



CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering

WordCount implementation in C++

B6B36PJC

Semestral project

Marek Szeles

Advisor: Ing. Radek Havlíček, Ph.D.

Prague, 2017/2018

1 Introduction

This semestral project was about the creation of a simple “wordcount” program, which could take in arguments/ files as an input and count the number of words in them.

Said functionality is achieved through a simple command line based GUI which guides the user through the options he has at hand.

1.1 Word definition

For the purpose of this program a word was defined as *“a string of characters between two spaces or a space to the left and a comma/dot to the right”*.

In the case of words split mid-sentence (ending on one line with a hyphen “-” and continuing on the next one), the word count of the first line is decreased by one and the whole word is being counted as one for the second line. This is to prevent duplicate counting of the same word.

In the “*word frequency*” function, lower or upper case letters are neutralized – e.g. “*Hawk*” is treated and counted as the same word as “*hawk*”.

1.2 Limitations

When running the “*word frequency*” function, all words are simplified to be lowercase and without punctuation. This causes inaccuracies, as “*John’s*”, “*Johns*” and “*Johns*” are all treated as the same word.

Furthermore, in the same function, the word split between lines is not taken into account and thus “*the-”\n“rapy*” would be counted as two distinct words: “*the*” and “*rapy*”. It only concerns a very small number of cases though, so it shouldn’t matter much in long texts. It leads to an interesting phenomenon however that in small files (for example the provided “*test4.txt*”), this can lead to a seemingly larger number of unique words than actual words (in said example 6/5). This is exactly due to this word-splitting and the “*word count*” function functioning more correctly.

These inaccuracies were decided to be out of scope for this semestral project, but should be taken into account nonetheless.

2 Program compilation and build

The program was coded and tested within the JetBrains CLion IDE, so preferably use the project files provided to build it.

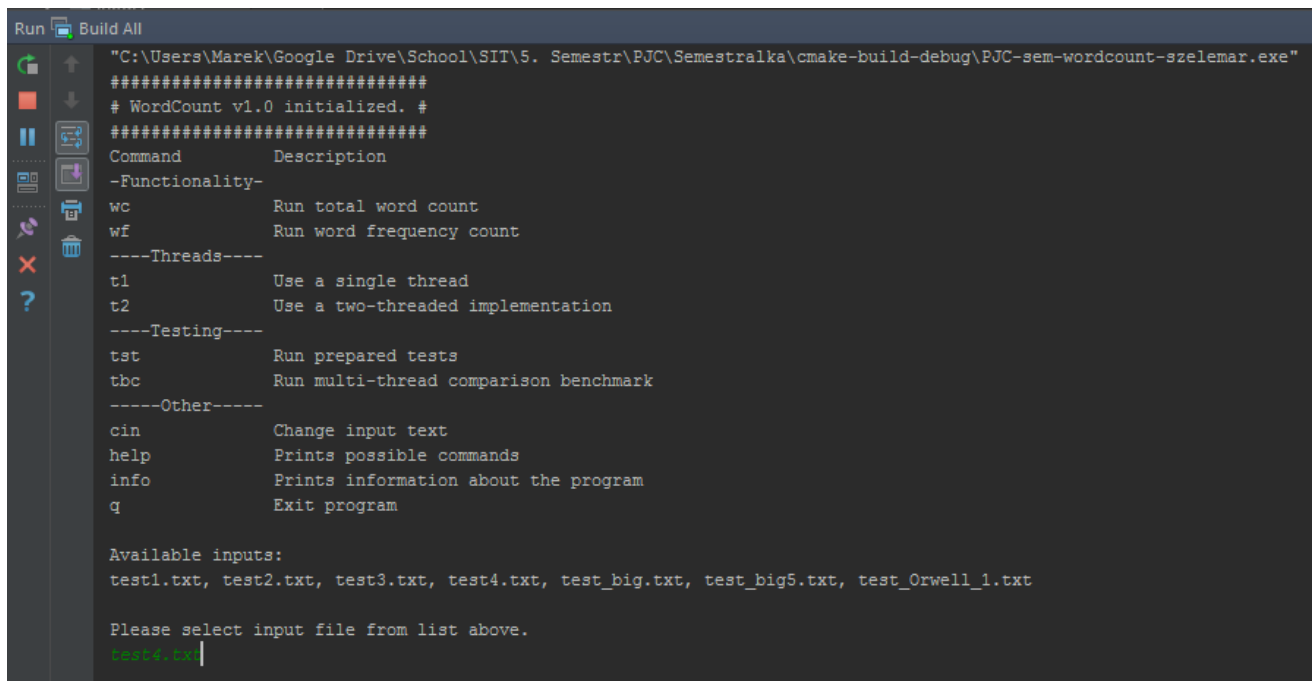
However, it is compilable using the command line/other IDEs too. If choosing to do this, please make sure that you place the input folder two levels above the main executable of the program – i.e. so that for “*file.txt*”, the route from the “*PJC-sem-wordcount-szelemar.exe*” would be “*../inputs/file.txt*”. Otherwise, the program might fail to load the input files correctly.

Please do not erase/move any of the provided sample .txt files, as they are being used in the program for test/benchmark functions.

The main function is in the “*main.cpp*” file, configuration is in “*CMakeLists.txt*”.

3 Running the program

After successful compilation and build, the program should write out the possible commands and a prompt to enter an input file from the available options. If no input options are listed, the directory hierarchy is wrong, see chapter 2. The input file name can be entered with or without the “.txt” extension. As seen in CLion IDE:



```
Run Build All
"C:\Users\Marek\Google Drive\School\SIT\5. Semestr\PJC\Semestralka\cmake-build-debug\PJC-sem-wordcount-szelemar.exe"
#####
# WordCount v1.0 initialized. #
#####
Command      Description
-----
-Functionality-
wc            Run total word count
wf            Run word frequency count
-----Threads-----
t1            Use a single thread
t2            Use a two-threaded implementation
-----Testing-----
tst           Run prepared tests
tbc           Run multi-thread comparison benchmark
-----Other-----
cin           Change input text
help          Prints possible commands
info          Prints information about the program
q            Exit program

Available inputs:
test1.txt, test2.txt, test3.txt, test4.txt, test_big.txt, test_big5.txt, test_Orwell_1.txt

Please select input file from list above.
test4.txt|
```

From there, one can use the various commands to measure the input. Here are the command descriptions:

Command	Description
wc	WordCount – Counts the number of words using the number of threads specified.
wf	WordFrequency – Counts the number of individual words and prints a table from least to most frequent.
t1	Changes the number of threads used to 1.
t2	Changes the number of threads used to 2.
tst	Runs a set of basic tests to compare output of, as well as a simple thread performance comparison. Tested on various files provided.
tbc	Thread Benchmark – a set of 10 tests comparing the performance of 1- and 2- threaded WordCount. Tested on the “test_big.txt” provided.
cin	ChangeInput – allows the user to change the file being worked with.
help	Prints the possible commands.
info	Prints information about the program.
q	Quit – closes the program

5 Multithreading

The program includes 1- and 2-threaded implementations of the “*word count*” function. The two threaded implementation is more complex and is thus only more effective when using larger files. Therefore, for the performance testing, the large provided text file “*test_big.txt*” was used (totaling 1109188 words).

These are the results of 10 runs on a Lenovo ThinkPad E440 Edge, with Intel Core i3 four-core 2.40 GHz processor:

Test id	1 thread	2 threads	Δ absolute	Δ relative
1	494ms	353ms	141ms	29%
2	445ms	372ms	73ms	16%
3	458ms	387ms	71ms	16%
4	451ms	388ms	63ms	14%
5	463ms	385ms	78ms	17%
6	639ms	562ms	77ms	12%
7	463ms	366ms	97ms	21%
8	435ms	373ms	62ms	14%
9	438ms	378ms	60ms	14%
10	485ms	371ms	114ms	24%

The overall average improvement in this case was 18%, with the average deviation of 4%.

One can run a similar benchmark in the program, using the `tbc` command. The output of such benchmark in-program can be seen here:

```

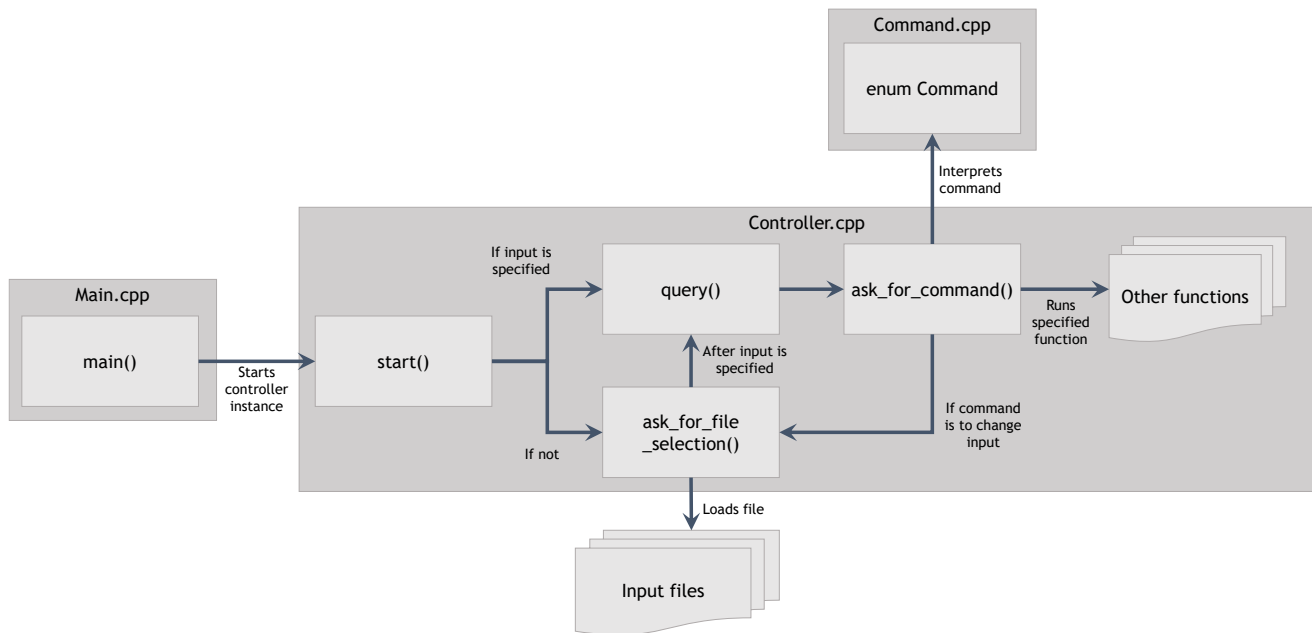
Test id 1T time    2T time    Improvement
1      476ms      455ms      4.41%
2      389ms      360ms      7.46%
3      378ms      300ms      20.6%
4      319ms      247ms      22.6%
5      271ms      221ms      18.5%
6      257ms      180ms      30%
7      222ms      194ms      12.6%
8      203ms      179ms      11.8%
9      206ms      157ms      23.8%
10     174ms      156ms      10.3%

----- Overview -----
Average improvement:      16.2%
Average deviation:       6.88%

```

6 Code structure

Since the program has limited functionality, most of it is included in just one file (*controller.cpp*). The structure of the code is:



- **Main.cpp** – contains main method
 - `main()` – starts controller instance
- **Command.cpp** – defines available commands enum
- **Controller.cpp** – contains the whole logic
 - `query()` – handles commands
 - `ask_for_command()` – asks for command input
 - `interpret_command()` – interprets the input
 - `get_files()` – retrieves files from a directory
 - `print_input_files()` – prints files at the *inputs* directory
 - `ask_for_file_selection()` – asks user to specify input
 - `select_file()` – selects file from *inputs*
 - `load_file()` – loads the selected file
 - `print_file()` – prints the loaded file line by line
 - `count_words_in_line()` – counts words in a line
 - `count_words_in_file()` – counts words in a file
 - `run_tests()` – runs prepared tests
 - `run_benchmark()` – runs the multithread benchmark
 - `run_frequency_count()` – runs words in a file count
 - `words_in_file()` – counts words in a file
 - `print_commands()` – prints available commands
 - `print_info()` – prints info about program
 - `start()` – entry point