CS 105
`Fgrep` Lab

1 About Hints

There is a large "Hints" section (Section 8) at the end of this handout. Be sure to read the entire handout and the hints before starting work, and refer back to the hints frequently while you are writing and debugging your program.

2 Introduction

In this lab you'll write a simple Unix filter. In the process you will learn how to process command-line arguments (both option switches and file names), how to create a program that can read data either from standard input or a given file, and how to use the `stdio` I/O package.

`Fgrep` is a simple command that searches one or more files for a given string. In the I/O lecture we saw the "guts" of `fgrep`:

```
n = strlen(search_string);
while (1) {
    nbytes = read(fd, buf, sizeof buf);
    for (int i = 0;  i < nbytes - n + 1;  i++) {
        if (strncmp(&buf[i], search_string, n) == 0)
            /* Print line containing search_string */
    }
}
```

and discussed the fact that the above code fails if the search string spans two I/O buffers.

We are now going to implement a working version of `fgrep` that doesn't have that major bug, and also offers a number of other features that are common in well-written Unix filters:

- "Switches" (options) on the command line can be used to modify the program's behavior in useful ways.
- Input can come from `stdin` or from a file.
- Multiple files can be processed in a single invocation.
- There is no built-in limit on the size of the file being processed, nor on the length of lines in that file.

2.1 Overview of a Unix Filter

A "proper Unix filter has a number of common characteristics:

1. By default, it reads from standard input and writes to standard output.
2. It isn't "chatty"; it does its job without progress reports.
3. One or more file names can be given on the command line, in which case it reads those files rather than `stdin`.
4. Switches (options) are introduced by a single dash (hyphen) followed by a single character, or alternately by two dashes followed by a longer name.
5. Switches can appear in any order.
6. Switches always precede file names and other arguments. Switches and arguments can't be intermixed.
7. The exit status indicates whether the filter succeeded (using a filter-specific definition of "success").
8. If the filter is invoked incorrectly, it prints a "usage" message that *briefly* summarizes the correct invocation.
9. Errors (including the usage message) are reported to `stderr`.
10. A separate manual page thoroughly documents the program. The documentation is explicityly *not* part of the program itself.

For this lab, we'll skip item 10 but implement the rest correctly.

3 Specifications

There is no skeleton code for this assignment; you will need to develop `fgrep.c` from scratch. However, we do provide a tar archive containing some simple test files and a Bash script ("`runtests`") that will run the tests shown in Subsection 3.4 and report whether the output and exit status were correct. Download and unpack the tar archive, and then do your work in the "`fgreplab`" directory that it creates.

Although our implementation will be simpler (and slower) than the version of fgrep that comes with your system, it will do its job entirely correctly within its limits.

IMPORTANT: Your output must match the output of the system `fgrep` *exactly*. We will test by mechanically comparing the two implementations. You can use `runtests` to help make sure of a match.

3.1 Invoking (Our) `fgrep`

If you type "`man fgrep`" you'll find that the modern version of fgrep has a lot of switches. We won't implement all of those. Instead, our version will be invoked as follows (assuming you call your program "`fgrep`"):

`./fgrep` [`-i`] [`-l`] [`-n`] [`-q`] *pattern* [*file file* ...]

Here, the square brackets indicate optional arguments, so the minimal invocation requires that only a *pattern* be given. The switches have the following meanings:

`-i` Ignore case when matching strings.

`-l` Instead of printing matching lines, print the name of the file containing the match.

`-n` When printing a match, prefix it with the line number.

`-q` Quiet mode: don't print any matches, and don't complain about files that can't be opened. Instead, simply return a success/failure status.

You aren't required to implement all of the switches (see Section 10 for scoring information).

If there are no input files given on the command line, `fgrep` searches the standard input for the given pattern. Otherwise, it ignores the standard input and searches each of the given files separately. **IMPORTANT NOTE:** when you see a command line like this:

```
fgrep test < test1.txt
```

it is important to realize that the shell processes and "swallows" the less-than sign and the filename that follows it. So what the program will see is only two arguments: "fgrep" and "test". The lack of an (obvious) file argument is how your filter will know that it should read from the standard input device, `stdin` (which will have already been opened for you).

If there are usage errors (either because there is no `pattern` given, or because there is an illegal switch), `fgrep` should print a usage message similar to the one above (preceded by the string "`Usage:`" and a space) and terminate without doing a search.

If file names are given but the file can't be accessed, or if an I/O error occurs (unlikely, and hard to test) then `fgrep` should print an appropriate error message but should continue its search on any other files that have been specified.

Some of the switches above conflict: `-q` makes `-l` and `-n` pointless, and `-l` makes `-n` pointless. You should not generate an error message in those cases; instead `-q` should cause `-l` and `-n` to be ignored, and `-l` should cause `-n` to be ignored. (Another way to think of it is "the switch that generates the least output wins.")

3.2 Output Format(s)

The primary purpose of `fgrep` is to print lines that contain the given pattern. However, the output format is designed to be human-friendly in the general case, which means that it varies depending on how `fgrep` was invoked. In particular:

- If there is only one input file, or if it is reading the standard input, the matched lines are printed verbatim, with no annotation (except for the `-n` switch).
- If there are multiple input files, each line is processed separately and matched lines are prefixed by the file name and a colon.
- If the `-n` switch is given, the matched lines are prefixed by the 1-indexed line number within that file. If there are multiple files, the file name comes first, followed by the line number, followed by the matched line.
- If the `-l` switch is given, `fgrep` prints only the name(s) of the file(s) containing a match. In this case the `-n` switch is ignored.
- If there are multiple matches on a line, it should only be printed once. Similarly, if the`-l` switch is given and there are multiple matches in a file, the file name should only be printed once. (A good implementation will stop the search early in these cases.)

3.3 Exit Status

If there are no errors and any matches are found anywhere, `fgrep` returns a zero (success) exit status. If there are no matches at all, if there is any error (such as an I/O error or a not-found file), or if it is invoked incorrectly, it returns nonzero (failure). An exception to this is if the `-q` switch is used: any matches will yield an exit status of zero, even in the presence of errors. Our own preference is to return 1 if there is an error or no match, and 2 if there are usage errors—but note that the "official" version uses a slightly different convention.

3.4 Sample Invocations

The contents of `test1.txt` are:

```
This is a test file.
We really like to test our tests.
It's important to have a line ending to test
Test the line beginning.
```

The file `test2.txt` contains:

```
I like spam.  But spam doesn't like me.
Our spam is very tasty.
```

The file `test3.txt` contains the following line *without a newline following*:

```
This tests whether we can match at the end when there is no newline
```

Here are some samples of running `fgrep` on those two test files (see page 6 for commentary). The command "`echo $?`" causes the shell to print the exit status of the last command as a single number on a separate line; thus we can see the success or failure status of each invocation. The shell (command-line) prompt is indicated by a number followed by a dollar sign.

```
1 $ ./fgrep test test1.txt; echo $?
This is a test file.
We really like to test our tests.
It’s important to have a line ending to test
0

2 $ ./fgrep test test2.txt; echo $?
1

3 $ ./fgrep test test1.txt test2.txt; echo $?
test1.txt:This is a test file.
test1.txt:We really like to test our tests.
test1.txt:It’s important to have a line ending to test
0

4 $ ./fgrep –i test < test1.txt; echo $?
This is a test file.
We really like to test our tests.
It’s important to have a line ending to test
Test the line beginning.
0

5 $ ./fgrep –n –i our test1.txt test2.txt; echo $?
test1.txt:2:We really like to test our tests.
test2.txt:2:Our spam is very tasty.
0

6 $ ./fgrep –q –n –i our test1.txt test2.txt; echo $?
0

7 $ ./fgrep –q –n –i spam test1.txt test2.txt; echo $?
0

8 $ ./fgrep –q –n –i chocolate test1.txt test2.txt; echo $?
1

9 $ ./fgrep –q –i spam test1.txt test2.txt missing.txt; echo $?
0

10 $ ./fgrep like test1.txt missing.txt test2.txt; echo $?
test1.txt:We really like to test our tests.
Couldn’t open ’missing.txt’: No such file or directory
test2.txt:I like spam.  But spam doesn’t like me.
1
```

```
11 $ ./fgrep -n test test1.txt; echo $?
1:This is a test file.
2:We really like to test our tests.
3:It's important to have a line ending to test
0

12 $ ./fgrep -l test < test1.txt; echo $?
(standard input)
0

13 $ ./fgrep -l -v test < test1.txt; echo $?
Usage: fgrep [-i] [-l] [-n] [-q] pattern [files]
2
```

Notes on the above:

- The first two runs show success and failure on a single file. Note that failure produces no output (the "`1`" is the exit status shown by the echo command). Also note that "Test" doesn't match, due to the capital "T".
- Run 3 shows that when there are two files, the filename is prepended to the match.
- Run 4 runs the program while feeding it a file on standard input. It also shows the `-i` (ignore case) switch.
- Run 5 shows line numbering.
- Runs 6–8 show the use of the `-q` (quiet) switch.
- Runs 9–10 show the behavior with the `-q` switch and a missing file. In this case the exit status reflects only whether the string is found, and not whether a file is missing. In addition, in run 9 `fgrep` exits with with success as soon as "spam" is found in a file. Thus, it never even discovers that `missing.txt` is nonexistent.
- Run 11 shows the `-n` switch on a single file.
- Run 12 shows the dummy "file name" that is printed if the `-l` switch is combined with standard input.
- Run 13 shows that an incorrect invocation gives a usage message.

4 Argument Processing

In the C language, the `main` program is given two arguments: `argc` and `argv`.[1] `Argc` is an integer that is equal to the number of arguments that were given on the command line; note that the command name itself is always the first argument so `argc` is always nonzero.

[1] Actually there's a third argument, `envp`, but we can get away with ignoring it in almost all programs.

The argument themselves are C-style strings in the array `argv`. Thus `argv[0]` is the name of the program (which you can verify in `gdb` by typing "p argv[0]" if you're stopped at a breakpoint in `main`). Similarly, `argv[1]` is the first argument expressed as a string, and since a string is an array, `argv[1][0]` is the first character of the first argument. That can be very useful because if that character is a dash, the argument must be a switch.[2]

As mentioned in Subsection 2.1, item 5, switches can appear in any order. Thus it is incorrect to write code similar to the following:

```
if (strcmp(argv[1], "-q") == 0) /* Process -q switch */
if (strcmp(argv[2], "-l") == 0) /* Process -l switch */
```

That way lies madness. Instead, you should loop over the arguments, using an `if/else if` construct, a `switch`[3] statement, or a combination of the two to figure out whether the argument is a switch or the pattern.

For each switch, you should have a corresponding integer variable that is zero if the switch is not present, or nonzero if it appears. Your switch-processing code can then simply set the appropriate variable to 1.

Because nearly every program needs to process arguments and switches, there have been many attempts to write library functions that can simplify the task. For C, two of the most popular are `getopt` and `getopt_long`, both of which have extensive manual pages.[4] You are welcome to investigate either or both of these, or to simply write your own argument-processing loop.

Regardless of how you approach argument processing, when you reach a non-switch argument you should assume that it is the pattern and break out of the loop. Any arguments after the pattern will be the names of files to process.

5 Handling Files

If there are no file arguments, `fgrep` should read the standard input (`stdin`), which is already open and is of type `FILE *`. But if there are file names on the command line, you will need to open each file in turn using `fopen` (which returns a `FILE *`) and search that file. Thus, it's best to have a helper function (we brilliantly called ours `fgrep`) that accepts a `FILE *` and some other arguments, and does the real work.

The best way to deal with the input file is to read it one line at a time. You can use `fgets` to do that; it allows you to read from an arbitrary `FILE *` and protects you from buffer-overflow attacks. However, you also need to handle arbitrarily long input lines, so you can't just declare (e.g.) a 1000-byte buffer and assume that all lines will be shorter than that. Instead, you should allocate a small buffer and use `fgets` to read into it. If the result doesn't end in a newline, then your buffer was too small. In that case you should expand it by **doubling** its size and then read again, appending the new data to the existing buffer. You can keep doing that until the buffer is

[2] Note that this implies that the pattern you're searching for can't begin with a dash. The standard version of `fgrep` has a solution to that problem, but we'll solve it by ignoring it.

[3] Pun!

[4] For Python, our overwhelming favorite is `docopt`. There is a C implementation of `docopt`, which would be wonderful for this lab... but as of spring 2022, it's broken.

big enough to hold the whole line, at which point you can do your search. (You may wish to break things up into helper functions.)

A few notes on this approach:

- In C, you allocate memory using `malloc`, which accepts a single integer that is the amount of memory, in bytes, that you want. `Malloc` returns a pointer to the desired memory, or `NULL` if you are out of memory.
- You can expand allocated memory with `realloc`. The expanded memory might be in a different place (so you need to update the pointer to it), but it is guaranteed to contain the data that was in the original. See the man page for the syntax.
- Once you have expanded the buffer, you should keep it at the expanded size. There is no point in freeing it and starting over.
- To make sure your expansion code works, we recommend that you start with a ridiculously tiny buffer (we chose two bytes; a single byte is too small). That will exercise your expansion code and quickly reveal any bugs.
- Be careful about memory leaks! You can free memory with `free`.

6 Useful Functions

There are a number of useful functions in the C library that will make writing `fgrep` simpler:

`strncmp(a, b, n)` Compares the strings `a` and `b`, returning 0 if they match and nonzero otherwise. The comparison stops after `n` characters, so `strncmp("ab", "abc", 2)` returns 0.

`strncasecmp(a, b, n)` Is the same as `strncmp` except that it ignores case.

`strlen(s)` Returns the length of string `s`. `Strlen` is inherently inefficient, so **DO NOT** use it inside a `for` construct; call it outside the loop and save its value.

`ferror(f)` Returns nonzero (true) if there has been an error on the `FILE * f`.

`strerror(errno)` Gives a string representation of the most recent I/O error, which is encoded in the global variable `errno` (you need to include `<errno.h>` and `<string.h>` to use it).

`malloc`, `realloc`, and `free` allocate and free memory (see Section 5).

`fopen` and `fclose` open and close files and set them up for using the standard input/output functions.

7 Header Files

We found the following C header files to be either useful or necessary in our solution:

`errno.h` Declares the global pseudo-variable `errno`, which is needed for understanding the cause of an error.

`stdio.h` Declares all the functions need to use the `stdio` (standard I/O) package, which is the standard[5] method for conveniently handling I/O in Unix.

`stdlib.h` Contains the declarations for a large number of library functions, particularly including `malloc`, `realloc`, and `free`.

`string.h` Declares all the important string functions.

You may find that you need other header files as well; if you compile with `-Wall` the compiler will give you hints on that topic.

8 Hints

- Don't try to write everything at once! Start with the argument processing. After you process the switches, put in some temporary code that prints the values of all your switch variables, and then loops through the remaining arguments, printing them with the `%s` format specifier. Once you get that debugged, you can start writing the search functions.
- Similarly, start the search function simply. Don't try to handle all the options; concentrate on finding and printing matching lines. Once you have that going, you can massage your code to add more features.
- Use `gdb`! Debuggers are amazingly useful; that's why we taught you how to use `gdb`.
- In `gdb`, use `step` and `next`, not `stepi` and `nexti`. You *really* don't want to debug this lab at the instruction level.
- As mentioned above, you should start your self-expanding buffer at a very small size; we recommend two bytes. (A one-byte buffer will cause problems with `fgets`, so don't go overboard.) Make the initial buffer size a constant by using `#define`. If necessary, step through the code with `gdb`. But be sure the use `next` to skip over `malloc` and `realloc`; you *really* don't want to wade through those functions. If you get into them accidentally, you can get out again with `finish`.
- Be sure to test as you go along!
- Once you're starting to get close, you can (and should) compare your output to the results from the "official" (system) version of `fgrep`. If you tested your program with (e.g.):

```
./fgrep -i test test1.txt; echo $?
```

[5] Pun!

then you can also run the system version by simply removing the "`./`":

```
fgrep -i test test1.txt; echo $?
```

Your output (and exit code) should match the system `fgrep` *exactly*. In fact, that's how we're going to test your implementation.

9 Testing

Here are some of the things you should test to make sure your code is working:

Short patterns. Make sure you can match single characters and short strings.

Short lines. Make sure you can match a line that is exactly the same as the pattern.

Very long lines. Make sure you can match lines that are extremely long, such as hundreds or even thousands of bytes. And make sure you don't have $O(N^2)$ behavior on those long lines.

Files with no final newline. The file "`test3.txt`" deliberately omits the final newline, which is a common violation of good Unix practice. Make sure you can find the word "newline" in it.

Line prefixes. When there are multiple files, the matched lines should be prefixed with the filename. If the "`-n`" switch is present, the line number should be given after the filename. Test with both of those switches together.

Multiple matches. When a line contains multiple matches, it should be printed only once. Similarly, with the "`-l`" switch, if a file contains multiple matches it should be listed only once.

Arbitrary switch ordering. Test with different switch orderings (e.g., both "`-i -n`" and "`-n -i`".

Switch combinations. What happens when two switches or more are combined, such as "`-i -l`"? What about when they don't make sense together, such as "`-l -n`"?

Illegal usage. Be sure to generate a usage message if a switch is illegal, or if there is no pattern.

Missing files. Handle inaccessible files correctly.

Exit status. Use "`echo $?`" (which must be the **very next** command after you run `fgrep`) to show the exit status after various tests.

Note that the above is not necessarily an exhaustive list. You should also come up with tests of your own.

10 Scoring

The lab is worth 100 points, scored as follows:

Basic functionality. 40 points for being able to find (and not find) strings in a single file, correctly handling files that do not end in a newline, and generating a correct exit status.

Switches. 5 points for each of the four switches (i.e., 20 points if you implement all four).

Standard input and multiple files. 15 points for correctly handling multiple files or no files on the command line (including correct line prefixes).

Long lines. 10 points for handling lines of arbitrary length.

Switch ordering. 5 points for accepting arbitrary switch orders (note that the library functions `getopt` and `getopt_long` do this for you—easy points!).

Missing files. 5 points for correctly handling missing files, including producing the proper exit code.

Miscellaneous. 5 points for generating correct usage messages.

11 Finishing the Lab

When you are done with the lab, use `cs105submit` to submit `fgrep.c`. When it prompts you for the assignment number, enter the lab number. Remember that you can submit more than once; we will grade only the last submission. So submit early and often!