C++ Strings

Introduction

The C++ string library exports the `string` type, an object that stores and manipulates a sequence of characters. In CS106X, `string` is one of the few C++ Standard Library objects we allow you to use, both because strings are essential for useful programming and because `string` is relatively simple to use. However, the `string` type has a host of member functions not covered in CS106X. This handout covers both the pitfalls and the powers of the `string` class.

Java `substring` vs. C++ `substr`

Although we cover the `substr` member function in CS106X, I find it necessary to repeat the syntax here because the C++ `substr` member function **does not** work the same way as the Java `substring` method. For example, given the following Java code:

```
String myString = "abcdefgh"; // Java, not C++
System.out.println(myString.substring(2, 6));
```

The output would be the substring "cedf," that is, the characters in the range [2, 6). However, if you were to directly copy this code and change to C++ syntax like this:

```
string myString = "abcdefgh"; // C++, not Java
cout << myString.substr(2, 6) << endl;
```

The output would be "cedfgh", the *six*-character subsequence beginning at character position two. C++ defines substrings using the range [start, start + length) instead of Java's more intuitive [start, stop). For those of you with a Java background, make sure to keep this in mind or you're bound to get stuck tracking down an incredibly elusive bug.

Of `int`s and `string::size_type`s

When specifying string indices or ranges – whether for `substr` or `find_first_not_of`, you will need to pass integer data as parameters. Because strings can't be of negative length, string functions are configured to accept and return the special nonnegative `string::size_type` type instead of regular `int`s.* While the distinction between `string::size_type` and `int` is trivial (it only shows up for strings of length greater than two billion), it is worth knowing because many compilers will generate warning messages if you try to store the results of the `find` class of functions in `int`s instead of `string::size_type`s. For almost all intents and purposes it's safe to ignore these warnings – most professional C++ programmers do – but you should be aware that to write technically correct code you should use `string::size_type` instead of `int`s.

---

* The STL container classes, which we will cover next week, have similar variable types.

Modifying String Contents

Odds are, at some point you will need to manipulate string contents in-place. For example, perhaps you will need to escape HTML tags read in from a forum submission, or perhaps you'll need to remove sensitive information from an order form before sending it over a network. Fortunately, the `string` class comes with a whole host of manipulation functions.

The easiest thing to do to a string is to simply delete a range of characters using the `erase` member function. For example:

```
string myString = "abcdefgh";

myString.erase(4);         // Delete everything after the 4th character.
cout << myString << endl;  // Output: abcd

myString.erase(2, 2);      // Delete two chars, starting at position 2.
cout << myString << endl;  // Output: ab
```

If you'll notice, `erase` is syntactically similar to `substr`. If you provide a single parameter to `erase`, it deletes everything from the string at or after that point. If you provide two parameters, those parameters represent the starting location for the deletion and the number of characters to delete. As with `substr`, make sure you remember that you are specifying the range [start, start + length) instead of [start, stop)!

If you want to insert more characters into a string, you can do so with the `insert` member function. There are many versions of `insert`, but in each case the first parameter indicates the position at which the new elements should be inserted. `insert` does not overwrite any data, and will increase the length of the string appropriately. Here is some code using `insert`:

```
string myString = "abcdefgh";

myString.insert(4, "0123");  // Inserts '0123' at position 4.
cout << myString << endl;    // Output: abcd0123efgh

myString.insert(0, 4, 'x');  // Insert four x's at position 0.
cout << myString << endl;    // Output: xxxxabcd0123efgh
```

A special case of the `insert` function is `append`, which puts characters at the end of a string. `append` has identical syntax to `insert`, except that the position parameter is omitted.

You can combine `erase` and `insert` operations into a single call with the `replace` member function, which replaces a range of text with the specified content. Syntactically, `replace` is identical to `insert`, except that it takes an additional parameter specifying the number of characters to replace. When you replace characters, the replacement does *not* need to be the same length. It is permissible to replace a two-element substring with a fifteen-element string, or to replace an entire paragraph with the string "...". Here are some examples using `replace`:

```
string myString = "abcdefgh";

myString.replace(4, 4, "0123"); // Replace 4 chars after char 4 with 0123.
cout << myString << endl;       // Output: abcd0123
```

```
myString.replace(6, 2, 4, 'x'); // Replace the last 2 chars with 4 x's.
cout << myString << endl;       // Output: abcd01xxxx
```

For your convenience, these functions are listed in the table below:

`string &erase(size_type start, size_type count = npos)`	`myString.erase(3);  // Erases all but first 3 chars.`
	`myString.erase(1,3); // Erases 3 chars,`
	`// starting at char 1.`
	
	Removes characters from the string and decreases the string length appropriately. Remember that, like `substr`, you are *not* specifying a start and stop point, but instead a start point and a length.

`string &append( const string &str)`
`string &append( size_type count, char ch)`
	`myString.append("string!");`
	`myString.append(10, 'a'); // Appends 10 a's.`
	
	Appends the specified string or run of characters to the end of the string.

`string &insert( size_type index, const string &str)`
`string &insert( size_type index, size_type count, char ch)`
	`myString.insert(4, "string!"); // Inserts "string"`
	`// at position 4.`
	`myString.insert(0, 10, 'a'); // Inserts 10 a's at the`
	`// start of the string.`
	
	Inserts extra characters or strings into the string object, beginning at the specified point. This function does not overwrite any data.

`string &replace( size_type index, size_type length, const string &str)`
`string &replace( size_type index, size_type length, size_type count, char ch)`
	`myString.replace(4, 2, "string!");`
	`// Replaces chars 4 and 5 with "string!"`
	`myString.replace(0, 2, 10, 'a');`
	`// Replaces first two chars with 10 a's`
	
	Replaces `length` characters in the string, starting at position `index`, with the specified string or characters.

Search Operations

The C++ `string` class supports a large number of search functions to locate substrings and characters. You can search for the first or the last instance of an item inside a string, and can also specify the location to begin searching from. Used correctly, you can perform queries along the lines of "find the last digit in the string that occurs before position 10."

All of the search functions return the index at which the search term exists in the string, or the special constant `string::npos` if the item was not found.

Again, we have not covered the `const` keyword yet, so for now it's safe to ignore it.

`size_type find( const string &str, size_type start = 0) const`
`size_type find( char ch, size_type start = 0) const`
	`if(myStr.find("popcorn") != string::npos)...`
	`if(myStr.find('q') != string::npos)...`
	
	Returns the index of the first instance of the substring `str` or the character `ch` in the string, starting at position `start`. If the item isn't found, `find` returns `string::npos`.

Searching functions, cont.

`size_type rfind( const string &str,`
`  size_type stop = npos) const`
`size_type rfind( char ch,`
`  size_type stop = npos) const`
	`if(myStr.rfind("popcorn") !=`
	`  string::npos) // Found "popcorn"`
	`if(myStr.rfind('q') != string::npos)`
	`  // Found letter 'q'`
	
	Returns the index of the last instance of the substring `str` or the character `ch` in the string occurring no later than position `stop`. If the item isn't found, `rfind` returns `string::npos`.

`size_type find_first_of( const string &str,`
`  size_type start = 0) const`
	`if(myStr.find_first_of("aeiouy") !=`
	`  string::npos) // String contains a vowel`
	
	Returns the index of the first character in the string that matches *any* character in `str`, starting at position `start`. If no characters from the string `str` are found, `find_first_of` returns `string::npos`.

`size_type find_last_of( const string &str,`
`  size_type stop = npos) const`
	`if(myStr.find_last_of("aeiouy") !=`
	`  string::npos) // String contains a vowel`
	
	Searches the string for the last character that matches any of the characters of the `str` parameter. If no characters in the strings match, the return value is `string::npos`.

`size_type find_first_not_of(const string &str,`
`  size_type start = 0) const`
	`if(myStr.find_first_not_of("aeiouy") !=`
	`  string::npos) // String has a nonvowel`
	
	Searches the string for the first character that does not match any of the characters of the `str` parameter. This is useful for checking if there is an invalid character somewhere in the string. If all the characters in the string are also in the `str` parameter, the return value is `string::npos`.

`size_type find_last_not_of( const string &str,`
`  size_type stop = npos) const`
	`if(myStr.find_last_not_of("aeiouy") !=`
	`  string::npos) // Contains a non-vowel.`
	
	Searches the string for the last character that does not match any of the characters of the `str` parameter. If all the characters in the strings match, the return value is `string::npos`.

Commonly you'll want to write code that iterates over an entire string looking for patterns or specific substrings. Thus, you might want to consider using this type of loop:

```
string::size_type startPosition = 0;
while(true)
{
      startPosition = myStr.find(subseq, startPosition);
      if(startPosition == string::npos)
            break;
      // Process here.
      // Increment startPosition here, if necessary.
}
```

In this loop, you're using the previous value of `startPosition` to determine the starting position of the next call to `find`. The first time this function loops, `startPosition` is zero and `find` proceeds as normal. On every subsequent iteration, your search will begin beyond the previously found items, and thus will search for the next remaining match in the string.

One point to note is whether you should manually increment `startPosition` at the end of every loop. If you do not modify the contents of the string once you've found a substring, you'll need to increment `startPosition` or on the next call to `find`, which begins at index `startPosition`, your program will pick up the same instance of the substring. This leads to an infinite loop and is probably not what you intended. However, if you are modifying the contents of the string in such a way that your search won't pick up the same string twice, you can skip on the increment.

C++ strings, C Strings, and c_str

Because C++ was developed as a superset of the C language, it inherited many C constructs to guarantee that code written in C could be compiled in C++. One of the more difficult concepts C++ inherited was the C string. All string literals in C++ are actually C strings. For example, when you write code along the lines of `string myString = "This is a string!"`, you are actually converting a C string (the right-hand side of the assignment) into a C++ `string`.

While most of the C++ Standard Library accepts both C and C++ strings, there is one notable exception: when specifying a filename for a file stream, you must use a C string instead of a C++ `string`. To obtain a C string from a C++ `string`, use the `c_str` member function, shown below.

```
ifstream input(myString.c_str());
```

Fortunately, in CS106X this is probably the only place you'll ever need to use a C string.

In most cases, that C strings can be implicitly converted to C++ `string`s means that you shouldn't have to worry too much about the distinction. However, there is one major issue to be aware of regarding C strings, and it shows up if you write seemingly valid code that looks like this:

```
string error1 = "String" + "!"; // Compile-time error!
string error2 = "String" + '!'; // Undefined runtime behavior!
```

The first line will generate a compiler error stating that it is illegal to add two pointers. The second line has undefined behavior at runtime – it might crash your program outright, or it might fill your string with garbage. What's going on?

The problem exists because C++ only converts C strings into C++ `string` when they are directly used in mixed expressions involving C++ `string`s. That is, unless you assign or add a C string to a C++ `string`, C++ will not perform an implicit type conversion. In the above examples, the C strings were added to other C strings, not C++ `string`s, and consequently no type conversions were performed. It is as though you had written the code with this grouping:

```
string error1 = ("String" + "!"); // No conversion – both are C strings.
string error2 = ("String" + '!'); // Same, except a C string and a char.
```

Because the C strings are not transformed into C++ `string`s, the + operator will not work as expected. Addition of C strings is undefined, hence the compiler error on the first line. As discussed in the next handout, when adding a `char` to a C string, the result is defined but means something

entirely different from concatenation. Commonly it results in a crash.

To fix these problems, you have several options. First, you can explicitly typecast the C strings to C++ `string`s, as shown here:

```
string notError1 = string("String") + "!";
string notError2 = string("String") + '!';
```

Alternatively, you could split the code into multiple lines, like this:

```
string notError1 = "String";
notError1 += "!";
string notError2 = "String";
notError2 += '!';
```

In this case, the addition is performed by the C++ `string` instead of the unsafe C string, so everything will work as planned.

The important message of this section is that strings in C++ don't always behave the way you expect. If you're not careful, you might accidentally manipulate C strings instead of C++ `string`s. While this is not necessarily a bad thing (as we'll see in the next handout), if you treat a C string like a C++ string while manipulating it, you're bound to run into some serious problems.

More to Explore

This handout is a relatively good exploration of the `string` class, but there are a few points we haven't covered. If you're interested in some advanced topics, you might consider looking into the Boost C++ Libraries. Although the C++ `string` class is suitable for many tasks, there are many string operations not included in the default C++ libraries. For example, the `string` class has no support for regular expression evaluation, nor do C++ `string`s natively support routine operations like trimming, case conversion, and splitting. However, an external set of libraries known as the Boost C++ Libraries have developed an overwhelming number of functions and classes related to virtually every C++ topic. The Boost support for strings is nothing short of incredible, and lets you perform all sorts of nifty operations relatively simply. If you are interested in maximizing the potential of the `string` class (and C++ in general), consider looking into the Boost libraries at www.boost.org.

Practice Problems

Again, I won't collect these for a grade, but I highly recommend that you experiment with them.

1. Write a function `Exaggerate` that takes a `string` as a reference parameter and increases any non-nine digits in that string by one. For example, given the input string "I worked 90 hours and drove 24 miles" the function would change it to read "I worked 91 hours and drove 35 miles." While you can write this program using the `isdigit` function, try to solve this problem using the `string`'s search functions instead.
2. Write a function `InsertCommas` that, given as a parameter a `string` containing mixed integer data and text, inserts commas into the integers to improve readability. For example, if the input is "1000000 melons," you should return "1,000,000 melons."