CS2201
Fall 2019
Assignment No. 8: Shake-n-Bacon

Purpose:
Gain experience in using container classes & iterators from the C++ STL.

Background:
William Shakespeare is arguably one of the most notable authors of the English language. The ultimate wordsmith, many of the phrases he coined remain in common use today and his plays are regularly performed around the world. But some have doubts that the bard actually wrote the texts commonly attributed to him.

The most common theory is that while it is true Shakespeare was an actor and his name was connected with the works, he did not have the education required to write the plays and sonnets. Depending on which of the many theories you read, many different authors are put forth as ghost writers for his works. One of the first and most popular candidates is Sir Francis Bacon. In completing this assignment, you will produce information that will support or detract from the simplified theory that Bacon wrote Shakespeare's works.

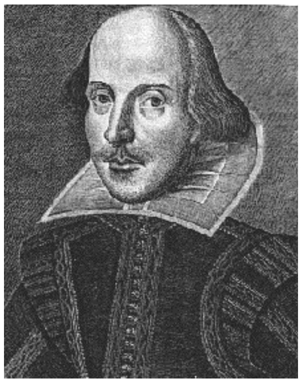
Figure 1: Shakespeare

First, as on any good expedition, you need to bring the right tools. In evaluating authorship, we will be using two major tools:

- Word usage comparison
- Word Stemming

Figure 2: Bacon

Word Usage Comparison

In general, each person has their own unique writing "signature" that is evident in the words and grammatical structure they use. Due to differences in vocabulary, speech patterns, and other environmental factors, these signatures can be identified by a light form of statistical analysis. Basically, you count each time a word is used in a document and compare the percentile appearance to the same word in another document. If the word is used a similar amount in both documents then it is more likely that the author was the same person. By repeating this process for all unique words in the document, a conclusion about authorship of a new document can be made with higher certainty.

A few limits are imposed on the comparison process to make it more accurate. First, many words are commonly used within the language itself or the files under consideration. For example, in English you would expect the word "the" to have a high count in most documents. You will be provided with a list of words that are to be completely ignored in processing the documents. Second, some words are very rare in one of the files and provide very little information about authorship differences. These words tend to be proper names or obscure words that are not likely to be shared even by works from the same author. We will not compare the counts of these words, but their appearances still count towards the overall total number of words in the file.

Word stemming

The second tool that you will use is called word stemming. Word stemming is a very difficult problem to solve programmatically and continues to intrigue those in the study of Natural Language Processing (NLP). The primary purpose of word stemming is to recognize when two words share a common meaning or root. A good example of stemming is the tense of a verb. "Run", "running", and "ran" all share the same root ("run"), but our program as described so far will count them as different words. The history of English as an amalgamated language makes generalizing rules for stemming difficult. If we tried to write a program that stemmed all words that ended with "nning" to a single "n" we would get the following transformations:

```
     Running -> run   (OK)
     Spanning -> span (OK)
     Inning -> in     (Bad, "inning" cannot be stemmed here)
```

There are similar exceptions for many of the other simple rules we might think up.

You are welcome to explore the concept of stemming further outside of the class, and there are a number of more interesting algorithms commonly used including porter stemming (http://tartarus.org/~martin/PorterStemmer/). For this assignment though, we will use a very primitive form of stemming by shortening every word found to STEM_LENGTH which is declared as 6 characters. More complex forms of word stemming are used in spell checkers and search engines.

The Assignment:
In this assignment, we are going to create a program that will count the number of occurrences of words in a file, while ignoring words that are considered to be "common". Using such a list, we will compare two files to attempt to answer the authorship of Shakespeare's works. This program is actually quite easy to implement when using two containers from the STL. We will use the **`set`** container to hold the set of common words we are not interested in counting. We will use the **`map`** container to hold pairs that represent a word-to-count mapping (i.e.; each word will map to an integer that represents how many times that word appeared in the file).

Step I:
Unzip the supplied file. It contains a `WordCount.cpp` file and a "texts/" subdirectory with the input files. Create a new CLion project for this assignment by selecting "New CLion Project from Sources" and select the directory you just unzipped. Make sure the supplied `WordCount.cpp` file is included as a part of the project. The project may compile cleanly (depending upon the compiler flags) but it will not run correctly until you add the required functionality, or it may not compile due to the parameters not being used yet. Be sure to set the working directory of the project to be the same directory that contains all your source code and the "texts/" subdirectory. See the `CMakeLists.txt` file provided for project #7 for how to specify the working directory and for specifying the clang compiler. Make sure you add the full set of compiler flags that we have been using on recent projects to the `CMakeLists.txt` file of this project by copying them from project #7 (they are also listed at the bottom of this document). Finally, please review the set-up instructions provided on Brightspace at the beginning of the semester (under Content→Course Resources) and ensure that your project is using the clang compiler (students have had problems in the past when their projects defaulted to using the gcc compiler). [We did not provide a project for this assignment since you need the ability to create your own projects when you get to your next programming class.]

Take a moment to view the supplied code. You will notice some defined constants and two typedef declarations for types we will be using. The main function consists mostly of some function calls. You are encouraged to add your own helper functions as needed.

The input files will be located in the subdirectory "texts". The output file will go into the same directory. When the main program prompts you for the names of the input files, you simply give the filename – the program will add the "texts/" subdirectory specifier for you.

Step II:
Implement the `readCommonWords()` function. The function needs to create an object that is a set of strings (an object of type `WordSet`). This function then opens the desired file (see the first half dozen lines of code in the Maze class constructor of Project #7 to see how to open a file for input). After verifying that the file was opened successfully, read all the words from the file (use the extraction operator with a variable of type string; performing the extraction as the test of a while-loop) and insert each word into the set. When done processing the file, close the file and return the set of words. Please refer to the supplemental information (MapSets.pdf file) provided with the project on how to use sets.

Step III:
Implement the `processFile()` function. This function takes two arguments: the name of the file to be processed and the set of words that are considered "common". This function needs to create an object that is a map (an object of type `WordCountMap`) which will map a string (a word) to an integer (the count of how many times that word appears). This function then opens the specified file, verifies it was opened successfully, and then reads & processes all the words. This process has three steps for every word (C++ string operations are described here):

1. Convert all letters to lower case and remove punctuation & other non-alphabetic characters.
2. Use our simple form of stemming: truncate any word that is longer than six letters to the first six letters.
3. Determine if the resulting word is in the set of common words. If it is a common word, we ignore it. Similarly, we ignore any resulting word that is an empty string. If the resulting word is not a common word and is non-empty, we find its current count in the map and increment it by one.

When done processing all the words in the file, we close the input file and return the map to the caller. Please refer to the supplemental information (MapSets.pdf file) provided with the project on how to use maps.

Step IV:
Implement the `compareTexts()` function. This function takes three parameters; the first two parameters are the maps of the two files to compare while the third parameter is the output file name. The function returns the distance between the word usage in the two files. The function needs to compare the files as described below. Each word will have a percentile measurement (or *score*) for its appearance found by dividing the count for that word by the sum of all counts within the `WordCountMap`. The percentile measure helps to normalize the information so we can more accurately compare texts with largely different sizes.

To compare the two texts, we will be using a version of Euclidean distance. In this case, each `WordCountMap` is an n-element vector where each word has an entry. To be fair in computing the distance we want to ignore words that appear very rarely and will only compare words that appear in *both* files with a percentile score such that: `0.001 < score`. So, for each word that appears sufficiently in both files, add the square of the difference (see the following expression) to a running summation:

$$(\texttt{score1} - \texttt{score2})^2$$

Where `score1` and `score2` are the percentile scores for the word in each `WordCountMap`. The final sum represents the square of the distance between the two vectors that represent the files – we then take its square root to compute our final distance. Before exiting, your program should print this distance to the output file as shown in the example output below. The numeric value printed does not have to be formatted.

Example output:
```
Vector Distance: 0.0195142
```

As your program is calculating the sum, it will also need to print to the output file the squared differences of all words that contribute to the sum of the file differences. Simply use the insertion operator[*] to print data to the file. The format for each line in the file is
```
Word:     distance = x.xxxe-xx   (score1 = x.xxxxx, score2 = x.xxxxx)
```
To help line things correctly, there is a tab character after the "`:`" and another tab character before the "(". The words should be printed out in lexicographical order (actually, when you iterate over a map you process the data in sorted order, so you get this automatically). Your output must match this format exactly. Once the final sum has been computed, write a blank line and report the number of counted words in each of the documents. Then write another blank line and report the vector distance. A sample execution is given later in this project specification. Finally, close the output file and return the distance to the caller.

Step V:
Test your program. Along with the problem specification, you have been provided with 5 files. Two of these files (`sonnets.txt, hamlet.txt`) are by William Shakespeare. Two of the files (`poems.txt, alchemist.txt`) are by Ben Jonson, a contemporary of Shakespeare who has not been strongly accused of writing for William. (Jonson is one of the authors from that era with the most historically documented life.) You should run your program and compare these four files against each other to find out what range of distances to expect between works by different authors and different types of work by the same author (poems vs. plays). Then, run your program on `hamlet.txt` and the final file, `atlantis.txt` which was written by Bacon. Observe the distance that was returned and, within the context of the previous runs, describe whether you think Bacon wrote Shakespeare's works. Place this short write up (1-2 paragraphs) in your `README` document (see the next step for details).

Step VI:
Answer the following questions in your `README` document (a .txt text file or a .doc Word document). Note: this write-up <u>constitutes 20% of your grade</u> for this project.
1. List your name and email address.
2. After reviewing this spec and the provided files, please estimate/report how long you think it will take you to complete it.
3. How many hours did you actually spend total on this assignment?
4. Report all pairs of documents you compared and give the vector distance your program reported for the pair.

[*] Depending upon your platform, the output of the insertion operator may vary in the number of digits printed or the number of digits in the exponent.

5. Give a short, but complete, write-up explaining whether you think Bacon wrote for Shakespeare or not. Defend your position using the results of your experiments. No fancy statistical analysis is required; just keep it fun & simple. You are explicitly disallowed to argue that our analysis is invalid. This should be 1-2 paragraphs long. Use full sentences and proper English when completing this portion of the `README`.
6. Who/What did you find helpful for this project? If you received assistance from a person, who were they and what assistance did they provide?
7. Did you access any other reference material other than those provided by the instructor and the class text? Please list them.
8. What did you enjoy about this assignment? What did you dislike? What could we have done better?
9. Was this project description lacking in any manner? Please be specific.

Step VII:
For grading, simply submit your updated `WordCount.cpp` file and your `README` document. Please do not submit a CLion project.

Programming notes:
1. On the assignment page in Brightspace is a small PDF file that provides some basic information about using the STL map container and set container (courtesy Prof. Claire Bono at USC). Here is another useful map reference: http://www.cplusplus.com/reference/map/map/. Please note that there is a gotcha with the subscript `[]` operator; read the second paragraph and see the example on this page: http://www.cplusplus.com/reference/map/map/operator[]/. And here is a useful set reference: http://www.cplusplus.com/reference/set/set/. There are many other tutorials available on the web.
2. Please give careful thought how you use these sets and maps in your program. Do not treat your sets and maps simply as arrays (e.g., by always creating iterators and indexing through the entire container) – doing so always results in O(n) performance when these containers are capable of O(lg n) performance in many cases, in particular searching. See the above mentioned references for operations available for these containers. There will be a penalty assessed for O(n) processing when O(lg n) processing is possible.
3. In this assignment, we will use the **set** container to keep a set of common words. Thus the collection of common words is a set of type string. We will use the **map** container to map a word to its corresponding count. Thus the map will take keys of type string and map them to values of type int. With the map you can use the subscript `[]` operator, using a string key as the index, to set or retrieve the corresponding count. Note: it is not always appropriate to use the subscript `[]` operator when reading data from a map – be sure you fully understand the implications of using the subscript operator with a map.
4. The .cpp file uses a **typedef** to define the types **WordSet** and **WordCountMap** which are aliases for the required set & map that we will be using. This type will make it easier for us to refer to the required types – which we need to do each time we pass them as a parameter, create local objects, and create iterators.
5. The provided program declares several functions as described above. You must implement those functions as given. You are not allowed to change the function names or parameter lists, as our grading script depends upon them. You can define as many helper functions as you wish. You should write helper functions to isolate any complex or specialized segments of code, regardless of how short those segments are.
6. When comparing the word counts, you need to process all the terms in the map. You can do this by using an iterator. You can declare an iterator in this manner: `WordCountMap::iterator p`. If you ever need to process all the terms in a map that is in a const map, you need to specify a `WordCountMap::const_iterator`. Once you declare the iterator, you can initialize it to `.begin()` of the map and then increment it using `++p` as long as it does not equal `.end()` of the map. Given an iterator `p`, you can access the associated key (or word) using `p->first`, and you can access the associated value (or count) using `p->second`.
7. Each file you turn in should have the standard block comments at the top of the file, including an honor statement. Be sure to use good programming style.
8. After you implement the required functions, please remove the TODO comments as they are no longer applicable.

Sample execution:
Here is what the execution of the WordCount.cpp program should look like:

```
Enter name of the first input file: sonnets.txt
Enter name of the second input file: poems.txt
Enter name of the output file: comparison.txt
Vector Distance: 0.0195142
Would you like to run the program again (Y|N): n
```

Given the supplied files "`common.txt`" (common words), "`sonnets.txt`" (compared) and "`poems.txt`" (compared), here are the first six and the last nine lines that should appear in the output file "`comparison.txt`":

```
age:   distance = 1.09468e-06     (score1 = 0.00140043, score2 = 0.0024467)
alone: distance = 6.88361e-10     (score1 = 0.00177388, score2 = 0.00174764)
am:    distance = 3.49524e-06     (score1 = 0.00326767, score2 = 0.00139811)
an:    distance = 7.38815e-07     (score1 = 0.00158715, score2 = 0.0024467)
are:   distance = 3.00948e-07     (score1 = 0.00644198, score2 = 0.00699056)
art:   distance = 5.35813e-06     (score1 = 0.00476146, score2 = 0.0024467)
...
why:   distance = 1.65242e-06     (score1 = 0.00233405, score2 = 0.00104858)
world: distance = 1.05946e-06     (score1 = 0.00242741, score2 = 0.00139811)
would: distance = 6.98267e-07     (score1 = 0.0019606, score2 = 0.00279623)
yet:   distance = 4.73473e-08     (score1 = 0.00476146, score2 = 0.00454387)

Counted words in document 1 = 10711
Counted words in document 2 = 2861

Vector Distance: 0.0195142
```

Again, the text is lined-up by inserting a tab before "distance" and also before "(score1". Your program must exactly produce this output format (modulo platform specific differences in number of digits printed with insertion operator).

Further Information:
Wikipedia's overview of the Shakespeare controversy: http://en.wikipedia.org/wiki/Shakespeare_authorship_question
Here's a recent article on plagiarism detection software finding new sources for 11 of Shakespeare's plays.
Project Gutenberg: Source for public domain texts: www.gutenberg.org
Stylometry is the application of the study of linguistic style, and it is used to attribute authorship to anonymous or disputed documents. The NSA used stylometry to identify Satoshi Nakamoto, the world's most elusive billionaire.

Compiler flag for CMakeLists.txt
Make sure that you CMakeLists.txt file contains this set of compiler flags:

```
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++17 -Wall -Werror -Wextra -pedantic -pedantic-errors")
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR})
```

Also make sure that you are using the clang compiler to build your project.

Acknowledgement:
This project is based on a project given at the University of Washington.