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.

In our case, the tokens are just going to be words taken from the chapters of a novel. You will have to read in several files, break them down into individual tokens and store information about those tokens—things like the number of occurrences, the largest and smallest tokens, etc.

Description

First things first, the files: There are 5 main files that you will be loading in this assignment:

files1-4.txt – the main files you will be loading/parsing
words_to_ignore.txt – containing a list of commonly used words to ignore for some operations (words like a, to, I, the… etc).

The general order of things will be:

1. Get the words from the file. Getting an individual word is the same as things you’ve seen before; all the data is separated by spaces, so you can use that as a delimiter to get the ball rolling.
2. 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 the words, 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 typically referred to as **trimming** the string.
2. In addition to removing spaces, words should have only letters at the **beginning** and **end** of the word. Anything not a letter should be removed.
3. All letters in the word should be converted to lowercase.

Exceptions

1. Some words are hyphenated, and **should** be allowed as a single word. For example, “merry-go-round” is a single word, and should not be converted to “merrygoround”
2. Single quotation marks ARE allowed if it is the only non-letter at the end of the word

Examples

Word in file	Word stored as
“Hello	hello
world!”	world
WasN’t	wasn't
wood-cutter	wood-cutter

String Trimming?

Removing a character from the beginning or end of a string is easy. You can write code to do that manually, or use the existing functionality of the `std::string` class. Like many standard library containers, the `std::string` class contains a function called erase(), which takes an iterator to the element that you want to remove from the container. In addition, you can use the pop_back() function to remove the last character. For example:

```
string example = " Batman "; // Need to clear the leading/trailing spaces

auto first = example.begin();
example.erase(first);

/*==== OR ====*/
//example.erase(example.begin());

// Erase the trailing character
example.pop_back();
```

What if you had a string with more than one character in front or in back? For example:

```
// How to clear all of the non-valid characters from the beginning and end of this?
// Answer: The same was as in the previous example… just do it more than once!
string example2 = "&( Batman   ;/.,?><123 ";
```

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. If “Dinosaur” appears 10 times in the data and “dinosaur” appears 12 times, your output should show this:

```
Word: dinosaur Count: 22
```

Instead of this:

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

(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, or a custom class?) This is not a requirement for this assignment, but something to think about if you were to work on something similar in the future. Then you might have an object to store the original form of a word and the lowercase version which you’ll actually use for searching.

Two functions exist to help you with this: toupper(), and tolower(). You will need to #include **<cctype>** in order to get access to them. Pass in the character you want to convert, and get the converted character in return.

Data Structures

While you could accomplish this task in a variety of ways, the standard template library **map** classes will probably be the most helpful: `unordered_map<Key, Value>` and `multimap<Key, Value>` in particular.

unordered_map

This stores the data in **pairs**, with a unique **Key** and some **Value** associated with that key. Since the purpose of this assignment is store each word and the number of occurrences in the file, this would be a perfect fit. An unordered_map is not sorted, which is fine in most cases. (If you need your data sorted, you would have to use something else).

multimap

A `multimap` also stores data in key/value pairs, but the difference here is that the stored keys don't have to be unique. Also, like the `map` class, a `multimap` is **sorted**. Depending on the context you may not need or want to have your data sorted, and so using a `map`/`multimap` would be unnecessary in those scenarios.

Let's say you wanted to sort the words in an `unordered_map` by their value (highest to lowest). By its very nature, the `unordered_map` is not sorted, so you can't do this. A map, on the other hand, DOES sort… but by key, and not value. So if you copied your `unordered_map` data into a map…

`unordered_map<string, int>` → `map<string, int>`, then you would sort by the words you are storing, regardless of their counts. What if you reversed the key/value data types? A `map<int, string>` would still sort by key, but now the key is the wordcount! This could be problematic, however…

```
// #hypothetical scenario
unordered_map<string, int> words;
words["hello"] = 10;
words["goodbye"] = 10;

map<int, string> words2;
words2.emplace(10, "hello");
words2.emplace(10, "goodbye"); // Nothing added, key already exists
```

A multimap works just like a map, but its keys are not required to be unique.

```
multimap<int, string> words3;
words3.emplace(10, "hello");
words3.emplace(10, "goodbye"); // No problem at all! New pair added
```

Copying from one to the other

To copy all of pairs of an unordered_map into a regular map (or multimap), you would need to iterate through all elements of the unordered_map, and one pair at a time, add a new pair to the destination, possibly flipping the key/value if that's your desired behavior.

Standard Library Functionality

Two functions that you may find helpful when searching your containers, or when finding/displaying the results:

```
// Search a container for a value, returning an iterator to where
// it was found (or a "last" iterator if it wasn't found)
std::find(iterator_first, iterator_last, search_term)
```

Parameter 1: The start of the container
Parameter 2: The end of the container
Parameter 3: The value that you're looking for

```
// Sort a container in ascending order
std::sort(iterator_first, iterator_last)
```

Parameter 1: The start of the container
Parameter 2: The end of the container

References for these functions:
http://www.cplusplus.com/reference/algorithm/find/
http://www.cplusplus.com/reference/algorithm/sort/

In both cases, you will need iterators for your containers. As has been discussed before, iterators make traversing standard libraries a breeze.

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 permutations 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.

Also, depending on how you want to iterate through an object, there exists the ability to iterate in **reverse**, using the reverse begin function. To do so, you would initialize an iterator to the rbegin(),

instead of the begin(). When checking for where the iterator should end, use the rend() function instead of end().

Check the slide presentation on iterators for more information.

class WordInfo

You are going to create a class for this assignment called WordInfo. It should contain the following public member functions, and any other functions you think would be helpful (especially one to do a partial search [i.e. one which returns a list of words which CONTAIN the search term]).

```
// Open and read the indicated file, storing the words
// in the file and how many times they appear
void ReadWordsFromFile(const char *filename);

// Print out the number of words, number of unique
// words, and average word length
void DisplayStats() const;

// Print out the "count" most frequently found words in the file.
// The optional boolean variable indicates whether to use
// the list of very common words to ignore
void MostCommonWords(int count, bool ignoreCommonFile = false) const;

// Finds the longest word(s) and stores them in the output parameter
void LongestWords(vector<string> &words) const;

// Set a list of words to ignore when searching for
// frequently occurring words in the file
void SetIgnoreWords(vector<string> &ignore);

// Find search for an exact match, return the number of occurrences
// Return 0 if no occurrences are found.
int SearchForWord(const char *word) const;
```

Miscellaneous

Finding a string in another string:

There exists a function in the std::string class, called find(). You can pass to it another string, and it will return the **starting index** of where that sub-string can be found. If the passed in string is NOT found, the index of where to find it will be returned as -1. However, since the return type of the function is a size_t, which is effectively an unsigned integer (or unsigned long long on 64-bit targets), the number underflows to the maximum value. A handy shortcut for this is the variable “npos” in the string class, which can be used to check for a “not found” result. So you would want to check the return of the find() function against the value of string::npos.

Accessing a map with brackets in a const function

The brackets operator can create a new pair in a map object, should the specified key not exist. As a result, the brackets operator is not usable in a const function. To work around this, you can retrieve a particular value associated with a particular key by using the at() function instead of brackets. (This is identical in concept to the vector class, which has both operator[] and at() defined.)

Word Variations

Depending on your needs in a program, you might want to store variations of words independently, or track them merely as variations. Let's look at the word "student" as an example. Variations of this word might be:

students – plural form
student's – singular possessive form
students' – plural possessive form

In this assignment, these should all be different entries.

Sample Output

A run of the program using file3.txt:

```
1-4: Which file to open?
3
Total number of words: 2934
Number of unique words: 681
Average word length: 4
Longest word: pebble-stones
4 most frequent words:
   the 181
   and 166
   to 72
   they 48
4 most frequent words (ignoring most commonly used):
   they 48
   hansel 45
   had 40
   gretel 36
Longest word(s):
   neighbourhood
   pebble-stones
Enter a word for an exact search: forest
'forest' was found 18 times in the list.
Enter a word for a partial search: rest

Words containing 'rest'
forest
rest
```