CSE 109: Systems Programming

Fall 2018

Program 1: **Due on Sunday, September 9th at 9pm on CourseSite.**

Checkpoint Due: Tuesday, September 4th at 9pm, via Checkpoint submissions.

Note: All file and folder names in this document are case-sensitive.

Collaboration Reminder:

1. You must submit your own work.
2. In particular, you may not:
   (a) Show your code to any of your classmates
   (b) Look at or copy anyone else's code
   (c) Copy material found on the internet
   (d) Work together on an assignment

Assignment:

1. Log into sunlab.cse.lehigh.edu
2. Create a folder named *Prog1* in your cse109.182 folder.

   This will contain all of your submission materials.
3. Use a text editor to create a C source code file named *prog1.c* in the *Prog1* folder.

   All relevant code for this assignment will end up in this source file.
4. Add the comment block shown at the end of this document to the top of the file named *prog1.c*.

   Make sure to transform all <item> notations into what is expected.

   Do not use your LIN, use the 6 character prefix of your Lehigh Email as your user id.

5. Create a main function and any additional functions necessary to do the following:

   (a) The program will take all input from standard input, possibly transform it, and output it to standard output.

   (b) The program will read in input line by line. Transformations, if any, will be done per line. Then print out the transformed line.

   (c) Transformations are specified by the command line arguments provided to the executable. You will have to read those arguments and handle them. (See Checkpointing below).

      i. *-r*: Reverse each word on the line

      ii. *-o*: 'Rotate': Rotate the first character of each word to the right (circularly).

      iii. *-t*: Change lowercase to uppercase and vice versa (toggle)

      iv. *-n*: Remove any digits ('0'-'9') from the line
          Do not include *:* in your command line arguments.

   (d) Multiple command line arguments may be provided, in that case, apply multiple transformations. You **must** ignore any duplicate command line arguments.

   (e) If a command line argument is given that does not match one that we accept, print "Invalid command line option" to standard error and return 1 (this should result in exiting the program - it should not process any of the lines of text).

   (f) You do not have to wait until all lines have been read in to print out the transformations. However, you absolutely must store the line in memory, modify that stored line, and ultimately print what remains in storage.

   (g) Make sure you understand what **whitespace** means and how it functions.

   (h) We will treat any non-whitespace as part of a word.

   (i) You may not use a gigantic statically declared array (or any large stack-based array) to handle the text input and conversions. Use the automatic capabilities of the functions provided. You are welcome to use malloc and do the memory acquisition yourself.

   (j) Hint: You will have to read from the user until there is no more text left. Ctrl+D can be typed into the terminal to indicate there is no text left. You will have to figure out how to get your input handler to understand this. Consider looking at the man page for *scanf*.

(k) Hint: You can use the functions in *ctype.h* to help you with the transformations. Look at the man page for *isalpha* for more info.

6. Compile your program using the command:

   Make sure the command *gcc -v* specifies gcc-7.1.0 (module load gcc-7.1.0)

   ```
   gcc -g -Wall -Werror -o changetext prog1.c
   ```

7. Create *myfile*, a file that contains lots of text, perhaps the text from the Wartortle page on Bulbapedia.

8. Run the program as part using *myfile* as input (two different ways shown):

   ```
   ./changetext <options> < myfile
   ```

   ```
   cat myfile | ./changetext <options>
   ```

   Make sure to replace <options> with whatever is appropriate for your test case.

9. Make sure to sufficiently test your code and that it makes sense and works for various combinations of the command line arguments.

10. Once ready to submit, you can package up the assignment as a .tgz file

    ```
    tar -czvf Prog1.tgz Prog1
    ```

    You must use this command in the directory that contains the *Prog1* folder, not within the directory.

11. Transfer *Prog1.tgz* to the Program 1 submission area of CourseSite.

Checkpointing

1. Each of the four transformations must be done via a function that is called by *main*. These functions must be prototyped above *main* and be defined after *main*.

2. void reverseWord(char* line, size_t length) => Satisfies *-r*

3. void toggle(char* line, size_t length) => Satisfies *-t*

4. void rotate(char* line, size_t length) => Satisfies *-o*
5. size_t removeDigits(char* line, size_t length) => Satisfies *-n*

   removeDigits returns the length of the resulting line.
6. Only *reverseWord* and *toggle* are due at the Checkpoint time. However, the Checkpoint will test all four of these (you **must** include the prototypes and at least a stubbed definition of all four for the Checkpoint).
7. These functions can call other functions, as long as everything is included in your checkpoint submission.
8. To submit your Checkpoint, copy your *prog1.c* to *checkpoint1.c*, remove the main function from the code. Then you can enter the following command:

   ```
   ~jloew/CSE109/submitCheckpoint.pl 1
   ```

   That is lowercase PL, followed by the number 1.

   Do not copy/paste the ∼ symbol. It may not copy/paste properly.
9. You may submit your Checkpoint multiple times before the deadline and even afterwards.
10. Ideally, it will tell you which functions are incorrect. It is possible that the functions are incorrect but pass the checkpoint. Although, they should be mostly correct or completely correct if they pass the checkpoint.
11. The checkpoint will not check for memory corruption or leaks. You will need to handle that yourself.

Testing

1. Make sure your code does not leak memory by running it with Valgrind.
2. Some test cases (*.in files) will be provided in the directory:

   ∼jloew/CSE109/prog1student/
3. A working executable will also be in that directory.

4. The working executable is built to the expected specifications (unless we screwed up). Any confusion in the requirements can potentially be tested by using the working executable.

5. You can generate output files from the working executable and use *diff* to compare them to the results that your code generates.

6. Warnings constitute errors, they are there for a reason, fix them!

7. Valgrind can tell you what lines of code that some errors (the ones it detects) come from. You may find that useful as well.

Comment Block:

```
/*
    CSE 109: Fall 2018
    <Your Name>
    <Your user id (Email ID)>
    <Program Description>
    Program #1
*/
```