SENG265
FALL 2020
ASSIGNMENT 3
UNIVERSITY OF VICTORIA

Due: Nov 20, 2020 by 10:00 pm, by "git push".
(Late submissions **not** accepted)

1 Assignment Overview

This assignment involves writing a program to solve the same problem as Assignment 1 Part C except by using dynamic memory allocation and linked lists techniques. As a reminder, the program will read lines of text from a given file, compute the frequency of words of certain length from the file and print these frequencies to standard output.

The overall goal of this assignment is to use the dynamic-memory available in C (i.e. *malloc(), realloc(), etc.*) as well as the concept of linked lists in C. **You may only use dynamic-memory for any arrays**. For example, you may not use `int array[n]` syntax.

Similar to Assignment 1, your eventual submission will consist of the source file(s) for the program. There are three parts to this assignment and Sections 1.2, 1.3, 1.4 below contain `Specifications` for each. **These sections do not necessarily need to be completed sequentially, but to be developed collectively to produce a final solution.**

Section 2 describes the (new) `Constraints` you should consider in your implementation, and Section 3 describes the `Testing` component. `What you should Submit` is outlined in Section 4 and the `Evaluation` scheme is given in Section 5.

Your code is expected to compile without warnings **in the vagrant VM Senjhalla** using the `-Wall -g` and `-std=c99` flags with the `gcc 9.3.0` compiler.

1.1 Data Type Constraints

In your previous C assignment, you were not allowed to use dynamic memory allocation methods, such as `malloc`, `calloc`, `realloc`, etc. For this assignment, you are **NOT** allowed to create any static-sized arrays using the syntax `int array[10]`. In other words, you are not allowed to store any arrays in *stack memory*. You must use dynamic memory allocation (as discussed in the lecture and labs) to store your arrays in *heap memory*.

To achieve full marks on this assignment, your solution must use a linked list as your data structure for:

1. Your histogram of word frequencies, and;
2. Your list of words in each histogram bin/bucket;
3. Your linked list must be dynamically allocated;

The additional behaviour expected by your program is outlined in the next two sections.

1.2 Part A. Printing Words Information

The requirements for this assignment are divided into three parts. The first part of the assignment is to write a C program, contained in a source file called `word_count.c`, which counts the number of words of all lengths. In this assignment you will **only implement Part C** from Assignment #1 using linked lists and dynamic memory allocation (e.g. `malloc`).

To achieve full marks on this assignment, your solution must use a linked list as your data structure for:

1. Your histogram of word frequencies, and;
2. Your list of words in each histogram bin/bucket;

You will output the histogram bin/buckets in the same order as Assignment #1 - Part A (i.e. order by word length bucket/bin size). The parameter `--print-words` will not be used as it is unnecessary: that is the only behaviour expected by the program.

Let's look at an example that uses input file **input.txt** as shown below.

input.txt:
```
Tomorrow, and tomorrow, and tomorrow,
To the last syllable of recorded time;
```

The program can be run in the following manner:
```
$ ./word_count --infile <input_file>
```

This will produce output that matches the behaviour of assignments 1 and 2 (i.e. using `--print-words` without `--sort`). The histogram buckets/bin are printed in ascending order based on the **word length**. The **unique words** in each histogram are printed in **ascending alphanumeric order**.

The expected output is shown below, for input file **input.txt**:
Count[02]=02; (words: "To" and "of")
Count[03]=03; (words: "and" and "the")
Count[04]=02; (words: "last" and "time")
Count[08]=05; (words: "Tomorrow", "recorded" and "syllable")

A second parameter `--sort` is used to change the **word list order** to **descending alphanumeric order**.

The program can be run in the follow manner:
`$ ./word_count --infile <input_file> --sort` , or
`$ ./word_count --sort --infile <input_file>`

The expected output is shown below, for input file **input.txt**:
`Count[02]`=02; (words: "of" and "To")
`Count[03]`=03; (words: "the" and "and")
`Count[04]`=02; (words: "time" and "last")
`Count[08]`=05; (words: "syllable", "recorded" and "tomorrow")

1.3 Part B. Error Handling

For this assignment, you will be returning *error codes* when there is an error. These are codes that are sent to the operating system that indicate the state of the program when it finished. In Linux and UNIX the error code standard is zero (0) to indicate the program completely successfully, and 1-255 for any other state. In C, the error code is the integer returned from `main` or using `exit(return_code)`. You can see the error code by typing `echo $?` in the terminal, after running a command.

Your solution must return the following error codes:

1. Invalid command line arguments: return code = 1;
2. Unable to open `--infile <file>`: return code = 2;
3. Unable to allocate memory: return code = 3;
4. All error messages should be sent to `stderr` (and not `stdout`);

You may assume that if `<file>` exists, it follows `--infile`. Your solution may ignore invalid switches (i.e. `--bad-param`). There is **no standard for the each error output message**, but all error messages should be sent to `stderr` (i.e. use `fprintf`).

1.4 Part C. Memory Handling and Debugging

When you use dynamic memory allocation, the programmer must release the memory directly using `free()`. Memory that is not released once it is no longer in use is called a *memory leak*. For full marks, your solution must produce no memory leaks.

In addition to `gdb` you can use `valgrind` for debugging memory errors.

1.4.1 Memory Leaks

If you have a pointer to a block of memory you allocated, and the pointer is deleted but the block is still tied up (i.e. you have it reserved), it's called a memory leak. This can happen if, for example, the pointer is a local variable inside a function, which returns without free()ing the associated block of memory. If enough memory becomes tied up in memory leaks, your program could run out of memory and crash. Or, the virtual memory keeps expanding and you run out of memory for your computer! Once you free() a pointer to a block of memory, trying to free() it again will cause an error. Once you allocate some memory, C still doesn't check if you're staying within the bounds of the new block; that's up to you.

1.4.2 Valgrind

Valgrind is a special debugger program that checks your program for memory allocation errors. Your program needs to be compiled with `-g`. Refer to the `makefile` provided to you.

Examples of how to use vagrind:
`valgrind ./word_count --infile cases/t03.txt`, or
`valgrind --leak-check=full ./word_count --infile cases/t03.txt`

Your program will run a bit slower than normal, but afterwards, Valgrind will print out a Leak Summary report on any memory leaks or allocation errors. If your program stops with a segfault, Valgrind will tell you where it is. `--leak-check=full` provides additional information for debugging.

To achieve full marks, proper solution will contain no memory leaks (0 bytes). In the Leak Summary report, we will be considering any **definitely loss** bytes as a memory leak. Any other potential leaks will be ignored during marking.

A full breakdown of how to interpret the output of `valgrind` can be found here.

2 Constraints

You are allowed to use any code provided during the lectures or the labs. Code snippets from third-party sources are allowed if appropriately attributed (i.e. include the URL link in the comments of your code). If you are unsure, ask your lab teaching assistant.

- You cannot assume a maximum length for an input file;
- You cannot assume a maximum number of lines in a file;
- You cannot assume a maximum number of words in a file;
- You must **alloc()* (malloc, realloc, calloc, etc.) to allocate memory any arrays or structs;
- You must use a linked list as your histogram data structure;
- You must use a linked list as your histogram word list data structure;
- The only allowed special characters to be included in the input file are .,;(). No other special characters are expected to be included in the input file.

3 Test Inputs

For this assignment, you will not be provided a test framework.

You are expected to evaluate and test your code using the skills you have learned so far in the course. You may re-use the test framework from assignment #1, however, be aware that you might experience *heap fragmentation* or your code has *memory leaks* which may cause *malloc()* or *realloc()* failures.

The program must compile, with no warnings, and produce no *memory leaks*. A `makefile` that when run using `make` creates the executable `word_count` has been provided for you. **This makefile is in the same format as the one used for evaluation.**

You may use `make word_count` to compile your word_count.c and any additional C-files required for your solution into a `word_count` executable. This command will remove any existing `word_count` executable, and compile the code located in your `a3/src` folder.

This will be the equivalent of running:
`$ gcc -Wall -g -std=c99 -o word_count word_count.c`

You will receive 8 test input files used to evaluate assignment 3. Tests **t0[1-7].txt** are the same tests used in assignments 1 and 2. An additional test `valgrind.txt` has been provided to allow you to test your program with large input for memory leaks.

You may use `make valgrind` to compile and run:
`valgrind word_count --infile cases/valgrind.txt [--sort]`.
Both with and without the `--sort` parameter is run.

Three additional test input files will be used during evaluation that will test the robustness of your solution. These 3 tests will test:

- A large file size that exceeds the previously used maximum file size;
- A file with the number of lines that exceeds the previously used maximum line count;
- A file with the number of words that exceeds the previously used maximum word count;

You should test all of your programs with a variety of test inputs, covering as many different use cases as possible, not just the test input provided. You should ensure that your programs handle error cases (such as files which do not exist) appropriately and do not produce errors on valid inputs.

Read the assignment carefully to ensure you understand all the requirements of the program.

Provided For You

For this assignment, you will be provided an `assign3.zip` containing an empty `word_count.c` file, `makefile`, the 7 test inputs and corresponding expected output. An additional test input file `valgrind.txt` for valgrind memory leak testing.

The expected output from part A are located in the folders *A* (ascending) and *D* (descending).

Folder (A)scending contains output from:
`$ ./word_count --infile <input_file>`

Folder (D)escending contains output from:
`$ ./word_count --sort --infile <input_file>`

4 What you must submit

- C source-code name `word_count.c` located in `a3/src` which contains your solution for Assignment #3.
- Any additional C (*.c or *.h) files required for your solution in `a3/src`
- Ensure your work is **committed** to your local repository in the provided **a3** folder **and pushed** to the remote **before the due date/time**. (You may keep extra files used during development within the repository.)

5 Evaluation

The teaching staff will primarily mark solutions based on the input files provided for this assignment. Students must adhere to the software requirements (command execution and output formatting) outlined in this assignment. For each assignment, some students will be randomly selected to demo their code to the course markers. Each student will have this opportunity to demo at least one assignment during the semester. Sign-up procedure will be discussed in class.

In addition to automated testing, your code will be evaluated based on:
- Proper error handling (i.e. memory allocation handling);
- Good coding practices;
- Adherence to the assignment constraints

Good Coding Practices:

- Your code is properly structured into functions, without too much functionality in any one function;
- Your variables names are descriptive and easy to infer their functionality;
- Your function names are descriptive and easy to infer their functionality;
- Program properly checks for possible error conditions;
- Program properly checks for user error;
- Your code doesn't contain (nearly) duplicate code in different functions;
- Global variables are used sparingly, and appropriately;
- Your code is fairly efficient and executes in a reasonable time (i.e. roughly less than 3 minutes);

Assignment Requirements

1. All arrays must be allocated using dynamic memory allocation;
2. Word histogram is constructed using a linked list;
3. The word list for each histogram bucket/bin is constructed using a linked list;
4. Your code returns the correct error codes;
5. Your code passes the provided tests (1-7);
6. Your code passes the 3 evaluation tests (8-10);
7. A complete solution will have no memory leaks (as described in Section 1.4) for `valgrind.txt`;

Our grading scheme is relatively simple. See above for requirements. Not following good coding practices will lower your grade.

- "A/A+": A submission completing ALL requirements of the assignment.
- "A-": A submission completing 1-6 requirements of the assignment. Some or minimal memory leaks in the program.

- "B/B+": A submission completing 1-5 requirements of the assignment.
- "B-": A submission completing 1-4 requirements. Non-trivial test cases (5-7) pass.
- "C+": A submission completing 1-4 requirements. Trivial test cases (1-4) pass.
- "C": A submission completing 1-3 requirements. Some test cases pass and some of the correct error codes are turned.
- "D": A submission that fails to complete the base requirements of the assignment. Trivial or lack of use of linked lists. Dynamic memory is used, but some arrays are also used. No attempt was made to free dynamic memory.
- "F": A submission that fails to complete the base requirements of the assignment. Either no submission given (or did not attend demo); submission represents very little work or understanding of the assignment.