WordCount readme

David Miranda

(Last update 4/MAR/2019)

Contents:

1. Introduction

WordCound is a command line tool (written in C++) that counts the frequency of the words in files of a specific directory. After running outputs, the 10 most repeated words from each file and the overall 10 most repeated words in a folder. The output is written to a text file with the location defined by the user, or if no output destination is provided the result is echoed to the terminal.

For the purposes of this program it as assumed the following:
1-A word is considered to be any sequence of letters of the English alphabet (‘a’ to ‘z’ and ‘A’ to ‘Z’) that is not interrupted by other characters, such as: !?@.£3()*->’ , spaces, character change of line, etc. For instance ‘re-open’ is counted as two words: ‘re’ and ‘open’.
2-Two words are equal if they have the same letters in the same sequence, independently of the case. For instance ‘Matlab’,’MATLAB’ and ‘matlab’ are the equal words.
3-The text files to count should are in ASCII. Other encodings such as Unicode are not supported.
4-The files are small enough to fit in the memory (<20Mb).

2. Install

2.1 Linux and macOS

In the terminal change to the source directory:

```
$cd {source_directory_path}
```

and run the command make:

```
$make
```

then you can copy the file executable to an installing directory :

```
$mv WordCount {path_for_installation_directory}
```

(please make sure that the gcc compiler and the utility make are installed in the system)

For an optimized compilation, possibly faster run:

```
$make optimized
```

Warning: this might introduce a bugs.

2.2 *Windows*

This version is no tested version available in windows.

3. Run and test

In the terminal, change to the working folder to where the executable WordCount is installed, run:

```
$cd {path_of_installation_directory}
```

for having the output report of the counts echoed the terminal run:

```
$./WordCount –d {installation_directory}
```

Otherwise, for having the output of the counts written to a directory run:

```
$./WordCount –d {installation_directory} –o {output_file}
```

Examples for testing the code with empty, small and large files are provided in the directory *test_files*. A basic test consists in running the program thought those files and then comparing the results with the counts obtained by using the functionality “find” or “find in files” of a text processor.

4. Implementation and application design

4.1 Overview

In this section the requirements of the program, design options and implementation are explained. Nevertheless, the code is extensively commented and can provide additional explanations.

The requirements of the program were:

R1 easy to understand

R2 robust

R3 documented

R4 testable and

R5 can be extended if necessary

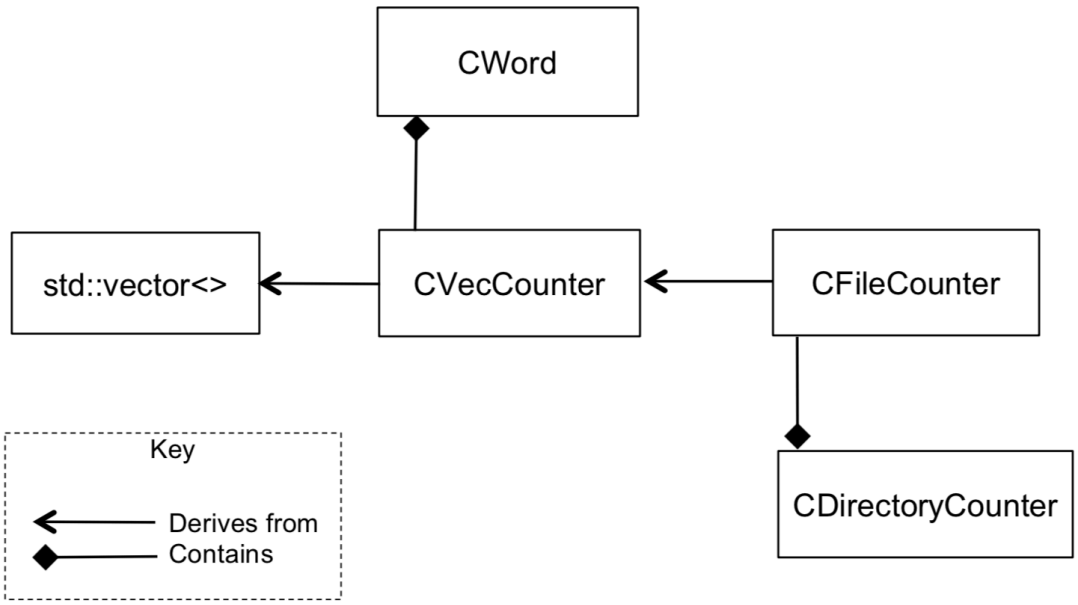
In terms of design decisions we assumed that:

1. To comply with R4, R5 object-oriented design would be better. That also made the code easier to understand (R1);
2. In terms of performance of memory and speed of search, different data structures to store and the words entries could be used: arrays, dynamic vectors, hash tables (maps), trees of search. But to make the code easier to understand (R1) and to be robust (R2), we simply decide using a vector structures that can grow dynamically as intended, nevertheless allow random access;
3. Standard STL libraries were used as much as possible in order to comply with (R1) and (R2), since those libraries hide some complexities, are robust and well known among C++ developers;

4. To comply with (R1) and (R3) we use long names for the methods and variables, along with some naming convention and commented the code extensively.

The code is structured in the objects: C*Word, CVecCounter,* C*FileCounter* and C*DirectoryCounter*. The main program parses the arguments given in the terminal and instantiates one single object of the class C*DirectoryCounter*. Then that object is used to count and output information regarding the files of the directory.

The following sub-sections 4.2 to 4.5 describe in more detail the classes of objects used. Nevertheless, the dependency between classes is represented in the following diagram.



Schematic diagram of relations between classes

4.2 CWord

An object of the class *CWord* abstractly represents a word entry and is the most fundamental class used since all other instances might contain several instances of this class. *CWord* is simply a string for the sequence of charaters

and a number with the count of occurrences, with functionalities such as increment the count, copy itself and output to a stream (for text output).

4.3 CVecCounter

The class CVecCounter provides support for storing different words (*CWord* objects). The structure was implemented with the standard STL, by deriving from the template vectors (vectors and can grow dynamically, but also provides random access along with useful pre-programed methods). The class *CVecCounter* extends that functionality with the methods:
-*findWord* to find or the a word entry in the set;
-*addWord* to a word if does not exists in the set, otherwise it updates the counter;
-*sortByNumberCounts* to sort the words by number of counts. Note that only sorts until getting the most repeated words that will be reported. For instance only until the 10^{th} if 10 are to be displayed;
-*mergeAnotherCounter* allows to merge in the words from another counter, i.e. add in all the words from another counter;
-overload of the stream operator<< for (for text output).

4.4 CFileCounter

The class *CFileCounter* derives from *CVecCounter*, to extend its functionalities for parsing the text files into words.

Those extended methods are:
-*contWordsInFile* counts the words in a file by calling count *WordsInSring*
-*countWordsInSring* counts the words in a string. This method is extremely important in case of future extensions of the code, because it is where the parsing of the words of the text happens. The implementation runs a fairly simple loop on all the characters alternating between two states (indicated by the flag *bIsReadingAWord)*: reading a word or not reading a word. See the code for more details;

-The methods *isInAlphabet* and *makeCharLowerCase* are used by the parser *countWordsInSring*.

4.5 CDirectoryCounter

CDirectory is a class in charge of handling the counting of the words over the files of an entire directory. Essentially handle the opening of the folder, gets all the file names, accumulate the file counters while keeping track of all the file counters and outputs the results.

Since the number of files is unknown at start, the counters of the files (objects of *CFileCounter*) are hold in the vector *VectorFileConter*. Furthermore, the files' names and their opening status are also kept in vectors. A general counter *TotalDirectoryCounter* it is used to accumulate the total of counts of word counts in the files.

The most relevant methods are:
-the constructor *CFileCounter*, that is in charge of opening the folder reading the file names and calling for *countWordsOfAllFiles*;
-*countWordsOfAllFiles* loops all the file names and reads them into *FileCounter* the file counters while accumulate them in to the general counter *TotalDirectoryCounter*;
-the stream operator<< , that was overloaded in order to facilitate the outputs booth to the terminal and external file.

5. Future extensions

Since the code is written in a object oriented style, is well divided in his components, all the methods are relatively small and with very well defined purpose its is relatively easy to extend. For this one should derive its classes and re-write some of the methods. Here are some ideas:

For instance the method CFileCounter:: *countWordsInSring* be re-written in order to encompass more broad definitions of words that the assumptions described in the introduction.

The class *CDirectoryCounter* can be derived in order to preform recursive processing of the sub directories as well, i.e. create a report for all the subdirectory three. Also the classes are written to report any number of most repeated words (10, 5, 30, 100). Therefore, the main program can be adapted to generate reports with any number of most repeated words defined by the user.

Finally, the class *CVecCounter* can be derived to introduce filter methods that would allow to remove certain kinds of words from the counters such as slang words, or even methods for recognizing synonymous and count them as a single word.

6. Unit test development

Unit tests should be develop in future to guarantee the quality of the code. Those automatized test units should be able to read test data folders with known results for automatic comparison. Furthermore, test units capable of generating random text files given a set of parameters should be developed and run varying those parameters.

7. Conclusions

Probably the code would be much easier to implement in a scripting language such as Python, since contrary to C++ it offers a more direct support for file manipulation. System administrators often use python to automate the management of files in the servers.

Nevertheless, C++ is the probably faster. Furthermore a C++ code is probably more interesting to the client as they develop most of the code in this language.

The tests provided with the source code deriver some degree of confidence in the application. Nevertheless, test units should be develop in the future in order to guarantee the quality of the code.