

Programming Assignment #2

This assignment gives you some practice using an API to write a filter and process command line arguments. The goal is to implement a filter that will manipulate text files in various ways. Your program should be able to handle both short and long options that were discussed in class. Most of the work will be related to the command line argument processing. The filtering functionality is straight-forward and every third-semester student should have few problems with it as the complexity of coding is at the freshman level.

Option	Description
-b, --remove-blank-lines	Removes all empty lines. An empty line is a line that contains nothing but a newline.
-d, --delete=X	Given a character, delete all occurrences of it. (Requires an argument)
-h, --help	Displays information about the options. See website for an example.
-l, --tolower	Converts all characters to lower case. (Short option is lower case 'L')
-r, --replace=XY	Replaces all occurrences of character X with character Y. (Requires an argument).
-n, --line-numbers	Prepends a line number to each line.
-u, --toupper	Converts all characters to upper case.
-x, --expand-tabs[=X]	Replaces all tabs with X spaces. (Optional argument, default is 8)

These examples all do the same thing, which is to remove empty lines, replace tabs with 2 spaces, and delete the lowercase letter a. from `foo.c`:

```
./filter foo.c -b -x2 -da
./filter foo.c -b -x2 -d a
./filter foo.c --remove-blank-lines --expand-tabs=2 --delete=a
./filter foo.c --remove-blank-lines --expand-tabs=2 --delete a

cat foo.c | ./filter --remove-blank-lines --expand-tabs=2 --delete a
./filter --remove-blank-lines --expand-tabs=2 --delete a < foo.c
```

The last two examples are slightly different in that the program reads from **stdin** instead of a file. (Technically, **stdin** is a file pointer, so your code will be able to handle both situations easily). If no files are given on the command line, you will read from **stdin**, otherwise you will open the text file(s) that were provided. You may be given any number of files to process and the filenames may be mixed with the options as such:

```
./filter --remove-blank-lines foo.c --expand-tabs=2 bar.c --delete a baz.c
```

The above command will filter all 3 files, in order. All output is written to **stdout**. See the website for more examples and information.

Some of the filters are sensitive to the order performed. For those filters, you should filter the characters in this order:

1. Remove blank lines
2. Delete characters
3. Replace characters
4. Upper case
5. Lower case

This will ensure that everyone generates the same output when filtering. Also, you do not need to deal with an option given multiple times like this:

```
./filter foo.c -da -db
```

This is asking you to delete two characters. You won't be given anything like this to test. If you want to deal with it in the simplest way, just accept the last option (last one wins), so you would just delete the character `b` from the input.

More on back →

Other criteria

1. All helper functions (and there must be a few or you will lose points for a poorly structured program having almost all code in main), must be tagged as **static**. This also goes for any variables you create outside of a function. Declaring a variable outside of a function will make it global, and this is one of the only assignments where you will ever do that. It may be necessary because there are a lot of options and most of the functions you create will need to access them. You could pass them to the functions, but you will need to put them all in a **struct** of some kind, which is equally acceptable.
2. You should use **fopen** to open the file (mode: "rt"), **fclose** to close the file, **getc** (or **fgetc**) to read a character and **putc** (or **fputc**) to write a character. Consult any C reference for details.
3. Be sure to free all of the memory that you allocate. You will be required to use *valgrind* on all assignments (to look for leaks and other bad things), as that is what I will be using. I will post a command line that you should use. Any memory leaks will cost you significant points.
4. If you are not given any filenames on the command line, you are to read from **stdin**. Any program that fails to read from **stdin** will receive an automatic 0.
5. If you do receive filenames, you need to process them in order they are received with the options. You must not use the default behavior of *getopt*, which is to move them to the end of the command. Doing so will result in an automatic 0 for the assignment.
6. You will be reading either from **stdin** or from the file you opened, never both.
7. Be sure to check the result of **malloc** and **fopen** for failures. I will give you bad file names and you need to deal with them properly. Failure to check the return values will cost you to lose points (even if the call succeeded).
8. If an unknown option is provided or an option requiring an argument does not have an argument, you must emit an error message. (See samples on website on how to deal with this situation.)
9. You must handle all options, both the short and long type.
10. To handle multiple files on the command line you may have to **malloc** a dynamic array of **char** pointers, one for each filename. In order to count the number of files, you must parse the command line twice, the first time just to count the file names and the second one to save the filenames to your array. Make sure you reset **optind** to 1 before the second pass. Don't forget to free any memory that you dynamically allocated. **Hard coding a size for the array will cause you to lose 50 points.**
11. To help you during the filter process, you should **read exactly one character at a time using getc (or fgetc)**. In almost every case, you should then **write one character at a time using putc (or fputc)**. Only one of the seven filters (`--line-numbers`) will require the use of a **printf** instead. Anything else complicates your code and is a waste of time.
12. All output goes to **stdout**. Period. **Writing anything to a file will cause you to lose 50 points.** (You don't need to do that.)
13. As always, this assignment is not very hard, as long as you think before typing. The programming level is that of CS 120, so I expect everyone in this class to be able to handle this assignment. You should do things in this order:
 1. Define the table of long options. All of the information is provided at the beginning of this document.
 2. Create a function that will do all of the parsing of the command line options.
 3. Test your code with all of the options, both short and long, especially the ones that have arguments. You should simply just print out information as the options and arguments are encountered, much like all of the examples in the notes. You are probably 75% finished at this point, as the actually reading of the file and outputting characters is trivial.
 4. After you have tested your code to make sure that you can handle all options and errors, you can implement the code that will do the actual filtering of a single file.
 5. Implement the simple filters first (e.g. `--help`, `--toupper`, `--tolower`, `--delete`, `--replace`), then move on to the less simple ones.
 6. After you can read from a file, add code to deal with reading from **stdin**. (It should only be a few additional lines.)
 7. After you can read from one file or **stdin**, add code to handle any number of files (there is no limit).
14. This is a trivial assignment, unless you decide that you want to ignore all of these suggestions and do it Your Way™.

Deliverables

You must submit the implementation file (`filter.c`), the `typescript` file, and the PDF documentation in a .zip file to the appropriate submission area online.

Source files	Description
<code>filter.c</code>	The implementation file. All implementation for the functions goes here. You must document the file (file header comments) and all functions (function header comments) using the appropriate Doxygen tags.
<code>typescript</code>	This is the session that was captured with the <code>script</code> command.
<code>refman.pdf</code>	The PDF produced from the Doxygen tags.

Usual stuff

Your code must compile (using the compilers specified) to receive credit. The code must be formatted as per the documentation on the website.

Make sure your name and other info is on all documents.