CS 132 Coursework 2 Question 2

Zain Mobarik - 2109306

January 2022

1 Preface

After some initial research, I was looking for ways in which C can detect what is happening in the command-line. One way was using the in-built arg parameters in C which tracked the user inputs and the number of inputs and another method was using a switch case block to read where the user wants to navigate to. I decided to use the arg parameters in my approach as I wanted to learn more about them and thought they made more logical sense to use. [3]

2 Manual

a **create {name}:** creates new file

b **copy {name} {copyFileName}:** copies a file's data and stores it in a new file under a specified name

c **delete {name}:** deletes a file

d **show {name}:** displays the contents of a file

e **lineCount {name}:** counts the number of lines in a file

f **add {name}:** adds lines to the end of a file

g **delete {name} {line number}:** deletes the line number specified in a given file if it exists

h **insert {name} {line number}:** inserts a line of text at the line number specified in a given file

i **show {name} {line number}:** displays a line of text at the line number specified in a given file

j **log:** displays log history of all file manipulations

3 Additional Features

a **update {name} {line number}:** updates a line of text at the line number specified in a given file

This was added because the user might want to change a single line in a file rather than insert new data in that line and shift all the other data down or append to the end of a file. This might come in handy if there is a spelling mistake or simple error on only one line in a file.

b **find {name} {word to find}:** returns the number of times a given word appears in a file

This was added due to the sheer usefulness of instantly knowing how many times a word appears in a file if it does at all. This simplifies and automates a laborious, manual process that is quite commonly used.

c **clear {name}:** deletes all the contents of a specified file

This is almost like a clear history functionality. It was originally intended to be used to clear the log.txt file which held the history of all operations. However, I thought it would be useful to generalise it to clear any file. This was even useful in my own testing. When I was trying to debug my "insert" function and it inserted text weirdly, instead of going to the file and deleting the content and trying again, I just called this function. This is also saves time as the user doesn't have to delete a file then re-create it with the same name to clear it.

4 Execution of Operations- args

As mentioned in my "Preface", I decided to use the C's in-built command line arguments. To execute a compiled file in C, the user must type the path to the file and the name. If you are in the directory in which the file is stored then you just type the name. For example, my code was written in a file called "CW2p2". So to run this code, type "./CW2p2" in the command line.

```
int main(int argc, char* argv[]) {

    printf("type './CW2p2 -h' into the terminal to display command line syntax\n\n");

    if (argc == 2) {
        if (strcmp(argv[1], "-h")==0) {
            help();
        }
        if (strcmp(argv[1], "log")==0) {
            viewLog();
        }
    }
```

As you can see, if the user simply runs the file, a message will be printed telling the user to type "-h" after the command that runs the code to display a menu.

"argc" is an integer parameter which stores the number of command line arguments. What is paramount to remember is that the command to execute the compiled code ("./CW2p2") counts as one command line argument.

"argv[]" is a pointer which points to a character array of all the command line arguments. This is incredibly useful as it means I can index this character array pointer to dereference it and access each of the command line arguments. "argv[0]" will always be "./CW2p2" when running my code.

As illustrated in the code snippet above, I count the number of arguments given and in accordance with the menu I have created, I call certain functions. Only two operations in my menu have an "argc" value of 2; "-h" and "log". Hence, they are in this "IF" block. I then use the "strcmp" function to compare the second command line argument with "-h" and "log". If one of them equals "0", then the comparison was successful and I call the relevant functions.

I then extrapolate this process for when the command line has three and four arguments.

5 Specific Code Highlights

a **Error Checking**

```
if (argc == 3) {
    char *ending = strrchr(argv[2], '.');
    if ((ending == NULL) || (strcmp(ending, ".txt") != 0)) {
        printf("Error: Filename must include the '.txt' extension.\n\n");
    }
    else {
        if (strcmp(argv[2], "log.txt") == 0 && strcmp(argv[1], "clear") != 0) {
            printf("This file name is reserved. Please use another one.\n");
        }
        else {
            if (strcmp(argv[1],"create")==0) {
                create(argv[2]);
            }
            ...
        }
    }
}
```

In the code snippet above I make two vital error checks. This block of code is for when the user has written three command line prompts. According

to my menu, this means the user is now manipulating files. Hence, my first check is to see if the file name entered is formatted correctly. I create a character pointer called "ending" that stores all the characters after and including a certain character in a string. This is achieved by the "strrchr" function that takes a string and a character. So if I do strrchr("TopTaylorSwiftSongs.txt", '.'), then I will get ".txt" returned. I then use an IF block with a boolean expression checking if the contents of "ending" is "NULL" or not ".txt" in which case the file name has been incorrectly formatted and the program quits. [4]

The second check I make is to do with the master "log.txt" file. This is the file that contains the history of all the operations and so I do not want a user to manipulate it. The only operation I allow to happen on it is to clear the log. Hence I have a quick check to make sure that the file name given is not "log.txt" and the operation is not "clear". If the operation is "clear" then it is fine to invoke "log.txt".

```
if (access(name, F_OK) ==0) {
    printf("File '%s' already exists.\n\n", name);
    return;
}
```

I also have simple checks in my "create" and "copy" functions to make sure that the new file names don't already exist because then the original file will be overwritten.

b **Add Function**

Off the bat, I decided it would be of benefit and more of a useful feature if the user could append multiple lines in one go to the end of a file.

```
while (add) {
    fgets(buffer, 1000, stdin);
    if (strcmp(buffer, "quit\n") == 0) {
        add=false;
        break;
    }
    fputs(buffer, fp);
}
```

In this snipped of the "add" function I show how I allowed the user to add multiple lines to the end of a file. I get a max of 1000 characters from the standard input and store it in my buffer. If at any point the contents of the buffer equals "quit\n" then stop appending the data into the file and break out the While loop. If, there was no While loop or that If condition, the user would only be able to input one line.

c **Delete Line Function**

```
do {
```

```
    fgets(buffer, 1000, fp);
    if (feof(fp)) {
        read = false;
    }
    else if (lineCount != lineNum) {
        fputs(buffer, tempFp);
    }
    lineCount++;
} while (read);
```

For the workings of the Delete Line function, I have used a Do While loop. I also utilise a temporary file and by extension a temporary file pointer. The idea is to copy all the lines from the original file into the temporary file except from the line the user wants to delete. Originally, I get up to 1000 characters from a line from the original file. I then check if the file has ended, in which case I exit the loop. This means the line number the user entered exceeded the number of lines in the file. If the file has not ended then if the line count does not equal the line number to delete, then we want to keep that line and we put it in the new temporary file. Finally, I increment the line count to update it so the program can keep track of what line of the file it is on. The original file is deleted and the temp file is renamed to what the original file was called.

d **Insert Function**

```
do {
    fgets(buffer, 1000, fp);
    if (feof(fp)) {
        printf("%d", lineCount);
        read = false;
    }
    if (lineCount == toIns && add == true) {
        addIns(tempFile);
        tempInsFp = fopen(tempFile, "a");
        fputs(buffer, tempInsFp);
        added = true;
        add = false;
    }
    else if (lineCount != toIns) {
        fputs(buffer, tempInsFp);
    }
    lineCount++;
} while (read);
```

For the Insert function, I also implement a temporary file. The idea behind this is to copy all the text before the line number where the user wants to insert text into inside the temp file. Then, when we reach the line number where text is to be inserted, I simply invoke the "add" function to add as

much text as the user wants until they write "quit\n". Then I copy the rest of the text from the original file into the temp file. The original file is deleted and the temp file is renamed to what the original file was called.

e **Update Function**

The "update" function is very similar to the "insert" except when the line count equals the line to update, I don't include an "fputs" statement in that IF block. This means that at that line number, the data from the original file will not be copied over to the temp file and will basically be overwritten by whatever the user adds.

f **Find Function**

```
while (!feof(fp)) {
    char c = fgetc(fp);
    int index = 0;
    while (c == wordToFind[index] && index<len) {
        word[index] = c;
        c = fgetc(fp);
        index++;
        tally++;
    }
    word[len] = '\0';
}
tally = tally / len;
```

The "find" function was quite fun to implement and works on a character by character basis. While the character retrieved equals the first character of the word the user wants to find, the while loop is entered. I then create a temporary "word" placeholder that is a character array of all the character matches between the character retrieved and the word to find at a given index. Then the while loop repeats and checks if the next character is equivalent to the second character of the word to find. If this is true, the temp word will then have this character added to it. Then, once the while loop is completed, I add the null terminator to the end of "word". I also divide tally by the length of the word to find. This is because I add 1 to "tally" every time there is a character match, so I divide it by the number of characters at the end.

I got the idea from reading this article [1]

6 Final Remarks

One recurring limit I had in my code was with the use of "fgets". I set the limit of my fgets call to 1000 as that should be big enough to get text from the majority of files. However, this is still a limit which could hypothetically be

surpassed by a given file. Furthermore, if a given line has very few words, then a lot of memory is wasted due to this generous buffer size.

I then researched the "getLine" function in C. This allocates the exact amount of memory needed for a line as it has its own in-built malloc and realloc functions. However, upon further research, I found out that getLine does not work on some Operating Systems; more specifically, it only works on POSIX which Windows is not a part of. [2]

Thus, I opted for fgets as it is universal in C unlike getLine even though it is not the most memory efficient.

References

[1] GeeksForGeeks. C program to find and replace a word in a file by another given word. 2021.

[2] RIP Tutorial. Get lines from a file using getline(). 2019.

[3] TutorialsPoint. C - command line arguments. 2020.

[4] TutorialsPoint. C library function - strrchr(). 2021.