# Question 1:

## Introduction:

This task lends itself to an iterative approach, as this is both intuitive to design, and understand once implemented. The issue with this type of approach is that for large sets of data it can become inefficient. This issue is more prevalent in this task as the dataset grows at an exponential rate as higher values of n are inputted.

There are multiple steps to creating the solution to this task that need to be solved which are listed below:

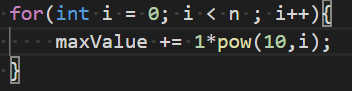
1. Iterating to the correct number of values.
2. Representing numbers in a way that allows individual digits to be checked.
3. Determining which numbers contain a “0” digit and omitting these from being checked.
4. Checking all numbers whether digits add to the value of n.
5. Returning the sum of these digits.

## Solution:

1. Iterating to the correct number of values.

The largest value who’s digits add to the value of n will be the one with the most digits, as this value has the highest power of 10. This value will always consist of 1s in order to contain the most digits. As a result the following pattern is produced which we can utilise in code:

|  |  |
| --- | --- |
| N | Maximum value |
| 1 | 1 |
| 2 | 11 |
| 3 | 111 |
| 4 | 1111 |
| 5 | 11111 |

As a result of this the following code is able to generate the maximum value which we should iterate to:

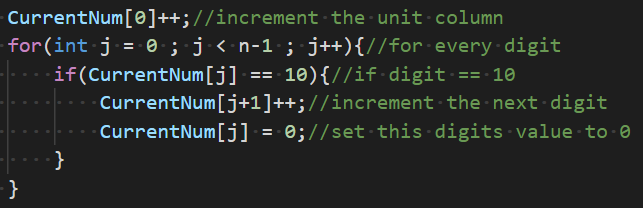
This for loop adds a new digit containing the value 1 until there are n digits present by adding successive powers of ten to the variable maxValue. This means we can now iterate over all numbers and be sure that every possible value whose digits can add to the value of n are considered.

1. Representing numbers in a way that allows individual digits to be checked.

I used an array to represent the numbers digit by digit in my code, this way digits can be checked individually by being referenced in the array.

The declared array has size n, as this is the maximum number of digits that a value adding to “n” may have (as we discovered in part 1).

The value being stored in CurrentNum needs to be incremented as each number is checked. Therefore the logic which is used when counting needs to be implemented so that every number is represented correctly in the array.



In the above code the digit column is incremented, and subsequently every digit is checked to see if its value has reached 10. If a digit is equal to 10 the digit above must be incremented and the digit in question must be set to zero.

It is worth noting that the digit value is stored in CurrentNum[0]. This is because the digits are stored in reverse for instance if n=5 the value 123 would be stored as shown below.

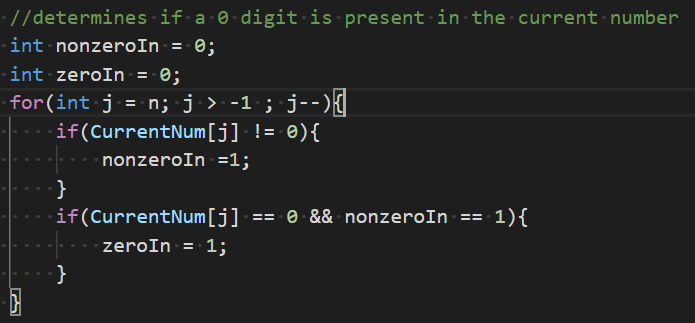
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Index | 1 | 2 | 3 | 4 | 6 |
| Value | 3 | 2 | 1 | 0 | 0 |

Fig 1 – example of number 123 being stored in the CurrentNum array

1. Determining which numbers contain a “0” digit and omitting these from being checked.

As we can see from fig1 to determine weather a number contains a zero digit we cannot simply check every digit to test if a zero is present otherwise any number that doesn’t contain n digits will be ignored.

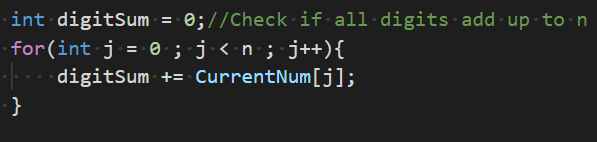
The solution for this is to test for a zero digit that is found after any non-zero digit, this way the zero digit is actually present in the number rather than being a leading zero from the arrays implementation.



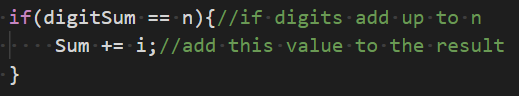
For a zero to be found in the current number two conditions must be satisfied simultaneously; A non-zero digit must already have been found, and a zero digit must be found. The first if statement updates a flag for if a non-zero digit has been found, once this flag is set to 1 (True) the loop then will look for a zero digit which if found will set the zeroIn flag to true and this number will not be processed.

1. Checking all numbers whether digits add to the value of n.

in order to check if all digits add to the value of n we can simply iterate through all digits in the array adding their values to the same variable and test if this variable is equal to n.

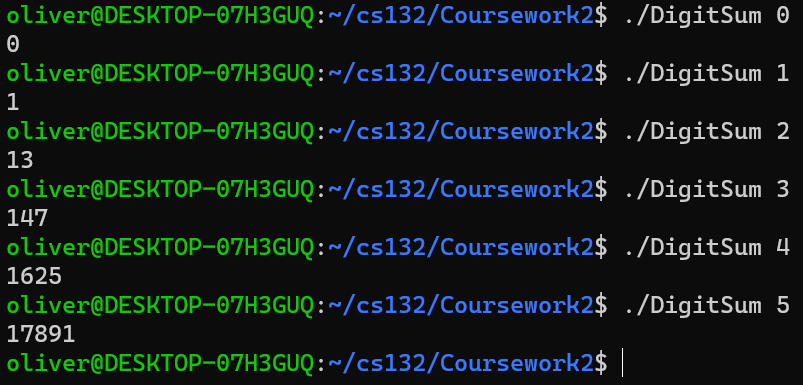


1. Returning the sum of these digits.

Whenever digitsum is equal to the value of n this number should be added to a variable storing the total sum. As the number stored in CurrentNum is incremented at the same time the iterator variable i is we are able to add the value of i to this variable whenever preconditions are correct.

The variable Sum is returned at the end of the function.

## Testing:

In order to test the correctness of my solution I chose to test values between 0 and 5 .

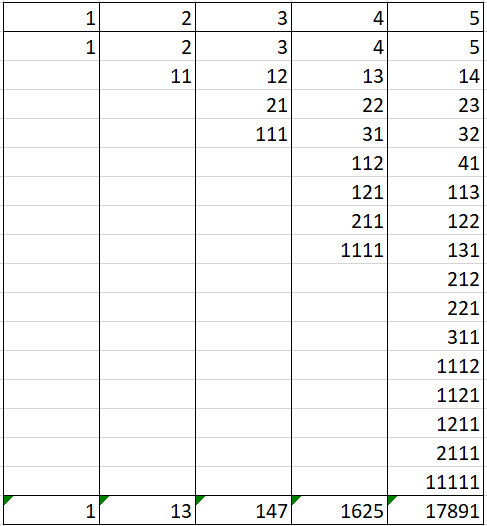


Fig 2 – Expected results and calculated results of DigitSum.c

This testing evidence suggests that the program is working and the desired outputs are achieved.

## Changes Made after testing:

During the testing process I noticed a pattern which occurred in the number of solutions for each value of n. There are always 2^(n-1) solutions, we can use this to improve the efficiency of the program by changing the iterative condition like so:

First we calculate the number of solutions for our given value of n and count the amount of solutions we have:

Notice that I have subtracted 1 from the number of solutions, this is because in order to improve the efficiency of our program I will still be calculating the maximum value as I did before. This means we no longer have to iterate to this number saving a significant amount of time; as we are performing less comparisons.

We use a while loop for our iteration with the following condition:



Our counting variable i remains in the code as it helps with adding together the sum of numbers, it is incremented at the end of the loops code block.

## Discussing efficiency:

Creating an efficient design for this program is challenging as the big O notation for this algorithm is (numbers that need to be considered increases exponentially as n increases) as seen in Fig2.

This means that even for small values of n evaluating f(n) can take some time to do.

The adjustments I have made to my program improve the efficiency by reducing the size of the dataset.

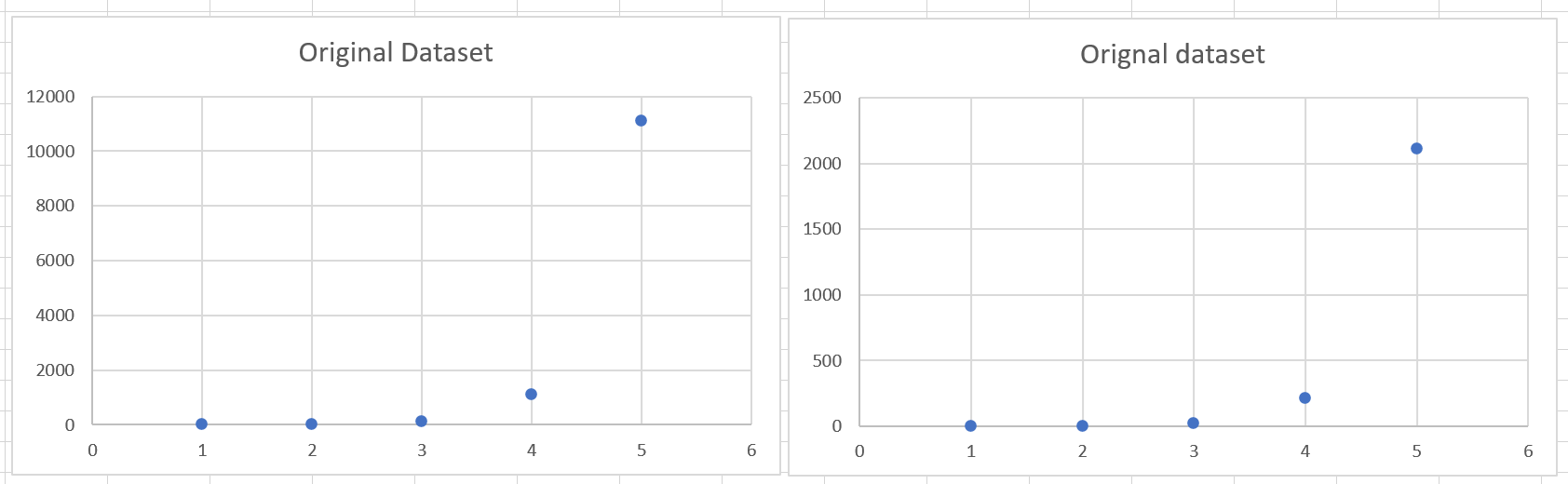


Fig 3 – Two graphs showing how the number of iterations increases as n increases (original implementation on the left improved implementation on the right)

Although the program follows the same big O relationship the size of the dataset has been reduced significantly (when n = 5 the dataset is reduced to roughly a fifth of its original value).

# Question 2:

## Introduction:

A variety of text editors are readily available for all platforms, this means in order to design something meaningful a lot of thought has to go into how the product you create will be different, and hopefully better, than other solutions. As I am designing a text editor for the Unix I need to think about solutions that exist specifically for this platform; most notably nano.

I also want to design my solution so that it very clearly meets the requirements that have been set out. Nano and other text editors make use of a cursor to edit characters individually however the requirements contradict this.

My design will take inspiration from other solutions particularly in the additional features I add. I plan to add a more sophisticated user interface, as this makes editing files significantly easier as you can view your changes in real time.

## Requirements analysis:

The requirements for this task are laid out explicitly in the question brief:

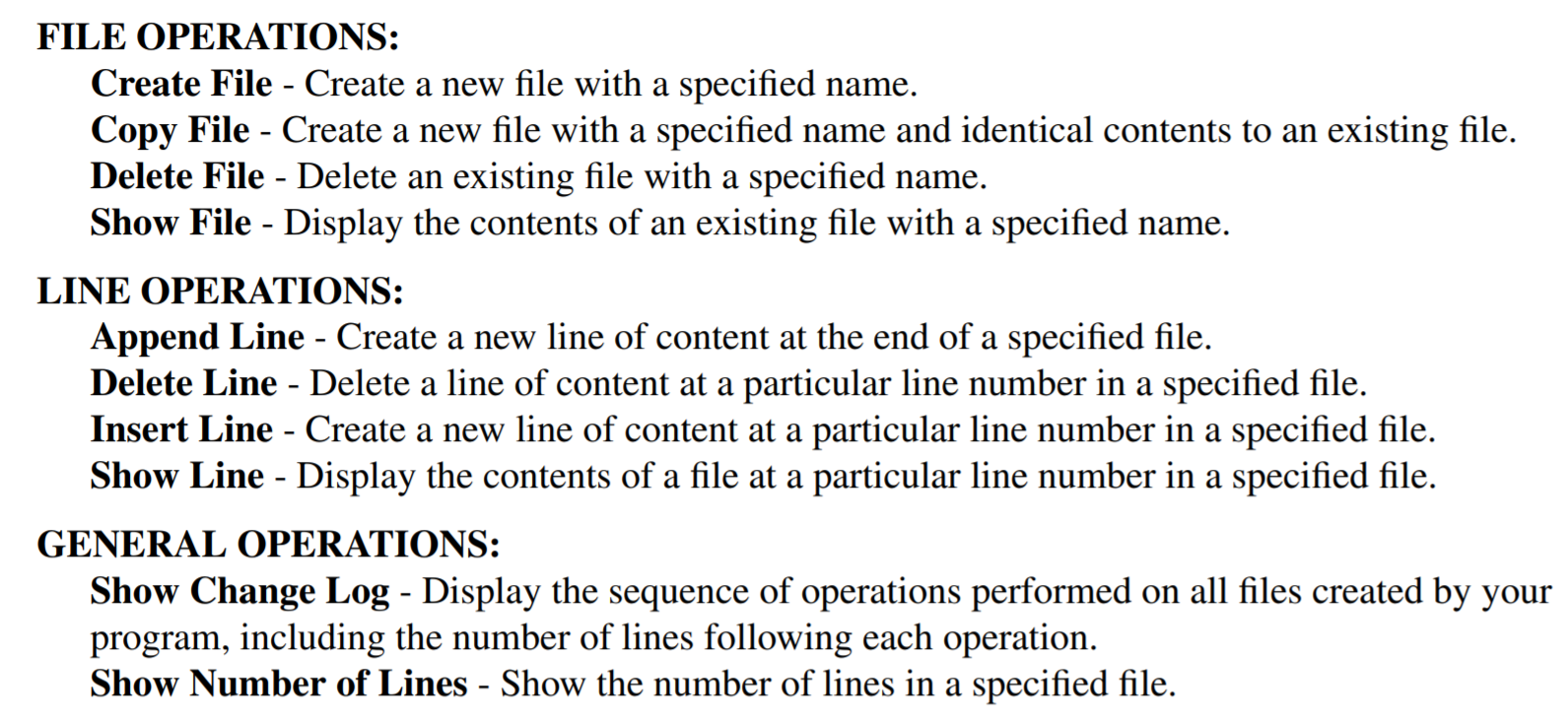


Fig 2.1 – an extract from cw2.pdf containing the requirements for question 2

The only other requirements for the text editor is that a command line interface is required; it should edit files in the current working directory; and that two additional features are also implemented.

My additional features are described below:

1. A more advanced UI – I plan to create an interface resembling applications such as nano or vim using the ncurses library.
2. Show line numbers – The requirements suggest that this application will edit files line by line rather than char by char, as a result it may become tedious to work with longer files. To remedy this I will implement a command that shows the user the line numbers on each line, making use of the software much less tedious.

## System design:

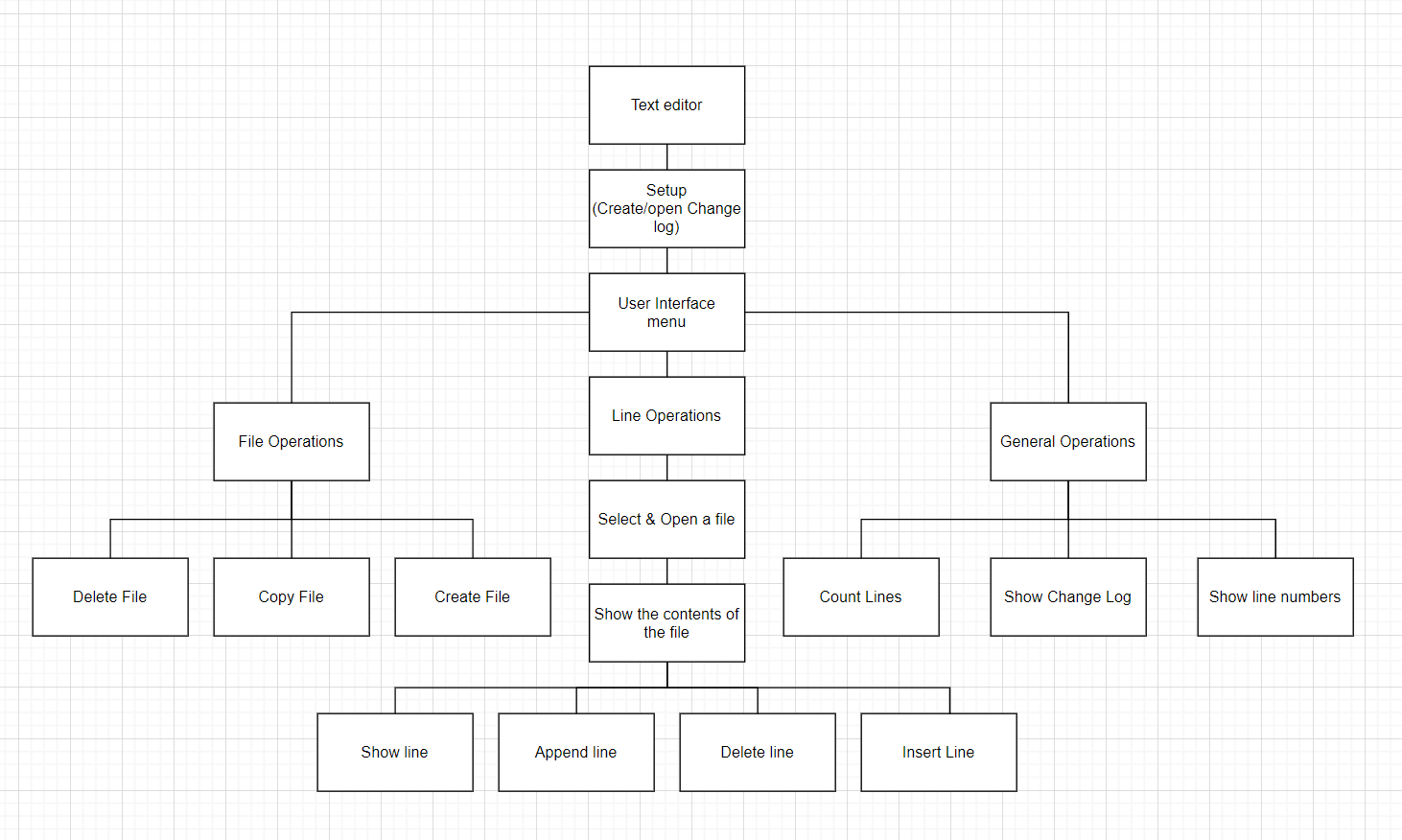
In order to create a structured solution I created a decomposition that showcases some of the control flow of my program: 

Fig 2.2 – A decomposition structured to follow the control flow of my program

The setup function will be written into main() as this is the first function called when a c program runs, and it is unnecessary to enter a new scope for code that does not need to be reused or portable (that would simply waste memory for no reason).

Some functionality in my program does not require to be implemented in its own function as it only appears in one section of code:

“Up” and “Down” commands – when using the UI to view the files contents its possible that the content of a file may not fit onto one screen. To account for this the “Show File” will be programmed to start at a particular line (initially set to 1) and will terminate at the final line it can fit on the screen. In order to view other sections of the file the user will be able to type either “up” or “down” to increment or decrement this starting value in order to view the rest of the file.

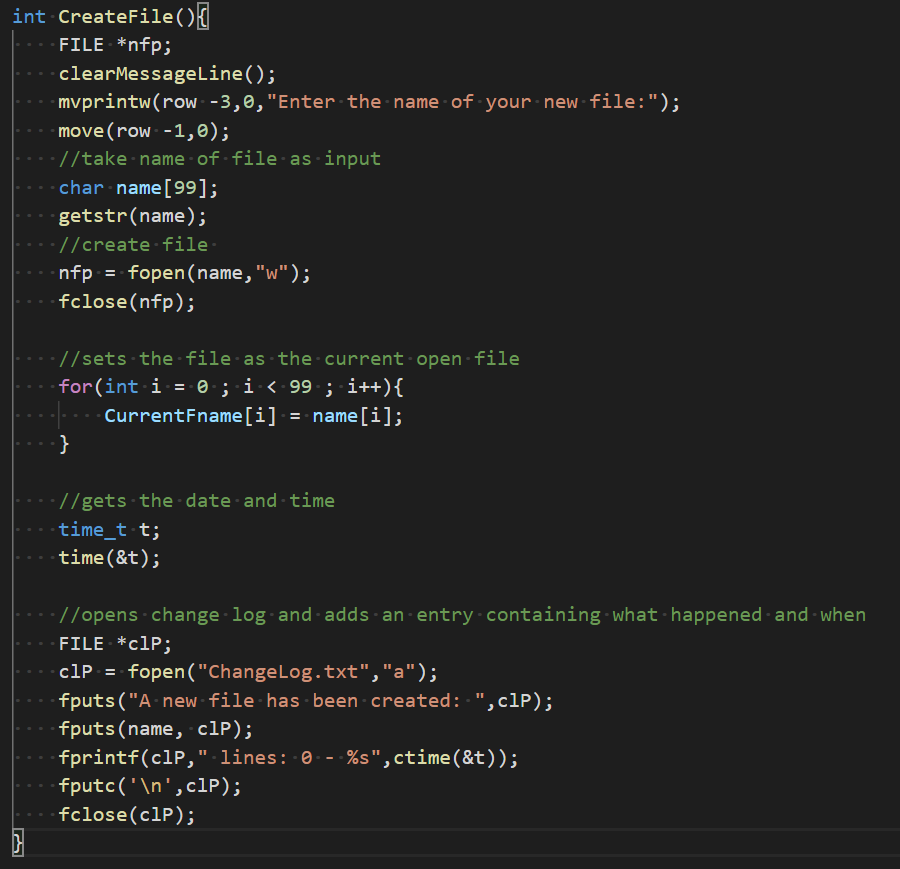
## Implementation:

### Key Design Decisions:

The strength of command line text editors is in their simplicity. when using command line interfaces users will often not require particularly complex formatting of their text documents therefore I have chosen to mirror this simplicity in my design. The user input is taken entirely from one text field located at the bottom of the screen.

The other features of the program are the display which shows the contents of the file that will fit to the screen, and the title of the file and user prompts are displayed at the top and bottom of the screen respectively, this allows the user to keep track of what file and operations they are performing.

### Create File:

In order to create files I utilised the fopen function. When fopen is given a file name and “w” as a parameter it creates a new file with the file name that has been passed.

I decided to also set the newly created file as the current open file that the text editor was working on, as usually when you create a new file you want to start working on it straight away making the process of starting a new document more efficient.

### Copy File:

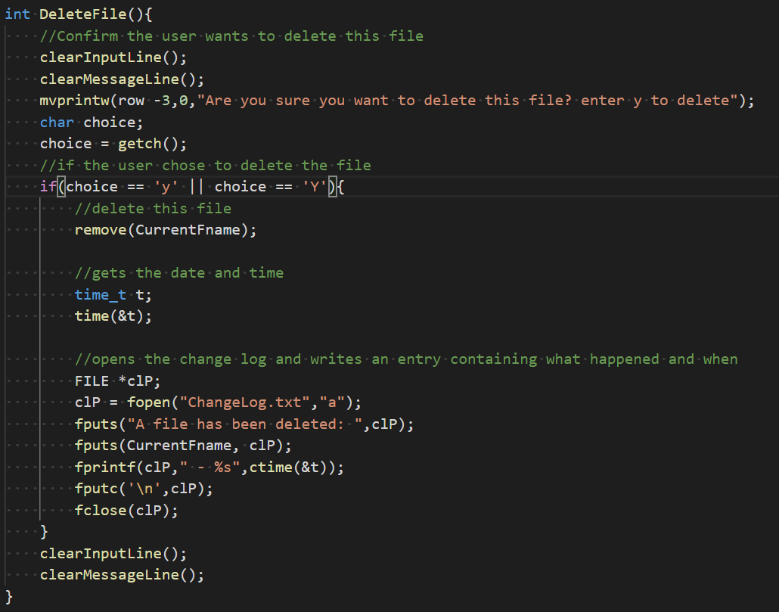
My implementation will copy the current selected file to a new file which the user is asked to name.

I decided that it makes sense for the user to have to open the file they want to copy so that they will see the contents of the file they have selected. Reducing the chance they copy the wrong file.

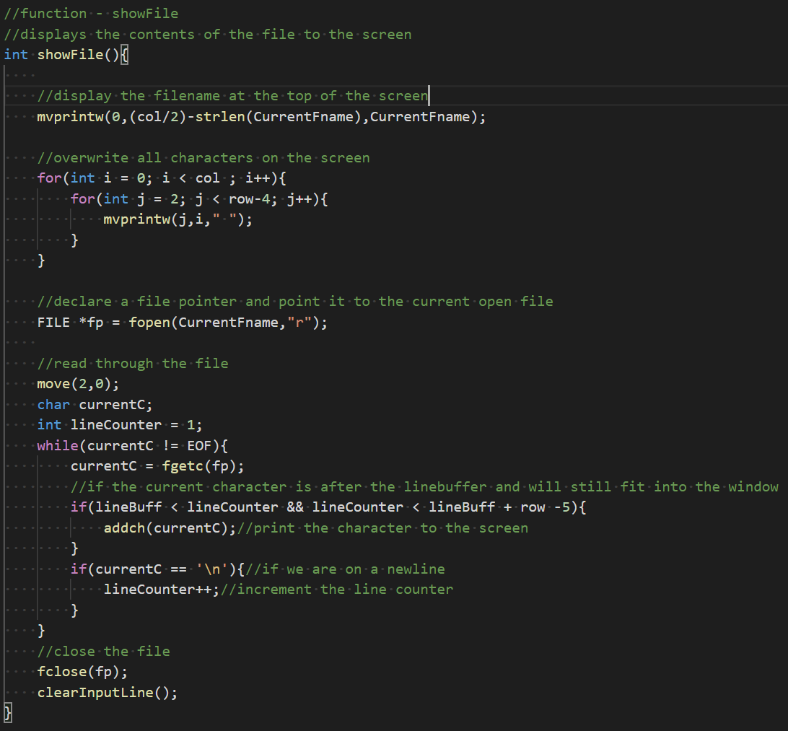
The code will check that a file with the same name doesn’t already exist as this would overwrite the file rather than create a new copy.

The program then will iterate through the entire of the open file writing each character to the new copied file until the end of the file has been reached in which case both files are then closed.

### Delete File:

My deletefile implementation works similarly to the copy file method. The currently selected file is deleted in order to force the user to check that this is the file they intend to delete.

The program also requires confirmation that the user does intend to delete this file, as it is very easy to mistakenly delete the wrong file therefore a double check is suitable.



### Show File:

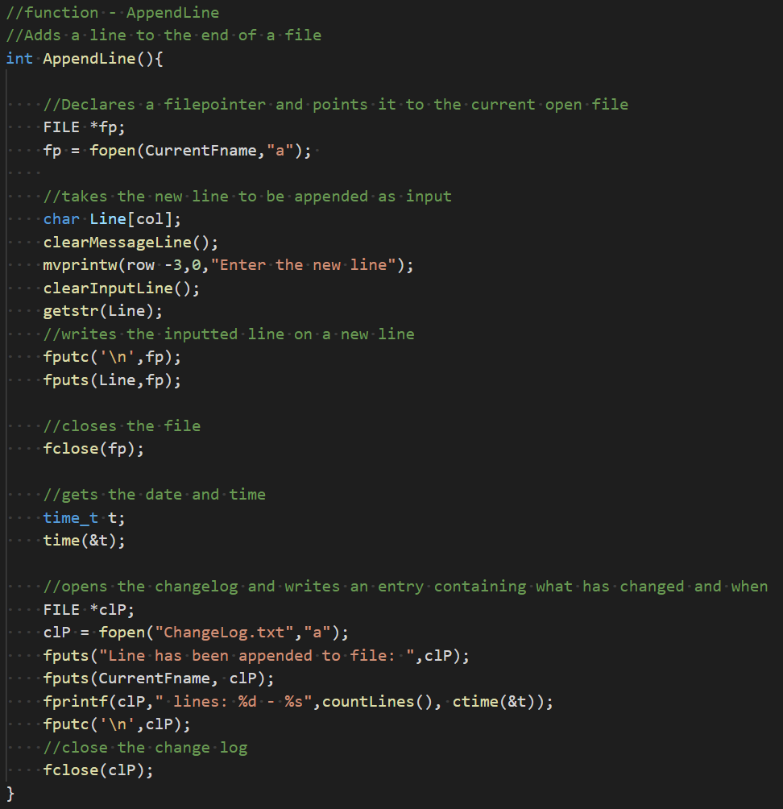
The showfile method displays the title of the current open file at the top of the screen. This is so that if the user is switching between many different files they wont lose track of which file is currently open.

The method also determines how much of the contents of the file can be displayed on the screen based off of the row variable.

In order to deal with files that contain more than a page of content I have implemented two commands that work with the ShowFile method.

The variable lineBuff stores the value of the first line which should be printed to the screen, this value can be altered using the commands “up” and “down”. This simulates scrolling up and down through the file and therefore allows the user to view the entire contents of the file.

### Append line:

The append line method takes a line as input which it adds to the end of the file. The length of the line has a maximum length of the width of the screen due to the UI library that I am using.

### Delete line:

The user is prompted to enter the line they want to remove from the current open file.

In order to remove characters from a file I have chosen to create a copy of the current open file and omit the selected line from the new copy.

A temporary copy file is created in which the entire contents of the file are copied to until a variable which counts the current line is equal to the line which is chosen to be deleted.

The original file is then deleted and the Temporary copy is renamed to the name of the original file, having the effect of removing the line from the file.

### Insert Line:

The insert line function works in a similar way to the delete line function. The user is prompted to enter the line number they would like to insert to. The program creates a temporary copy file and copies every character 1 by 1 whilst simultaneously counting the current line.

Once the chosen line has been reached the user is then prompted to enter the line they wish to insert. The program will then enter that line into the file before proceeding to copy the rest of the original file over.

The original file is then deleted and the temporary copy file is renamed to match the original file.

### Show line:

The show line function has a similar implementation to the show file function. The only difference is that the condition for printing characters has changed.

The program prompts users for the line number they want to view the contents of, and the program iterates through the file and will display characters once the line counting variable reaches the value of the chosen line.

### Show Changelog:

In order to display the changelog to the screen I have chosen to reuse the code for the show file function, however, rather than taking user input for the name of the file, the program will automatically open “ChangeLog.txt”.

A different linebuff variable has been used in this function as well. This is so that the scrolling feature of both the change log and current open file do not interfere with each other.

## Count lines:

In order to identify a new line in a text file the program looks for ‘\n’ which indicates that a new line should be started. This function has been designed to return the number of lines that have been counted. This is because I can make use of this function in other parts of my program.

To implement this feature in line with the specification I wrote a function called printLineCount(), that will display the number of lines returned by the function +1 as the countLines function is zero indexed.

## Additional Features:

The main weakness that I wanted to address in this program was how files are navigated, as a result the purpose of both of my additional features is to make navigating through files easier for the user.

I have chosen to implement A more sophisticated text based interface. This is something that comes standard with any text editor as it allows users to see the changes made to their file as they are made. This is a useful feature for users as it allows users to work more efficiently, as they are not required to enter commands to view the changes they have made or commit them to memory and as a result can commit multiple changes to a file without interruption.

I also implemented a function that would display the line numbers of each line, this means the user is not required to count each line in order to use the commands to use the software. This can be especially frustrating for longer files as this process begins to waste a lot of time especially when mistakes are made and additional commands are then required.

## Testing:

In order to test my solution thoroughly I will conduct a set of tests for valid and invalid inputs, this allows me to be confident that my solution will work as expected and also be robust against invalid inputs that do not intend to cause errors.

### Testing Valid Inputs:

|  |  |  |  |
| --- | --- | --- | --- |
| Test Number | Input | Expected output | Actual output |
| 1 | CreateFile Testing.txt | Testing.txt file created in current directory and should be opened in the text editor | Testing.txt file created in current directory and should be opened in the text editor |
| 2 | AppendLine “The rain is heavy” | Testing.txt now contains line “The rain is heavy” | Testing.txt now contains line “The rain is heavy” |
| 3 | CopyFile “Copied.txt” | A copy of Testing.txt is created called Copied.txt | A copy of Testing.txt is created called Copied.txt |
| 4 | DeleteFile “Copied.txt open” | The file Copied.txt is deleted | The file Copied.txt is deleted |
| 5 | AppendLine “The snow was light” | Testing.txt now contains line “The snow was light” | Testing.txt now contains line “The snow was light” |
| 6 | insertLine 2 “This is line 2” | Line 2 of Testing.txt says “This is line 2” all other lines are shifted | Line 2 of Testing.txt says “This is line 2” all other lines are shifted |
| 7 | seeline 3 | The third line of Testing.txt is displayed on the screen | The third line of Testing.txt is displayed on the screen |
| 8 | changelog | A description of the previous operations is given | A description of the previous operations is given |
| 9 | File | Contents of Testing.txt are displayed on the screen | Contents of Testing.txt are displayed on the screen |
| 10 | linenums | The contents of the file are shown with the relevant line numbers | The contents of the file are shown with the relevant line numbers |
| 11 | deletelines 3 | The third line of Testing.txt is omitted | The third line of Testing.txt is omitted |
| 12 | Createfile “New.txt” | New.txt is added to the current directory | New.txt is added to the current directory |
| 13 | Seeline 2 | The contents of line 2 are displayed on the screen | The contents of line 2 are displayed on the screen |
| 14 | Countlines | The number of lines is displayed on the screen | The number of lines is displayed on the screen |

Fig 2.3 - table containing the valid tests for editFile.c

The table is highlighted to show the severity of each outcome where green is successful, orange is minor improvements required, and red is execution terminated.

### Testing invalid inputs:

|  |  |  |  |
| --- | --- | --- | --- |
| 15 | deletefile (without open file) | No file is deleted | No file is deleted |
| 16 | Append line (no file open) | No file is appended | Segmentation fault |
| 17 | Insertline 89 “Failed line” | The line isn’t added and a helpful message is displayed | The line isn’t added no message is displayed |
| 18 | Deleteline 89 | The line isn’t deleted and a helpful message is displayed | The line isn’t deleted no message is displayed |
| 19 | createfile “” | file is created | Segmentation fault |
| 20 | Append line “” | No characters are added to the file and a helpful message is displayed | EOF character remains in the file, a ‘\n’ is added |
| 21 | show line 89 | No line is shown and a helpful message is displayed | No line is shown no message is displayed |
| 22 | Copyfile “” | A copy of the file is created | Segmentation fault |

Fig 2.4 – Table of invalid tests for editFile.c

## Improvements made in response to testing:

My priority was to fix the segmentation fault which occurred on test 19, the only inconsistency I found with this input command was that the array which the input was being passed into on line 378 was set to a fixed size, after changing this the program would now create a file called ‘ ‘.

In order to fix the segmentation fault in test 16 I didn’t allow users to run certain commands without opening a file. After doing this I also decided to allow the user to quit the program while picking a file to open in case the user forgets the file name. To do this I set the filename to a rouge value, then when this rouge value is detected in menu the program quits.

I was unable to find a fix for test twenty two so I amended the input instructions to request the user not name their file with an empty string.

Next I needed to add a suitable message to the functions in test 14-15 and 18-19 seeing as these incorrect inputs do not have any adverse effect on the program I simply added the following code to the end of each function:  
if(UNWANTED PRECONDITIONS){

clearMessageLine();

mvprintw(row -3,0, “HELPFUL MESSAGE”);

clearInputLine();

getch();

}

In the case where code should not be ran using these inputs this code is wrapped in an else clause that followed the above code.

## Final Evaluation:

The success of my solution can be measured based on 3 requirements:

* Ability to meet the design brief
* Robustness
* Strength of justification for additional features

My final solution was able to meet all of the requirements set out in the design prompt and therefore serves as a good example of a functional text editor, allowing the key functionality that any basic user would require to create text documents effectively on a Unix terminal system.

My solution is fairly robust and was able to pass the majority of tests after refinements, however, this is a weakness of my solution as no program should crash in the event the user accidently enters undesired inputs. As a result given more time this would be something I would aim to improve in my solution.

The additional features of my solution are well justified as they aim to solve an identified weakness of the original program. This being the difficulty to traverse and make changes to larger files. As a result my solution has succeeded in this requirement.

# Using the applications:

## Question 1:

My solution for question one makes use of math.h and is compiled using the following version of gcc:

gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0

in order to execute this code the value of n should be passed as the object file is called for execution for example:

./DigitSum 5

Will return

17891

## Question 2:

My solution for question two makes use of the following headers:

* curses.h
* string.h
* math.h
* time.h
* unistd.h

and is compiled with the following version of gcc:

gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0

when compiled the math and curses headers need to be linked, the command for compilation should resemble this:

gcc -o editfile editfile.c -lm -lncurses

Below is a list of all commands and descriptions of what they do:  
(Note all commands are case and whitespace sensitive)

|  |  |
| --- | --- |
| Command | Description |
| open | Will prompt the user for a valid filename and set this as the current open file. If filename “quit” is given the code terminate |
| createfile | Will prompt the user for a file name and create a file in the current directory with this file name |
| appendline | Will prompt the user to enter a line which is added to the end of the current open line |
| insertline | Will prompt the user for a line number and a line which will be inserted into that line of the file |
| deleteline | Will prompt the user for a line number and omit this line from the file |
| copyfile | Will prompt the user to enter a unique file name and make a copy of the current open file in the current directory |
| deletefile | Will delete the current open file from the current working directory |
| changelog | Displays the change log of the current working directory |
| file | Displays the current open file |
| down | Scrolls down the file |
| up | Scrolls up the file |
| seeline | Prompts the user for a line number and displays this line of the current open file on the screen |
| linenums | Displays the number of each line on the screen of the current open file |
| countlines | Shows the total number of lines of the current open file |
| quit | Terminates the program |

Fig 2.4 – The available commands for editFile.c