UNIX Utilities

- **Due** Sep 16 by 11:59pm

- **Points** 120

- **Available** after Sep 5 at 12am

Small fixes since project specification was initially released:

- Error messages for my-diff should print its name "my-diff" (not "mydiff")

Overview:

In this assignment, you will build a set of small utilities that are (mostly) versions or variants of commonly-used commands. We will call your version of these utilities **my-look and my-diff** to differentiate them for the original utilities look and diff; you will also create a third utility called **across** which is more "special purpose".

Objectives:

- Re-familiarize yourself with the C programming language
- Re-familiarize yourself with a shell / terminal / command-line of UNIX
- Learn (as a side effect) how to use a proper code editor not necessarily emacs/vim
- Learn a little about how UNIX utilities are implemented

While the project focuses upon writing simple C programs, you can see from the above list of objectives that this requires a great deal of other previous knowledge, including a basic idea of what a shell is and how to use the command line on some UNIX-based systems (e.g., Linux or macOS), and how to use an editor. If you do not have these skills already, or do not know C, this is not the right place to start.

This project is to be performed alone. Projects in CS 537 cannot be turned in late.

Summary of what you will turn in:

- Set of *.c* files, one for each utility : **my-look.c**, **across.c, my-diff.c**,
- Each file should compile successfully when compiled with the -Wall and -Werror flags.
- Each should pass tests we supply as well as tests that meet our specification that we do not supply
- Include a single README.md for all the files describing your implementation
- Assume that input files for each utility can be larger than RAM on system

my-look

The program **my-look** is a simple program based on the utility **look**.

If you haven't heard of **look** before, you should do is read about it. On UNIX systems, the best way to read about utilities and functions is to use what are called the **man** pages (short for **manual**). In our HTML/web-driven world, the man pages feel a bit antiquated, but they are useful and informative and generally quite easy to use.

To access the man page for **look,** for example, just type the following at your UNIX shell prompt: `prompt> man look`

Then, read! Reading man pages effectively takes practice; why not start learning now?

Both **look** and **my-look** take a string as input and return any lines in a file that contain that string as an input; if a file isn't specified, the file **/usr/share/dict/words** is used by default. (If a file is specified, then it will have the same format and contain a subset of the lines in /usr/share/dict/words.) You'll see that **look** has a number of command line options that **my-look** should not support.

A typical usage if you would like to find a list of words that start with "gn" (because who wouldn't like to know this?) would be:

```
prompt> ./my-look gn
```

The "**./**" before the **my-look** above is a UNIX thing; it just tells the system which directory to find **my-look** in (in this case, in the "." (dot) directory, which means the current working directory). As shown, **my-look** reads the file **/usr/share/dict/words** and prints out all the lines that start with the letters "gn". Thus, the output would be

```
GNU
GNU's
Gnostic
Gnostic's
Gnosticism
Gnosticism's
gnarl
gnarled
gnarlier
gnarliest
gnarling
gnarls
gnarly
gnash
gnash's
gnashed
gnashes
gnashing
gnat
gnat's
gnats
gnaw
gnawed
gnawing
gnawn
gnaws
gneiss
gneiss's
gnome
gnome's
gnomes
gnomish
gnu
gnu's
gnus
```

To create the **my-look** binary, you'll create a single source file, **my-look.c.** To compile this program, you will do the following:

```
prompt> gcc -o my-look my-look.c -Wall -Werror
```

This will make a single *executable binary* called **my-look** which you can then run as above.

You'll need to learn how to use a few library routines from the C standard library (often called **libc**) to implement the source code for this program, which we'll assume is in a file called **my-look.c**. All C code is automatically linked with the C library, which is full of useful functions you can call to implement your program. Learn more about the C library here (Links to an external site.).

For this project, we recommend using the following routines to do file input and output: **fopen()**, **fgets()**, and **fclose()**. You should read the man pages carefully for those three functions.

We will also give a simple overview here. The **fopen()** function "opens" a file, which is a common way in UNIX systems to begin accessing a file. In this case, opening a file just gives you a pointer to a structure of type **FILE**, which can then be passed to other routines to read, write, etc.

Here is a typical usage of **fopen()**:

```
FILE *fp = fopen("main.c", "r");  if (fp == NULL) {    printf("cannot open file\n"); exit(1);  }
```

A couple of points here. First, note that **fopen()** takes two arguments: the *name* of the file and the *mode*. The latter just indicates what we plan to do with the file. In this case, because we wish to read the file, we pass "r" as the second argument. Read the man pages to see what other options are available.

Second, note the *critical* checking of whether the **fopen()** actually succeeded. This is not Java where an exception will be thrown when things goes wrong; rather, it is C, and it is expected (in good programs, i.e., the only kind you'd want to write) that you always will check if the call succeeded. Reading the man page tells you the details of what is returned when an error is encountered; in this case, the linux man page says:

```
Upon successful completion fopen(), fdopen(), and freopen() return a FILE pointer.  Otherwise, NULL is returned and the global variable errno is set to indicate the error.
```

Thus, as the code above does, check that **fopen()** does not return NULL before trying to use the FILE pointer it returns.

Third, note that when the error case occurs, the program prints a message and then exits with error status of 1. In UNIX systems, it is traditional to return 0 upon success, and non-zero upon failure. Here, we will use 1 to indicate failure.

Side note: if **fopen()** does fail, there are many reasons possible as to why. You can use the functions **perror()** or **strerror()** to print out more about *why* the error occurred; learn about those on your own (using … you guessed it … the man pages!).

Once a file is open, there are many different ways to read from it. The one we're suggesting here to you is **fgets()**, which is used to get input from files, one line at a time.

To compare the string to a line from the file, we recommend **strncmp()**. Find more details about string functions on their man page (as well information about the header file you will need to include).

To print out file contents, just use **printf()**. For example, after reading in a line with **fgets()** into a variable **buffer**, you can just print out the buffer as follows:

```
printf("%s", buffer);
```

Note that you should *not* add a newline (\n) character to the printf(), because that would be changing the output of the file to have extra newlines. Just print the exact contents of the read-in buffer (which, of course, many include a newline).

Finally, when you are done reading and printing, use **fclose()** to close the file (thus indicating you no longer need to read from it).

Details

- Your program **my-look** can be invoked with one or two arguments on the command line; the first argument is always the string to look for; if the second argument exists, it is the name of the file to use instead of /usr/share/dict/words.
- If a file is specified, you can assume that the lines it contains are a subset of those in /usr/share/dict/words (i.e., you don't have to handle arbitrarily long lines in this utility).
- If my-look has 0 or 3 or more arguments, it is an error. In this case, you should exit with status code 1 and print the exact error message "my-look: invalid number of arguments" (followed by a newline) .
- In all non-error cases, **my-look** should exit with status code 0, usually by returning a 0 from **main()** (or by calling **exit(0)**).
- If the program tries to **fopen()** a file and fails, it should print the exact message "my-look: cannot open file" (followed by a newline) and exit with status code 1.
- Note that the comparisons performed between the string and list of words are not case sensitive; that is, "gn" will match with lines containing "GNU" and and "gnu".

- Note that unlike **across** described below, you should NOT discard any words from the dictionary.

Across

The second utility you will build is called **across,** it isn't a variant of any existing utility I know of, but it has some similarities to **my-look** and **grep**. The idea of **across** is that it finds words of a specified length that contain a specified substring at a specified position; you could imagine using **across** to automatically construct the words in a cross-word puzzle.

For example, to find all the 6 letter words (again in /usr/share/dict/words) that have the letters "too" at position 1 (assuming the first letter of a word starts at position 0), one would type:

```
prompt> ./across too 1 6
```

The output would then be:

```
stooge
stools
stoops
```

Details

- Your program **across** is passed with a substring to find, the position that substring must appear in the matching word (assuming the first letter of word is at position 0), the number of letters in the matching word, and an optional word file to use instead of /usr/share/dict/words.
- You should dismiss all words in the word file that have uppercase or non-alpha characters (e.g., apostrophes or special characters with diacritics or accents).
- You may assume that an optionally specified word file will contain only a subset of the lines in /usr/share/dict/words.
- If across has an incorrect number of command line arguments, it should print “across: invalid number of arguments” (followed by a newline) and exit with status code 1.
- If **across** encounters a file that it can’t open it should print “across: cannot open file” (followed by a newline) and exit with status code 1 immediately.
- If the length of the substring and its required starting position extend past the length of the allowed matching words, it should print "across: invalid position" (with newline) and exit with status code 1.

The following are different ways **across** can be executed and example output:

```
prompt> ./across a 0 17
anesthesiologists
authoritativeness

prompt> ./across y 17 18
characteristically
disproportionately

prompt> ./across yel 0 9
yellowest
yellowing
yellowish

prompt> ./across a 5 5
across: invalid position

prompt> ./across hi 2 5
aphid
ethic
sahib

prompt> ./across hi 4 5
across: invalid position
```

my-diff

The third and final utility you will build is called **my-diff,** a much simplified variant of the **diff** utility. The idea of **diff** is that it takes two files as input and then outputs the lines that are different across the two of them. Your variant will be very simple: **my-diff** will compare line i from file A and line i from file B; if the two lines are identical, **my-diff** prints nothing; if the two lines are different, **my-diff** prints out the line number followed by the line i from file A and from file B.

There are a few complications in my-diff that you will need to address that you did not need to handle in **my-look** or **across**. In particular, you will need to be able to handle lines and files that are of an arbitrary length; this means, that you cannot simply call **fgets()** once for each line of the file, since you will not know the size of the buffer needed to fit the line. You can either look into using a different library routine such as **getline()** or call **fgets()** multiple times for each line.

Details

Line counting should start such that the first line of each file is numbered 1. The most basic case looks like the following; if file A contains

```
a
b
c
```

and file B contains

```
a
b
d
```

then

```
prompt> ./my-diff fileA fileB
```

prints the output

```
3
< c
> d
```

That is, when a difference is detected, **my-diff** prints the line number (followed by a newline), a "<" character followed by a single space and the line from file A (i.e., the first file given on the command line), and then a ">" character and a space followed by the line from file B (i.e., the second file given on the command line).

To make the output easier to read, if two (or more) consecutive lines are different in both files, **my-diff** does not print each line number. For example, if file C contains:

```
a
x
y
```

then

```
prompt> ./my-diff fileA fileC
```

prints the output

```
2
< b
```

```
> x
< c
> y
```

If one file is longer than the other, **my-diff** should detect this by printing the line number for the longer file followed by only the extra lines in that file. For example, if file D contains

```
a
b
c
d
e
```

`prompt> ./my-diff fileA fileD`

prints the output

```
4
> d
> e
```

Note that my-diff does not print "<" with empty lines for fileA (and, as in the previous example, it does not print line numbers for each consecutively different line).

Note that **my-diff** will not be able to detect if a line has been inserted (or deleted) across the two files; after seeing and printing a single "inserted" line in file A, every line in file A and file B will then be a mismatch.

For example, given a file A containing

```
hello
world
this
is
a
file
for
testing
```

and a file B

```
hello
earth
this
is
a
boring
file
for
testing
```

The output would then be:

```
2
< world
> earth
6
< file
> boring
< for
> file
< testing
> for
> testing
```

- If **my-diff** is given an incorrect number of command line arguments, it should print “my-diff: invalid number of arguments” (followed by a newline) and exit with status code 1.
- If **my-diff** encounters a file that it can’t open it should print “my-diff: cannot open file” (followed by a newline) and exit with status code 1 immediately.
- All comparisons should be case sensitive (i.e., "a" is different than "A").

Testing and Handing in your Code

- We will also be giving some points for good programming style and memory management. Read the details.
- This project is to be done on the lab machinesLinks to an external site., so you can learn more about programming in C on a typical UNIX-based platform (Linux).
- Some tests are provided at *~cs537-1/tests/p1*. Read more about the tests, including how to run them, by executing the command `cat`

`~cs537-1/tests/p1/README.md` on any lab machine. Note these test cases are not complete, and you are encouraged to create more on your own.

- **Handing it in: Copy your files to *~cs537-1/handin/login/p1* where login is your CS login.** Do NOT use this handin directory for your work space. You should keep a separate copy of your project files in your own home directory and then simply copy the relevant files to this handin directory when you are done. The permissions to this handin directory will be turned off promptly when the deadline passes and you will no longer be able to modify files in that directory. If you cannot find your handin directory, it is likely that you added the course after the first week of classes; in this case, send email to the instructor asking for a handin directory; **tell us your CS login.** Note that a special late directory called *~cs537-1/handin/login/late* exists in case something unusual goes horribly wrong and you need to share files with us after the deadline has passed; in general, the late directory should not be used and the files in there will NOT be graded.

Submission

Submission Details
Grade: 113 (120 pts possible)
Graded Anonymously: no

Comments: