beyond the size of the `string` *using the subscript operator is an unreported logic error.*

18.2 `string` Assignment and Concatenation (cont.)

- String `string4` is declared (line 29) and initialized to the result of concatenating `string1` and `"apult"` using the overloaded + operator, which for class `string` denotes concatenation.
- Line 33 uses the addition assignment operator, +=, to concatenate `string3` and `"pet"`.
- Line 34 uses member function `append` to concatenate `string1` and `"acomb"`.
- Line 38 appends the string `"comb"` to empty `string` `string5`.
  - This member function is passed the `string` (`string1`) to retrieve characters from, the starting subscript in the `string` (4) and the number of characters to append (the value returned by `string1.length() - 4`).

18.3 Comparing `strings`

- Class `string` provides member functions for comparing `string`s.
- Figure 18.2 demonstrates class `string`'s comparison capabilities.

```
1  // Fig. 18.2: Fig18_02.cpp
2  // Demonstrating string comparison capabilities.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string string1( "Testing the comparison functions." );
10    string string2( "Hello" );
11    string string3( "stinger" );
12    string string4( string2 );
13
14    cout << "string1: " << string1 << "\nstring2: " << string2
15       << "\nstring3: " << string3 << "\nstring4: " << string4 << "\n\n";
16
17    // comparing string1 and string4
18    if ( string1 == string4 )
19       cout << "string1 == string4\n";
20    else // string1 != string4
21    {
22       if ( string1 > string4 )
23          cout << "string1 > string4\n";
```

Fig. 18.2 | Comparing `string`s. (Part 1 of 4.)

```
24          else // string1 < string4
25             cout << "string1 < string4\n";
26       } // end else
27
28       // comparing string1 and string2
29       int result = string1.compare( string2 );
30
31       if ( result == 0 )
32          cout << "string1.compare( string2 ) == 0\n";
33       else // result != 0
34       {
35          if ( result > 0 )
36             cout << "string1.compare( string2 ) > 0\n";
37          else // result < 0
38             cout << "string1.compare( string2 ) < 0\n";
39       } // end else
40
41       // comparing string1 (elements 2-5) and string3 (elements 0-5)
42       result = string1.compare( 2, 5, string3, 0, 5 );
43
44       if ( result == 0 )
45          cout << "string1.compare( 2, 5, string3, 0, 5 ) == 0\n";
46       else // result != 0
47       {
```

Fig. 18.2 | Comparing `string`s. (Part 2 of 4.)

```
48          if ( result > 0 )
49             cout << "string1.compare( 2, 5, string3, 0, 5 ) > 0\n";
50          else // result < 0
51             cout << "string1.compare( 2, 5, string3, 0, 5 ) < 0\n";
52       } // end else
53
54       // comparing string2 and string4
55       result = string4.compare( 0, string2.length(), string2 );
56
57       if ( result == 0 )
58          cout << "string4.compare( 0, string2.length(), "
59             << "string2 ) == 0" << endl;
60       else // result != 0
61       {
62          if ( result > 0 )
63             cout << "string4.compare( 0, string2.length(), "
64                << "string2 ) > 0" << endl;
65          else // result < 0
66             cout << "string4.compare( 0, string2.length(), "
67                << "string2 ) < 0" << endl;
68       } // end else
69
70       // comparing string2 and string4
71       result = string2.compare( 0, 3, string4 );
```

Fig. 18.2 | Comparing `string`s. (Part 3 of 4.)

```
72
73      if ( result == 0 )
74         cout << "string2.compare( 0, 3, string4 ) == 0" << endl;
75      else // result != 0
76      {
77         if ( result > 0 )
78            cout << "string2.compare( 0, 3, string4 ) > 0" << endl;
79         else // result < 0
80            cout << "string2.compare( 0, 3, string4 ) < 0" << endl;
81      } // end else
82   } // end main
```

```
string1: Testing the comparison functions.
string2: Hello
string3: stinger
string4: Hello

string1 > string4
string1.compare( string2 ) > 0
string1.compare( 2, 5, string3, 0, 5 ) == 0
string4.compare( 0, string2.length(), string2 ) == 0
string2.compare( 0, 3, string4 ) < 0
```

Fig. 18.2 | Comparing `string`s. (Part 4 of 4.)

18.3 Comparing `strings` (cont.)

- All the `string` class overloaded relational and equality operator functions return `bool` values.
- Line 29 uses `string` member function `compare` to compare `string1` to `string2`.
  - The function returns `0` if the `string`s are equivalent, a positive number if `string1` is lexicographically greater than `string2` or a negative number if `string1` is lexicographically less than `string2`.
- When we say that a `string` is lexicographically less than another, we mean that the `compare` method uses the numerical values of the characters (see Appendix B, ASCII Character Set) in each `string` to determine that the first `string` is less than the second.

18.3 Comparing strings (cont.)

- Line 42 compares portions of `string1` and `string3` using an overloaded version of member function `compare`.
  - The first two arguments (`2` and `5`) specify the starting subscript and length of the portion of `string1` (`"sting"`) to compare with `string3`.
  - The third argument is the comparison `string`.
  - The last two arguments (`0` and `5`) are the starting subscript and length of the portion of the comparison `string` being compared (also `"sting"`).

18.3 Comparing `strings` (cont.)

- Line 55 uses another overloaded version of function `compare` to compare `string4` and `string2`.
  - The first two arguments are the same—the starting subscript and length.
  - The last argument is the comparison `string`.

18.4 Substrings

- Class `string` provides member function `substr` for retrieving a substring from a `string`.
- The result is a new `string` object that is copied from the source `string`.
- Figure 18.3 demonstrates `substr`.
- The first argument of `substr` specifies the beginning subscript of the desired substring; the second argument specifies the substring's length.

```
1  // Fig. 18.3: Fig18_03.cpp
2  // Demonstrating string member function substr.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string string1( "The airplane landed on time." );
10
11    // retrieve substring "plane" which
12    // begins at subscript 7 and consists of 5 characters
13    cout << string1.substr( 7, 5 ) << endl;
14 } // end main
```

```
plane
```

Fig. 18.3 | Demonstrating `string` member function `substr`.

18.5 Swapping `strings`

- Class `string` provides member function `swap` for swapping `string`s.
- Figure 18.4 swaps two `string`s.
- Lines 9–10 declare and initialize `string`s `first` and `second`.
- Each `string` is then output.
- Line 15 uses `string` member function `swap` to swap the values of `first` and `second`.
- The `string` member function `swap` is useful for implementing programs that sort strings.

```
1  // Fig. 18.4: Fig18_04.cpp
2  // Using the swap function to swap two strings.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string first( "one" );
10    string second( "two" );
11
12    // output strings
13    cout << "Before swap:\n first: " << first << "\nsecond: " << second;
14
15    first.swap( second ); // swap strings
16
17    cout << "\n\nAfter swap:\n first: " << first
18       << "\nsecond: " << second << endl;
19 } // end main
```

Fig. 18.4 | Using function `swap` to swap two `string`s. (Part 1 of 2.)

```
Before swap:
 first: one
second: two

After swap:
 first: two
second: one
```

Fig. 18.4 | Using function `swap` to swap two `string`s. (Part 2 of 2.)

18.6 `string` Characteristics

- Class `string` provides member functions for gathering information about a `string`'s size, length, capacity, maximum length and other characteristics.
- A `string`'s size or length is the number of characters currently stored in the `string`.
- A `string`'s capacity is the number of characters that can be stored in the `string` without allocating more memory.
  - The capacity of a `string` must be at least equal to the current size of the `string`, though it can be greater.
  - The exact capacity of a `string` depends on the implementation.
- The maximum size is the largest possible size a `string` can have.
  - If this value is exceeded, a `length_error` exception is thrown.
- Figure 18.5 demonstrates `string` class member functions for determining various characteristics of `string`s.

```
1  // Fig. 18.5: Fig18_05.cpp
2  // Demonstrating member functions related to size and capacity.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  void printStatistics( const string & );
8
9  int main()
10 {
11    string string1; // empty string
12
13    cout << "Statistics before input:\n" << boolalpha;
14    printStatistics( string1 );
15
16    // read in only "tomato" from "tomato soup"
17    cout << "\n\nEnter a string: ";
18    cin >> string1; // delimited by whitespace
19    cout << "The string entered was: " << string1;
20
21    cout << "\nStatistics after input:\n";
22    printStatistics( string1 );
23
```

Fig. 18.5 | Printing `string` characteristics. (Part 1 of 4.)

```
24      // read in "soup"
25      cin >> string1; // delimited by whitespace
26      cout << "\n\nThe remaining string is: " << string1 << endl;
27      printStatistics( string1 );
28
29      // append 46 characters to string1
30      string1 += "1234567890abcdefghijklmnopqrstuvwxyz1234567890";
31      cout << "\n\nstring1 is now: " << string1 << endl;
32      printStatistics( string1 );
33
34      // add 10 elements to string1
35      string1.resize( string1.length() + 10 );
36      cout << "\n\nStats after resizing by (length + 10):\n";
37      printStatistics( string1 );
38      cout << endl;
39   } // end main
40
41   // display string statistics
42   void printStatistics( const string &stringRef )
43   {
44      cout << "capacity: " << stringRef.capacity() << "\nmax size: "
45         << stringRef.max_size() << "\nsize: " << stringRef.size()
46         << "\nlength: " << stringRef.length()
47         << "\nempty: " << stringRef.empty();
48   } // end printStatistics
```

Fig. 18.5 | Printing `string` characteristics. (Part 2 of 4.)

```
Statistics before input:
capacity: 0
max size: 4294967293
size: 0
length: 0
empty: true

Enter a string: tomato soup
The string entered was: tomato
Statistics after input:
capacity: 15
```

Fig. 18.5 | Printing `string` characteristics. (Part 3 of 4.)

```
max size: 4294967293
size: 6
length: 6
empty: false

The remaining string is: soup
capacity: 15
max size: 4294967293
size: 4
length: 4
empty: false

string1 is now: soup1234567890abcdefghijklmnopqrstuvwxyz1234567890
capacity: 63
max size: 4294967293
size: 50
length: 50
empty: false

Stats after resizing by (length + 10):
capacity: 63
max size: 4294967293
size: 60
length: 60
empty: false
```

Fig. 18.5 | Printing `string` characteristics. (Part 4 of 4.)

18.6 `string` Characteristics (cont.)

- The program declares empty `string` `string1` (line 11) and passes it to function `printStatistics` (line 14).
- Function `printStatistics` (lines 42–48) takes a reference to a `const` `string` as an argument and outputs
  - capacity (using member function `capacity`),
  - maximum size (using member function `max_size`), size (using member function `size`),
  - length (using member function `length`) and
  - whether the `string` is empty (using member function `empty`).
- A size and length of `0` indicate that there are no characters stored in a `string`.
- When the initial capacity is `0` and characters are placed into the `string`, memory is allocated to accommodate the new characters.

Performance Tip 18.1

To minimize the number of times memory is allocated and deallocated, some `string` *class implementations provide a default capacity that is larger than the length of the* `string`.

18.6 `string` Characteristics (cont.)

- Line 30 uses the overloaded `+=` operator to concatenate a 46-character-long string to `string1`.
- The capacity has increased to `63` elements and the length is now `50`.
- Line 35 uses member function `resize` to increase the length of `string1` by `10` characters.
- The additional elements are set to null characters.
- The output shows that the capacity has not changed and the length is now `60`.

18.7 Finding Substrings and Characters in a `string`

- Class `string` provides `const` member functions for finding substrings and characters in a `string`.
- Figure 18.6 demonstrates the find functions.

```
1  // Fig. 18.6: Fig18_06.cpp
2  // Demonstrating the string find member functions.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string string1( "noon is 12 pm; midnight is not." );
10    int location;
11
12    // find "is" at location 5 and 24
13    cout << "Original string:\n" << string1
14       << "\n\n(find) \"is\" was found at: " << string1.find( "is" )
15       << "\n(rfind) \"is\" was found at: " << string1.rfind( "is" );
16
17    // find 'o' at location 1
18    location = string1.find_first_of( "misop" );
19    cout << "\n\n(find_first_of) found '" << string1[ location ]
20       << "' from the group \"misop\" at: " << location;
21
```

Fig. 18.6 | Demonstrating the `string` `find` functions. (Part 1 of 3.)

```
22      // find 'o' at location 29
23      location = string1.find_last_of( "misop" );
24      cout << "\n\n(find_last_of) found '" << string1[ location ]
25         << "' from the group \"misop\" at: " << location;
26
27      // find '1' at location 8
28      location = string1.find_first_not_of( "noi spm" );
29      cout << "\n\n(find_first_not_of) '" << string1[ location ]
30         << "' is not contained in \"noi spm\" and was found at: "
31         << location;
32
33      // find '.' at location 12
34      location = string1.find_first_not_of( "12noi spm" );
35      cout << "\n\n(find_first_not_of) '" << string1[ location ]
36         << "' is not contained in \"12noi spm\" and was "
37         << "found at: " << location << endl;
38
39      // search for characters not in string1
40      location = string1.find_first_not_of(
41         "noon is 12 pm; midnight is not." );
42      cout << "\nfind_first_not_of(\"noon is 12 pm; midnight is not.\")"
43         << " returned: " << location << endl;
44   } // end main
```

Fig. 18.6 | Demonstrating the `string` `find` functions. (Part 2 of 3.)

```
Original string:
noon is 12 pm; midnight is not.

(find) "is" was found at: 5
(rfind) "is" was found at: 24

(find_first_of) found 'o' from the group "misop" at: 1

(find_last_of) found 'o' from the group "misop" at: 29

(find_first_not_of) '1' is not contained in "noi spm" and was found at: 8

(find_first_not_of) '.' is not contained in "12noi spm" and was found at: 12

find_first_not_of("noon is 12 pm; midnight is not.") returned: -1
```

Fig. 18.6 | Demonstrating the `string` `find` functions. (Part 3 of 3.)

18.7 Finding Substrings and Characters in a `string` (cont.)

- Line 14 attempts to find `"is"` in `string1` using function `find`.
  - If `"is"` is found, the subscript of the starting location of that string is returned.
  - If the `string` is not found, the value `string::npos` (a `public` `static` constant defined in class `string`) is returned.
  - This value is returned by the `string` `find`-related functions to indicate that a substring or character was not found in the `string`.

18.7 Finding Substrings and Characters in a `string` (cont.)

- Line 15 uses member function `rfind` to search `string1` backward (i.e., right-to-left).
- If `"is"` is found, the subscript location is returned.
- If the string is not found, `string::npos` is returned.
- [*Note: The rest of the find functions presented in this section return the same type unless otherwise noted.*]

18.7 Finding Substrings and Characters in a `string` (cont.)

- Line 18 uses member function `find_first_of` to locate the first occurrence in `string1` of any character in `"misop"`.
  - The searching is done from the beginning of `string1`.
- Line 23 uses member function `find_last_of` to find the last occurrence in `string1` of any character in `"misop"`.
  - The searching is done from the end of `string1`.
- Line 28 uses member function `find_first_not_of` to find the first character in `string1` not contained in `"noi spm"`.
  - Searching is done from the beginning of `string1`.
- Line 34 uses member function `find_first_not_of` to find the first character not contained in `"12noi spm"`.
  - Searching is done from the end of `string1`.

18.7 Finding Substrings and Characters in a `string` (cont.)

- Lines 40–41 use member function `find_first_not_of` to find the first character not contained in `"noon is 12 pm; midnight is not."`.
  - In this case, the `string` being searched contains every character specified in the string argument.
  - Because a character was not found, `string::npos` (which has the value `-1` in this case) is returned.

18.8 Replacing Characters in a `string`

- Figure 18.7 demonstrates `string` member functions for replacing and erasing characters.
- Line 20 uses `string` member function `erase` to erase everything from (and including) the character in position 62 to the end of `string1`.
- [*Note: Each newline character occupies one element in the `string`.*]

```
1  // Fig. 18.7: Fig18_07.cpp
2  // Demonstrating string member functions erase and replace.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     // compiler concatenates all parts into one string
10    string string1( "The values in any left subtree"
11       "\nare less than the value in the"
12       "\nparent node and the values in"
13       "\nany right subtree are greater"
14       "\nthan the value in the parent node" );
15
16    cout << "Original string:\n" << string1 << endl << endl;
17
18    // remove all characters from (and including) location 62
19    // through the end of string1
20    string1.erase( 62 );
21
22    // output new string
23    cout << "Original string after erase:\n" << string1
24       << "\n\nAfter first replacement:\n";
```

Fig. 18.7 | Demonstrating functions `erase` and `replace`. (Part 1 of 3.)

```
25
26      int position = string1.find( " " ); // find first space
27
28      // replace all spaces with period
29      while ( position != string::npos )
30      {
31         string1.replace( position, 1, "." );
32         position = string1.find( " ", position + 1 );
33      } // end while
34
35      cout << string1 << "\n\nAfter second replacement:\n";
36
37      position = string1.find( "." ); // find first period
38
39      // replace all periods with two semicolons
40      // NOTE: this will overwrite characters
41      while ( position != string::npos )
42      {
43         string1.replace( position, 2, "xxxxx;;yyy", 5, 2 );
44         position = string1.find( ".", position + 1 );
45      } // end while
46
47      cout << string1 << endl;
48   } // end main
```

Fig. 18.7 | Demonstrating functions `erase` and `replace`. (Part 2 of 3.)

```
Original string:
The values in any left subtree
are less than the value in the
parent node and the values in
any right subtree are greater
than the value in the parent node

Original string after erase:
The values in any left subtree
are less than the value in the


After first replacement:
The.values.in.any.left.subtree
are.less.than.the.value.in.the


After second replacement:
The;;alues;;n;;ny;;eft;;ubtree
are;;ess;;han;;he;;alue;;n;;he
```

Fig. 18.7 | Demonstrating functions `erase` and `replace`. (Part 3 of 3.)

18.8 Replacing Characters in a `string` (cont.)

- Lines 26–33 use `find` to locate each occurrence of the space character.
- Each space is then replaced with a period by a call to `string` member function `replace`.
- Function `replace` takes three arguments:
  - the subscript of the character in the `string` at which replacement should begin,
  - the number of characters to replace and
  - the replacement string.

18.8 Replacing Characters in a `string` (cont.)

- Lines 37–45 use function `find` to find every period and another overloaded function `replace` to replace every period and its following character with two semicolons.
- The arguments passed to this version of `replace` are
  - the subscript of the element where the replace operation begins,
  - the number of characters to replace,
  - a replacement character string from which a substring is selected to use as replacement characters,
  - the element in the character string where the replacement substring begins and
  - the number of characters in the replacement character string to use.

18.9 Inserting Characters into a `string`

- Class `string` provides member functions for inserting characters into a `string`.
- Figure 18.8 demonstrates the `string` `insert` capabilities.
- Line 19 uses `string` member function `insert` to insert `string2`'s content before element 10 of `string1`.
- Line 22 uses `insert` to insert `string4` before `string3`'s element 3.
  - The last two arguments specify the starting and last element of `string4` that should be inserted.
  - Using `string::npos` causes the entire `string` to be inserted.

```
1  // Fig. 18.8: Fig18_08.cpp
2  // Demonstrating class string insert member functions.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string string1( "beginning end" );
10    string string2( "middle " );
11    string string3( "12345678" );
12    string string4( "xx" );
13
14    cout << "Initial strings:\nstring1: " << string1
15       << "\nstring2: " << string2 << "\nstring3: " << string3
16       << "\nstring4: " << string4 << "\n\n";
17
18    // insert "middle" at location 10 in string1
19    string1.insert( 10, string2 );
20
21    // insert "xx" at location 3 in string3
22    string3.insert( 3, string4, 0, string::npos );
23
```

Fig. 18.8 | Demonstrating the `string` `insert` member functions. (Part 1 of 2.)

```
24      cout << "Strings after insert:\nstring1: " << string1
25         << "\nstring2: " << string2 << "\nstring3: " << string3
26         << "\nstring4: " << string4 << endl;
27   } // end main
```

```
Initial strings:
string1: beginning end
string2: middle
string3: 12345678
string4: xx

Strings after insert:
string1: beginning middle end
string2: middle
string3: 123xx45678
string4: xx
```

Fig. 18.8 | Demonstrating the `string` `insert` member functions. (Part 2 of 2.)

18.10 Conversion to C-Style Pointer-Based char * Strings

- Class `string` provides member functions for converting `string` class objects to C-style pointer-based strings.
- As mentioned earlier, unlike pointer-based strings, `string`s are not necessarily null terminated.
- These conversion functions are useful when a given function takes a pointer-based string as an argument.
- Figure 18.9 demonstrates conversion of `string`s to pointer-based strings.
- The `string string1` is initialized to `"STRINGS"`, `ptr1` is initialized to `0` and `length` is initialized to the length of `string1`.
- Memory of sufficient size to hold a pointer-based string equivalent of `string string1` is allocated dynamically and attached to `char` pointer `ptr2`.

```
1  // Fig. 18.9: Fig18_09.cpp
2  // Converting to C-style strings.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string string1( "STRINGS" ); // string constructor with char* arg
10    const char *ptr1 = 0; // initialize *ptr1
11    int length = string1.length();
12    char *ptr2 = new char[ length + 1 ]; // including null
13
14    // copy characters from string1 into allocated memory
15    string1.copy( ptr2, length, 0 ); // copy string1 to ptr2 char*
16    ptr2[ length ] = '\0'; // add null terminator
17
18    cout << "string string1 is " << string1
19       << "\nstring1 converted to a C-Style string is "
20       << string1.c_str()   << "\nptr1 is ";
21
```

Fig. 18.9 | Converting `string`s to C-style strings and character arrays. (Part 1 of 2.)

```
22      // Assign to pointer ptr1 the const char * returned by
23      // function data(). NOTE: this is a potentially dangerous
24      // assignment. If string1 is modified, pointer ptr1 can
25      // become invalid.
26      ptr1 = string1.data();
27
28      // output each character using pointer
29      for ( int i = 0; i < length; i++ )
30         cout << *( ptr1 + i ); // use pointer arithmetic
31
32      cout << "\nptr2 is " << ptr2 << endl;
33      delete [] ptr2; // reclaim dynamically allocated memory
34   } // end main
```

```
string string1 is STRINGS
string1 converted to a C-Style string is STRINGS
ptr1 is STRINGS
ptr2 is STRINGS
```

Fig. 18.9 | Converting `string`s to C-style strings and character arrays. (Part 2 of 2.)

18.11 Conversion to C-Style Pointer-Based `char *` Strings (cont.)

- Line 15 uses `string` member function `copy` to copy object `string1` into the `char` array pointed to by `ptr2`.
- Line 16 manually places a terminating null character in the array pointed to by `ptr2`.
- Line 20 uses function `c_str` to obtain a `const char *` that points to a null terminated C-style string with the same content as `string1`.
- The pointer is passed to the stream insertion operator for output.

18.12 Conversion to C-Style Pointer-Based `char *` Strings (cont.)

- Line 26 assigns the `const char * ptr1` a pointer returned by class `string` member function `data`.
- This member function returns a non-null-terminated C-style character array.
- We do not modify `string string1` in this example.
- If `string1` were to be modified (e.g., the `string`'s dynamic memory changes its address due to a member function call such as `string1.insert( 0, "abcd" );`), `ptr1` could become invalid—which could lead to unpredictable results.
- Lines 29–30 use pointer arithmetic to output the character array pointed to by `ptr1`.
- In lines 32–33, the C-style string pointed to by `ptr2` is output and the memory allocated for `ptr2` is `delete`d to avoid a memory leak.

Common Programming Error 18.4

Not terminating the character array returned by `data` *with a null character can lead to execution-time errors.*

Good Programming Practice 18.1

Whenever possible, use the more robust `string` *class objects rather than C-style pointer-based strings.*

18.13 Iterators

- Class `string` provides iterators for forward and backward traversal of `string`s.
- Iterators provide access to individual characters with syntax that is similar to pointer operations.
- Iterators are not range checked.
- In this section we provide “mechanical examples” to demonstrate the use of iterators.
- We discuss more robust uses of iterators in Chapter 22, Standard Template Library (STL).

18.13 Iterators (cont.)

- Figure 18.10 demonstrates iterators.
- Lines 9–10 declare `string string1` and `string::const_iterator iterator1`.
- A `const_iterator` is an iterator that cannot modify the `string`—in this case the `string` through which it's iterating.
- Iterator `iterator1` is initialized to the beginning of `string1` with the `string` class member function `begin`.
- Two versions of `begin` exist—one that returns an `iterator` for iterating through a non-`const string` and a `const` version that returns a `const_iterator` for iterating through a `const string`.

```
1  // Fig. 18.10: Fig18_10.cpp
2  // Using an iterator to output a string.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string string1( "Testing iterators" );
10    string::const_iterator iterator1 = string1.begin();
11
12    cout << "string1 = " << string1
13       << "\n(Using iterator iterator1) string1 is: ";
14
15    // iterate through string
16    while ( iterator1 != string1.end() )
17    {
18       cout << *iterator1; // dereference iterator to get char
19       iterator1++; // advance iterator to next char
20    } // end while
21
22    cout << endl;
23 } // end main
```

Fig. 18.10 | Using an iterator to output a `string`. (Part 1 of 2.)

```
string1 = Testing iterators
(Using iterator iterator1) string1 is: Testing iterators
```

Fig. 18.10 | Using an iterator to output a `string`. (Part 2 of 2.)

18.13 Iterators (cont.)

- Lines 16–20 use iterator `iterator1` to "walk through" `string1`.
- Class `string` member function `end` returns an `iterator` (or a `const_iterator`) for the position past the last element of `string1`.
- Each element is printed by dereferencing the iterator much as you'd dereference a pointer, and the iterator is advanced one position using operator ++.

18.13 Iterators (cont.)

- Class `string` provides member functions `rend` and `rbegin` for accessing individual `string` characters in reverse from the end of a `string` toward the beginning.
- Member functions `rend` and `rbegin` return `reverse_iterators` or `const_reverse_iterators` (based on whether the `string` is non-`const` or `const`).
- We'll use iterators and reverse iterators more in Chapter 22.

Error-Prevention Tip 18.1

Use `string` *member function* `at` *(rather than iterators) when you want the benefit of range checking.*

Good Programming Practice 18.2

When the operations involving the iterator should not modify the data being processed, use a `const_iterator`. *This is another example of employing the principle of least privilege.*

18.14 String Stream Processing

- In addition to standard stream I/O and file stream I/O, C++ stream I/O includes capabilities for inputting from, and outputting to, `string`s in memory.
- These capabilities often are referred to as in-memory I/O or string stream processing.
- Input from a `string` is supported by class `istringstream`.
- Output to a `string` is supported by class `ostringstream`.

18.14 String Stream Processing (cont.)

- The class names `istringstream` and `ostringstream` are actually aliases defined by the `typedef`s
  - `typedef basic_istringstream< char > istringstream;`
    `typedef basic_ostringstream< char > ostringstream;`
- Class templates `basic_istringstream` and `basic_ostringstream` provide the same functionality as classes `istream` and `ostream` plus other member functions specific to in-memory formatting.
- Programs that use in-memory formatting must include the `<sstream>` and `<iostream>` header files.

18.14 String Stream Processing (cont.)

- One application of these techniques is data validation.
- A program can read an entire line at a time from the input stream into a `string`.
- Next, a validation routine can scrutinize the contents of the `string` and correct (or repair) the data, if necessary.
- Then the program can proceed to input from the `string`, knowing that the input data is in the proper format.
- Outputting to a `string` is a nice way to take advantage of the powerful output formatting capabilities of C++ streams.
- Data can be prepared in a `string` then written to a disk file.
- An `ostringstream` object uses a `string` object to store the output data.

18.14 String Stream Processing (cont.)

- The `str` member function of class `ostringstream` returns a copy of that `string`.
- Figure 18.11 demonstrates an `ostringstream` object.
- The program creates `ostringstream` object `outputString` (line 10) and uses the stream insertion operator to output a series of `string`s and numerical values to the object.

```
1  // Fig. 18.11: Fig18_11.cpp
2  // Using an ostringstream object.
3  #include <iostream>
4  #include <string>
5  #include <sstream> // header file for string stream processing
6  using namespace std;
7
8  int main()
9  {
10    ostringstream outputString; // create ostringstream instance
11
12    string string1( "Output of several data types " );
13    string string2( "to an ostringstream object:" );
14    string string3( "\n        double: " );
15    string string4( "\n           int: " );
16    string string5( "\naddress of int: " );
17
18    double double1 = 123.4567;
19    int integer = 22;
20
21    // output strings, double and int to ostringstream outputString
22    outputString << string1 << string2 << string3 << double1
23       << string4 << integer << string5 << &integer;
```

Fig. 18.11 | Using an `ostringstream` object. (Part 1 of 2.)

```
24
25      // call str to obtain string contents of the ostringstream
26      cout << "outputString contains:\n" << outputString.str();
27
28      // add additional characters and call str to output string
29      outputString << "\nmore characters added";
30      cout << "\n\nafter additional stream insertions,\n"
31         << "outputString contains:\n" << outputString.str() << endl;
32   } // end main
```

```
outputString contains:
Output of several data types to an ostringstream object:
        double: 123.457
           int: 22
address of int: 0012F540

after additional stream insertions,
outputString contains:
Output of several data types to an ostringstream object:
        double: 123.457
           int: 22
address of int: 0012F540
more characters added
```

Fig. 18.11 | Using an `ostringstream` object. (Part 2 of 2.)

18.14 String Stream Processing (cont.)

- Lines 22–23 output `string` `string1`, `string` `string2`, `string` `string3`, `double` `double1`, `string` `string4`, `int` `integer`, `string` `string5` and the address of `int` `integer`—all to `outputString` in memory.
- Line 26 uses the stream insertion operator and the call `outputString.str()` to display a copy of the `string` created in lines 22–23.
- Line 29 demonstrates that more data can be appended to the `string` in memory by simply issuing another stream insertion operation to `outputString`.

18.14 String Stream Processing (cont.)

- An `istringstream` object inputs data from a `string` in memory to program variables.
- Data is stored in an `istringstream` object as characters.
- Input from the `istringstream` object works identically to input from any file.
- The end of the `string` is interpreted by the `istringstream` object as end-of-file.
- Figure 18.12 demonstrates input from an `istringstream` object.

18.14 String Stream Processing (cont.)

- Lines 10–11 create `string input` containing the data and `istringstream` object `inputString` constructed to contain the data in `string input`.
- The `string input` contains the data
  - `Input test 123 4.7 A`
- which, when read as input to the program, consist of two strings (`"Input"` and `"test"`), an `int` (`123`), a `double` (`4.7`) and a `char` (`'A'`).
- These characters are extracted to variables `string1`, `string2`, `integer`, `double1` and `character` in line 18.

```
1  // Fig. 18.12: Fig18_12.cpp
2  // Demonstrating input from an istringstream object.
3  #include <iostream>
4  #include <string>
5  #include <sstream>
6  using namespace std;
7
8  int main()
9  {
10    string input( "Input test 123 4.7 A" );
11    istringstream inputString( input );
12    string string1;
13    string string2;
14    int integer;
15    double double1;
16    char character;
17
18    inputString >> string1 >> string2 >> integer >> double1 >> character;
19
20    cout << "The following items were extracted\n"
21       << "from the istringstream object:" << "\nstring: " << string1
22       << "\nstring: " << string2 << "\n   int: " << integer
23       << "\ndouble: " << double1 << "\n  char: " << character;
```

Fig. 18.12 | Demonstrating input from an `istringstream` object. (Part 1 of 2.)

```
24
25      // attempt to read from empty stream
26      long value;
27      inputString >> value;
28
29      // test stream results
30      if ( inputString.good() )
31         cout << "\n\nlong value is: " << value << endl;
32      else
33         cout << "\n\ninputString is empty" << endl;
34   } // end main
```

```
The following items were extracted
from the istringstream object:
string: Input
string: test
   int: 123
double: 4.7
  char: A

inputString is empty
```

Fig. 18.12 | Demonstrating input from an `istringstream` object. (Part 2 of 2.)

18.14 String Stream Processing (cont.)

- The program attempts to read from `inputString` again in line 27.
- The `if` condition in line 30 uses function `good` (Section 15.8) to test if any data remains.
- Because no data remains, the function returns `false` and the `else` part of the `if`…`else` statement is executed.