Project #1 – Reverse & Unix Utilities

Part 1a: Reverse

This assignment was taken from the great textbook Operating Systems: Three Easy Pieces (OSTEP); written by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau from UW-Madison. The online version is completely free and contains many open source assignments that are on GitHub; We are not using all of them for this course so please check out the other ones for your own benefit!

1. Introduction

Before beginning: Read this lab tutorial; it has some useful tips for programming in the C environment.

This project is a simple warm-up to get you used to how this whole project thing will go. It also serves to get you into the mindset of a C programmer, something you will become quite familiar with over the next few months. Good luck!

You will write a simple program called "reverse". This program should be invoked in one of the following ways:

```
prompt> ./reverse
prompt> ./reverse input.txt
prompt> ./reverse input.txt output.txt
```

The above line means the users typed in the name of the reversing program reverse (the ./ in front of it simply refers to the current working directory (called dot, referred to as .) and the slash (/) is a separator; thus, in this directory, look for a program named reverse) and gave it either no command-line arguments, one command-line argument (an input file, input.txt), or two command-line arguments (an input file and an output file output.txt).

An input file might look like this:

```
hello
this
is
a file
```

The goal of the reversing program is to read in the data from the specified input file and reverse it; thus, the lines should be printed out in the reverse order of the input stream. Thus, for the aforementioned example, the output should be:

```
a file
is
this
hello
```

The different ways to invoke the file (as above) all correspond to slightly different ways of using this simple new Unix utility. For example, when invoked with two command-line arguments, the program should read from the input file the user supplies and write the reversed version of said file to the output file the user supplies.

When invoked with just one command-line argument, the user supplies the input file, but the file should be printed to the screen. In Unix-based systems, printing to the screen is the same as writing to a special file known as standard output, or stdout for short.

Finally, when invoked without any arguments, your reversing program should read from standard input (stdin), which is the input that a user types in, and write to standard output (i.e., the screen).

Sounds easy, right? It should. But there are a few details…

2: Details

Assumptions and Errors

- **Input is the same as output**: If the input file and output file are the same. file, you should print out an error message "*Input and output file must differ*" and exit with return code 1.

- **String length**: You may not assume anything about how long a line should be. Thus, you may have to read in a very long input line.

- **File length**: You may not assume anything about the length of the file, i.e., it may be VERY long.

- **Invalid files**: If the user specifies an input file or output file, and for some reason, when you try to open said file (e.g., input.txt) and fail, you should print out the following exact error message: "*reverse: cannot open file 'input.txt'*" and then exit with return code 1 (i.e., call exit(1);).

- **Malloc fails**: If you call malloc() to allocate some memory, and malloc fails, you should print the error message malloc failed and exit with return code 1.

- **Too many arguments passed to program**: If the user runs reverse with too many arguments, print "*usage: reverse <input> <output>*" and exit with return code 1.

- **How to print error messages**: On any error, you should print the error to the screen using fprintf(), and send the error message to stderr (standard error) and not stdout (standard output). This is accomplished in your C code as follows:

```
fprintf(stderr, "whatever the error message is\n");
```

Useful Routines

- To exit, call exit(1). The number you pass to exit(), in this case 1, is then available to the user to see if the program returned an error (i.e., return a non-zero) or exited cleanly (i.e., returned 0).

- For reading in the input file, the following routines will make your life easy: fopen(), getline(), and fclose().

- For printing (to screen, or to a file), use fprintf(). Note that it is easy to write to standard output by passing stdout to fprintf(); it is also easy to write to a file by passing in the FILE * returned by fopen, e.g.,

```
fp=fopen(...);
fprintf(fp, ...);
```

- The routine malloc() is useful for memory allocation.

- If you don't know how to use these functions, use the man pages. For example, typing man malloc at the command line will give you a lot of information on malloc.

3. Tips

- Start small, and get things working incrementally. For example, first get a program that simply reads in the input file, one line at a time, and prints out what it reads in. Then, slowly add features and test them as you go.

- For example, the way we wrote this code was first to write some code that used fopen(), getline(), and fclose() to read an input file and print it out. Then, we wrote code to store each input line into a linked list and made sure that worked. Then, we printed out the list in reverse order. Then we made sure to handle error cases. And so forth…

- Testing is critical. A great programmer we once knew said you have to write five to ten lines of test code for every line of code you produce; testing your code to make sure it works is crucial. Write tests to see if your code handles all the cases you think it should. Be as comprehensive as you can be. Of course, when grading your projects, we will be. Thus, it is better if you find your bugs first, before we do.

- Keep old versions around. Keep copies of older versions of your program around, as you may introduce bugs and not be able to easily undo them. A simple way to do this is to keep copies around, by explicitly making copies of the file at various points during development. For example, let's say you get a simple version of reverse.c working (say, that just reads in the file);

    - type cp reverse.c reverse.v1.c to make a copy into the file
    - reverse.v1.c. More sophisticated developers use version control systems git (perhaps through github); such a tool is well worth learning, so do it!

- Use the Makefile to compile reverse.c and use test-reverse.sh to run the testcases.

Part 1b: Unix Utilities

1. Introduction

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 wcat.

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: wcat.c, wgrep.c, wzip.c, and wunzip.c.
- Each should compile successfully when compiled with the -Wall and -Werror flags.
- You can also use the Makefile to compile the files and use *test-***.sh* to run the testcases.
- Each should pass the tests we supply to you.

2. wcat

The program wcat 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> ./wcat main.c
#include <stdio.h>
...
```

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

To create the wcat binary, you'll be creating a single source file, wcat.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 wcat wcat.c -Wall -Werror
prompt>
```

This will make a single executable binary called wcat 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 wcat.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.

For this project, we recommend using the following routines to do file input and output: fopen(), fgets(), and fclose(). 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 then 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 wcat 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, wcat 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, wcat 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 "wcat: 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 wcat exits).

2. wgrep

The second utility you will build is called wgrep, 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> ./wgrep 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 wgrep 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). wgrep 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 wgrep is passed no command-line arguments, it should print "*wgrep: searchterm [file …]*" (followed by a newline) and exit with status 1.

- If wgrep encounters a file that it cannot open, it should print "*wgrep: cannot open file*" (followed by a newline) and exit with status 1.

- In all other cases, wgrep should exit with return code 0.

- If a search term, but no file, is specified, wgrep should work, but instead of reading from a file, wgrep 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, wgrep can either match NO lines or match ALL lines, both are acceptable.

4. wzip and wunzip

The next tools you will build come in a pair, because one (wzip) is a file compression tool, and the other (wunzip) 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 (wzip) 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 wzip, 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 wzip 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> ./wzip file.txt > file.z
```

The "greater than" sign is a UNIX shell redirection; in this case, it ensures that the output from wzip 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 wunzip tool simply does the reverse of the wzip 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> ./wunzip file.z
```

wunzip 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 "*wzip: file1 [file2 …]*" (followed by a newline) or "*wunzip: file1 [file2 …]*" (followed by a newline) for wzip and wunzip 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 *wzip, 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 wzip is lost). The same thing holds for wunzip.

5. Running Tests

- Use the *Makefile* to compile.

- The 'tests' directory contains the test files.
  - #.desc - Description of the test
  - #.run - command executed to run the test
  - #.in - input for the test
  - #.out - expected output
  - #.rc - expected return code
  - #.other - other files maybe used by the test.

- For running tests, run the corresponding test script. For example:

```
prompt> ./test-reverse.sh
prompt> ./test-wzip.sh
```

If you implemented things correctly, you should get some notification that the tests passed. Cheers.!

Appendix – Test options

The run-tests.sh script is called by various testers to do the work of testing. Each test is actually fairly simple: it is a comparison of standard output and standard error, as per the program specification.

In any given program specification directory, there exists a specific tests/ directory which holds the expected return code, standard output, and standard error in files called n.rc, n.out, and n.err (respectively) for each test n. The testing framework just starts at 1 and keeps incrementing tests until it can't find any more or encounters a failure. Thus, adding new tests is easy; just add the relevant files to the tests directory at the lowest available number.

The files needed to describe a test number n are:

- n.rc: The return code the program should return (usually 0 or 1)
- n.out: The standard output expected from the test
- n.err: The standard error expected from the test
- n.run: How to run the test (which arguments it needs, etc.)
- n.desc: A short text description of the test
- n.pre (optional): Code to run before the test, to set something up
- n.post (optional): Code to run after the test, to clean something up

There is also a single file called pre which gets run once at the beginning of testing; this is often used to do a more complex build of a code base, for example. To prevent repeated time-wasting pre-test activity, suppress this with the -s flag (as described below).

In most cases, a wrapper script is used to call ***run-tests.sh*** to do the necessary work.

The options for run-tests.sh include:

- -h (the help message)
- -v (verbose: print what each test is doing)
- -t n (run only test n)
- -c (continue even after a test fails)
- -d (run tests not from tests/ directory but from this directory instead)
- -s (suppress running the one-time set of commands in pre file)