Assignment 1

Submit Assignment

Due Jun 15 by 11:59pm **Points** 100
Submitting a text entry box or a file upload **Available** until Jun 30 at 11:59pm

Objectives

- Re-familiarize yourself with the C programming language
- Re-familiarize yourself with the shell / terminal / command-line of UNIX
- Learn (as a side effect) how to use a proper code editor, such as cLion (or emacs or vi or...)
- 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 a code editor such as cLion or emacs or vi, and of course a basic understanding of C programming.

Before beginning: If you need a refresher on the Unix/C environment, read **this tutorial (http://pages.cs.wisc.edu/~remzi/OSTEP/lab-tutorial.pdf)** .

To start cLion you can type cLion in the VM that you installed.

Summary of what gets turned in:

- A single .c file **avg.c**.
- It should compile successfully in cLion on the VM given.
- Each should (hopefully) pass the tests we supply to you.

avg

We are going to write a utility that is missing from linux: **avg**. To create the **avg** binary, you'll be creating a single source file, **avg.c**, which will involve writing C code to implement it.

Sometimes you have a file full of numbers and you want to average them all up. You could

write a complicated function with awk, but the syntax is hard to remember. Instead we want to write a program that does that for us.

For this assignment assume that you have a text file with numbers separated by newlines. You want to read the numbers in and add them up ; then divide by the line count to get the average. Since we want to be flexible on the type and size of numbers we can handle, we will use double floating point.

You must be able to handle a file with any number of numbers (including no numbers). You can assume that the files are well formed: it will contain only valid numbers separated by newlines.

For example, **numbers.txt** contains a list of numbers. Your program takes the filename as a command-line parameter and averages the numbers, then prints the average.

```
prompt> ./avg numbers.txt
```

which averages the numbers to 10.5/5=2.1 and will output

```
2.100000
```

(trailing zeros are ok.)

The "**./**" before the **avg** above is a UNIX thing; it just tells the system which directory to find **avg** in (in this case, in the "." (dot) directory, which means the current working directory).

To compile this program outside of cLion, you can also do the following:

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

This will make a single *executable binary* called **avg**, which you can then run as above.

grading

compiles cleanly	10%
overall requirements of assignment are satisfied	20%
output correctly for given test case	10%

outputs correctly for tester's test cases	20%
program flow is understandable	10%
number of numbers that can be processed is not limited	10%
code is commented and indented (use of whitespace)	10%
submission of **avg.c** inside a zip file called **proj1.zip**	10%

submission

Upload a zip file called proj1.zip that contains a single file: avg.c

details

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 **avg.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)** and perhaps **here (https://www-s.acm.illinois.edu/webmonkeys/book/c_guide/)**.

For this project, we recommend using the following routines to do file input and output: **fopen()**, **fgets()**, **fscanf()**, 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("numbers.txt", "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. We suggest **fscanf()** or **fgets()**, which is used to get input from files, one line at a time.

To print out file contents, you may 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).

- In all non-error cases, **avg** 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, or an empty file with no numbers is given, **avg** should just exit and return 0.
- If the program tries to **fopen()** a file and fails, it should print the exact message "avg: cannot open file" (followed by a newline) and exit with status code 1.