Assignment week 7: Concordances

Course ‘Imperative Programming’ (IPC031)

Make sure to start working on the assignment **before the lab** session, otherwise you will be too late for the deadline

1 Background

Computers play an important role in the analysis of texts. Concordances are such an example. A concordance is a (usually alphabetically sorted) list of all (or at least the most important) words in a text including their context, to allow careful study of these texts. Before the computer age, concordances were produced manually, which is a time-consuming process, and were conducted only for important texts. In this assignment you design and implement a console program that reads words from a text file and stores them in their order of appearance. Hence, no words are omitted (unless the text file has too many words) and sorting does not take place. After reading the file the user can get information about word frequency, occurrence, and contexts via the console.

2 Learning objectives

After doing this assignment you are able to:

- work with strings;
- work with console I/O that handles user queries and displays information;
- work with text file I/O.

3 Assignment

On Brightspace, “`assignment-07-mandatory-files.zip`” contains a “`main.cpp`” and “`main_test.cpp`”, as well as a number of test text files (“`desktop.txt`”, “`Vogon poem.txt`”, and “`Alice’s Adventures In Wonderland.txt`”). You have a great deal of freedom how to structure your program. However, you must ensure that the functions `enter_command` and `count_command` can be tested as described in Part 1 and “`main_test.cpp`”.

Part 1: Basic user interaction

Design and implement a console program that allows the user to enter the commands:

1. `enter␣`*filename*: the program attempts to open a text file with file name *filename* (*filename* is allowed to contain spaces, *e.g.*, `enter␣`Vogon␣poem.txt). If the open operation is successful, the number of words read is reported to the user. If it fails, then this information is told to the user as well.
   Restrictions:
   - You must use the function `enter_command` as part of your implementation for this command.
   - The function `enter_command` must use the function `read_word` to read words from an input file stream. Note that the purpose of this week’s bonus assignment is to create a smarter version of this `read_word` function.
   - The words are to be stored in the array `content`, defined in the function `main`, which can hold at most `MAX_NO_OF_WORDS` strings. The value `MAX_NO_OF_WORDS` is deliberately smaller than the number of words in test text file “`Alice’s Adventures In Wonderland.txt`”.
2. `content`: the program displays the words that have been read from the input text file in their order of appearance.
3. `stop`: the program terminates.

Example:

```
> enter desktop.txt
Successfully read 10 words

> content
Rose
is
a
rose
is
a
rose
is
a
rose.

> stop
>
```

Testing: Implement the following tests for `enter_command` by attempting to load the following files:

- `desktop`: Load “`desktop.txt`”. Verify that you read the correct number of words, and that the correct words have been loaded.
- `vogon_poem`: Load “`Vogon poem.txt`”. Verify that you read the correct number of words, and that the correct words have been loaded.
- `alice`: Load “`Alice’s Adventures In Wonderland.txt`”. Verify that you read the correct number of words. Recall that by design this file has more words than `MAX_NO_OF_WORDS`. You do not have to verify that the correct words have been loaded.

See “`main_test.cpp`” for the test templates and further instructions. Since implementing unit tests for the `stop` and `content` commands is not trivial[1], you do not have to implement tests in “`main_test.cpp`” for them. You must however do manual testing for them by running your program and interacting with it.

Part 2: Word occurrences

Extend the program created in Part 1 with the following user commands:

4. `count`␣$word_1$␣. . . ␣$word_n$: the program counts the number of occurrences of the word sequence $word_1$␣. . . ␣$word_n$. The program first shows this number to the user, and second shows the total number of words in the text, and third shows the percentage of occurrences of the word sequence with respect to the total number of words in the text.
   Restrictions:
   - You must use the function `count_command` as part of your implementation for this command.
5. `where`␣$word_1$␣. . . ␣$word_n$: the program first displays all index-positions (the first word having index number 1) of the occurrences of the word sequence $word_1$␣. . . ␣$word_n$, and second it displays the number of found occurrences.
6. `context`␣m␣$word_1$␣. . . ␣$word_n$: the program first displays all occurrences of the word sequence $word_1$␣. . . ␣$word_n$, including up to m words immediately preceding and following it, and second it displays the number of found occurrences.

Hints:

- Introduce a helper function to read a word sequence.
- To read a word sequence, use `cin.get(c)` to try to read a character `c`.
- If this character is a space (␣), continue with `cin >> word` to read the next word into string `word`.
- Otherwise you have found the end of the word sequence.

Example:

```
> enter desktop.txt
Successfully read 10 words

> count is a
Found sequence 3 times in 10 words (30%)
```

[1]The issues is that these commands operate entirely using *side effects* (stopping the program and printing text to the standard output), rather than producing values we can easily capture and test against. There are ways to do this, but it would require us to restructure the code and making it more complex, which we opt not to do until later in the course.

```
> where is a
Found occurrence at index 2
Found occurrence at index 5
Found occurrence at index 8

Found sequence 3 times in 10 words (30%)

> context 3 is a
Rose is a rose is a
is a rose is a rose is a
is a rose is a rose.

Found sequence 3 times in 10 words (30%)
```

Testing: The “`main_test.cpp`” file contains several tests for the function `count_command`. Verify that your implementation of `count_command` works with these tests. Like the `content` command from Part 1, it is not trivial to properly test the `where` and `context` commands. Therefore you do not need to add unit tests for them. You must however perform manual testing by interacting with your program and observing that the output is correct.

4 Products

As product-to-deliver you upload to Brightspace:

- “`main.cpp`” that you have created with solutions for each part of the assignment.
- “`main_test.cpp`” that has been extended with non-trivial unit tests for each new function that you have developed in “`main.cpp`”, along with your test implementations for `enter_command`.

Deadline

Lab assignment: Friday, October 18, 2024, 23:59h

Important notes:

1. check that you have actually submitted your solution in Brightspace.
2. the deadline is firm, and it is impossible to upload a solution after expiry. In that case you fail the assignment.
3. you can upload solutions to Brightspace several times, but only the last submission is stored.
4. identify yourself and your lab partner in every uploaded document. The identification consists of your first and last names, student numbers, and number of (sub) assignment. By identifying yourself, you declare that the uploaded work has been created solely by you and your lab partner.
5. your work will be judged and commented upon. We expect that you obtain the feedback, read it, and use it to for the next exercises.
6. it is essential that you only submit your own solution, never copy somebody/something else’s solution, and never share your solution—in particular: **AI tools (including but not limited to Github Copilot or ChatGPT) are not permitted**, solutions from previous year cannot be reused, and finally, you and your lab partner take joint responsibility for the assignment you submit.