School of Computer Science, McGill University

COMP-206 Introduction to Software Systems, Fall 2020

Assignment 6: C Programming - Dynamic Memory

Due Date December 1st, 23:55 EST

This is an individual assignment. You need to solve these questions on your own. If you have questions, post them on Piazza, but do not post major parts of the assignment code. Though small parts of code are acceptable, we do not want you sharing your solutions (or large parts of them) on Piazza. If your question cannot be answered without sharing significant amounts of code, please make a private question on Piazza or utilize TA/Instructors office hours. Please ensure to check "Assignment 6 general clarifications" pinned post in Piazza before you post a new question. It might have been already clarified there if it is a popular question.

Late penalty is -5% per day. Even if you are late only by a few minutes it will be rounded up to a day. Maximum of 2 late days are allowed.

This is not a mini assignment. It is one out of the two regular assignments. It is worth 8% of your course grade.

You MUST use `mimi.cs.mcgill.ca` to create the solution to this assignment. You must not use your Mac command-line, Windows command-line, nor a Linux distro installed locally on your laptop. You can ssh or putty from your laptop to `mimi.cs.mcgill.ca`. All of your solutions should be composed of commands that are compilable/executable in `mimi.cs.mcgill.ca`.

Questions in this assignment require you to turn in a number of files (listed in the description below). Instructors/-TAs upon their discretion may ask you to demonstrate/explain your solution. No points are awarded for commands that do not execute at all or programs that do not compile in `mimi`. (Commands/programs that execute/compile, but provide incorrect behavior/output will be given partial marks.). All questions are graded proportionally. This means that if 40% of the question is correct, you will receive 40% of the grade. There may be some objectives for which no partial marks are given. This is indicated where applicable.

Please read through the entire assignment before you start working on it. You can lose several points for not following the instructions.

Unless otherwise stated, all the names of the scripts and programs that you write, commands, options, input arguments, etc. are implied to be case-sensitive.

Total Points: 20

Ex. 1 — A Word Counter App

Word clouds are often used to visualize the context of an article or someone's academic research. This is because they are a good visual representation of the main research categories on which one or more articles focus. This is accomplished by constructing an image where words are organized in the shape of a cloud with words which occur at a higher frequency getting more emphasis (i.e., larger fonts). The first task in building a word cloud is to scan the relevant articles and compute the frequency of different words in it.
To facilitate this, in this exercise, you will create a C program, `wcloud`, a simple application that reads through all the text files passed in as its argument and prints the number of times each word occurs **across all** those text files. The output is printed in alphabetical order. Your program is expected to contain multiple modules (multiple C files).

```
$ cat article1.txt
A vegetable that is a good source of iron.
But spinach offers more than just iron. It is a very versatile vegetable.
$ ./wcloud article1.txt
a 3
but 1
good 1
iron 2
is 2
it 1
just 1
more 1
of 1
offers 1
source 1
spinach 1
than 1
that 1
vegetable 2
versatile 1
very 1
$
```

The program itself processes the contents in a case insensitve manner; i.e., `But` and `but` are considered the same. It stores and produces the output in lowercase.

Below, the points shown are the deductions if you do not meet each objective. That said, no matter what, your overall score cannot be negative.

1. Create a directory for this assignment and initialize `git` (just a local repository, do not make it public). As you progress through this assignment, you will keep committing code to `git` (I leave the actual points in development where you feel like you should commit the code to `git` up to you, but **there should be at least 4 commits separated out by 15 minutes or more**). You must turn in the `git log` of your work to prove this. **(-2 points, no partial points.)**
2. All your program files (`Makefile` is exempted) should include a comment section at the beginning that describes its purpose (couple of lines), the author (your name), your department, a small "history" section indicating what changes you did on what date. The code should be properly indented for readability as well as contain any additional comments required to understand the program logic. **(-2 points).**
3. Your main will be in a C file `wcloud.c`. Your program will be invoked as follows by passing one or more text files as its argument. (Therefore, do not assume the number of names of the input files, but read it from the arguments passed to your program).

```
$ ./wcloud article1.txt anotherarticle.txt
```

4. If your program is invoked without any arguments, it should throw an error message and terminate with error code 1. ALL error messages of your program should be printed to standard error. Only the output from the non-erroneous steps should be printed to standard output. **(-1 point, no partial points.)**

```
$ ./wcloud
Usage: wcloud article [article]...
$ echo $?
1
```

5. In your main, use `fopen` to read one file at a time from the input arguments passed to the program.
6. If a file passed to the program cannot be opened (for any reason) then the program should produce an error message. **If other files were passed as arguments, the program should continue processing those. (-2 points, no partial points)**.
7. If at least one file could not be opened, the program's exit code should be 2. **(-1 point, no partial points.)**

```
$ ./wcloud nosuchfile.txt a2.txt
Error! unable to open nosuchfile.txt
a 3
but 1
good 1
... (truncated for assignment description)
$ echo $?
2
```

Note that file ***a2.txt*** *was processed and the results were printed despite the fact that* ***nosuchfile.txt*** *could not be opened for reading.*

8. You can assume that only text files will be passed as arguments. You can also assume that the length of any line in the file will not exceed 1000 characters.

9. At a high-level, your program maintains a linked list of words that it encounters as it reads through the articles and that it uses to keep track of their count.

10. Use `vim` to create a header file, `wordlist.h`. In this, you will include the following structure definition, which will function as the node of your program's linked list.

```
struct WordNode
{
  char *word;
  unsigned long count;
  struct WordNode *next;
};
```

NOTE that, you might have to add other things to this header file depending on your overall design, but that is up to YOU to figure it out, including where to include this header file.

11. Use `vim` to create another C file, `wordlist.c`. In this file, you will define a function with the following signature.

```
struct WordNode *addWord(char* word, struct WordNode *wordListHead, int* outofmemory);
```

This function will be called by main to add a new word to the existing linked list, whose head is accessible through the pointer `wordListHead`. The function will add the word to the list if it does not exist and will otherwise increment the appropriate word count. If it runs out of memory while trying to add a new node, it will set the `outofmemory` indicator to 1.

Next you will add a second function in this C file with the following signature.

```
void printWordList(struct WordNode *wordListHead);
```

This function will be called by main (after reading through all the files) to print the word list (the head node being the argument). While it is expected that the output thus displayed will be in alphabetical order, there is no restriction on how you achieve the ordering. It is left to your imagination on how to facilitate the final sorted output.

You may add extra arguments to either of the above functions, if it suits your design.

12. You may (and are encouraged to) add other functions to your source code (either the above source code or new ones) to help modularize your design.

13. If your program runs out of memory, it should terminate with code 3 after printing the following message (**(-1 points, no partial points)**). No other output is printed under such circumstances.

```
$ ./wcloud article1.txt
Error!! ran out of memory!
$ echo $?
3
$
```

NOTE:- You might not be able to create an out of memory condition easily in real execution. Review your code / use some simulated approach just for testing the validity of your logic.

14. If you need additional header files (`.h`) to make sure your various C files can interact with each other, create them as necessary and include them in appropriate files as needed. The objective is for `make` / `gcc` not to give any compilation warnings and errors.
15. Create a make file, name it `Makefile` . This make file should contain necessary compilation steps to build your final executable, which should be named `wcloud`. Your make file should build an executable such that TAs can simply download your source code into an empty directory and type `make` to build it. Therefore, make sure whatever commands you include in your make file do not involve custom scripts, aliases, etc., but regular commands available to everyone on `mimi`. **If your makefile does not build the program executable, you will receive a grade of 0 on the assignment.**
16. Ensure that your make file does not rebuild executables (including .o) files if there are no changes to any source code files. **(-1 point, no partial points)**.
17. Ensure that your make file rebuild **ONLY** those executables (including .o) that are dependant on the changed source code files. **(-1 point, no partial points)**.
18. **(-1 Point)** if the make process results in ANY warning.
19. For your convenience, an extra file, `delim.txt` has been give with this assignment description that contains the string that can be used as the delimiter (to use the `strtok` function that you already familiar from mini 5). You may copy and paste this into your relevant source code. Please note that by this definition, many non-alphabet characters such as digits, etc., will be considered part of the word. This is OK.
20. Keep in mind that while files can contain empty lines (without words or text in a line). That said, none of the files will be empty; they will contain at least one line with valid words.
21. Words can be sometimes separated by multiple delimiters (e.g. two spaces between adjacent words). The program should take care of it.
22. **(-3 Points)** if your program crashes for ANY test case.
23. **(-3 Points)** if the program output misses ANY word.
24. **(-3 Points)** if some words are not accounted for in a case insensitive manner.
25. **(-3 Points)** for incorrectly parsed words (e.g., delimiters included in at least one word) or for including invalid words (e.g words made solely out of delimiter characters).
26. **(-3 Points)** For printing ANY non-sorted output.
27. You must remember to `free` any manually allocated memory using `malloc/calloc` before terminating the program **(-2 Points)**.

**** Important **** If your program "hangs" / is stuck while executing it through the tester script and requires TAs to interrupt it, you will lose **4 points**.

WHAT TO HAND IN

Turn in the make file `makefile`, your git log `gitlog.txt`, C program source codes `wcloud.c`, `wordlist.c`, header file `wordlist.h` named properly **AS WELL AS** any other header files (`.h`) and/or `.c` files that your program needs to compile correctly. You may but do not have to zip the files. The files must be uploaded to mycourses. If you zip your files, double check to ensure that they are not accidentally corrupted. DO NOT turn in the executable `wcloud` or your testing data files. TAs will compile your C program on their own as indicated in the problem descriptions above.

MISC. INFORMATION

There are no tester scripts provided with this assignment. You may write one, if it suits your repeated testing needs (encouraged), however, do not turn it in. Also do not turn in your executables/.o files.

TAs will be testing using their own standard simple scripts in `mimi`.

You have been taught various debugging techniques in class, especially `gdb`. You are expected to apply those techniques to figure out odd behaviours of your program.

There are no restrictions on the functions you can use for this assignment. Just make sure they are from the C libraries that are already available in mimi as part of the gcc.