Programming Assignment #2
with Extra Credit (on Page 3)

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. Delete 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.

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.
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 complete 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
filter.c	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.
typescript	This is the session that was captured with the script command.
refman.pdf	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.

Extra Credit Portion

Extra Credit
To earn extra credit, you must implement these 3 additional "features" from the filter assignment.

1. **Deleting multiple characters.** Instead of providing only a single character to the `-d` or `--delete=` option, you can provide multiple characters. Examples:

   ```
   -dXYZ   -d"ABC "   --delete="()[]/\"
   ```

   Special characters (like spaces or backslashes) must be put into quotes (single- or double-quotes, depending on the special characters). However, your program will not get the quotes, as they will be stripped off by the shell. You must also accept the delete option multiple times. This is the same as the first example above:

   ```
   -dX   -dY   -dZ
   ```

2. **Replacing multiple characters.** Instead of providing only a pair of characters to the `-r` or `--replace=` option, you can provide multiple pairs of characters. Examples:

   ```
   -rXxYyZz   -r"A B C "   --replace="([)]"
   ```

   Special characters (like spaces or backslashes) must be put into quotes. You must also accept the replace option multiple times. This is the same as the first example above:

   ```
   -rXx -rYy -rZz
   ```

   There must be an even number of characters because they come in pairs. Simply doing this:

   ```
   -rX
   ```

   should emit an appropriate error message and then terminate the program.

3. **Line number behavior.** There are two modifications to this option. The first modification is to allow the user to specify the interval in which to number the lines. For example, to number every 5th line, you would specify the option in either of these ways:

   ```
   -n5   --line-numbers=5
   ```

   This would put numbers on lines 5, 10, 15, 20, … etc. This is an **optional** argument and the default is 1, to number every line. The second modification is to skip numbering blank lines. (This is the default behavior of the `nl` filter command in Linux.) To indicate this, a trailing 's' (lowercase S, without the quotes) is appended to the argument. For example, to number every 5th line, but skip blank lines, the option would look like one of these:

   ```
   -n5s   -n5s   --line-numbers=5s
   ```

   To number every line, but skip blank lines, any of these should be supported:

   ```
   -n1s   -ns   --line-numbers=1s   --line-numbers=s
   ```

   The number must be a positive (i.e. greater than 0) integer. Invalid numbers should set the interval to 1, meaning to number every line. Hint: Use `atoi` to convert the argument to a number. Invalid strings (e.g. X), will result in the value 0, which is invalid and will cause the interval to be set to 1. You don't have to worry about negative integers. However, you might get multiple digits (e.g. 15, 25, 100, etc.) Do not assume that you are only getting a single digit.

More on back →

Extra Credit Portion
You can earn 50 points of extra credit by successfully completing the assignment. Since this is an extra-credit assignment, very little, if any, partial credit will be given. Extra credit assignments are always held to a higher standard than "regular" assignments.

Notes
Since this is an extension of assignment #2, you MUST first make sure that you can pass all of the tests from that assignment. If any of those tests fail, your extra credit code will not be accepted. So, if you already passed all of the original tests, then you can just start working on the new stuff. However, if you had any problems with the original tests, you must fix those before starting the extra credit stuff.

You will have to dynamically allocate the space to hold the characters in both the *delete* and *replace* options. This is simply because you do not know how many characters will be given to you or how many times the option will be provided. You can use dynamic arrays or linked-lists. This means that, if the option is provided multiple times, you must "grow" the array (by one) each time to accommodate the new characters. In the non-extra credit portion, if an option was given multiple times, you simply overwrote the previous values. Now, you are appending the new values to the set of previous values. Hard-coding any sizes or growing the array larger than the number of elements will invalidate your program and you will receive 0 points. As a third-semester student at Digipen, there is no excuse for hard-coding anything. You've learned (or at least have been taught), how to do this. Also, don't forget to free the old memory after growing the array. Use Valgrind to help you make sure.

If you don't want to use arrays that must grow at run-time, you can use a linked-list. Since we're using C, you can't use anything from the STL (bummer!), so you'll have to write your own linked-list routines. The point is that you can't hard-code any sizes, so you either have to use dynamic arrays (that you can grow), or use linked-lists. The choice is yours.

Since skipping blank lines when numbering is a new feature, you need to make sure that you remove the blank lines before attempting to number them. This is the new order to process the options:

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

The website has several examples that demonstrate all of the modifications listed above.

Submitting
On the submission server (pontus), this extra-credit assignment is listed as a lab. This is because it will perform additional checks when you upload. Usually, an assignment just compiles your code and that's it. For this extra-credit assignment, the server will also run other checks like running and diffing, tablen, Doxygen, Valgrind, etc. If the server accepts your code (meaning you have successfully passed all of the automated checks), it will be eligible for grading. If the server rejects it for any reason, it won't be eligible. Once the due date/time has passed, I will download all of the accepted submissions and them have them graded. The grader will be looking at things like documentation, code style, etc. This means that just because the server accepted your code, that doesn't mean you will get 100% on it.