Lecture 24:
File Processing & Strings

Ioan Raicu
Department of Electrical Engineering & Computer Science
Northwestern University

EECS 211
Fundamentals of Computer Programming II
May 7th, 2010

17.6 Updating Sequential Files

- Data that is formatted and written to a sequential file as shown in Section 17.4 cannot be modified without the risk of destroying other data in the file.
- For example, if the name "`White`" needs to be changed to "`Worthington`," the old name cannot be overwritten without corrupting the file.
- The record for `White` was written to the file as
  - `300 White 0.00`
- If this record were rewritten beginning at the same location in the file using the longer name, the record would be
  - `300 Worthington 0.00`
- The new record contains six more characters than the original record.
- Therefore, the characters beyond the second "`o`" in "`Worthington`" would overwrite the beginning of the next sequential record in the file.

17.6 Updating Sequential Files (cont.)

- The problem is that, in the formatted input/output model using the stream insertion operator << and the stream extraction operator >>, fields—and hence records—can vary in size.
  - For example, values 7, 14, –117, 2074, and 27383 are all `int`s, which store the same number of "raw data" bytes internally (typically four bytes on today's popular 32-bit machines).
  - However, these integers become different-sized fields when output as formatted text (character sequences).
  - Therefore, the formatted input/output model usually is not used to update records in place.

17.6 Updating Sequential Files (cont.)

- Such updating can be done awkwardly.
- For example, to make the preceding name change, the records before `300 White 0.00` in a sequential file could be copied to a new file, the updated record then written to the new file, and the records after `300 White 0.00` copied to the new file.
- This requires process-ing every record in the file to update one record.
- If many records are being updated in one pass of the file, though, this technique can be acceptable.

17.7 Random-Access Files

- Sequential files are inappropriate for instant-access applications, in which a particular record must be located immediately.
- Common instant-access applications are
  - airline reservation systems,
  - banking systems,
  - point-of-sale systems,
  - automated teller machines and
  - other kinds of transaction-processing systems that require rapid access to specific data.
- A bank might have hundreds of thousands (or even millions) of other customers, yet, when a customer uses an automated teller machine, the program checks that customer's account in a few seconds or less for sufficient funds.
- This kind of instant access is made possible with random-access files.

17.7 Random-Access Files (cont.)

- Individual records of a random-access file can be accessed directly (and quickly) without having to search other records.
- C++ does not impose structure on a file. So the application that wants to use random-access files must create them.
- Perhaps the easiest method is to require that all records in a file be of the same fixed length.
- Using same-size, fixed-length records makes it easy for a program to calculate (as a function of the record size and the record key) the exact location of any record relative to the beginning of the file.
- Figure 17.9 illustrates C++'s view of a random-access file composed of fixed-length records (each record, in this case, is 100 bytes long).

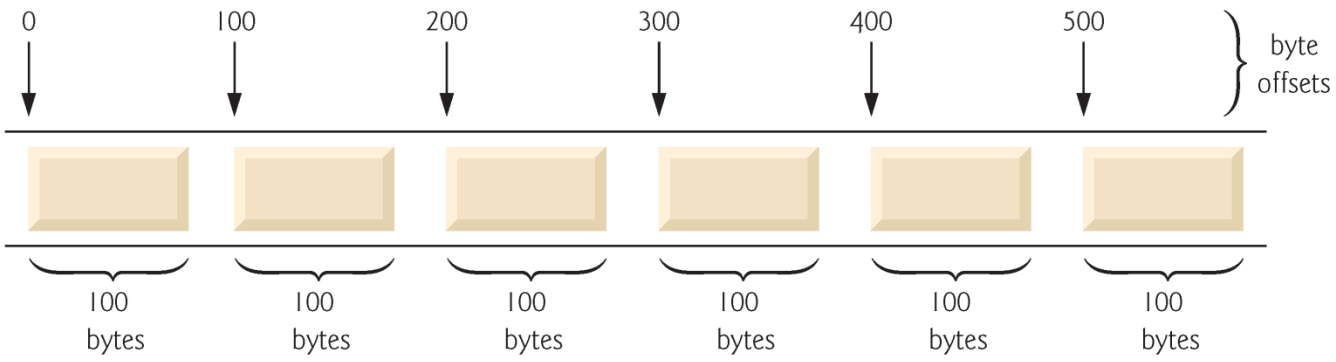

Fig. 17.9 | C++ view of a random-access file.

17.7 Random-Access Files (cont.)

- Data can be inserted into a random-access file without destroying other data in the file.
- Data stored previously also can be updated or deleted without rewriting the entire file.
- In the following sections, we explain how to create a random-access file, enter data into the file, read the data both sequentially and randomly, update the data and delete data that is no longer needed.

17.8 Creating a Random-Access File

- The `ostream` member function `write` outputs a fixed number of bytes, beginning at a specific location in memory, to the specified stream.
- When the stream is associated with a file, function `write` writes the data at the location in the file specified by the "put" file-position pointer.
- The `istream` member function `read` inputs a fixed number of bytes from the specified stream to an area in memory beginning at a specified address.
- If the stream is associated with a file, function `read` inputs bytes at the location in the file specified by the "get" file-position pointer.

17.8 Random-Access Files (cont.)

- Outputting a four-byte integer as text could print as few digits as one or as many as 11 (10 digits plus a sign, each requiring a single byte of storage)
- The following statement always writes the binary version of the integer's four bytes (on a machine with four-byte integers):
  - ```
    outFile.write( reinterpret_cast<
       const char * >( &number ), sizeof( number ) );
    ```
- Function `write` treats its first argument as a group of bytes by viewing the object in memory as a `const char *`, which is a pointer to a byte.
- Starting from that location, function `write` outputs the number of bytes specified by its second argument—an integer of type `size_t`.
- `istream` function `read` can be used to read the four bytes back into an integer vari-able.
    ```

17.8 Random-Access Files (cont.)

- Most pointers that we pass to function `write` as the first argument are not of type `const char *`.
- Must convert the pointers to those objects to type `const char *`; otherwise, the compiler will not compile calls to function `write`.
- C++ provides the `reinterpret_cast` operator for cases like this in which a pointer of one type must be cast to an unrelated pointer type.
- Without a `reinterpret_cast`, the `write` statement that outputs the integer `number` will not compile because the compiler does not allow a pointer of type `int *` (the type returned by the expression `&number`) to be passed to a function that expects an argument of type `const char *`—as far as the compiler is concerned, these types are incompatible.
- A `reinterpret_cast` is performed at compile time and does not change the value of the object to which its operand points.

17.8 Random-Access Files (cont.)

- In Fig. 17.12, we use `reinterpret_cast` to convert a `ClientData` pointer to a `const char *`, which reinterprets a `ClientData` object as bytes to be output to a file.
- Random-access file-processing programs typically write one object of a class at a time, as we show in the following examples.

Portability Tip 17.2

A program that reads unformatted data (written by `write`*) must be compiled and executed on a system compatible with the program that wrote the data, because different systems may represent internal data differently.*

```
1  // Fig. 17.12: Fig17_12.cpp
2  // Creating a randomly accessed file.
3  #include <iostream>
4  #include <fstream>
5  #include <cstdlib>
6  #include "ClientData.h" // ClientData class definition
7  using namespace std;
8
9  int main()
10 {
11    ofstream outCredit( "credit.dat", ios::out | ios::binary );
12
13    // exit program if ofstream could not open file
14    if ( !outCredit )
15    {
16       cerr << "File could not be opened." << endl;
17       exit( 1 );
18    } // end if
19
20    ClientData blankClient; // constructor zeros out each data member
21
```

Fig. 17.12 | Creating a random-access file with 100 blank records sequentially. (Part 1 of 2.)

```
22      // output 100 blank records to file
23      for ( int i = 0; i < 100; i++ )
24         outCredit.write( reinterpret_cast< const char * >( &blankClient ),
25            sizeof( ClientData ) );
26   } // end main
```

Fig. 17.12 | Creating a random-access file with 100 blank records sequentially. (Part 2 of 2.)

17.9 Writing Data Randomly to a Random-Access File

- Figure 17.13 writes data to the file `credit.dat` and uses the combina-tion of `fstream` functions `seekp` and `write` to store data at exact locations in the file.
- Function `seekp` sets the "put" file-position pointer to a specific position in the file, then `write` outputs the data.
- Line 6 includes the header file `clientData.h` defined in Fig. 17.10, so the program can use `clientData` objects.

```
1  // Fig. 17.13: Fig17_13.cpp
2  // Writing to a random-access file.
3  #include <iostream>
4  #include <fstream>
5  #include <cstdlib>
6  #include "ClientData.h" // ClientData class definition
7  using namespace std;
8
9  int main()
10 {
11    int accountNumber;
12    string lastName;
13    string firstName;
14    double balance;
15
16    fstream outCredit( "credit.dat", ios::in | ios::out | ios::binary );
17
18    // exit program if fstream cannot open file
19    if ( !outCredit )
20    {
21       cerr << "File could not be opened." << endl;
22       exit( 1 );
23    } // end if
```

Fig. 17.13 | Writing to a random-access file. (Part 1 of 4.)

```
24
25      cout << "Enter account number (1 to 100, 0 to end input)\n? ";
26
27      // require user to specify account number
28      ClientData client;
29      cin >> accountNumber;
30
31      // user enters information, which is copied into file
32      while ( accountNumber > 0 && accountNumber <= 100 )
33      {
34         // user enters last name, first name and balance
35         cout << "Enter lastname, firstname, balance\n? ";
36         cin >> lastName;
37         cin >> firstName;
38         cin >> balance;
39
40         // set record accountNumber, lastName, firstName and balance values
41         client.setAccountNumber( accountNumber );
42         client.setLastName( lastName );
43         client.setFirstName( firstName );
44         client.setBalance( balance );
45
```

Fig. 17.13 | Writing to a random-access file. (Part 2 of 4.)

```
46          // seek position in file of user-specified record
47          outCredit.seekp( ( client.getAccountNumber() - 1 ) *
48             sizeof( ClientData ) );
49
50          // write user-specified information in file
51          outCredit.write( reinterpret_cast< const char * >( &client ),
52             sizeof( ClientData ) );
53
54          // enable user to enter another account
55          cout << "Enter account number\n? ";
56          cin >> accountNumber;
57       } // end while
58    } // end main
```

Fig. 17.13 | Writing to a random-access file. (Part 3 of 4.)

17.10 Writing Data Randomly to a Random-Access File (cont.)

- Lines 47–48 position the "put" file-position pointer for object `outCredit` to the byte location calculated by
  - `( client.getAccountNumber() - 1 ) * sizeof( ClientData )`
- Because the account number is between 1 and 100, 1 is subtracted from the account number when calculating the byte location of the record.
  - Thus, for record 1, the file-position pointer is set to byte 0 of the file.
- Line 16 uses the `fstream` object `outCredit` to open the existing `credit.dat` file.
  - The file is opened for input and output in binary mode by combining the file-open modes `ios::in`, `ios::out` and `ios::binary`.
- Multiple file-open modes are combined by separating each open mode from the next with the bitwise inclusive OR operator (`|`).

17.11 Reading from a Random-Access File Sequentially

- In this section, we develop a program that reads a file sequentially and prints only those records that contain data.
- The `istream` function `read` inputs a specified number of bytes from the current position in the specified stream into an object.
- For example, lines 30–31 from Fig. 17.14 read the number of bytes specified by `sizeof(clientData)` from the file associated with `ifstream` object `inCredit` and store the data in the `client` record.
- Function `read` requires a first argument of type `char *`.
- Since `&client` is of type `clientData *`, `&client` must be cast to `char *` using the cast operator `reinterpret_cast`.

```
1  // Fig. 17.14: Fig17_14.cpp
2  // Reading a random-access file sequentially.
3  #include <iostream>
4  #include <iomanip>
5  #include <fstream>
6  #include <cstdlib>
7  #include "ClientData.h" // ClientData class definition
8  using namespace std;
9
10 void outputLine( ostream&, const ClientData & ); // prototype
11
12 int main()
13 {
14    ifstream inCredit( "credit.dat", ios::in | ios::binary );
15
16    // exit program if ifstream cannot open file
17    if ( !inCredit )
18    {
19       cerr << "File could not be opened." << endl;
20       exit( 1 );
21    } // end if
22
```

Fig. 17.14 | Reading a random-access file sequentially. (Part 1 of 3.)

```
23       cout << left << setw( 10 ) << "Account" << setw( 16 )
24          << "Last Name" << setw( 11 ) << "First Name" << left
25          << setw( 10 ) << right << "Balance" << endl;
26
27       ClientData client; // create record
28
29       // read first record from file
30       inCredit.read( reinterpret_cast< char * >( &client ),
31          sizeof( ClientData ) );
32
33       // read all records from file
34       while ( inCredit && !inCredit.eof() )
35       {
36          // display record
37          if ( client.getAccountNumber() != 0 )
38             outputLine( cout, client );
39
40          // read next from file
41          inCredit.read( reinterpret_cast< char * >( &client ),
42             sizeof( ClientData ) );
43       } // end while
44    } // end main
45
```

Fig. 17.14 | Reading a random-access file sequentially. (Part 2 of 3.)

```
46   // display single record
47   void outputLine( ostream &output, const ClientData &record )
48   {
49      output << left << setw( 10 ) << record.getAccountNumber()
50         << setw( 16 ) << record.getLastName()
51         << setw( 11 ) << record.getFirstName()
52         << setw( 10 ) << setprecision( 2 ) << right << fixed
53         << showpoint << record.getBalance() << endl;
54   } // end function outputLine
```

```
Account   Last Name       First Name    Balance
29        Brown           Nancy          -24.54
33        Dunn            Stacey         314.33
37        Barker          Doug             0.00
88        Smith           Dave           258.34
96        Stone           Sam             34.98
```

Fig. 17.14 | Reading a random-access file sequentially. (Part 3 of 3.)

17.13 Overview of Object Serialization

- This chapter and Chapter 15 introduced the object-oriented style of input/output.
- An object's member functions are not input or output with the object's data; rather, one copy of the class's member functions remains available internally and is shared by all objects of the class.
- When object data members are output to a disk file, we lose the object's type information.
- We store only the values of the object's attributes, not type information, on the disk.
- If the program that reads this data knows the object type to which the data corresponds, the program can read the data into an object of that type as we did in our random-access file examples.

17.13 Case Study: A Transaction-Processing Program (cont.)

- An interesting problem occurs when we store objects of different types in the same file.
- How can we distinguish them (or their collections of data members) as we read them into a program?
- The problem is that objects typically do not have type fields (we discussed this issue in Chapter 13).
- One approach used by several programming languages is called object serialization.
- A so-called serialized object is an object represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.
- After a serialized object has been written to a file, it can be read from the file and deserialized—that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

17.13 Case Study: A Transaction-Processing Program (cont.)

- C++ does not provide a built-in serialization mechanism; however, there are third party and open source C++ libraries that support object serialization.
- The open source Boost C++ Libraries (`www.boost.org`) provide support for serializing objects in text, binary and extensible markup language (XML) formats (`www.boost.org/libs/serialization/doc/index.html`).

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`.
    ```

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.

```
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.*

18.3 Comparing `string`s

- 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.

18.4 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.)

Questions