CS11 C++ Lab 7 - Word Processing

Fun with STL

The STL!

This week's lab will give you more opportunities to play with the C++ Standard Template Library (STL). The STL contains a number of very useful container-classes, which are implemented as templates so that we can store just about anything in them. Also, there are two different kinds of containers - *sequences*, which store a collection of items in a particular order, and *associative containers*, which associate each item with a key, that is used to store and retrieve that item. You will get to play with both of these kinds of containers this week.

Word Processing

You probably use the Google search engine, or some similar search engine, many times a day. Of course, these search engines don't just run against a huge plaintext database of information; if they did, they would be terribly slow and nearly useless. Rather, these search engines *preprocess* documents on the Internet, so that it is very fast and easy for them to figure out which documents contain the search words you specify.

One simple way to preprocess text documents for searching is to generate a list of unique words in each document. This information can be used to immediately include or exclude documents from a search operation: if a search word doesn't appear in a document's unique-words list, then there's no reason to search it! A document's unique-words list will also generally be much smaller than the document itself, and storage can be optimized so that lookups are very fast, so that the time savings is very large for using such an indexing mechanism.

However, this isn't the whole picture, because you would like to generate search hits in a particular order, based on their "relevance" to what you think the user is interested in seeing. For example, if someone searches for the keywords "tufted puffin", you would like to put the documents that talk *most* about tufted puffins, *first* in the list of results. This is why it is also valuable to keep track of the number of times each unique word appears in a particular document. Once you have this information, it is straightforward to assign a relevance to each document that contains a particular search word.

Your mission...

This week you will work on a command-line program that can generate a list of unique words that appears in a text document, and the number of times that each word appears in the document. Your program will read its input from the console input stream `cin`, and it will produce its result on console output stream `cout`. Internally, your program can use the STL container-classes to make quick work of this task. You will be able to run your program like this:

```
wcount < doc.txt
```

This tells the computer to run your program `wcount`, and feed it the contents of `doc.txt` as its input. The output will appear in the console as normal.

Reading Input

You may remember, from the distant hazy past of C++ lectures 1 and 2, that it is very easy to read words from the console input in C++. Specifically, you can write something like this:

```
#include <iostream>
#include <string>

int main()
{
  string name;

  cout << "Enter your name:  ";
  cin >> name;
  cout << "Hello, " << name << endl;

  return 0;
}
```

It is just as easy to read all of the words from console input, by putting the `cin >> word;` into a loop, like this:

```
void consumeAllWords()
{
  string word;

  // Read input until there is no more!
  while (cin >> word)
  {
    ... // Do something with each word.
  }
}
```

(If you are curious why and how this works, the high level hand-wavy answer is that `cin` is implemented so that it will evaluate to true if it hasn't reached end-of-stream yet, or it will evaluate to false if there is no more input to read. The `while` statement takes a conditional expression of type `bool`, so everything works together automatically.)

Processing Words

If you read any text document, there are a lot of details that can hinder this kind of processing! For example, punctuation tends to get in the way, as do capitalized words. Thus, it is very important to convert each word into a standard, normalized form. For our program, we will do the following things:

1. Convert the word to all lowercase
2. Remove any leading and trailing punctuation from the word

This task will be handled by the `string processWord(string word)` function, a function that you must write. As you can see from the function signature, this function takes a word (passed by-value so that the function can manipulate the word directly, if desired), and returns another `string` object containing the result of this word-processing.

Of course, once this has happened, the word may now be empty. For example, if your program is given this text:

```
and -- believe it or not -- Jerry said that the answer was
```

The two "`--`" words will be processed down to an empty string by this function. Fortunately, you can find out how many characters are left in the result string by calling the `length()` (or `size()`) function on the object. If the length of the word is zero after processing it, you can just skip it and go on to the next word.

Tracking Word-Counts

So now you have a way to read words from `cin`, and you have a way to clean them up so that your program can make sense of them. Now it's time to figure out how many times each word appears. This is where you can use the STL associative container called `map`. STL `map` containers associate a particular key with some value; they are effectively stored like this: (*key* , *value*). Each key can only appear once in a `map`, and the keys are kept in sorted order by the `map`. We will use a `map` to keep track of each word and how many times it has been used in the document.

The `map` container is a template, so you have to specify the key type and the value type when you declare your `map` variable. In our program, the keys will simply be the words we find. This is supposed to be a list of unique words, so the "unique key" constraint is perfect for this task. The value associated with each key will simply be the number of times we have seen that word. So, our variable declaration will look like this:

```
#include <map>

using namespace std;

...

// Our collection of word counts.  Each word is a key, and the count is the value.
map<string, int> wordCounts;
```

How do you look up entries on a `map`? STL has made this *very* easy, by overloading the brackets-operator `[]`. In fact, what you can do is this:

```
cout << "Count of \"hello\" is " << wordCounts["hello"] << endl;
...
wordCounts["goodbye"] = 1;
```

Or, if you want to increment a particular word's count, you can do something like this:

```
string word;
...
++wordCounts[word];   // Or wordCounts[word]++, of course!
```

This will even work if the word is not already in the map; the default initial value will be 0.

Easy!

Attempt #1: Simple Word-Counter

For your first attempt at creating a word counting utility, you can start by copying wcount.cc to your `~/cs11/cpp/lab7` directory. This file contains a simple framework for your program.

You will have two tasks to complete in this file; you must implement the `processWord` function and the `processText` function.

The `processWord` function must do the following tasks:

1. Iterate over every character in the input word, converting the character to lowercase.
2. Remove all leading and trailing punctuation from the input word. You can iterate over the word's characters starting from the front, stopping when you reach the first non-punctuation character. If you didn't get all the way to the end of the string, repeat this operation but

start at the *end* of the word, and stop when you reach the first non-punctuation character. Then, you can just return the substring of the input word using these two indexes.

- Make sure to handle cases where the input word is all punctuation!
- Also, make sure that words like "they're" get processed correctly! Internal punctuation should *not* be removed.

You might find the functions in the standard header-file `<cctype>` *very* helpful for this processing. For example, there is an `ispunct(int c)` function that returns a nonzero value when the input is a punctuation character, or zero when the input is not punctuation. There is also a `tolower(int c)` function that returns the lowercase version of its input. Don't let the types of the inputs or outputs confuse you; these are C standard functions, and hence use `int`s instead of `char` or `bool` values. Unfortunately, we will just have to put up with that here...

The `processText` function must do the following things:

1. Read a word from `cin`.
2. Process the word using `processWord()`, and skip over any words that end up with zero size after processing. (That means, go back to step 1 if the word's length ends up being 0 after being processed.)
3. Update a counter of the total words you have seen. (This is a counter of the total number of *real* words you have seen; don't count words that get processed down to nothing.)
4. Update the `wordCounts` map to indicate that the word appears in the document.
5. Go back to step 1, until all words are consumed.
6. Print out the total number of words your function has seen. Also, print out the total number of unique words your function found, using `wordCounts.size()`.

Make sure to put a good comment right before your implementation of `processText()`, describing in detail what it does. This is a requirement.

Once you have finished this function, you can run it on this document. If you have written your program correctly, you should see something like this:

```
wcount < usdeclar.txt
Total words in document:  1326
Unique words:  536
of      77
the     77
to      65
and     56
for     29
our     26
...
```

Wait a minute...

Now, if you look at this output, you may be thinking to yourself, "How useful is it to know that 'of' appears in this document?" And that would be a very astute question! Words like "of" and "the" are tremendously common, and therefore they don't help us much at all when it comes to searching documents. Specifically, their presence or absence in a document does not fundamentally affect the subject or focus of the document. In fact, the first "meaningful" word that appears in the test document is "people", with a count of 10. Every word that appears before "people" in the list, really could be excluded from the list.

This is where the concept of a **stop-word list** becomes very useful. A stop-word list (aka "skip list") is simply a list of common words that we would like to ignore when we are processing a document. Just about every search engine has a skip-list of words that it will simply ignore in a search query; it has also ignored those words in its document preprocessing. (When Google or another search engine tells you that "such-and-such is a common word and has been excluded from your search," it is because that word is in its stop-word list. As an aside, it used to be hard to search for the band "The Who" because these words *both* frequently appear in stop-word lists, but of course, now search engines properly handle this special case.)

The STL `set` Collection

The STL `set` collection is another kind of associative container, one that we can use to represent a stop-word list in our program. A `set` only stores keys; it doesn't associate a value with each key. Thus, it is particularly useful in cases where you want to know if a particular item appears in a collection of items.

Sets are also templates, so you must declare what kind of items they will store. To keep a set of stop-words as `string` values, you would do something like this:

```
#include <set>

using namespace std;

...

set<string> stopWords;
```

```
    stopWords.insert(string("of"));
    stopWords.insert(string("the"));
    ...
```

As you can see here, the `set<T>::insert(const T&)` function can be used to put values into the set.

Sets are just like maps in that they also keep their contents sorted into a particular order. The implication of this (stated explicitly by the STL documentation) is that sets typically have log(n) insert and lookup times for operations. But, for our simple document processor, this is hardly a concern!

Iterating Over a Collection

Once you have things inside a collection, it is helpful to be able to iterate over its contents. For example, we might want to print out our list of stop-words. STL provides a concept of *iterators* to accomplish this task. An iterator is like a pointer, and it can perform these basic operations:

- `*iter` evaluates to the item in the collection that the iterator is pointing at
- `iter++` moves the iterator to the next item in the collection

Different kinds of iterators can perform other operations, such as `iter--`, but the two operations listed above are the most common.

Every STL collection has two functions, `begin()` and `end()`, that return iterators to the start and the end of the collection's contents. You can use these to iterate over a collection's contents. Here is an example of how to print out the stop-word list:

```
    set<string> stopWords;

    ...

    // Iterate over the stop-word list, printing each one to cout.

    set<string>::iterator iter;
    for (iter = stopWords.begin(); iter != stopWords.end(); iter++)
      cout << *iter << endl;
```

Note that `end()` does *not* point to the last item in the collection! Rather, it points "just past" the last item in the collection. Thus, `end()` really points to nothing. This is an important detail when using STL iterators!

Finding a Stop-Word

Once the stop-word list is populated, you can use it to find out if a particular word in a document is a stop-word, and therefore whether you ought to skip it. You can use the `set<T>::find(const T& key)` function to do this. This function actually returns an *iterator*. If the function finds the specified item, the iterator points to that item in the set. Otherwise, if the item isn't in the set, the returned iterator is `end()`.

So, we can now implement our stop-word checker like this:

```
    if (stopWords.find(word) != stopWords.end())
    {
      // It's a stop-word.
      ...
    }
```

If the `find()` function returns `stopWords.end()`, it means that the word simply isn't in the set. Pretty easy!

Attempt #2: A Smarter Word-Counter

Now that we know a little more about what appears in text documents, and we know about stop-words, we can implement an improved version of our word counter. Just like before, some framework code is already written for you, so download `swcount.cc` into your directory. (Don't get rid of `wcount.cc` though; you need to turn that in!) This program already contains functions to populate a skip-list, so you don't have to worry about that part.

You will again implement the `processText()` function, but it will be slightly different from the last one, to take into account the stop-word list. Here is what you must do:

1. Read a word from `cin`.
2. Process the word using `processWord()`, and skip over any words that end up with zero size after processing.
3. Update a counter of the total words you have seen. (This is a counter of the total number of *real* words you have seen; don't count words that get processed down to nothing.)
4. Check to see if the word appears in the stop-words set. If the word is a stop-word, update another counter of the number of stop-words you have seen, and then skip on to the next word.
5. Update the `wordCounts` map to indicate that the (non skip-list) word appears in the document.
6. Go back to step 1, until all words are consumed.

7. Print out the total number of words your function has seen, and the number of unique words you have catalogued. (Again, you can use `wordCounts.size()` for this.) Also print out the number of words that you skipped because they appeared in the stop-words list.

Like before, put a good comment right before your implementation of `processText()`, to describe what it does. Make sure to note the new additions in processing!

Once you have finished this function, you can run it over the U.S. Declaration of Independence again. If you have written your program correctly, you should see something like this:

```
swcount < usdeclar.txt
Total words in document:  1326
Unique words:  497
Skipped 587 words.
our     26
these   13
people  10
which   10
laws    9
states  8
...
```

You can see that our results are a little more helpful now, although the stop-word list probably could use a few more additions!

What to Turn In

Leave your completed versions of `wcount.cc` and `swcount.cc` in your `~/cs11/cpp/lab7` directory.

Extra Credit

If you want to make your document processor more clever, here are some ideas:

- Extend the stop-word feature to optionally load the stop-word list from an input file. If no stop-word file is provided, use the default stop-word list.
- This document-processor considers different forms of the same word to be distinct. For example, "cooks", "cooking", and "cooked" are all variants of "cook". You can extend the document processor to track all variants of a word as one "word group." (This technique is called stemming, and it is also essential for generating useful search indexes for documents!)
