CS 306/491 – Linux Programming – Spring 2016

Lab #1

(Due: 03/12/16 by 11:59pm)

The subject of your first programming assignment is writing a C program to duplicate (a subset of) the functionality of the Linux/UNIX `grep` command. `grep` is a command that searches one or more files for lines that contain a given string/pattern, and prints those lines out. It is considered a *filter* because it writes to standard output and if no file arguments are given, it reads from *standard input* (and so can be used in shell *pipelines*). Your program will be much simpler than `grep`: (1) it *need not* accept any *options*; (2) it will use only a *fixed string* rather than a *regular expression* pattern; and (3) it need handle only a *single file argument*. **This program will use only C standard library functions—no system calls**.

The syntax for your `mygrep` is to be: `mygrep` STRING [FILE]. Notice that the **file argument is optional**. If no file argument is given, the program is to read from *standard input* (`stdin`). By default, `grep` accepts a regular expression pattern for the search string. However, if the "-F" or "--fixed-strings" option is given, PATTERN is taken as a fixed string. This can also be accomplished by invoking `grep` as `fgrep`. The `mygrep` syntax has STRING in the syntax rather than PATTERN, because it is to behave exactly like `fgrep` or `grep` given the"-F" or "--fixed-strings" options: it will search for the string STRING in each of the lines in a file, and print those that contain STRING to standard output (`stdout`). Your code is to exactly duplicate the output that would be produced by calling `fgrep` with the same STRING and FILE arguments.

Here are examples of syntactically valid calls to `mygrep`:

- `mygrep printf lab1.c`
- `mygrep "printf (" lab1.c`
- `cat lab1.c | mygrep printf`
- `mygrep testing` (read from `stdin`, use ctrl-d to terminate input)

You are to submit your code for the assignment in a single file named **lab1.c**. The file must be able to compile and run on its own, which means it must contain a `main` and any needed functions. `main` must obviously be set up to take *command-line arguments* using the C `argc` and `argv` mechanism. Your code should assume that the executable created will be named `mygrep`. Your code is to include and use at least two functions, with prototypes:

- `int grep_stream(FILE *fpntr, char *string, char *file_pathname)`
  - `fpntr` – an open file stream
  - `string` – the search string as a valid C string
  - `file_pathname` – the file path as a valid C string, else "`stdin`"
  - return indicates whether any matching line(s) were found (Boolean)
- `char *get_next_line(FILE *fpntr)`
  - `fpntr` – an open file stream
  - return is the next line in the stream, as a valid C string, else NULL

`grep_stream()` is to perform the primary function of `grep` on an already opened file (also known as a *stream* in UNIX jargon). This means reading line-by-line through the file, identifying lines to be printed, and printing them out to standard output (`stdout`). `main` must eventually know what *exit status* to return, and as we discuss below, this requires knowing whether any matching lines were found or not, thus `grep_stream()`'s Boolean (really `int`) return value.

`get_next_line()` is to read the next line from a stream and return it as **a valid C string**. It should return NULL if an I/O error occurs during its execution or if it immediately encounters the end of the file (i.e., if there are no further characters in the file when the function is called). A line in Linux/UNIX text files is a string of characters terminated by either: (1) a *newline* character ('\n'), or (2) the *end of the file* (EOF). With newline terminated lines, *the newline character is not considered part of the line itself*, and is not to be returned.

The basic logic for `main` should be as follows:

1. examine the command line arguments to to retrieve the search string, and to determine whether to read from standard input or from a file argument
2. if a file is to be processed, open it else just use `stdin` as the stream
3. call `grep_stream()` to process the stream
4. when `grep_stream()` returns, close the stream if it was for a file
5. exit with appropriate exit status

The basic logic for `grep_stream()` should be as follows:

1. iteratively call `get_next_line()` to obtain the next line from the stream
2. for each returned line, check if the line contains the search string
3. if the line should be printed, print it to standard output
4. when `get_next_line` returns NULL, return 0/1 depending on whether any matching lines were found or not (0 if none found, 1 if some found)

The basic logic for `get_next_line()` should be as follows:

1. iteratively call `fgetc()` to obtain the next character from the stream, until the end of the current line is detected or an error occurs reading
2. store each read character into a **"line buffer"** (array), created with `malloc()`
3. when the line end has been reached, turn the array into a valid C string and return it
4. return NULL if there were no more lines in the file or if an error occurred

A key issue in implementing `get_next_line()` is how to allocate memory to hold the string that is to be returned. Memory to hold the line/string is to be allocated within `get_next_line()`.Unfortunately, the maximum length of the lines in any file are unknown in general, so we cannot know how large to make the array either at compile time or when calling `get_next_line()`.The solution is to use dynamic memory allocation, equal to the size of the line to be processed in a call to `get_next_line()`.

The second memory pitfall we face here is that in a function, memory for local variables is allocated on the stack, and will be automatically reclaimed/reused when the function returns. This means that one should never return pointers to these variables from a function (and remember that strings are arrays, and arrays are represented by a pointer to the array start). While the compiler will allow you to do this, it is a logic error in your program! Remember, all dynamic memory allocation calls allocate memory from the heap rather than the stack, and heap memory is not automatically reclaimed—it must be manually reclaimed using `free()`. For this lab, you are to use `malloc()` to allocate fresh memory to store each line.

`main` must ultimately return an appropriate exit status. Normally this should be either EXIT_SUCCESS or EXIT_FAILURE, but `grep` is slightly different (nonstandard). `grep` returns one of 0, 1, or 2. Read the man page (and/or experiment with `grep`) to see what the appropriate exit status is when there are or are not matches, and when an error occurs.

Additional instructions and requirements:

- Remember that you are to return a valid C string containing the line, and C strings will include a null char sentinel ('\0'), so you must allocate more memory than what is required for just the line.
- You are to use C library functions (from `<stdio.h>`) to handle all I/O. The functions you will need include: `fopen()`, `fgetc()`, `fprintf()`, and `fclose()`.
- You are not allowed to use any of the following functions: `fgets()`, `gets()`, `scanf()`, or `fscanf()`.
- All terminal output is to be done with `printf()` and/or `fprintf()`. All error messages are to go to standard error rather than to standard output.

- You will need to use one or more functions from the C string library (`<string.h>`), such as `strstr()`, to check if the argument string occurs in a line.
- `grep_`stream() must always return (it should not call exit()).
- `get_next_line()` should not itself print any error messages or any other output, and it must always return.
- Your program must error check the original call and all library calls.
- In the case of an error in the original call (number of command-line arguments), the output should be a "*usage*" message similar to what you would obtain by typing just `grep` as a command: Usage: `mygrep` [OPTION] STRING FILE.
- In the case of an error with a library call, print the standard system message (using either `perror()` or `strerror()`). You may wish to duplicate the format that most Linux/UNIX programs use, which is to prefix the message with the name of the program followed by a colon, give an informative message followed by a colon, and end with the system error message. E.g., `mygrep`: cannot open file: no such file).
- Note that `fgetc()` has an int return type because while it normally returns the next character (char) in the file, it returns EOF when it reaches the end of the file or gets an error trying to read from the file (EOF is normally the int value -1).
- Whether `fgetc()` has returned EOF due to the end-of-file or due to a read error can be distinguished by using either `ferror()` or `feof()` (or even `errno`).
- If a read error occurs (while calling `fgetc()`), you must print an appropriate error message, but this is to be done by `grep_stream()`, not by `get_next_line()`. Note that is extremely difficult to cause a read error to occur, so testing whether your code properly prints an error message is not straightforward.

Submission Guidelines

You are to submit your **single C source file**, **lab1.c**, using the Lab #1 Dropbox on the course D2L website.

- **Do not zip the file, nor submit any additional, unnecessary files**. In particular, do not submit `project` files created by an IDE you may be using (e.g., Eclipse). There is no reason to submit a compressed or archived version, so try to properly follow instructions and submit the single C source file, properly named.
- If you create/edit your file on a Windows machine, you must make sure that it is in proper Linux/UNIX format when you submit it or you will lose points. No carriage-returns!
- You are free to use any reasonable indentation style, but **you must turn in reasonably indented and readable C code, or you will lose points**!
- **Code that does not compile will typically receive an immediate zero grade; we are not going to try to read through code to assess what parts are written or not.**
- **Code that compiles but fails to run on any test cases will also probably receive a zero.**
- **Points will be deducted for each test case that the code fails to run on.**
- **You should not be getting any warnings from GCC, so make certain you check for this, as we will.**
- Since we still have to look at various parts of your code to grade it, **code must be appropriately indented and a reasonable style**.
- **There should be little/no duplicated code for the different cases**. Once a file has been opened, there is just a minor difference between the way standard input and a file are processed, so you should not have chunks of similar code for standard input and for files. If this is going to occur, you need to break out the similar code into a function. **Points will be deducted if you submit a lab with duplicated code or other style issues**.

Development Suggestions

You are strongly urged to develop your program in stages rather than typing the whole thing in and then trying to get it working. Most complete programs will initially have multiple interacting bugs, making debugging potentially very difficult. While there are many ways you might proceed, one possible approach could be as follows:

- Start by writing a main that simply reads through standard input. Make sure you understand how to stop at the file/input end when reading, so you don't have errors. Also make sure you understand how to call your `mygrep` in a pipeline and with redirection.
- You can start off using `get_line()` as your `get_next_line()`, and create a first pass at `grep_stream()` that just prints all lines in the stream.
- Then modify your code to accept a search string argument and fix `grep_stream()` to print only those lines that contain the search string.
- Finally, write `get_next_line()`, using `malloc()` and `fgetc()`.

The goal is to make small enough changes that only very few things could be wrong at any one time, so you continually having a working or close to working program.

CS 491 Students

Students registered for the course as CS491 (graduate credit) are required to implement an additional functionality:

- Accept one optional option. The option should be invert, which should cause those lines that **do not contain STRING** to be printed rather than those that do contain it. The option can be specified as either: "-v" or "--invert-match". Your program must allow this option specified either way, but if it is specified, you can assume it must occur first in the command argument list.

Additional Programming Challenge (extra credit of 10 points):

- Accept an arbitrary number of file arguments (including none).