18

Class `string` and String Stream Processing

The difference between the almost-right word and the right word is really a large matter — it's the difference between the lightning bug and the lightning.

— Mark Twain

I have made this letter longer than usual, because I lack the time to make it short.

— Blaise Pascal

Mum's the word.

— Miguel de Cervantes

Suit the action to the word, the word to the action; with this special observance, that you o'erstep not the modesty of nature.

— William Shakespeare

OBJECTIVES

In this chapter you will learn:

- To use class `string` from the C++ Standard Library to treat `string`s as full-fledged objects.
- To assign, concatenate, compare, search and swap `string`s.
- To determine `string` characteristics.
- To find, replace and insert characters in a `string`.
- 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.

Outline

18.1 Introduction

- C++ class template `basic_string`
  - Provides typical string-manipulation operations
  - Defined in `namespace std`
  - `typedef`s
    - For `char`
      - `typedef basic_string< char > string;`
    - Also provides one for `wchar_t`

18.1 Introduction (Cont.)

- `string` object
  - Initialization
    - `string empty();`
      - Creates an empty `string` containing no characters
    - `string text( "hello" );`
      - Creates a `string` containing the characters `"hello"`
    - `string name( 8, 'x' );`
      - Creates a `string` containing eight `'x'` characters
    - `string month = "March";`
      - Implicitly performs `string month( "March" );`

18.1 Introduction (Cont.)

- `string` object (Cont.)
  - No conversion from `int` or `char` in a `string` definition
    - Examples (produce syntax errors)
      - `string error1 = 'c';`
      - `string error2( 'u' );`
      - `string error3 = 22;`
      - `string error4( 8 );`
  - Assigning a single character to a `string` object is allowed
    - Example
      - `string1 = 'n';`

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.)

- `string` object (Cont.)
  - Member functions `length` and `size`
    - Return the length of the `string`
  - The subscript operator `[]`
    - Used to access and modify individual characters
    - First subscript is `0`, last subscript is `length() - 1`

18.1 Introduction (Cont.)

- `string` object (Cont.)
  - Stream extraction operator (>>)
    - Example
      - `cin >> stringObject;`
    - Input is delimited by white-space characters
  - Function `getline` is overloaded for `string`s
    - Example
      - `getline( cin, string1 );`
    - Input is delimited by a newline (`'\n'`);

18.2 `string` Assignment and Concatenation

- **Member function `assign`**
  - **Copies the contents of a `string` into another `string`**
  - **Single-argument version**
    - **Copies contents of the `string` argument into the current `string`**
  - **Three-argument version**
    - **Copies a specified range of characters**
    - **Example**
      - `targetString.assign( sourceString, start, numberOfCharacters );`

Outline

Fig18_01.cpp

(1 of 3)

```
// Fig. 18.1: Fig18_01.cpp
// Demonstrating string assignment and concatenation.
#include <iostream>
using std::cout;
using std::endl;

#include <string>
using std::string;

int main()
{
   string string1( "cat" );
   string string2;
   string string3;

   string2 = string1; // assign string1 to string2
   string3.assign( string1 ); // assign string1 to string3
   cout << "string1: " << string1 << "\nstring2: " << string2
      << "\nstring3: " << string3 << "\n\n";

   // modify string2 and string3
   string2[ 0 ] = string3[ 2 ] = 'r';

   cout << "After modification of string2 and string3:\n'
      << string1 << "\nstring2: " << string2 << "\nstring
      
   // demonstrating member function at
   for ( int i = 0; i < string3.length(); i++ )
      cout << string3.at( i );
```

Assign the value of **string1** to **string2** with the assignment operator

Copy **string1** into **string3** with the **assign** member function

Use the subscript operator to assign to individual characters

Use member functions **length** and **at** to output the contents of **string3** one character at a time

```cpp

   // declare string4 and string5
   string string4( string1 + "apult" ); // concatenation
   string string5;

   // overloaded +=
   string3 += "pet"; // create "carpet"
   string1.append( "acomb" ); // create "catacomb"

   // append subscript locations 4 through end of string1 to
   // create string "comb" (string5 was initially empty)
   string5.append( string1, 4, string1.length() - 4 );

   cout << "\n\nAfter concatenation:\nstring1: " << string1
      << "\nstring2: " << string2 << "\nstring3: " << string3
      << "\nstring4: " << string4 << "\nstring5: " << string5 << endl;
   return 0;
} // end main
```

Initialize **string4** to the result of concatenating **string1** and **"apult"** using the addition operator **+**

Concatenate **string3** and **"pet"** using the addition assignment operator **+=**

(2 of 3)

Concatenate **string1** and **"acomb"**

Append the string **"comb"** (the characters from subscript **4** to the end of **string1**) to empty **string string5**

Outline

```
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
```

`Fig18_01.cpp`

(3 of 3)

18.2 `string` Assignment and Concatenation (Cont.)

- **Member function `at`**
  - **Allows access to individual characters**
    - **Much like the subscript operator does**
  - **Provides checked access (or range checking)**
    - **Going past the end of the `string` throws an `out_of_range` exception**
    - **Subscript operator does not provide checked access**

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` concatenation
  - Addition operator and addition assignment operator
    - Overloaded for `string` concatenation
  - Member function `append`
    - Single-argument version
      - Concatenates contents of the `string` argument to end of the current `string`
    - Three-argument version
      - Concatenates specified range of characters from the `string` argument to end of the current `string`

18.3 Comparing `strings`

- **Overloaded comparison operators**
  - Operators ==, `!`=, <, >, <=, >= are overloaded for `string`s
    - All such operators return `bool` values
- **Member function `compare`**
  - Compares the values of two `strings`
    - Returns `0` if the `string`s are equivalent
    - Returns positive number if the current `string` is lexicographically greater than the argument `string`
    - Returns negative number if the current `string` is lexicographically less than the argument `string`

18.3 Comparing `strings` (Cont.)

- Member function `compare` (Cont.)
  - Overloaded versions
    - With five arguments
      - First two arguments specify starting subscript and length in the current `string`
      - Third argument specifies the comparison `string`
      - Last two arguments specify starting subscript and length in the comparison `string`
    - With three arguments
      - First two arguments specify starting subscript and length in the current `string`
      - Third argument specifies the comparison `string`

Outline

Fig18_02.cpp

(1 of 4)

```
// Fig. 18.2: Fig18_02.cpp
// Demonstrating string comparison capabilities.
#include <iostream>
using std::cout;
using std::endl;

#include <string>
using std::string;

int main()
{
   string string1( "Testing the comparison functions." );
   string string2( "Hello" );
   string string3( "stinger" );
   string string4( string2 );

   cout << "string1: " << string1 << "\nstring2: " << string2
      << "\nstring3: " << string3 << "\nstring4: " << string4 << "\n\n";

   // comparing string1 and string4
   if ( string1 == string4 )
      cout << "string1 == string4\n";
   else // string1 != string4
   {
      if ( string1 > string4 )
         cout << "string1 > string4\n";
      else // string1 < string4
         cout << "string1 < string4\n";
   } // end else
```

Test **string1** against **string4** for equality using the overloaded equality operator

Test **string1** against **string4** using the overloaded greater-than operator

Outline

```

   // comparing string1 and string2
   int result = string1.compare( string2 );

   if ( result == 0 )
      cout << "string1.compare( string2 ) == 0\n";
   else // result != 0
   {
      if ( result > 0 )
         cout << "string1.compare( string2 ) > 0\n";
      else // result < 0
         cout << "string1.compare( string2 ) < 0\n";
   } // end else

   // comparing string1 (elements 2-5) and string3 (elemen
   result = string1.compare( 2, 5, string3, 0, 5 );

   if ( result == 0 )
      cout << "string1.compare( 2, 5, string3, 0, 5 ) == 0\n";
   else // result != 0
   {
      if ( result > 0 )
         cout << "string1.compare( 2, 5, string3, 0, 5 ) > 0\n";
      else // result < 0
         cout << "string1.compare( 2, 5, string3, 0, 5 ) < 0\n";
   } // end else
```

Compare **`string1`** to **`string2`**

Compare **`"sting"`** (from **`string1`**) to **`"sting"`** (from **`string3`**)

`Fig18_02.cpp`

(2 of 4)

Outline

_02.cpp

(3 of 4)

```

   // comparing string2 and string4
   result = string4.compare( 0, string2.length(), string2 );

   if ( result == 0 )
      cout << "string4.compare( 0, string2.length(), "
         << "string2 ) == 0" << endl;
   else // result != 0
   {
      if ( result > 0 )
         cout << "string4.compare( 0, string2.length(), "
            << "string2 ) > 0" << endl;
      else // result < 0
         cout << "string4.compare( 0, string2.length(), "
            << "string2 ) < 0" << endl;
   } // end else
```

Compare "**Hello**" (from **string4**) to **string2**

Outline

```
   // comparing string2 and string4
   result = string2.compare( 0, 3, string4 );

   if ( result == 0 )
      cout << "string2.compare( 0, 3, string4 ) == 0" << endl;
   else // result != 0
   {
      if ( result > 0 )
         cout << "string2.compare( 0, 3, string4 ) > 0" << endl;
      else // result < 0
         cout << "string2.compare( 0, 3, string4 ) < 0" << endl;
   } // end else

   return 0;
} // end main
```

Compare **"Hel"** (from **string2**) to **string4**

```
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
```

Fig18_02.cpp

(4 of 4)

18.4 Substrings

- Member function `substr`
  - Retrieves a substring from a `string`
    - Returns a new `string` object copied from the source `string`
  - First argument
    - Specifies beginning subscript of desired substring
  - Second argument
    - Specifies length of desired substring

Outline

Fig18_03.cpp

(1 of 1)

```cpp
// Fig. 18.3: Fig18_03.cpp
// Demonstrating string member function substr.
#include <iostream>
using std::cout;
using std::endl;

#include <string>
using std::string;

int main()
{
   string string1( "The airplane landed on time." );

   // retrieve substring "plane" which
   // begins at subscript 7 and consists of 5 elements
   cout << string1.substr( 7, 5 ) << endl;
   return 0;
} // end main
```

Retrieve a substring from **`string1`**

```
plane
```

18.5 Swapping `strings`

- **Member function `swap`**
  - **Swaps contents of the current `string` and the argument `string`**
  - **Useful for implementing programs that sort strings**

Outline

Fig18_04.cpp

(1 of 1)

```
// Fig. 18.4: Fig18_04.cpp
// Using the swap function to swap two strings.
#include <iostream>
using std::cout;
using std::endl;

#include <string>
using std::string;

int main()
{
   string first( "one" );
   string second( "two" );

   // output strings
   cout << "Before swap:\n first: " << first << "\nsecond: " << second;

   first.swap( second ); // swap strings

   cout << "\n\nAfter swap:\n first: " << first
      << "\nsecond: " << second << endl;
   return 0;
} // end main
```

Swap the values of **`first`** and **`second`**

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

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

18.6 `string` Characteristics

- Characteristics of `string`s
  - Capacity
    - Number of characters that can be stored without allocating more memory
      - Must be at least equal to the size, can be greater
      - Depends on the implementation
    - Returned by member function `capacity`
  - Maximum size
    - Largest possible size a `string` can have
      - If exceeded, a `length_error` exception is thrown
    - Returned by member function `max_size`

Outline

```
// Fig. 18.5: Fig18_05.cpp
// Demonstrating member functions related to size and capacity.
#include <iostream>
using std::cout;
using std::endl;
using std::cin;
using std::boolalpha;

#include <string>
using std::string;

void printStatistics( const string & );

int main()
{
   string string1;

   cout << "Statistics before input:\n" << boolalpha;
   printStatistics( string1 );

   // read in only "tomato" from "tomato soup"
   cout << "\n\nEnter a string: ";
   cin >> string1; // delimited by whitespace
   cout << "The string entered was: " << string1;

   cout << "\nStatistics after input:\n";
   printStatistics( string1 );
```

Fig18_05.cpp

(1 of 4)

Outline

(2 of 4)

```

   // read in "soup"
   cin >> string1; // delimited by whitespace
   cout << "\n\nThe remaining string is: " << string1 << endl;
   printStatistics( string1 );

   // append 46 characters to string1
   string1 += "1234567890abcdefghijklmnopqrstuvwxyz1234567890";
   cout << "\n\nstring1 is now: " << string1 << endl;
   printStatistics( string1 );

   // add 10 elements to string1
   string1.resize( string1.length() + 10 );
   cout << "\n\nStats after resizing by (length + 10):\n";
   printStatistics( string1 );

   cout << endl;
   return 0;
} // end main

// display string statistics
void printStatistics( const string &stringRef )
{
   cout << "capacity: " << stringRef.capacity() << "\nmax size: "
      << stringRef.max_size() << "\nsize: " << stringRef.size()
      << "\nlength: " << stringRef.length()
      << "\nempty: " << stringRef.empty();
} // end printStatistics
```

Use the overloaded **+=** operator to concatenate a 46-character-long string to **string1**

Increase the length of **string1** by 10 characters

Output the capacity, maximum size, size, length and whether the **string** is empty

Outline

Fig18_05.cpp

(3 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
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
```

(Continued at top of next slide…)

Outline

(…Continued from bottom of previous slide)

```
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
```

`Fig18_05.cpp`

(4 of 4)

18.6 `string` Characteristics (Cont.)

- Member function `empty`
  - Returns `true` if the `string` is empty
- Member function `resize`
  - Changes the length of the current `string`
    - Additional elements are set to null characters

Performance Tip 18.1

To minimize the number of times memory is allocated and deallocated, some `string` class implementations provide a default capacity above and beyond the length of the `string`.

18.7 Finding Strings and Characters in a `string`

- Member function `find`
  - Attempts to find specified string in the current `string`
    - Returns starting location of the string if found
    - Returns the value `string::npos` otherwise
      - All `string` `find`-related functions return this `const static` value to indicate the target was not found
- Member function `rfind`
  - Searches current `string` backward (right-to-left) for the specified string
    - If the string is found, its subscript location is returned

Outline

Fig18_06.cpp

(1 of 3)

```cpp
// Fig. 18.6: Fig18_06.cpp
// Demonstrating the string find member functions.
#include <iostream>
using std::cout;
using std::endl;

#include <string>
using std::string;

int main()
{
   string string1( "noon is 12 pm; midnight is not." );
   int location;

   // find "is" at location 5 and 25
   cout << "Original string:\n" << string1
      << "\n\n(find) \"is\" was found at: " << string1.find( "is" )
      << "\n(rfind) \"is\" was found at: " << string1.rfind( "is" );

   // find 'o' at location 1
   location = string1.find_first_of( "misop" );
   cout << "\n\n(find_first_of) found '" << string1[ location ]
      << "' from the group \"misop\" at: " << location;

   // find 'o' at location 29
   location = string1.find_last_of( "misop" );
   cout << "\n\n(find_last_of) found '" << string1[ location ]
      << "' from the group \"misop\" at: " << location;
```

Attempt to find **"is"** in **string1** using function **find**

Search **string1** backward for **"is"**

Locate the first occurrence in **string1** of any character in **"misop"**

Find the last occurrence in **string1** of any character in **"misop"**

Outline

```

   // find '1' at location 8
   location = string1.find_first_not_of( "noi spm" );
   cout << "\n\n(find_first_not_of) '" << string1[ location ]
      << "' is not contained in \"noi spm\" and was found at:"
      << location;

   // find '.' at location 12
   location = string1.find_first_not_of( "12noi spm" );
   cout << "\n\n(find_first_not_of) '" << string1[ location ]
      << "' is not contained in \"12noi spm\" and was "
      << "found at:" << location << endl;

   // search for characters not in string1
   location = string1.find_first_not_of(
      "noon is 12 pm; midnight is not." );
   cout << "\nfind_first_not_of(\"noon is 12 pm; midnight is not.\")"
      << " returned: " << location << endl;
   return 0;
} // end main
```

Find the first character in **`string1`** not contained in the string argument

(2 of 3)

`string1` contains only characters specified in the string argument, so **`string::npos`** is returned

Outline

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

(find) "is" was found at: 5
(rfind) "is" was found at: 25

(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
```

`Fig18_06.cpp`

(3 of 3)

18.7 Finding Strings and Characters in a `string` (Cont.)

- Member function `find_first_of`
  - Locates first occurrence in the current `string` of any character in the specified string
- Member function `find_last_of`
  - Locates last occurrence in the current `string` of any character in the specified string
- Member function `find_first_not_of`
  - Locates first occurrence in the current `string` of any character not contained in the specified string

18.8 Replacing Characters in a `string`

- Member function `erase`
  - One-argument version
    - Erases everything from (and including) the specified character position to the end of the `string`
- Member function `replace`
  - Three-argument version
    - Replaces characters in the range specified by the first two arguments with the specified string (third argument)
  - Five-argument version
    - Replaces characters in the range specified by the first two arguments with characters from the range in the specified string (third argument) specified by the last two arguments

Outline

Fig18_07.cpp

(1 of 3)

```
// Fig. 18.7: Fig18_07.cpp
// Demonstrating string member functions erase and replace.
#include <iostream>
using std::cout;
using std::endl;

#include <string>
using std::string;

int main()
{
   // compiler concatenates all parts into one string
   string string1( "The values in any left subtree"
      "\nare less than the value in the"
      "\nparent node and the values in"
      "\nany right subtree are greater"
      "\nthan the value in the parent node" );

   cout << "Original string:\n" << string1 << endl << endl;

   // remove all characters from (and including) location 62
   // through the end of string1
   string1.erase( 62 );

   // output new string
   cout << "Original string after erase:\n" << string1
      << "\n\nAfter first replacement:\n";

   int position = string1.find( " " ); // find first space
```

Erase characters from **string1** starting at position 62

Fig18_07.cpp

```

   // replace all spaces with period
   while ( position != string::npos )
   {
      string1.replace( position, 1, "." );
      position = string1.find( " ", position + 1 );
   } // end while

   cout << string1 << "\n\nAfter second replacement:\n";

   position = string1.find( "." ); // find first period

   // replace all periods with two semicolons
   // NOTE: this will overwrite characters
   while ( position != string::npos )
   {
      string1.replace( position, 2, "xxxxx;;yyy", 5, 2 );
      position = string1.find( ".", position + 1 );
   } // end while

   cout << string1 << endl;
   return 0;
} // end main
```

Locate each occurrence of the space character and replace it with a period

Continue searching for the next space character at **`position + 1`**

Find every period and replace it and the neext character with two semicolons

Outline

```
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
```

Fig18_07.cpp

(3 of 3)

18.9 Inserting Characters into a `string`

- Member function `insert`
  - For inserting characters into a `string`
    - Two-argument version
      - First argument specifies insertion location
      - Second argument specifies `string` to insert
    - Four-argument version
      - First argument specifies insertion location
      - Second argument specifies `string` to insert from
      - Third and fourth arguments specify starting and last element in source `string` to be inserted
        - Using `string::npos` causes the entire `string` to be inserted

Outline

Fig18_08.cpp

(1 of 2)

```cpp
// Fig. 18.8: Fig18_08.cpp
// Demonstrating class string insert member functions.
#include <iostream>
using std::cout;
using std::endl;

#include <string>
using std::string;

int main()
{
   string string1( "beginning end" );
   string string2( "middle " );
   string string3( "12345678" );
   string string4( "xx" );

   cout << "Initial strings:\nstring1: " << string1
      << "\nstring2: " << string2 << "\nstring3: " << string3
      << "\nstring4: " << string4 << "\n\n";
```

Outline

Fig18_08.cpp

```
   // insert "middle" at location 10 in string1
   string1.insert( 10, string2 );

   // insert "xx" at location 3 in string3
   string3.insert( 3, string4, 0, string::npos );

   cout << "Strings after insert:\nstring1: " << string1
      << "\nstring2: " << string2 << "\nstring3: " << string3
      << "\nstring4: " << string4 << endl;
   return 0;
} // end main
```

Insert **string2**'s contents before element 10 of **string1**

Insert **string4** before **string3**'s element 3

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

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

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

- Member function `copy`
  - Copies current `string` into the specified `char` array
    - Must manually add terminating null character afterward
- Member function `c_str`
  - Returns a `const char *` containing a copy of the current `string`
    - Automatically adds terminating null character
- Member function `data`
  - Returns non-null-terminated C-style character array
    - If original `string` object is later modified, this pointer becomes invalid

Outline

Fig18_09.cpp

(1 of 2)

```
// Fig. 18.9: Fig18_09.cpp
// Converting to C-style strings.
#include <iostream>
using std::cout;
using std::endl;

#include <string>
using std::string;

int main()
{
   string string1( "STRINGS" ); // string constructor with char* arg
   const char *ptr1 = 0; // initialize *ptr1
   int length = string1.length();
   char *ptr2 = new char[ length + 1 ]; // including null

   // copy characters from string1 into allocated memory
   string1.copy( ptr2, length, 0 ); // copy string1 to ptr2 char*
   ptr2[ length ] = '\0'; // add null terminator
```

Copy object **`string1`** into the **`char`** array pointed to by **`ptr2`**

Manually place a terminating null character at the end of the array

Outline

```
   cout << "string string1 is " << string1
      << "\nstring1 converted to a C-Style string is "
      << string1.c_str() << "\nptr1 is ";

   // Assign to pointer ptr1 the const char * returned by
   // function data(). NOTE: this is a potentially dangerou
   // assignment. If string1 is modified, pointer ptr1 can
   // become invalid.
   ptr1 = string1.data();

   // output each character using pointer
   for ( int i = 0; i < length; i++ )
      cout << *( ptr1 + i ); // use pointer arithmetic

   cout << "\nptr2 is " << ptr2 << endl;
   delete [] ptr2; // reclaim dynamically allocated memory
   return 0;
} // end main
```

Output the null-terminated array pointed to by the `const char *` returned by member function `c_str`

(2 of 2)

Assign the **`const char *`** **`ptr1`** a pointer returned by member function **`data`**

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

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.

18.11 Iterators

- `string` iterators
  - Provide access to individual characters
    - Syntax similar to pointers
  - `string::iterator` and `string::const_iterator`
    - A `const_iterator` cannot modify the `string`
    - `string` member function `begin`
      - Returns `iterator` positioned at the beginning of the `string`
      - Another version returns `const_iterator`s for `const` `string`s
    - `string` member function `end`
      - Returns `iterator` (or `const_iterator`) positioned after the last element of the `string`

Outline

Fig18_10.cpp

(1 of 1)

```
// Fig. 18.10: Fig18_10.cpp
// Using an iterator to output a string.
#include <iostream>
using std::cout;
using std::endl;

#include <string>
using std::string;

int main()
{
   string string1( "Testing iterators" );
   string::const_iterator iterator1 = string1.begin();

   cout << "string1 = " << string1
      << "\n(Using iterator iterator1) string1 is: ";

   // iterate through string
   while ( iterator1 != string1.end() )
   {
      cout << *iterator1; // dereference iterator to get char
      iterator1++; // advance iterator to next char
   } // end while

   cout << endl;
   return 0;
} // end main
```

const_iterator iterator1 is initialized to the beginning of **string1**

Use iterator **iterator1** to "walk through" **string1**

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

18.11 Iterators (Cont.)

- `string` iterators (Cont.)
  - Using iterators
    - Dereference iterator to access individual characters
    - Use operator ++ to advance iterator one position
  - `reverse_iterator` and `const_reverse_iterator`
    - Used for reverse traversal of `string`s (from the end toward the beginning)
    - `string` member functions `rend` and `rbegin`
      - Return `reverse_iterators and const_reverse_iterators`

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.

18.12 String Stream Processing

- String stream processing (a.k.a. in-memory I/O)
  - Enables inputting from, and outputting to, `string`s in memory
  - Class `istringstream`
    - A `typedef` for `basic_istringstream< char >`
    - Supports input from a `string`
      - Provides same functionality as `istream`
  - Class `ostringstream`
    - A `typedef` for `basic_ostringstream< char >`
    - Supports output to a `string`
      - Provides same functionality as `ostream`
  - Program must include `<sstream>` and `<iostream>`

18.12 String Stream Processing (Cont.)

- **Application of string stream processing**
  - Data validation
    - Read an entire line from the input stream into a `string`
    - Scrutinize and repair contents of the `string`
    - Input from the `string` to program variables
  - Preserving the screen image
    - Data can be prepared in a `string`
      - Mimicking the edited screen format
    - The `string` could then be written to a disk file

18.12 String Stream Processing (Cont.)

- `ostringstream` object
  - Uses a `string` to store output data
    - Member function `str` returns copy of that `string`
  - Data can be appended to the `string` in memory by using stream insertion operator
- `istringstream` object
  - Inputs data from a `string` in memory to program variables
    - Input works identically to input from files
      - End of the `string` is interpreted as end-of-file
  - Member function `good` returns `true` if any data remains

Outline

Fig18_11.cpp

(1 of 2)

```
// Fig. 18.11: Fig18_11.cpp
// Using a dynamically allocated ostringstream object.
#include <iostream>
using std::cout;
using std::endl;

#include <string>
using std::string;

#include <sstream> // header file for string stream processing
using std::ostringstream; // stream insertion operators

int main()
{
   ostringstream outputString; // create ostringstream instance

   string string1( "Output of several data types " );
   string string2( "to an ostringstream object:" );
   string string3( "\n        double: " );
   string string4( "\n           int: " );
   string string5( "\naddress of int: " );

   double double1 = 123.4567;
   int integer = 22;

   // output strings, double and int to ostringstream outputString
   outputString << string1 << string2 << string3 << double1
      << string4 << integer << string5 << &integer;
```

Create **ostringstream** object **outputString**

Output a series of strings and numerical values to **outputString**

Outline

```

   // call str to obtain string contents of the ostringstream
   cout << "outputString contains:\n" << outputString.str();

   // add additional characters and call str to output string
   outputString << "\nmore characters added";
   cout << "\n\nafter additional stream insertions,\n"
      << "outputString contains:\n" << outputString.str() << endl;
   return 0;
} // end main
```

Display a copy of the **string** contained in **outputString**

Append more data to the **string** in memory by issuing another stream insertion operation

```
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
```

Outline

`Fig18_12.cpp`

(1 of 2)

```cpp
// Fig. 18.12: Fig18_12.cpp
// Demonstrating input from an istringstream object.
#include <iostream>
using std::cout;
using std::endl;

#include <string>
using std::string;

#include <sstream>
using std::istringstream;

int main()
{
   string input( "Input test 123 4.7 A" );
   istringstream inputString( input );
   string string1;
   string string2;
   int integer;
   double double1;
   char character;

   inputString >> string1 >> string2 >> integer >> double1 >> character;

   cout << "The following items were extracted\n"
      << "from the istringstream object:" << "\nstring: " << string1
      << "\nstring: " << string2 << "\n   int: " << integer
      << "\ndouble: " << double1 << "\n  char: " << character;
```

Create **`istringstream inputString`** to contain the data in **`string input`**

Extract characters to program variables

Outline

Fig.18_12.cpp

(2 of 2)

```

   // attempt to read from empty stream
   long value;
   inputString >> value;

   // test stream results
   if ( inputString.good() )
      cout << "\n\nlong value is: " << value << endl;
   else
      cout << "\n\ninputString is empty" << endl;

   return 0;
} // end main
```

Test if any data remains

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

inputString is empty
```