The Standard C++ String Class

Topics Covered
- Reading: 780-792, Appendix F [980-996]
- The Standard C++ string class
- The string constructors and destructors
- String access, size and capacity, and C-strings
- String modification, substrings and concatenation
- String input and output
- String searching

The Standard C++ Library
- What is the standard C++ library?
  - Classes supplied with every conforming implementation
  - Just like <stdio.h>, etc. in C
  - Much larger than the C standard library
  - Unlike legacy [C++ 1.0/2.0] library, based on templates
- Major pieces
  - The iostream library
  - The string class
  - Numeric and utility classes
  - Classes based upon the STL [Not the STL]
The Standard C++ string Class

- Why have a standard string class?
  - Allows you to treat strings like built-in types
    » Freely copy, compare, search, manipulate
    » Don't need to worry about allocation, deallocation
    » Easier for programmer, eliminates many errors
    » Template-based; fairly high performance
- Standard C++ string class is a template class

  ```
  typedef basic_string<char> string;
  typedef basic_string<wchar_t> wstring;
  ```

Characteristics of strings I

- A C++ string is a class that represents a sequence of characters [char, wchar_t, or...]
  - Not a NUL terminated array of char
    » Number of characters retrieved via member functions length() or size()
  - Not a char * or const char *
    » Uses pointers and dynamic array internally
    » Class automatically manages internal array
  - Transparently expands as needed

Characteristics of strings II

- How many characters can a string contain?
  - "Implementation dependent" like sizeof(int)
    » Can't use sizeof(string) of course
    » Get info using member function max_size()  
- How much memory is allocated for a string?
  - Don't allocate max_size() chars; too inefficient
  - Don't want to resize internal array more than required
    » Slow, invalidates pointers & references
  - Find out allocated size with capacity() function
Characteristics of strings III

- The string class gives you some access to the underlying array used to store the characters:
  - The member function `data()` returns a `const char *` that points to the first element in the array.
  - The member function `c_str()` returns the same thing, except that the array is NUL terminated.
  - Note that there is no operator `const char *()`.
  - Normally you access the characters using the subscript operator or the `at()` member function.

String constructors I

- How do you create a string? [string01.cpp]
  - 1. Use a default constructor [unlike Java]
     ```cpp
     string s1;  // size() = 0
     ```
  - 2. Use an array [const char *] conversion constructor
     ```cpp
     string s2("s2");
     string s3 = "s3";
     ```
  - 3. From part of an array [pass in number of characters]
     ```cpp
     string s4("Bowser", 3);
     string s5("Bowser", 80); // ??
     ```

String constructors II

- Creating string objects [cont.]
  - 4. Fixed size, filled with a particular character
     ```cpp
     string s6(20, '5');
     ```
  - 5. Using a copy constructor
     ```cpp
     string temp("ABCDEFG");
     string s7(temp);  // ABCDEFG
     string s8(temp, 3);  // DEF
     string s9(temp, 3, 3);  // DEF
     string s10(temp, 3, 200);  // DEF
     ```
String constructors III

- Creating string objects [cont.]
  - 6. By passing two "iterators" [pointers]
    
    ```c++
    char ca[] = "All of the King's men";
    string s11(ca + 4, ca + 10); // [of the]
    ```
    
    » Second iterator is excluded
  - Iterator must be a "pointer-like" object
    
    » A C++ string is not a pointer:
      
      ```c++
      string s12(temp, temp+4);           // ILLEGAL
      string s12(&temp[0], &temp[4]); // OK
      ```

String Assignment

- You can assign to an existing string using:
  
  - Three assignment operators [string02.cpp]
    
    ```c++
    s2 = "Who knows"; // a const char *
    s1 = s2;          // a string
    s3 = 'X';          // a char
    ```
  
  - For more complex cases use the assign() method
    
    ```c++
    s1.assign(s2, 1, 4);      // ho kn
    s2.assign("You", 2);     // Yo
    s3.assign(2, 'D');        // DD
    ```

Appending Values

- Two ways to add to end of existing strings
  
  - For simple values, use operator +=()
    
    ```c++
    s1 += s2; // string += string
    s2 += '?'; // string += const char *
    s3 += 'C'; // string += char
    "What" += s2; // Illegal
    ```
  
  - Use append() for more complex values
    
    ```c++
    s1.append("Shell", 3); // add She
    s2.append(s1, 3, 7);   // add 7 chars, start at 3
    s3.append(10, '"');   // add 10 "" chars
    ```
String Concatenation

- Overloaded operator `+` to concatenate
  - Not member function, produces new string
  - Overloaded for string, const char *, and char

```cpp
string s1("red");
string s2("rain");
string s3, s4, s5, s6;
s3 = s1 + "uce"; // string + char *
s4 = 't' + s2;  // char + string
s5 = s1 + s2;   // string + string
s6 = s6 + "Fred" + " and " + "Ethel";
```

Substring Operations

- Sometimes you only want part of a string
  - You can usually use `assign()` for this, but often the `substr()` methods are clearer and easier to read

```cpp
string s("interchangeability");
string s1, s2, s3;
s1 = s.substr(); // whole string
// Remainder of string starting at 11
s2 = s.substr(11); // ability
// 6 characters starting at position 5
s3 = s.substr(5, 6); // change
```

Accessing String Elements

- Overloaded operator `[ ]` for character access
  - Fast, but not range-checked
  - Two versions (const, non-const)

```cpp
string barney("barney");
string fred("fred");
cout << "barney[0] = " << barney[0];
fred[0] = 'F';
cout << "fred[0] = " << fred[0];
barney[0] = 'B';  // const
```

- For range-checked access use the `at()` method
Comparing Strings

- Class has a method like strcmp(): compare()
  - Strings compared using locale specific char traits
  - Returns 0 when equal, < 0 if less, > 0 if greater
- Class also has overloaded comparison operators
  - The equality operators: ==, !=
  - The inequality operators: >, >=, <, <=
  - Written as non-members using compare()
- No real reason to use compare() directly

String I/O I

- Insertion and extraction operators [string03.cpp]
  - Works like for character arrays, but extractor resizes
    - char ca[10]; // 10 characters
    - cin >> s; // no size required
    - cout << ca; // not really safe
- Skips leading whitespace unless skipws flag set
- Reads until width(), whitespace, or end of file
- In general, it "reads a word" [string03b.cpp]

String I/O II

- Read a line using the getline() method [string04.cpp]
  - Unlike cin.getline() this is not a member function
  - Instead, you pass an input stream as the first argument, and the string to fill as the second
  - Supply a third argument to specify the character used to end a line of input. [default = \n] [string05.cpp]
  - Delimiter is removed from input, but not stored.
    - string s1, s2;
      getline(cin, s1); // reads until \n' or eof
      getline(cin, s1, '\'); // reads until ' or eof
String Searching I

- Search using *find()* and *rfind()* ([string06.cpp])
  - Can search for a string, char *, or char
  - Starting search position defaults to 0
  - Returns a string::size_type value
    - Unsigned integral type, undefined size
    - Careful not to mix with signed types like int
  - If not found, returns string::npos
    - Unsigned string::size_type value of -1
    - Largest possible string index, thus illegal

String Searching II

- Here are some examples of *find()* and *rfind()*
  - Returned value is zero-based position of argument
    ```cpp
    String text("Howdy Doody");
    String srch("oody");
    text.find(srch);   // 7 [string]
    text.find("o");    // 1 [char *]
    text.find('o', 2);  // 7 [char, start at 2]
    text.rfind('o');   // 8 [char, last o]
    text.find("Doodly"); // string::npos
    ```
  - Can search for a char *, but not search a char *

String Searching II

- You can also search for a portion of a string
  - Suppose you have the following operators
    ```cpp
    String ops("()+-/*%");
    ```
  - You can separate the operators like this: ([string07.cpp])
    ```cpp
    String expr("3*(4+7)/3");
    expr.find_first_of(ops); // 1
    expr.find_last_of(ops);  // 7
    expr.find_first_not_of(ops); // 0
    expr.find_last_not_of(ops); // 8
    ```
String Modification I

- Remove characters using `erase()` ([string08.cpp])
  - Subsequent characters are moved to left
    - These only work on string objects, not `char*`
  - Four versions:
    ```cpp
    string text("The Cat in the Hat");
    text.erase(15); // erase Hat
    text.erase(0, 3); // erase The
    text.erase(text[0]); // erase space
    text.erase(text[3], text[10]);
    ```

String Modification II

- Add `char`, `char*`, `string` using `insert()`
  - First argument is position at which to insert
  - Characters at that position are moved to the right
  ```cpp
  string s("p"), iString("age");
  // Insert String at end of s [could also use 1]
  s.insert(s.size(), iString); // s: page
  // Insert const char * "ersifl" after first p
  s.insert(1, "ersifl"); // s: persiflage
  // Insert at the beginning of the string
  s.insert(0, "A "); // s: A persiflage
  ```

String Modification III

- Here are some other versions of `insert()`
  ```cpp
  text.insert(0, 'x'); // Single char
  text.insert(&text[0], 1, 'x'); // Single char
  text.insert(&text[0], 20, '$'); // 20 chars
  text.insert(0, "What's up", 4); // 4 chars What
  ```

- Combine `insert()` and `erase()` with `replace()`
  ```cpp
  text.replace(0, 3, "Dog");
  ```

- Several different versions of each
Homework

Homework 24: String searching
- Due Sunday, May 9
- Accept two arguments on command-line
  - Word to search for, file to look through
- Open file and search for instances of word
- Print line numbers where word is found
- Print number of times when word is found
  - At least as great as line number, possibly larger
- Use C++ strings, not char arrays for everything except the command-line arguments