Project 1: Word-Count Using Standard C Library

Due Date: 2/12 by 11:59p

Important Reminder: As per the course Academic Honesty Statement, cheating of any kind will minimally result in receiving an F letter grade for the entire course.

Aims of This Project

The aims of this project are as follows:

- To provide experience with writing a non-trivial C program.
- To build simple data-structures using only the standard C library.
- To provide familiarity with the tools used to build C programs under Linux.

Project Specification

[Note: this specification is intentionally ambiguous and incomplete; this is similar to many real world specifications].

Write a program to output the `N` most frequently occurring words from one or more input files. Specifically, submit a `prj1.tar.gz` archive such that unpacking that archive into a directory and typing `make` within that directory builds a `word-count` executable which when invoked as:

```
$ ./word-count N STOP_WORDS FILE1...
```

outputs on standard output the `N` most frequently occurring words in one-or-more files `FILE1...` which are not in the file `STOP_WORDS`.

- A **word** is a maximal sequence of characters for which the standard C library function `isalnum()` returns non-zero or is a single quote `'` (apostrophe `'\''`). So `is'nt` will be treated as a single word.
- Words which differ merely in case are regarded as identical.
- The output should consist of `N` lines with each line containing a lower-cased word followed by a single space character followed by the count of that word across all files `FILE1...`. The lines should be sorted in non-increasing order by count.
- The program should handle files which do not necessarily end with newline.
- If the arguments to `word-count` are in error, then the program should print a suitable error message on standard error and terminate.

- The program should also detect any runtime errors (memory allocation errors, I/O errors) and terminate after outputting a suitable error message on standard error.

The program must meet the following implementation restrictions:

- The program may not directly or indirectly use any languages other than C.
- The program may not use any libraries other than the standard C11 libraries available with gcc on `remote.cs`.
- The program may assume that all distinct words in all of `FILE1...` and their counts can fit within memory.
- The store used by the program should be organized in such a manner such that adding a new word to the store normally takes time sub-linear in the number of words in the store. The word *normally* means that it is permissible that the time is not sub-linear in certain pathological cases.
- There should not be any implementation restrictions on the size of entities except those defined by available resources. Hence there should not be any restriction on the size of a word or a line.

Provided Files

The ./files directory contains the following:

Makefile

> This file assumes that the project is organized into a `main.c` main program, `word-count.h` / `word-count.c` specification / implementation files containing the main `word-count` logic and `word-store.h` / `word-store.c` specification / implementation files for the word store. It provides the following targets:
>
> `word-count`
>
> > This will build the `word-count` program.
>
> `clean`
>
> > This will clean out all generated files.
>
> `submit`
>
> > This will build the required `prj1.tar.gz` archive.
>
> Typing simply `make` will build the `word-count` program, typing `make clean` will remove all generated files and typing `make submit` will create a `prj1.tar.gz` compressed archive which can be submitted.
>
> You may edit this file if you choose to use a different organization for your project. When editing, watch out for tabs (the first character of any command-line **must be a tab character**).

README
: A template README; replace the `XXX` with your name, B-number and email. You may add any other information you believe is relevant to your project submission. In particular, you should document the data-structure used for your word-store.

`main.c`
: A driver with a dummy `main()` function to be filled-in by you.

`word-count.[ch], word-store.[ch]`
: Empty files to be filled-in by you, assuming you are using the project structure supported by the provided Makefile.

Hints

You may choose to follow the following hints (they are not by any means required). They assume that you are using the project structure supported by the provided Makefile,.

It is probably best to read a word at a time from the input files. Since there is no restriction on the size of a word, the buffer used to hold the read word must be dynamically (re-)allocated.

1. Write suitable specifications in `word-count.h` and `word-store.h`; i.e., declare the functions and ADT's to be exported by `word-count` and `word-store`.

   Provide dummy NOP implementations for these specifications in `word-count.c` and `word-store.c`.

2. Write a driver program in `main.c`. Make sure it prints appropriate error messages if given incorrect arguments.

3. Write a implementation of `word-count`. Make it track the `N` most-frequently occurring words. Note that there is no performance restriction on finding the `N` most-frequently occurring words; a simple-minded implementation may result in `O(N*M)` performance, where `M` is the number of distinct words in the file.

4. Write a trivial implementation of `word-store`. You may use non-performant data-structures and simple-minded memory allocation. At this point, you should have a functional program, but one which does not meet its performance requirements.

5. Choose a more performant data-structure for your word-store. Possibilities include but are not limited to a:

   - Hashtable.
   - Trie.
   - Binary tree.

6. Implement your specification for `word-store` using your chosen data-structure. When doing dynamic memory allocation using the C library `malloc()`, take care not to allocate a small amount of memory each time: for example, it is probably a bad idea to make a separate `malloc()` call for each word; words could be allocated in a string-space which could be a list of memory blocks with each block containing `NUL`-terminated words (however, the size of a block should not limit the size of a word).

7. Test and review your code until it meets all requirements.

Submission

You will need to submit a compressed archive file `prj1.tar.gz` which contains all the files necessary to build your `word-count` executable. Additionally, this archive **must** contain a `README` file which should minimally contain your name, email, the status of your project and any other information you believe is relevant (it is probably a good idea to mention which data-structure you have chosen for your word-store).

If you are using the suggested project structure, then the provided Makefile provides a `submit` target which will build the compressed archive for you; simply type `make submit`.

Note that it is your responsibility to ensure that your submission is complete so that simply typing `make` builds the `word-count` executable. To test whether your archive is complete, simply unpack it into a empty directory and see if it builds and runs correctly.

Submit your project using the submission link for this project, under **Projects** in Blackboard for this course.