P1: UNIX Utilities

Re-submit Assignment

Due Feb 11 by 11:59pm **Points** 120 **Submitting** a file upload
Available after Jan 26 at 12am

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-rev** and **my-look** to differentiate them for the original utilities rev and look.

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 such as emacs/vim
- Learn a little about how UNIX utilities are implemented
- See how pipes can be used to redirect stdout to stdin within a shell

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 course to start learning these things.

This project is to be performed alone. You can use up to three skip days throughout the semester for late projects; we recommend you do not use those skip days for this initial project.

Summary of what you will turn in:

- Set of *.c* files, one for each utility : **my-look.c**, **my-rev.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
- A screenshot showing the output of **my-rev** piped as the input of **my-look**

1. my-look

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

1.1 Overview of my-look and look

If you haven't heard of **look** before, the first thing 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 a prefix. For an example, we will assume the file has been specified as **/usr/share/dict/words.** (You may assume that all input will have the same format and contain a subset of the lines in **/usr/share/dict/words**.)

You'll see that **look** and **my-look** have different command line options; be sure to implement the command-line options specified for **my-look**.

1.2 Example Usage

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 -f /usr/share/dict/words 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
```

1.3 Compiling

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.

1.4 Suggested C Routines

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 (https://en.wikipedia.org/wiki/C_standard_library).

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 return
ed 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. (In general, 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!). However, for this project make sure you print out the error messages we request to **stdout**. We will ignore anything you print to **stderr.)**

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 **strncasecmp()**. 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 to standard output (stdout), 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, may 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).

1.5 Using stdin for Dictionary Input

For many utilities, it is useful to be able to read from standard input, stdin (i.e., characters typed directly into the terminal). Your program, when given the correct command line arguments (see below), should be able to read from stdin instead of using a file as the dictionary.

In Linux, **stdin** is defined as a FILE pointer, just like what is returned by fopen() and passed to fgets(). (You can even look at the man pages for stdin.). Your goal should be to add as little new code as possible to handle reading from stdin instead of a regular file; don't replicate code needlessly!

1.6 Command Line Arguments

my-look has one required command-line argument: the string to search for. However, **my-look** has a number of optional command-line arguments that must appear before the required string.

- -V : prints information about this utility; the message should be exactly **"my-look from CS537 Spring 2021"** followed by a newline. The utility should then exit with status code 0 without processing any more options. (Note this an upper-case V, not lower-case. Lower-case v is often used to signify verbose output.)
- -h: prints help information about this utility; you can display this information in the way you think is best. The utility should then exit with status code 0 without processing any more options.
- -f <filename>: uses <filename> as the input dictionary. For example, "my-look -f ./mywords" will read the file "./mywords" for input. **If this option is not specified, then my-look will read from standard input (i.e., stdin);** it will not open a file called stdin!!!
- If **my-look** encounters any other arguments or has any error parsing the command line arguments, you should exit with status code 1 and print the exact error message "**my-look: invalid command line**" (followed by a newline).

For parsing command-line arguments, we recommend using **getopt()**. Again, read the man pages for more information. While getopt() may be overkill for this simple utility, it is useful to know how to use when you have programs with more complex arguments.

When processing the command line, remember **argv** is an array of strings (where a string is a pointer to characters), and **argc** is the number of words on the command line. For example, if the process is run as "my-look -V -h -f ./testfile" then argc = 5, argv[0] = my-look, argv[1] = "-V", argv[2] = "-h", argv[3] = "-f" and argv[4] = "./testfile"

Details

- If **my-look** is passed both -V and -h, it will only process the option that it sees first (and then exit).
- For all input, you can assume the lines are a subset of those in /usr/share/dict/words (i.e., you don't have to handle arbitrarily long lines in this utility).
- 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 "gnu".
- All output (including error mesages) should be printed to **stdout**.

- You can safely assume the user will never search for a prefix string starting with "-". Any command-line arg beginning with "-" is considered an option.

2. my-rev

The second utility you will build is called **my-rev,** which is a variant of **rev**.

2.1 Overview of my-rev and rev

These two utilities are quite straightforward: they simply reverse each line of a file on a character-by-character basis.

That is, if the input file looks like this:

```
hello
there
world
```

The output will be:

```
olleh
ereht
dlrow
```

2.2 Compiling

To create **my-rev**, you'll create a single source file my-rev.c which you compile with:

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

2.3 Suggested C Routines

You will probably end up using many of the same routines for **my-rev** that you used for **my-look**.

2.4 Command Line Arguments

my-rev has no required command-line arguments, but it has a number of optional command-line arguments.

- -V : prints information about this utility; the message should be exactly **"my-rev from CS537 Spring 2021"** followed by a newline. The utility should then exit with status code 0 without processing any more options. (Note this an upper-case V, not lower-case. Lower-case v is often used to signify verbose output.)
- -h: prints help information about this utility; you can display this information in the way you think is best. The utility should then exit with status code 0 without processing any more options.
- -f <filename>: uses <filename> as the input dictionary. For example, "my-rev -f ./mywords" will read the file "./mywords" for input. If this option is not specified, then **my-rev** will read from standard input

(i.e., stdin).

- If **my-rev** encounters any other arguments or has any error parsing the command line arguments, you should exit with status code 1 and print the exact error message "**my-rev: invalid command line**" (followed by a newline).

Details

- If **my-rev** is passed both -V and -h, it will only process the option that it sees first (and then exit).
- For all input, you can assume the lines will not exceed a length of 100.
- In all non-error cases, **my-rev** 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-rev: cannot open file**" (followed by a newline) and exit with status code 1.
- All output (including error messages) should be printed to **stdout**.
- All testcase input will contain only valid ASCII characters. Note that **/usr/share/dict/words** contains non-ASCII characters and thus the test dictionary in **~cs537-1/tests/p1/dictionary** which we will use contains a subset of that file.

3. Piping stdout to stdin

A neat result of having one utility write output to stdout (e.g., **my-rev**) and another utility read from stdin (e.g., **my-look**), is that you can use your Linux shell to **pipe** the output of the first process as the input to a second process using the "|" character. For example, you could run the command:

```
prompt> ps -edaf | grep dusseau
```

from your shell to pipe the output from the **ps** utility to **grep** to select all the lines that have the word "dusseau" in them. Piping allows you to connect any processes together in this way; you could even pipe the output of the second process to the input of a third process.

When you implement a shell in a later project in CS 537, you'll understand more how this is easily done.

So, as a final step, use your two new utilities to look for a string in the reversed dictionary; that is, show that you can pipe the output of **my-rev** as the input to **my-look** with **/usr/share/dict/words** as the original input file. You can look for any string that you think shows that the basic functionality is working (i.e., **don't** pick a palindrome -- a word that is the same forwards and backwards).

Once you have an example from one of the machines in the instructional cluster, take a screenshot and upload that screenshot to Canvas as part of this assignment. Your screenshot should clearly show your shell command and the relevant output. Your screenshot should also show the output of running the commands **hostname** and **whoami**.

4. Testing and Handing in your Code

- We will also be giving some points for good programming style and memory management. We will be running **valgrind** and a linter on your code. Read the details.
- This project is to be done on the lab machines (https://csl.cs.wisc.edu/services/instructional-facilities), so you can learn more about programming in C on a typical UNIX-based platform (Linux).
- Some public 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 source files to *~cs537-1/handin/*login*/p1/ontime* where login is your CS login.** Do NOT use this handin directory for your workspace. 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 an email to the instructor asking for a handin directory; **tell us your CS login.**
- You can use up to 3 slip (late) days across all the projects throughout this semester; for example, you can use 1 slip day on three separate assignments or 3 slip days on a single assignment (or other combinations adding up to a total of 3 days). If you are using slip days, you must create a file called **slip_days** with the full pathname ***~cs537-1/handin/*login*/p1/slip_days.*** The file should contain a single line containing the integer number of slip days you are using; for example, you can create this file with "echo 1 > slip_days". You must then copy your code into the corresponding slip directory: ***~cs537-1/handin/*login*/p1/slip1, ~cs537-1/handin/login/p1/slip2,*** *or* ***~cs537-1/handin/login/p1/slip3***.
- **Upload your screenshot of piping to this assignment in Canvas**, as a single image file in any image format.