CSCI 1312 (Introduction to Programming for Engineering), Fall 2017

Homework 10

Credit: 50 points.

1 Reading

Be sure you have read (or at least skimmed) the assigned readings from chapters 7 and 11.

2 Honor Code Statement

Please include with each part of the assignment the Honor Code pledge or just the word "pledged", plus one or more of the following about collaboration and help (as many as apply).[1] Text *in italics* is explanatory or something for you to fill in. For written assignments, it should go right after your name and the assignment number; for programming assignments, it should go in comments at the start of your program(s).

- This assignment is entirely my own work. *(Here, "entirely my own work" means that it's your own work except for anything you got from the assignment itself — some programming assignments include "starter code", for example — or from the course Web site. In particular, for programming assignments you can copy freely from anything on the "sample programs page".)*

- I worked with *names of other students* on this assignment.

- I got help with this assignment from *source of help — ACM tutoring, another student in the course, the instructor, etc. (Here, "help" means significant help, beyond a little assistance with tools or compiler errors.)*

- I got help from *outside source — a book other than the textbook (give title and author), a Web site (give its URL), etc.. (Here too, you only need to mention significant help — you don't need to tell me that you looked up an error message on the Web, but if you found an algorithm or a code sketch, tell me about that.)*

- I provided help to *names of students* on this assignment. *(And here too, you only need to tell me about significant help.)*

3 Programming Problems

Do the following programming problems. You will end up with at least one code file per problem. Submit your program source (and any other needed files) by sending mail to `bmassing@cs.trinity.edu` with each file as an attachment. Please use a subject line that mentions the course and the assignment (e.g., "csci 1312 hw 10" or "CS1 hw 10"). You can develop your programs on

[1] Credit where credit is due: I based the wording of this list on a posting to a SIGCSE mailing list. SIGCSE is the ACM's Special Interest Group on CS Education.

any system that provides the needed functionality, but I will test them on one of the department's Linux machines, so you should probably make sure they work in that environment before turning them in.

Yes, this writeup is long. But I think the code you write need not be, and it's an interesting problem!

You may have heard claims that E is the most frequently-used character in English text, followed by T, and so forth. Your mission for this assignment is to write two programs that together will allow you to find out how true this claim is for selected text (and, okay, to give you practice working with some course topics):

- The first program analyzes a single file of plain-text, counting occurrences of each alphabetic character and writing results (characters and counts, but only for characters that occur at least once) to an output file.
- The second program merges one or more files produced by the first program and writes results to an output file.

(Why two programs? Mostly pedagogical reasons.) Writing the programs from scratch is nontrivial (though you could probably do it), so to make it more doable I'm providing starter code that reduces what you need to do and also gives you some practice with UNIX `make`, discussed in class. Once you have an output file produced by the second program, you can use the Linux command

```
sort -n -r outfilename
```

to display the results in a way that shows the most-often-used letter first, etc.

To give you some practice working with `struct`s in C, I want you to do this problem using an array of a `struct`s, with each `struct` containing a letter and a count of how many times it occurs in the input(s). Since you need such an array in both programs, as well as code to look up a particular letter and increment its counter, it would seem to make sense to have a "library" used by both programs that declares/defines the `struct` and some needed functions. I've written code that declares the needed `struct` and declares some functions for building and operating on the needed array and also starter code for the two programs. Your mission will be to fill in the missing pieces. There are several ways to combine this "library" code with the two programs, but what I want you to do is to use the Linux utility `make`, as discussed in class. Starter code, with `FIXME` comments showing where you need to add code:

- alphacounters.h[2], declarations of "library" functions with comments saying what they do.
- alphacounters.c[3], starter definitions of "library" functions.
- countalpha.c[4], starter code for `countalpha` program.
- mergecounts.c[5], starter code for `mergecounts` program.

[2] `http://www.cs.trinity.edu/~bmassing/Classes/CS1312_2017fall/Homeworks/HW10/Problems/alphacounters.h`

[3] `http://www.cs.trinity.edu/~bmassing/Classes/CS1312_2017fall/Homeworks/HW10/Problems/alphacounters.c`

[4] `http://www.cs.trinity.edu/~bmassing/Classes/CS1312_2017fall/Homeworks/HW10/Problems/countalpha.c`

[5] `http://www.cs.trinity.edu/~bmassing/Classes/CS1312_2017fall/Homeworks/HW10/Problems/mergecounts.c`

- Makefile[6], "makefile" to build the two programs.

Rather than copying or downloading each of these files separately, you'll probably find it easier to download the ZIP file hw10.zip[7] and unzip it with `unzip hw10.zip`. If you prefer to download individual files, *NOTE* that you should use your browser's "download" or "save" function to obtain the Makefile rather than copying and pasting text. This is because copy-and-paste will likely replace the tab characters in the file with spaces, with bad consequences (since tabs are semantically significant in makefiles.)

The `Makefile` includes instructions for "building" the project. Note that just using `gcc` with a single program, as we've been doing, won't work, but once you have all the above files downloaded, typing `make` will produce two executables, `countalpha` and `mergecounts`, that you can run (although they won't do anything very interesting). You might try that before starting to write code.

Instructions for specific files you need to change:

1. (5 points) The first file you need to change is `alphacounters.c`, which provides code for functions declared in `alphacounters.h`. (Notice that `alphacounters.h` also includes comments describing what these functions do — very important!) There's only one function you need to write code for, the one that given a character finds the element of the array for it and increments its counter, and I'm hoping that the functions I'm providing code for will give you some hints about how to work with the array. You can check that your code at least compiles by typing `make` again.

2. (20 points) The next file you need to change is the code for the first program, the one that analyzes a single input file and produces an output file. The starter code checks that there are two command-line arguments (filenames for input and output) and opens the input file. Add code to do the following:

   - Read the input file a character at a time and count, using the function `update_count` (written in the first step), how many times each alphabetic character occurs (but use `tolower()` first to turn any upper-case characters into lower-case). Note that this function also tells you whether the character is even alphabetic — it returns `false` if not — so you don't need a separate check using `isalpha`. Note also that to get full credit for this part you *must* use this function rather than trying to figure out another way to update the right counter.
   - Count the total number of characters and how many were alphabetic.
   - For every alphabetic character that occurs at least once, write to the output file a line with the character and the count.
   - Print the total number of characters and the number of alphabetic characters.

   This is probably easiest to understand with examples. If the input file looks like this:

   ```
   testing 1 2 3 4?

   TESTING 4 3 2 1!
   ```

[6] http://www.cs.trinity.edu/~bmassing/Classes/CS1312_2017fall/Homeworks/HW10/Problems/Makefile

[7] http://www.cs.trinity.edu/~bmassing/Classes/CS1312_2017fall/Homeworks/HW10/Problems/hw10.zip

the output file should look like this:

```
e 2
g 2
i 2
n 2
s 2
t 4
```

and the program should print this:

```
alphabet 'abcdefghijklmnopqrstuvwxyz'
14 alphabetic characters, 36 total characters
```

And if the input file looks like this:

```
Now is the time for all good persons
to come to the aid of their party!
```

the output file should look like this:

```
a 3
c 1
d 2
e 6
f 2
g 1
h 3
i 4
l 2
m 2
n 2
o 9
p 2
r 4
s 3
t 7
w 1
y 1
```

and the program should print this:

```
alphabet 'abcdefghijklmnopqrstuvwxyz'
55 alphabetic characters, 72 total characters
```

3. (20 points) The last file you need to change is the code for the second program, the one that merges output from repeated executions of the first program. The starter code checks that there is at least one command-line argument, builds the array of `structs`, and calls a function `process_file` for each input filename to process that single file. Add code to do the following:

- Actually do something in `process_file`, in addition to printing the filename — read the file a line at a time (more below about how to do this) and use this information to update the array of counters. Print an error message if the file cannot be opened or has errors (more below). The function should return `true` if everything was okay, `false` if there was an error.
- After all input files have been processed, write to the output file a line for each element of the array of `struct`s for which the count is nonzero, printing first the count and then the letter (this is to make it easier to sort the output with the `sort` command).

About reading lines from the input file, to get some practice with an additional way of reading text input, I want you to use `fgets` to get a line at a time and `sscanf` to then pick out the character and the count. (Sample program `grades.c` has an example of using this technique.) Your program should do something sensible if an input line is too long to fit into the array you declare to hold it (such as printing an error message and throwing away the rest of the line). It should also print an error message for any input line that isn't in the right form (character, space, integer, end-of-line). The starter code has some additional hints.

Here too this is probably easiest to understand with an example. Given the two output files shown earlier, the program should combine them to produce an output file containing

```
3 a
1 c
2 d
8 e
2 f
3 g
3 h
6 i
2 l
2 m
4 n
9 o
2 p
4 r
5 s
11 t
1 w
1 y
```

and print this:

```
alphabet 'abcdefghijklmnopqrstuvwxyz'
processing input file sample1-out.txt
processing input file sample2-out.txt
```

Finally, the program should give an error message for every line of this input file:

```
hello
x
100
x 1000x
```

4. (5 points) Finally, you should try your programs with some non-trivial input. The Project Gutenberg web site[8] is a good source of freely-available text. I downloaded copies of two books (one by Jane Austen, one by P.G. Wodehouse) in UTF-8 format, converted to plain-text, and made another ZIP file hw10-data.zip[9] with the results. Run your programs on these two files and send me your output files (results of running `countalpha` on each of the input files, and result of running `mergecounts` to combine them).

   If you find this sort of thing interesting, you could download additional books and try the program with them. I used the following command to convert from UTF-8 to really-plain-ASCII-text:

   ```
   iconv -f UTF-8 -t US-ASCII -c infile.txt >outfile.txt
   ```

   Or word-processing programs will also export to plain text, though if you try that route you should probably open the resulting file in `vim` and make sure it looks like text.

   If you do this, send me your additional input files for extra credit.

[8] http://www.gutenberg.org

[9] http://www.cs.trinity.edu/~bmassing/Classes/CS1312_2017fall/Homeworks/HW10/Problems/hw10-data.zip