Unix Utilities

In this project, you'll build a few different UNIX utilities, simple versions of commonly used commands like **cat**, **ls**, etc. We'll call each of them a slightly different name to avoid confusion; for example, instead of **cat**, you'll be implementing **seucat** (i.e., "Southeast University" cat).

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
- Learn a little about how UNIX utilities are implemented

While the project focuses upon writing simple C programs, you can see from the above that even that requires a bunch 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), how to use an editor such as emacs, and of course a basic understanding of C programming. If you **do not** have these skills already, this is not the right place to start.

Summary of what gets turned in:

- A bunch of single .c files for each of the utilities below: **seucat.c**, **seugrep.c**, **seuzip.c**, and **seuunzip.c**.
- Each should compile successfully when compiled with the **-Wall** and **-Werror** flags.
- Each should (hopefully) pass the tests we supply to you.

seucat

The program **seucat** is a simple program. Generally, it reads a file as specified by the user and prints its contents. A typical usage is as follows, in which the user wants to see the contents of main.c, and thus types:

```
prompt> ./seucat seucat.c
#include <stdio.h>
...
```

As shown, **seucat** reads the file **seucat.c** and prints out its contents. The "**./**" before the **seucat** above is a UNIX thing; it just tells the system which directory to find **seucat** in (in this case, in the "." (dot) directory, which means the current working directory).

To create the **seucat** binary, you'll be creating a single source file, **seucat.c**, and writing a little C code to implement this simplified version of **cat**. To compile this program, you will do the following:

```
prompt> gcc -o seucat seucat.c -Wall -Werror
prompt>
```

This will make a single *executable binary* called **seucat** 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 **seucat.c**. All C code is automatically linked with the C library, which is full of useful functions you can call to implement your program.

For this project, we recommend using the following routines to do file input and output: **fopen()**, **fgets()**, and **fclose()**, etc. Whenever you use a new function like this, the first thing you should do is read about it -- how else will you learn to use it properly?

On UNIX systems, the best way to read about such 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 **fopen()**, for example, just type the following at your UNIX shell prompt:

```
prompt> man fopen
```

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

We will also give a simple overview here. The **fopen()** function "opens" a file, which is a common way in UNIX systems to begin the process of file access. In this case, opening a file just gives you back 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 macOS man page says:

```
Upon successful completion fopen(), fdopen(), freopen() and fmemopen() 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, please 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 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 **seucat** can be invoked with one or more files on the command line; it should just print out each file in turn.
- In all non-error cases, **seucat** should exit with status code 0, usually by returning a 0 from **main()** (or by calling **exit(0)**).
- If *no files* are specified on the command line, **seucat** should just exit and return 0. Note that this is slightly

different than the behavior of normal UNIX **cat** (if you'd like to, figure out the difference).
- If the program tries to **fopen()** a file and fails, it should print the exact message "cannot open file" (followed by a newline) and exit with status code 1. If multiple files are specified on the command line, the files should be printed out in order until the end of the file list is reached or an error opening a file is reached (at which point the error message is printed and **seucat** exits).

seugrep

The second utility you will build is called **seugrep**, a variant of the UNIX tool **grep**. This tool looks through a file, line by line, trying to find a user-specified search term in the line. If a line has the word within it, the line is printed out, otherwise it is not.

Here is how a user would look for the term **foo** in the file **bar.txt**:

```
prompt> ./seugrep foo bar.txt
this line has foo in it
so does this foolish line; do you see where?
even this line, which has barfood in it, will be printed.
```

Details

- Your program **seugrep** is always passed a search term and zero or more files to grep through (thus, more than one is possible). It should go through each line and see if the search term is in it; if so, the line should be printed, and if not, the line should be skipped.
- The matching is case sensitive. Thus, if searching for **foo**, lines with **Foo** will *not* match.
- Lines can be arbitrarily long (that is, you may see many many characters before you encounter a newline character, \n). **seugrep** should work as expected even with very long lines. For this, you might want to look into the **getline()** library call (instead of **fgets()**), or roll your own.
- If **seugrep** is passed no command-line arguments, it should print "searchterm [file ...]" (followed by a newline) and exit with status 1.
- If **seugrep** encounters a file that it cannot open, it should print "cannot open file" (followed by a newline) and exit with status 1.
- In all other cases, **seugrep** should exit with return code 0.
- If a search term, but no file, is specified, **seugrep** should work, but instead of reading from a file, **seugrep** should read from *standard input*. Doing so is easy, because the file stream **stdin** is already open; you can use **fgets()** (or similar routines) to read from it.
- For simplicity, if passed the empty string as a search string, **seugrep** can either match NO lines or match ALL lines, both are acceptable.

seuzip and seuunzip

The next tools you will build come in a pair, because one (**seuzip**) is a file compression tool, and the other (**seuunzip**) is a file decompression tool.

The type of compression used here is a simple form of compression called *run-length encoding* (*RLE*). RLE is quite simple: when you encounter **n** characters of the same type in a row, the compression tool (**seuzip**) will turn that into the number **n** and a single instance of the character.

Thus, if we had a file with the following contents:

```
aaaaaaaaaabbbb
```

the tool would turn it (logically) into:

```
10a4b
```

However, the exact format of the compressed file is quite important; here, you will write out a 4-byte integer in binary format followed by the single character in ASCII. Thus, a compressed file will consist of some number of 5-byte entries, each of which is comprised of a 4-byte integer (the run length) and the single character.

To write out an integer in binary format (not ASCII), you should use **fwrite()**. Read the man page for more details. For **seuzip**, all output should be written to standard output (the **stdout** file stream, which, as with **stdin**, is already open when the program starts running).

Note that typical usage of the **seuzip** tool would thus use shell redirection in order to write the compressed output to a file. For example, to compress the file **file.txt** into a (hopefully smaller) **file.z**, you would type:

```
prompt> ./seuzip file.txt > file.z
```

The "greater than" sign is a UNIX shell redirection; in this case, it ensures that the output from **seuzip** is written to the file **file.z** (instead of being printed to the screen). You'll learn more about how this works a little later in the course.

The **seuunzip** tool simply does the reverse of the **seuzip** tool, taking in a compressed file and writing (to standard output again) the uncompressed results. For example, to see the contents of **file.txt**, you would type:

```
prompt> ./seuunzip file.z
```

seuunzip should read in the compressed file (likely using **fread()**) and print out the uncompressed output to standard output using **printf()**.

Details

- Correct invocation should pass one or more files via the command line to the program; if no files are specified, the program should exit with return code 1 and print "seuzip: file1 [file2 ...]" (followed by a newline) or "seuunzip: file1 [file2 ...]" (followed by a newline) for **seuzip** and **seuunzip** respectively.
- The format of the compressed file must match the description above exactly (a 4-byte integer followed by a character for each run).
- Do note that if multiple files are passed to **seuzip**, they are compressed into a single compressed output, and when unzipped, will turn into a single uncompressed stream of text (thus, the information that multiple files were originally input into **seuzip** is lost). The same thing holds for **seuunzip**.