CSCI-442: Project 1 - Warm Up

Due: Please see the assignment on Canvas for dates.

> Important
>
> - You'll want to read this **entire document** before beginning the project. Please ask any questions you have on Piazza, but only if this README does not answer your question.
> - Finally, be sure to start early. If you wait until a few days before the due date, you are unlikely to finish in time.

Introduction

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!

Deliverable 1

Your first task is to familiarize yourself with the development environment. To provide a standardized development environment, you will be using a Docker container. Follow the instructions here to setup Docker and your environment. We are using Docker so your code can run in an environment similar to the autograder regardless of what your native platform is.

If you have successfully completed the above linked tutorial, you should have access to this starter code inside container.

Run the command

```
make && ./reverse
```

Take a screenshot of your terminal, including the result of the above command. This is your submission for deliverable 1. It should say:

```
TODO: write a main function
```

Deliverable 2

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

```
prompt> ./reverse input_file
prompt> ./reverse input_file output_file
```

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.

- 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.

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

Assumptions

- **String length**: You may NOT assume *anything* about how long a line can be. I.e., it may be VERY long, or quite short.
- **File length**: You may NOT assume *anything* about how large a file is. Thus, you may have to read in a very large input file...

Errors You Need to Handle

On any of the errors below, you should print the error message to `stderr` (standard error), **NOT** `stdout` (standard output). You can accomplish this using `fprintf()`.

- **Input file is the same as the output file**: Print the error message `error: input and output file must differ` and exit with return code 1.
- **Invalid files**: If the user specifies an input or output file, and for some reason, when you try to open said file (e.g., `input.txt`) and fail, print the error message `error: cannot open file 'input.txt'` (or whatever the file is named) and exit with return code 1.
- **System call fails**: All system calls can fail. For grading purposes, you can print any error message, but you must exit with a return code 1.
    - Ex: If `malloc()` fails, you can print any error message like `error: malloc failed` (or whatever the system call is named) and exit with return code 1.
- **Incorrect number of arguments passed to program**: Should too many, or too few, arguments be provided to `reverse`, then print the error message `usage: reverse <input> <output>` and exit with return code 1.
- **Memory Safety**: If your program is susceptible to buffer-overflow based on certain inputs, it is not memory safe.
    - You will lose points for using these memory unsafe functions: `strcat`, `strcpy`, `strcmp`, or `sprintf`.

- Ex: If you try to use `getline()` and there is no space for the line you are trying to grab, you can print any useful message like `error: memory safety`, but you must exit with return code 1.

Useful Routines

You may find the following manuals useful:

- `fopen(3)`
- `getline(3)`
- `fclose(3)`
- `malloc(3)` and `free(3)`
- `fprintf(3)`
- `exit(3)`

You can open these using the `man` command. For example:

```
prompt> man 3 fopen
```

Tips

1: 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. Compile your code with the command `make` to test your work and check for memory safety. Then, slowly add features and test them as you go.

Here are our recommended steps:

1. First, write some code using `fopen()`, `getline()`, and `fclose()` to read an input file and print it out.
2. Design how you will store the lines to be easily reversed later. A proper data structure makes all the difference here. Think back to 262: would a map or set even be useful here? What about a stack or a queue? How might you implement one?

    - Don't pass by this too quickly. "A week of coding can save you an hour of planning" as the saying goes.
    - But also don't over-engineer this. No need to have a perfect circular array or balanced binary tree. A linked list might be perfect in this scenario: quick to implement, and can easily be used to solve the problem at hand (*hint hint*).
3. Write the code to store each input line into your data structure, and make sure that works.
4. Use your data structure to print the lines in reverse order of the input file.
5. Handle error cases, and so forth...

2: 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.

We have provided some *basic* tests for you to check against, you can run them via:

```
./test-reverse.sh -v
```

However, **YOU WILL NEED TO TEST YOUR CODE ON MORE THAN JUST THE PROVIDED TESTS**.

- The provided tests are *not* comprehensive. All they do is check for basic error handling, and run the provided example.
- We will have no sympathy for students who only use the provided tests and then receive a poor score.

> Warning
>
> We will be using `diff` to verify your program produces the correct output. This means you will *not* get partial credit on a within-test basis (i.e., you will either pass, or fail, each individual test. There is no in-between)
>
> This means if your program produces an even slightly incorrect output (e.g., missing the last character of the last line, produces extraneous output, throws an error, etc.), **you will get a 0 on that specific test**. Take extra care with testing to ensure that a minor error doesn't propogate and cause your program to fail all test cases.

3: Keep old versions around

Keep copies of old versions of your program around, as you may introduce bugs and not be able to easily undo them.

- Use **git** for this. This project is already a Git repository, so take advantage of all the version control features git provides!

General Requirements

- Your program should have a zero exit status if no errors are encountered.
- Your project must be written in the C programming language, and execute on Isengard.
- You should follow Linux Kernel coding style, a common style guide for open-source C projects.
- Your project must not execute external programs or use network resources.
- You should `free` any memory that you heap-allocate, and close any files that you open. If you do not, the flag `-fsanitize=address` will catch a memory leak which will be shown either in your console or in the *.err file.
- To compile your code, the grader should be able to `cd` into the root directory of your repository and run `make` using the provided `Makefile`.

Your grade will be negatively impacted if you do not heed these requirements.

Collaboration Policy

Please see the syllabus for the course plagarism policies.

This is an **individual project**. Plagarism cases will be punished harshly according to school policies.

Please do keep any Git repos private, even after you finish this course. This will keep the project fun for future students!

Submitting Your Project

Submission of your project will be handled via **Gradescope**.

1. Create the submission file using the provided `make-submission` script:

```
prompt> ./make-submission
```

2. This will create a `.zip` file named `$USER-submission` (e.g., for me, this would be named `lhenke-submission.zip`).
3. Submit this `.zip` file to Gradescope. You will get a confirmation email if you did this correctly.

> Warning
>
> You are **REQUIRED** to use `make-submission` to form the `.zip` file. Failure to do so may cause your program to not compile on Gradescope.