Lab 10 – File Lexer

Overview

In this assignment you are going to create a file lexer. A lexer is a program that performs some sort of lexical analysis, which is essentially translating a stream of characters (from a file, user input, or some other source), into a series of **tokens**. A token is a string that has some sort of meaning to a program. A line of code in a source file could have multiple tokens like “void” “main()” or “int” which are then sent to a compiler for interpretation. The tokens themselves are inherently useless without some sort of additional processing. (“Additional processing” could be pretty much anything you want.)

In our case, the tokens are going to be words taken from the chapters of a novel. The files you’ll read will have some information you need to filter out, so in the end you just have the words themselves with no punctuation or even capitalization. Just the words themselves. You’ll use a `map<string, int>` to store them (the integer will be how many times the word appears in the file).

Description

First things first, the files: There are 4 main files that you will be loading in this assignment:

Ipsum1-4.txt

> What’s in these files? Lorem Ipsum, seemingly random “sample” text used to simulate real content on a printed page. The code you use to read these files will be the same whether it’s sample text or a “real” file.

To start, you need to get the words from the file. A good approach for this (like you may have done before) is to get a single line from the file, create a `stringstream` object from that line, then using getline() with a space as a delimiter, break that line into multiple pieces (i.e. individual words).

Once you have the word, you need to make sure it’s properly formatted, so you can later search through the data without concern for whether the data is “clean” or not. You want to store them, and the number of times they appear in the list, for later access.

Rules

1. Words should have no blank spaces at the beginning or end. Removing these leading or trailing whitespaces is often referred to as **trimming** the string.
2. In addition to trimming any whitespace, you also want to remove any non-letter characters from your string. This means punctuation marks like periods, commas, etc… if it isn’t a letter, we don’t want to keep it.

For example, let’s say you had this string:

```
string test = "2%$ 3948BATMAN   1 1 49138&%(!";
```

After converting, you would be left with just `"batman"`

Note about rules: Depending on the file (and your desired results), there could be a LOT of rules for something like this. You could have a word like "brother" but modify it to be plural or possessive such as brothers or brother's. Do you store these 2 separately? Or are they really the same, just a variation of the base word (if so, how do you determine that)?

If that's the case, you may have to write a lot of code (with a lot of rules) to determine exactly how you want to treat a particular word. Developing a "grammar" from a particular set of data can be quite difficult. Think of all the rules and exceptions in English. Now, think of how to write a program to support all of that.

In this assignment, the rules come down to really just one rule: Anything not a letter is removed from the string.

How to remove something from a string?

Removing character from either end of a string is easy. `std::string` has two functions you could use:

`string::pop_back()`	removes the last character from the string
`string::erase()`	erase some number of characters based on the parameters you specify. You can specify an index and a count of characters, or use an iterator (such as `begin()`) to indicate a location

For more info: http://www.cplusplus.com/reference/string/string/erase/

The characters that you want to erase in this assignment are anything NOT letters. **A-Z** as well as **a-z** are what we want to keep. Keep in mind that these characters are just numeric values:

Character	ASCII Value
a	97
b	98
…	…
z	122

Character	ASCII Value
A	65
B	66
…	…
Z	90

You can use characters in numeric checks as well—they get converted to their integer equivalent:

```
// Check if a character is upper-case
char someLetter = 'C';
if (someLetter >= 'A' && someLetter <= 'Z')
    cout << "It's upper-case";
```

That essentially checks if 67 (`'C'`) is greater than or equal to 65 and less than or equal to 90 (which it is).

When removing something from a container (ANY container, not just strings), you could end up removing everything from that container. Make sure you aren't try to perform any operations on nothing at all.

Converting to Lower/Uppercase Characters

Words should be all converted to **lowercase** for comparison. For example, without modification, "Dinosaur" and "dinosaur" are two different words. So… if you had this:

```
Word: Dinosaur Count: 10
Word: dinosaur Count: 12
```

It should instead report (after converting):

Word: dinosaur Count: 22

If you wanted to track the original capitalization of a word, you would have to store that as a separate piece of data—another map, perhaps? For example, `map<string, vector<string>>` could be used to store the lower-case version as a key, and all the permutations originally found could be stored in a vector.

There exists a function in the header file `<cctype>` called tolower(). It takes a single character and returns the lower-case version of it. If the conversion can't be made (what's lower-case **#**, anyway?) it just returns the original parameter, unchanged.

```
// Example
char tiny = 'B';
tiny = tolower(tiny);
cout << tiny; // Prints b
```

http://www.cplusplus.com/reference/cctype/tolower/

Convert then trim? Trim then convert?

There isn't anything inherently wrong with doing one of those before the other. If you trim first, you will have to check for upper-case letters. If you convert first, you will potentially be comparing more characters with tolower() than you might have to… both approaches can get the job done in the end (and the extra computations that might seem "wasteful" in either approach are very minor).

What's next?

After you have cleaned up the string, your token is ready for… whatever you want! In this assignment, you'll perform some simple operations:

Show the word list and a count.
What's the most common word from the list?
How many words longer than a certain number of characters?
How many times does <some word> appear in the list?

Iterators

The key to going through STL containers is to use iterators. While some objects like vectors and strings store their data contiguously, making simple for loop iteration possible, that's not true of all containers. For that, you need an iterator. In C++ iterators commonly revolve around the use of 2 functions: begin(), and end().

Want to start with the first element in a list? That's begin(). Want to reach the end of the list? That's just BEFORE end(). Before? Why not on end()? The end() function returns an iterator that is beyond the range of elements.

Think of it like this: if you were looping through an array of 10 elements, the valid indices would be 0-9. The "end" index would be 10, or 1 past the last element. With iterators, end() functions the same way. In both cases, you should never try to USE that "just past the end" element, but it can be helpful to check against that. Check the slide presentation on iterators for more information.