Count Assignment: A Simplified Unix `wc` Program

COMP 222: Introduction to Computer Organization
Assigned: 9/12/25, **Due: 9/26/25, 11:55 PM**

Important: This project must be done individually. Be sure to carefully read the course policies for assignments (including the honor code policy) on the assignments page of the course web site:

`http://www.clear.rice.edu/comp222/html/assignments.html`

Overview

You will write C code to implement a program that counts the number of characters, words, and lines in a file. The goals of this project are as follows:

- Use C pointers and data structures.
- Dynamically allocate and deallocate memory.
- Use C's Standard I/O library to read from files.
- Become comfortable with writing a larger C program.

Summary

You will write a simplified version of the Unix `wc` word count program. Your `count` program must do the following:

- Process multiple files.
- Count the number of characters, words, and lines in each file.
- Keep track of the total number of characters, words, and lines in all files.
- Print the results for each file in "ASCIIbetical" order by file name.
- Print the total number of characters, words, and lines in all files.

The program accepts three command-line flags that control what it counts: "-c", "-w", and "-l". The flags indicate whether or not the program should count characters, words, and lines, respectively. Your program should support any combination of these flags. If no flags are specified, then nothing should be counted, but empty results for each file should still be printed.

We provide a main routine that reads and parses the command-line arguments appropriately. The usage of the word count program is as follows:

```
count [-c] [-l] [-w] <input filenames>
```

At least one input file must be specified on the command line. Otherwise, the main routine prints an error message and terminates the program. The main routine does not impose a limit on the number of input files, aside from the implied limit of `INT_MAX` because `argc` is of `int` type.

The main routine will call the following function based on what is specified on the command line:

```
int
do_count(char *input_files[], const int nfiles,
    const bool char_flag, const bool word_flag,
    const bool line_flag)
```

The first argument, `input_files`, is an array of pointers to strings. Each string is the name of a file that should be counted. The second argument, `nfiles`, indicates how many elements are in the `input_files` array. The remaining "flag" arguments are true or false based on whether the corresponding options were specified on the command line. In other words, `char_flag` is true if the command line included `-c` and false otherwise.

Your `do_count` function should count the characters, words, and/or lines, based on the value of the flags, in each file specified in the `input_files` array. Once complete, the results should be printed out to `stdout` using the following provided function:

```
void
print_counts(FILE *fp, struct counts *cnts,
    const char *name, const bool char_flag,
    const bool word_flag, const bool line_flag)
```

This will print out the results of counting a single file on one line. If all flags are true, it will print the number of lines, then the number of words, then the number of characters, and finally the (file) name. For each flag that is false, that count will be omitted, but everything else will remain in its respective order. The name is always printed. For each file, you should use the same values for the flags that were passed to `do_count`.

The results should be printed out in "ASCIIbetical" order based on the filename. Essentially, ASCIIbetical ordering collates strings based upon the numerical ASCII representation of the characters. It differs from the conventional notion of "alphabetical" ordering in that all upper-case letters come before all lower-case letters. For example, in ASCIIbetical ordering, the string "XYZ" will come before the string "abc". The function `strcmp()` can be used to help sort strings in this order. See `man strcmp` for details.

After all of the individual file results have been printed, a total should be printed. And, to be clear, a total should be printed regardless of the number of individual file results printed. The `print_counts` function should again be used, but the filename should be specified as "total" and the character, word, and line counts should all be printed, regardless of the command line options. If an option was not specified on the command line, the corresponding output in the total line should be 0.

For example:

```
./count -l file.txt abc.txt
      32 abc.txt
     254 file.txt
     286       0       0 total
```

Note that the return type of `do_count` is `int`. The value returned by your `do_count` function will subsequently be returned by the main routine, so your `do_count` function should follow the standard Unix convention for the return value from programs: return `0` if there were no errors and an integer between `1`

and `255` otherwise. In other words, if there is an error opening or processing *any* of the files, then the return value from `do_count` should be an integer between `1` and `255`. However, your `do_count` function should *not* stop opening and processing files just because it encounters an error. Instead, it should move on to the next file in the `input_files` array. Further details on handling errors are discussed below.

For the purposes of this assignment, you should use the following definition of characters, words, and lines:

- **Character:** A character is any ASCII character that appears in a file, whether it is visible or not.
- **Word:** A word is a sequence of ASCII characters separated by whitespace. Therefore, a word is any sequence of characters that does not include a character for which the function `isspace` returns true (see `man isspace` for further details on this function).
- **Line:** A line is anything that ends with a '\n' character.

For example, consider the following input file:

```
This is a line.
This is another line.
```

This file has 38 characters, 8 words, and 2 lines. This is assuming that both lines end with a '\n' character, that there are no other whitespace characters (space, tab, etc.) at the end of either line, and that there are no other non-visible characters in the file.

You may write whatever additional procedures or data structures you deem necessary to complete this project, but they should all be contained within the `count.c` file.

Since a pedagogical goal of this assignment is that you dynamically allocate and deallocate memory, you may **not** add any additional arrays into your submitted `count.c` file beyond what is initially provided in the template[1]. This restriction includes both fixed-length and variable-length arrays. Moreover, you may not make changes to the existing arrays. Instead, you should use pointers to maintain a sorted linked list of dynamically allocated structures, with each structure containing at least the counts for a single file and a pointer to the string representing that file's name.

While the computations performed by your `do_count` function on each file should take little time, reading the contents of each file will be a comparatively slow operation. Therefore, you may **not** open a file more than once or read the contents of a file more than once. In other words, you must simultaneously compute the requested character, word, and line counts in a single pass through each file.

If you encounter an error with a file that prevents the program from opening the file, you should use `app_error_fmt` to print out the following error message to `stderr`:

```
ERROR: cannot open file '<fname>'
```

If you encounter an error while reading the file that prevents the program from counting characters/words/lines, you should use `app_error_fmt` to print out the following error message to `stderr`:

```
ERROR: cannot read file '<fname>'
```

In the error messages, `<fname>` *must* be the file name, and that file name *must* be surrounded by single quotes (ASCII character 39). For example, if an error was encountered while trying to open the file `/usr/share/dict/words`, the following error message should be printed:

```
ERROR: cannot open file '/usr/share/dict/words'
```

[1] Using dynamically allocated memory to store an array still qualifies as adding an array, violating this restriction.

Do not include a newline in the arguments that you pass to `app_error_fmt`, as `app_error_fmt` itself outputs the required newline at the end of the error message. These messages should be printed to `stderr` in the order in which the files occur in the `input_files` array passed to `do_count` (in contrast to the counts, which should be printed to `stdout` in ASCIIbetical order by filename). When such an irrecoverable error occurs, you should not print anything to `stdout` for that file and no counts from that file should be included in the total counts. Even if all files cause errors, the `total` line should still be printed.

Notes

Important: be sure to read this *entire* assignment handout carefully. Part of the point of this assignment is to ensure that you can follow an English language specification of a program precisely. This means that the output of your program must match the specifications of this handout exactly. *Do not modify the function* `print_counts`*, create your own output format, or add any additional output beyond what is specified in this handout.* You may want to add additional output while you are debugging your program, but be sure that you comment out (or remove) the debugging printouts before you turn in the assignment.

Before you concern yourself with sorting the output, we suggest that you first get your count program working correctly with a single file. As in the Data assignment, you can use the provided `driver.pl` to compile and run your program on the same test cases that we will use to grade your program.

Note that you cannot assume anything about what will be in the input file(s) except that they will not contain any non-ASCII characters. You should not, for example, assume that the input files will only contain words less than 10 characters long or anything like that.

You will find the functions `strcmp` and `isspace` useful. See their man pages for details.

Three functions from the code described throughout the textbook, and provided in the files `csapp.h` and `csapp.c`, may be helpful in this project to simplify error handling:

```
void *Malloc(size_t size);
void Free(void *ptr);
void app_error(char *msg);
```

The `Malloc` and `Free` functions are wrappers around their lowercase counterparts that print an error message and terminate the program if they fail. Keep in mind that this may not always be the behavior you want. For example, if `malloc` fails, it means there is not enough memory in the system, so a smart program may be able to free up memory or use an alternative routine in that case. For this project, however, terminating the program is a reasonable response to out-of-memory errors. The procedure `app_error` simply prints the provided error message and terminates the program, which is useful whenever you encounter an unrecoverable error in your code. These functions will be compiled into your program automatically by the provided `Makefile`.

An additional function is provided in `count.c`:

```
void app_error_fmt(char *fmt, ...);
```

This function takes the same arguments as `printf`. It will format and print the error message appropriately to `stderr`. It adds the string "ERROR: " to the front of the message, and it adds a newline at the end. Unlike `app_error`, it does not terminate the program. This function should be used to report errors encountered during the processing of a file. You can then go on to process the next file in the array.

Getting Started

To get started on this assignment, please visit the web page at `https://classroom.github.com/a/k57qT56c`. (If you are copying this URL, do not include the period at the end of the sentence, as it is not part of the URL.) This page should say "RICE-COMP222-F25-Classroom" and "Accept the assignment — Count". Moreover, it should have a green button labeled "Accept this assignment"[2]. Please accept the assignment.

Upon accepting the assignment, you will be redirected to another web page. This page will confirm that you have accepted the assignment, and it will eventually (after you click refresh) provide you with a link to your personal repository for the assignment. Click this link to go to your personal repository.

The web page for your personal repository has a green button labeled "Code". Click this button. You should now see a text field with a URL. Copy or remember this URL.

Login to the `CLEAR` system if you have not already done so. Type the following:

```
git clone [Copy the URL for your repo here]
```

You will be prompted for your `github` username and password.

Once the clone operation is complete, you will have a directory named

```
count-[YOUR github ID]
```

Please `cd` into this directory, and run the command `ls`. You should see the following files:

- `count.c` – provided code
- `Makefile` – specification for compiling `count` using `make`
- `driver.pl` – compiles and runs your program on a set of test cases

If you do ***NOT*** see these files, contact the course staff immediately!

Building Your Program

To build the program, use the Unix command:

```
make
```

This will compile your code and build the `count` program. The `make` command will compile the `csapp.c` file and include it in your program.

Testing Your Program

To test your program, use the Unix command:

```
driver.pl
```

This will first use `make` to build your program, and then run it on the same test cases that we will use to grade your program.

[2] You may have to login to GitHub to see this page. If so, you will be prompted for your GitHub username and password.

Turning in Your Assignment

To turn in your assignment, you ***must*** use `git push` to copy your work to the `github` remote repository. We will *only* look at the last version that you `push`ed before the deadline. As a precaution against accidental loss of your code, we encourage you to `push` periodically. Please note, the *only* file that you need to turn in is `count.c`. In other words, this is the only file on which you should ever perform `git add`. For grading your submission, we will use the `Makefile` that was originally provided to compile your code. Therefore, your code should not rely on any modifications to the `Makefile` for correct compilation.

As a sanity check, you should use your web browser to visit your assignment repo. Make sure that what you see in the browser is consistent what you think you have pushed.

Evaluation

Your program will be graded by performing *exact* matching on the output of your program and the correct output. **You will receive no credit for every test that is run in which your program's output deviates from the formatting described above in any way.** This includes both the counts and the error messages.

The maximum achievable score on the test cases is 60 points.