KAIST

EE 209: Programming Structures for EE

Assignment 1: A Word Counting Program

(This assignment borrows some statements and examples from Princeton COS 217's "De-Comment" Program Assignment)

Purpose

The purpose of this assignment is to help you learn or review (1) the fundamentals of the C programming language, (2) a portion of the "de-commenting" task of the C preprocessor, and (3) how to use the GNU/Unix programming tools, especially `bash`, `emacs`, and `gcc209`.

Rules

Make sure you study the course *Policy* web page before doing this assignment or any of the EE 209 assignments. In particular, note that you may consult with the course instructors, lab TAs, mailing list, etc. while doing assignments, as prescribed by that web page. However, there is one exception...

Throughout the semester, each assignment will have an "on your own" part. You must do that part of the assignment *completely on your own*, without consulting with the course instructors, lab TAs, mailing list, etc., except for clarification of requirements. You might think of the "on your own" part of each assignment as a small take-home exam.

For this assignment, "detecting and reporting unterminated comments" (as described below) is the "on your own" part. That part is worth 10% of this assignment.

The Task

Your task is to write a C program named `wc209` that prints the number of lines, words, and characters in the input text fed from standard input to standard ouput. The program behaves similarly to Linux `wc`, but `wc209` skips "commented text"(e.g., text in `/* ... */`) and does not count such text in the output.

Functionality

Your program should read characters from the standard input stream, and writes the output to the standard output stream and possibly to the standard error stream. Specifically, your program should (1) read text from the standard input stream, (2) writes the number of lines, words, and characters in the input text to the standard output stream with each comment replaced by a space, and (3) writes error and warning messages as appropriate to the standard error stream. A typical execution of your program from a shell might look like this:

```
./wc209 < somefile.txt 2> errorandwarningmessages
 3 13 300
```

The output (`3 13 300`) indicates that there are 3 lines, 13 words, and 300 characters in the file, `somefile.txt`.

Here are a few rules.

- The syntax of comment follows that of the C language (C90).
- A word consists of one or more sequence of non-space characters. A space character is recognized by `isspace()` -- space, form-fead ('₩f'), newline('₩n'), carriage return ('₩r'), horizontal tab('₩t'), and vertical tab ('₩v').
- An empty line (except for the first line) counts as one line. For example, "abc₩n" produces `2 1 4`. However, an empty file (with no characters at all) produces `0 0 0`.

In the following examples a space character is shown as "$_s$" and a newline character as "$_n$".

Your program should internally replace each comment with a space. Examples:

Standard Input Stream	**Internal Representation After Decommenting**	**Standard Output Stream**	**Standard Error Stream**
abc/*def*/ghi$_n$	abc$_s$ghi$_n$	2$_s$2$_s$8$_n$	
abc/*def*/$_s$ghi$_n$	abc$_{ss}$ghi$_n$	2$_s$2$_s$9$_n$	
abc$_s$/*def*/ghi$_n$	abc$_{ss}$ghi$_n$	2$_s$2$_s$9$_n$	

Your program should define "comment" as in the C90 standard. In particular, your program should consider text of the form (/*...*/) to be a comment. It should *not* consider text of the form (//...) to be a comment. Example:

Standard Input Stream	**Internal Representation After Decommenting**	**Standard Output Stream**	**Standard Error Stream**
abc//def$_n$	abc//def$_n$	2$_s$1$_s$9$_n$	

Your program should allow a comment to span multiple lines. That is, your program should allow a comment to contain newline characters.

Your program should add blank lines as necessary to preserve the original line numbering. Also, each newline character in comment is counted as one character. Examples:

Standard Input Stream	Internal Representation After Decommenting	Standard Output Stream	Standard Error Stream
abc/*def␤ghi*/jkl␤mno␤	abc␣␤jkl␤mno␤	4␣3␣13␤	
abc/*def␤ghi␤jkl*/mno␤pqr␤	abc␣␤␤mno␤pqr␤	5␣3␣14␤	

Your program should not recognize nested comments. Example:

Standard Input Stream	Internal Representation After Decommenting	Standard Output Stream	Standard Error Stream
abc/*def/*ghi*/jkl*/mno␤	abc␣jkl*/mno␤	2␣2␣13␤	

Your program should detect an unterminated comment. If your program detects end-of-file before a comment is terminated, it should write the message "Error: line X: unterminated comment" to the standard error stream. "X" should be the number of the line on which the unterminated comment begins. Examples:

Standard Input Stream	Standard Output Stream	Standard Error Stream
abc/*def␤ghi␤		Error:␣line␣1:␣unterminated␣comment␤
abcdef␤ghi/*␤		Error:␣line␣2:␣unterminated␣comment␤
abc/*def/ghi␤jkl␤		Error:␣line␣1:␣unterminated␣comment␤
abc/*def*ghi␤jkl␤		Error:␣line␣1:␣unterminated␣comment␤
abc/*def␤ghi*␤		Error:␣line␣1:␣unterminated␣comment␤
abc/*def␤ghi/␤		Error:␣line␣1:␣unterminated␣comment␤

Your program (more precisely, its `main` function) should return `EXIT_FAILURE` if it was unsuccessful, that is, if it detects an

unterminated comment and so was unable to remove comments properly. Otherwise it should return `EXIT_SUCCESS` or, equivalently 0. Note that EXIT_SUCCESS and EXIT_FAILURE are defined in the standard header file, `stdlib.h`.

Your program should work for standard input lines of any length whose number of characters is less than 2 billion characters.

Design

Design your program as a *deterministic finite state automaton* (*DFA*, alias *FSA*). The DFA concept is described in lectures, and in Section 5.1 of the book *Introduction to Programming* (Sedgewick and Wayne). That book section is available through the web at http://introcs.cs.princeton.edu/java/51language/.

We suggest that your program use the standard C `getchar` function to read characters from the standard input stream.

Logistics

You should create your program on the lab machines cluster using `bash`, `emacs`, and `gcc209`.

Step 1: Design a DFA

Express your DFA using the traditional "ovals and labeled arrows" notation. More precisely, use the same notation as is used in the examples from Section 7.3 of the Sedgewick and Wayne book. Let each

oval represent a state. Give each state a descriptive name. Let each arrow represent a transition from one state to another. Label each arrow with the character, or class of characters, that causes the transition to occur. We encourage (but do not require) you also to label each arrow with action(s) that should occur (e.g. "print the character") when the corresponding transition occurs.

Express as much of the program's logic as you can within your DFA. The more logic you express in your DFA, the better your grade on the DFA will be.

To properly report unterminated comments, your program must contain logic to keep track of the current line number of the standard input stream. You need not show that logic in your DFA.

Step 2: Create Source Code

Use `emacs` to create source code in a file named `wc209.c` that implements your DFA.

Step 3: Preprocess, Compile, Assemble, and Link

Use the `gcc209` command to preprocess, compile, assemble, and link your program. Perform each step individually, and examine the intermediate results to the extent possible.

Step 4: Execute

Execute your program multiple times on various input files that test all logical paths through your code.

We have provided several files here.

(1) Download all files to your project directory. You will find `samplewc209` and 10 test C source files. You need to make `samplewc209` executable (by changing file permission) by

```
chmod u+x samplewc209
```

(2) `samplewc209` is an executable version of a correct assignment solution. Your program should write *exactly* (character for character) the same data to the standard output stream and the standard error stream as `samplewc209` does. You should test your program using commands similar to these:

```
./samplewc209 < somefile.c > output1 2> errors1
./wc209 < somefile.c > output2 2> errors2
diff output1 output2
diff errors1 errors2
rm output1 errors1 output2 errors2
```

The Unix `diff` command finds differences between two given files. `diff output1 output2` produces output, then `samplewc209` and your program have written different characters to the standard output stream. Similarly, if the command `diff errors1 errors2` produces output, then `samplewc209` and your program have written different characters to the standard error stream.

You also should test your program against its own source code using a command sequence such as this:

```
./samplewc209 < wc209.c > output1 2> errors1
./wc209 < wc209.c > output2 2> errors2
diff output1 output2
diff errors1 errors2
rm output1 errors1 output2 errors2
```

Step 5: Create a `readme` File and an Ethics document

Use `emacs` to create a text file named `readme` (not `readme.txt`, or `README`, or `Readme`, etc.) that contains:

- Your name, student ID, and the assignment number.
- A description of whatever help (if any) you received from others while doing the assignment, and the names of any individuals with whom you collaborated, as prescribed by the course Policy web page.
- (Optionally) An indication of how much time you spent doing the assignment.
- (Optionally) Your assessment of the assignment: Did it help you to learn? What did it help you to learn? Do you have any suggestions for improvement? Etc.
- (Optionally) Any information that will help us to grade your work in the most favorable light. In particular you should describe all known bugs.

Descriptions of your code should not be in the `readme` file. Instead they should be integrated into your code as comments.

Your `readme` file should be a plain text file. Don't create your `readme` file using Microsoft Word, Hangul (HWP) or any other word processor.

For **every** assignment submission, you **must** submit your own Ethics document that pledges that you did not violate any rules of course Policy or any rules of ethics enforced by KAIST while doing this assignment.

Please edit an Ethics document for assignment 1 and submit it along with other files. Please write the assignment number, your name, sign on it, and make it into a PDF file (you can convert it into the PDF format in the FILE menu of MS Word).

Step 6: Submit

Your submission should include your `wc209.c` file, the files that `gcc209` generated from it, and your `readme` file. Also submit your DFA. You can do that using either of these two options:

1. Create hard copy of your "labeled ovals and labeled arrows" DFA and give it to your preceptor during precept before the assignment due date. A DFA drawn using drawing software (e.g. Microsoft PowerPoint) would be best. But it is sufficient to submit a neatly hand-drawn DFA.
2. Create a soft copy textual representation of your DFA, and submit it with your other files. For example, the last DFA given in Section 7.3 of the Sedgewick and Wayne book could be expressed as this Textual DFA. Use that textual notation.

If you use option 2, then name the text file `dfa` (not `dfa.txt`, `DFA`, etc.) We cannot accept your DFA via e-mail. We cannot accept your DFA electronically in any form other than plain text.

Create a local directory named 'YourID_assign1' and place all your files in it. Then, `tar` your submission file by issuing the following command on a lab machine (assuming your ID is 20091234):

```
mkdir 20091234_assign1
mv wc209.c wc209.i wc209.s wc209.o wc209 readme Ethics0ath.pdf dfa
20091234_assign1
tar zcf 20091234_assign1.tar.gz 20091234_assign1
```

Upload your submission file (20091234_assign1.tar.gz) to our KMLS assignment submission page. We do not accept e-mail submission (unless our course KLMS page is down).

Grading

We will grade your work on two kinds of quality: quality from *the user's* point of view, and quality from *the programmer's* point of view. To encourage good coding practices, we will deduct points if `gcc209` generates warning messages.

From the user's point of view, a program has quality if it behaves as it should. The correct behavior of your program is defined by the previous sections of this assignment specification, and by the behavior of the given `samplewc209` program.

From the programmer's point of view, a program has quality if it is well styled and thereby easy to maintain. In part, style is defined by the rules given in *The Practice of Programming* (Kernighan and Pike), as summarized by the Rules of Programming Style document. For this assignment we will pay particular attention to rules 1-24. These additional rules apply:

- **Names**: You should use a clear and consistent style for variable and function names. One example of such a style is to prefix each variable name with characters that indicate its type. For example, the prefix `c` might indicate that the variable is of type `char`, `i` might indicate `int`, `pc` might mean `char*`, `ui` might mean `unsigned int`, etc. But it is fine to use another style -- a style that does not include

the type of a variable in its name -- as long as the result is a clear and readable program.

- **Comments**: Each source code file should begin with a comment that includes your name, the number of the assignment, and the name of the file.
- **Comments**: Each function -- especially the `main` function -- should begin with a comment that describes *what the function does* from the point of view of the caller. (The comment should not describe *how the function works*.) It should do so by *explicitly* referring to the function's parameters and return value. The comment also should state what, if anything, the function reads from the standard input stream or any other stream, and what, if anything, the function writes to the standard output stream, the standard error stream, or any other stream. Finally, the function's comment should state which global variables the function uses or affects. In short, a function's comments should describe the flow of data into and out of the function.
- **Function modularity**: Your program should not consist of one large `main` function. Instead your program should consist of multiple small functions, each of which performs a single well-defined task. For example, you might create one function to implement each state of your DFA.
- **Line lengths**: Limit line lengths in your source code to 72 characters. Doing so allows us to print your work in two columns, thus saving paper.

Special Note

As prescribed by Kernighan and Pike style rule 25, generally you should avoid using global variables. Instead all communication of data into and out of a function should occur via the function's parameters and its return value. You should use ordinary *call-by-value* parameters to communicate data from a calling function to your function. You should use your function's return value to communicate data from your function back to its calling function. You should use *call-by-reference* parameters to communicate additional data from your function back to its calling function, or as bi-directional channels of communication.

However, call-by-reference involves using pointer variables, which we have not discussed yet. So for this assignment you may use global variables instead of call-by-reference parameters. (But we encourage you to use call-by-reference parameters.)

In short, you should use ordinary call-by-value function parameters and function return values in your program as appropriate. But you need not use call-by-reference parameters; instead you may use global variables. In subsequent assignments you should use global variables only when there is a good reason to do so.