Computer Organisation And Architecture

CS132

Coursework 2 2021-22

Author:

Param Bhatia

January 2, 2021

Question 2

Introduction:

In this section I will address the important processes used in creating the program for *Question 2* in coursework 2. I have answered all parts of *Question 2* in this document.

Aim

The broader aim of this question is to create a file handling program, which can conduct a large variety of operations. To begin with, I started by creating a functional description of what I want the program to do:

- Firstly, provide user with a list of operations that the program is able to carry out.

- Then ask user which operation they want to carry out, and read the user input.

- The program will then need to validate the input to make sure that the input is a number, and that the number is within the valid range (1-11). Once it is validated, call a function which carries out the operation.

- Each operation should have a different, appropriately labelled function, because this increase modularisation, and improves code maintainability and code readability.

- For all functions a file name will be required, so we will need to ask the user for that too, and store it as a char array.

- The name will need to be validated, in order to make sure file with that name exists.

- Once that has been validated, carry out the function as expected (discussed later on).

- Once the program has completed, ask user if they want to run it again without having to recompile or executing the program again, and depending on response, carry out program again.

List of Methods

Since there are a lot of methods in this program, due to modularisation, I have listed them below, and explained them further below:

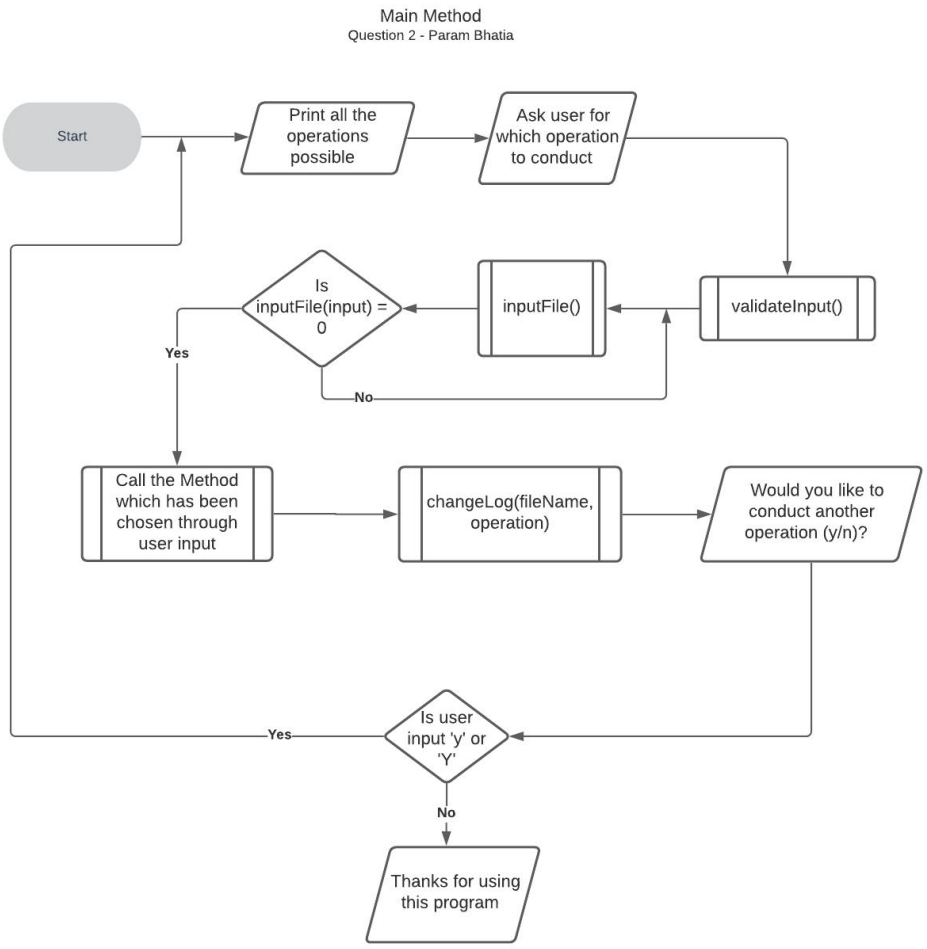
-**main()** - Lists all operations, and calls all other functions.
-**validateInput()** - Ensures input is a number within range 1-11.
-**inputFile()** - Prompts user to input file name.
-**checkExists()** - Checks to see if a file of the name exists.
-**addFile()** - Add a new file.
-**deleteFile()** - Delete existing file.
-**copyAndCreateFile()** - Copy contents of existing file, and copy into a new file.
-**showFile()** - Print contents of existing file.
-**appendLine()** - Add one line of text into a file.
-**deleteLine()** - Delete specified line if text from a file.
-**insertLine()** - Insert line at specified position in a file.
-**showLine()** - Print a specified line of data to user.
-**numberOfLines()** - Count number of lines in a file.
-**checkWordInFile()** - Check if a word exists in a file.
-**wordCounter()** - Counts the number of words in a file.
-**changeLog()** - Adds data on files after each operation.

All of these methods are appropriately named and will be explained in greater detail below.

Low-Level Design Analysis

main()

Here is a flowchart demonstration of the main method:



Let us now go through the code for the this function.

```c
int main(){
    int repeater = 0;
    while (repeater == 0)
    {
        char repeatProgram;


        printf("Welcome Here!\n");
        printf("1 - Add A New File\n");
        printf("2 - Delete File\n");
        printf("3 - Create New File And Copy Existing File\n");
        printf("4 - Show Contents Of File\n");
        printf("5 - Append New Line to End Of File\n");
        printf("6 - Delete a specific line from a specified file\n");
        printf("7 - Insert Text At Specified Line Of File\n");
        printf("8 - Show Text Of Specific Line In File\n");
        printf("9 - Show Number Of Lines In File\n");
        printf("10 - Search for word in a file\n");
        printf("11 - Word Counter in a File\n");

        int option;
        option = validateInput();   //call function to validate input and store user choice

        char input[MAXLENGTH];      //declare char arry of fileName
```

This is the main method for the program. It starts off by intialising an integer variable ‘repeater’ to 0, since 0 represents success by C standards. This variable is used to determine whether the program will run again if the user wants to conduct another operation. This is possible because the rest of the code in the main method is within the scope of the while loop, so if a user wants to repeat the program, then repeater will stay as 0, else it will be changed to 1. Inside the loop, a character value ‘repeatProgram’ is declared, but not initialised yet, as this will be used for storing user’s reply to if they want to repeat the program. Next, there are a series of print statements, all of which define the operations that the program can do, including the two additional operations. Following that is the declaration of an integer variable ‘option’ and it is initialised with the return value from the function validateInput. Calling this function will prompt user to input a number to make a choice of operation. Then, there is a character array declared as ‘input’ with a fixed size of a macro value defined in the header.

```
if (option == 1)
{
    if (addFile(input) == 0)
    {
        //addFile does not need name validation
        changeLog(input, "has been created");
    }

}
else if (option == 2)
{
    if (inputFile(input) == 0)
    {
        //will delete the file
        changeLog(input, "has been deleted");
        deleteFile(input);

    }
}
```

Subsequently, we can see what happens once the option value is provided. If the option value is 1, then the addFile method is called, with the argument being ‘input’. Since input is an ‘array’, the argument passed into the addFile method is essentially the memory address of the ‘input’ array. Creating a new file does not require any validation in terms of file name. However addFile will return 1 if the user decided to exit the method, or 0 if the user successfully entered a file name. If the return value of the addFile method is 0, then call the changeLog method, with arguments ‘input’ and ‘has been created’ which represents file name and operations respectively.

However, when option is greater than 1, i.e. not used for creating a new file, the

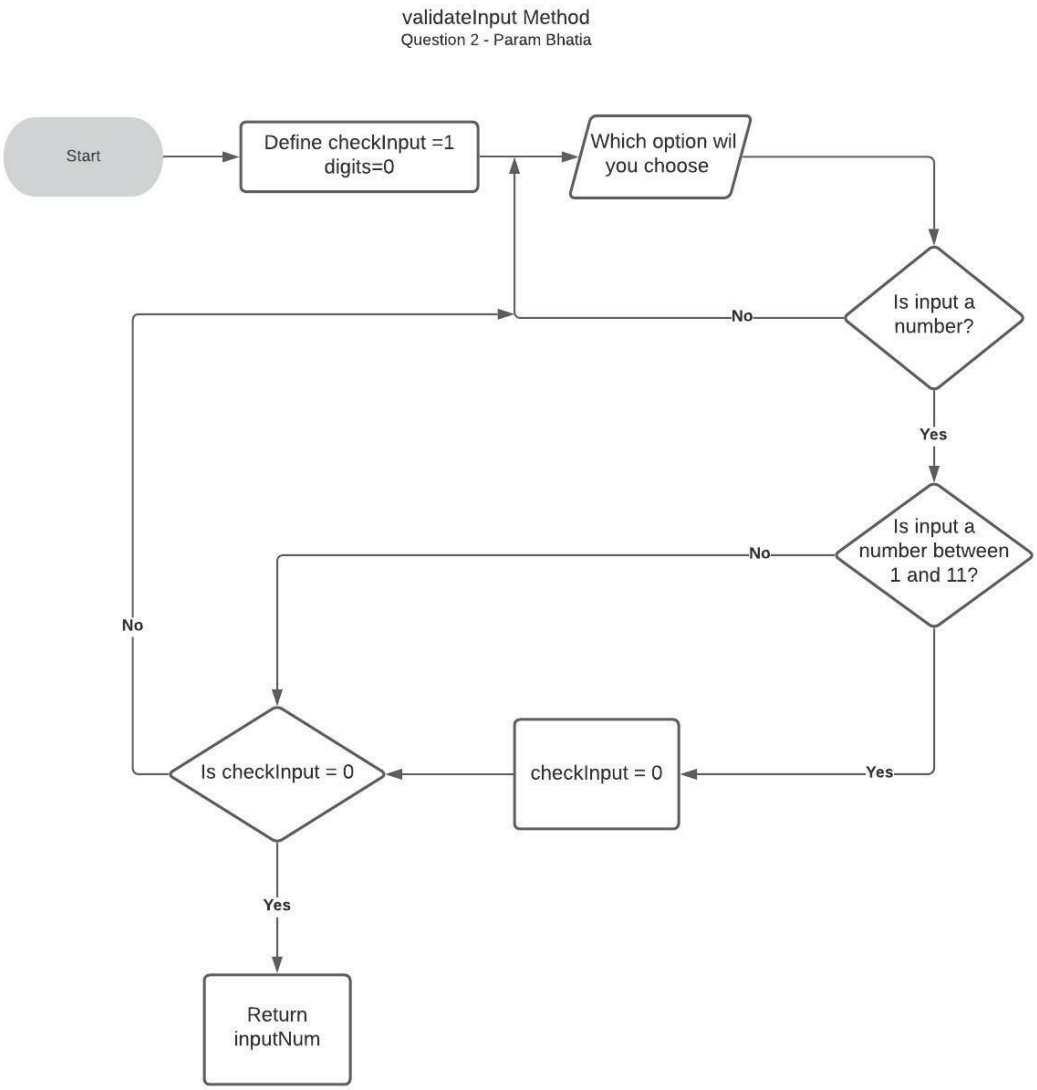
inputFile method will be called. This method will validate if such a file exists or not. This method will then return 0 is the file name stored in ‘input’ has been validated successfully. If it has, then call the changeLog function with given arguments. Then call the deleteFile function with ‘input’ as the argument. All the other operations are conducted in a similar manner to deleteFile, so it would be redundant to go through them.

```
printf("Would you like to do an operation again? (y/n)\n");
scanf("%s", &repeatProgram);
if (repeatProgram == 'y' || repeatProgram == 'Y')
{
    //check to see if input begins with 'y', and if it does, then set flag to 0
    repeater = 0;
}
else
{
    repeater = 1; //if user does not want to repeat, then exit the while loop}
}
```

This is the last part of the main method, and it allows the user to repeat the program if they choose to. The user is prompted to enter either ‘y’ or ‘n’ on their choice to run the program again. Their reply is stored in the character variable ‘repeatProgram’, which stores only one character. If that character is ‘y’ or ‘Y’, then set repeater to 0, and program will run again. Otherwise, the program will end by leaving the while loop.

validateInput():

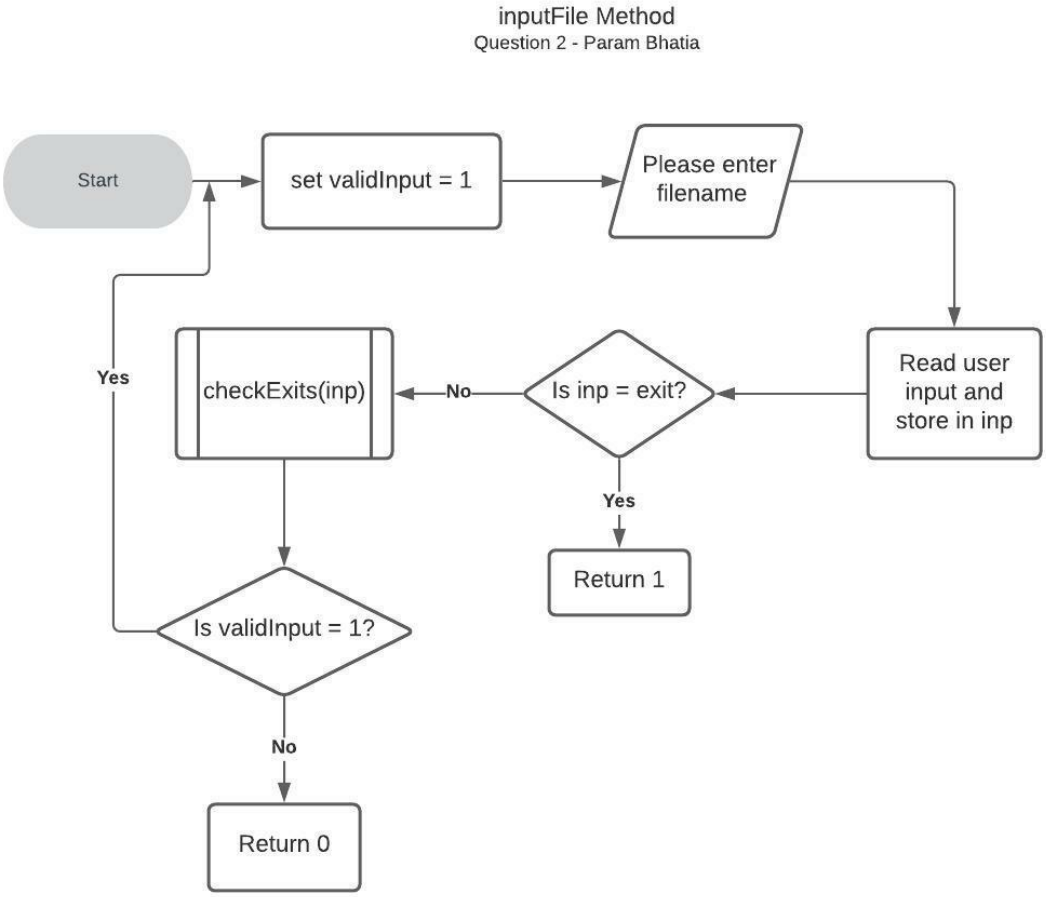
This is a flow chart implementation of the function:



The code for this method is very similar to the validation method used in question 1. Please refer to the low-level analysis section in the *Question* 1 for explanation of the code in this method.

inputFile():

Here is a flowchart implementation of the method:



Let us go through the code:

```
int inputFile(char (inp)[MAXLENGTH]){
  int validInput = 1;        //int type to check whether input file has been entered properly, or if program should stay in loop.
    char text[] = ".txt";       //used to concatenate file name with .txt since the program is meant for text files.
    while (validInput == 1)
    {
        //the following print statements explain to the user how to input a file name.
        printf("Enter file to be checked\n");
        printf("Please note: File Names cannot have spaces. If the name has multiple words, separate using hyphens\n");
        printf("Please note: Do not add .txt to the end of the file name\n");
        printf("Please note: If you choose to leave program, please type exit\n");
        scanf("%s", inp);       //input as only one word, as spaces in file names can cause errors in external software, eg shell scripts.
        if (strcmp("exit", inp) == 0)       //checks if user input value is the same as "exit"
        {
            //this statement checks whether the user has entered exit, in order to leave the program from the loop.
            return 1;
        }
        printf("The file name is %s\n", strcat(inp,text));  //provide user with update on what is happening
        validInput = checkExists(inp);   //calls function to check if input file exits
    }
    return 0;   //C standard for success, since the program will only get to here if the file name is validated.
```

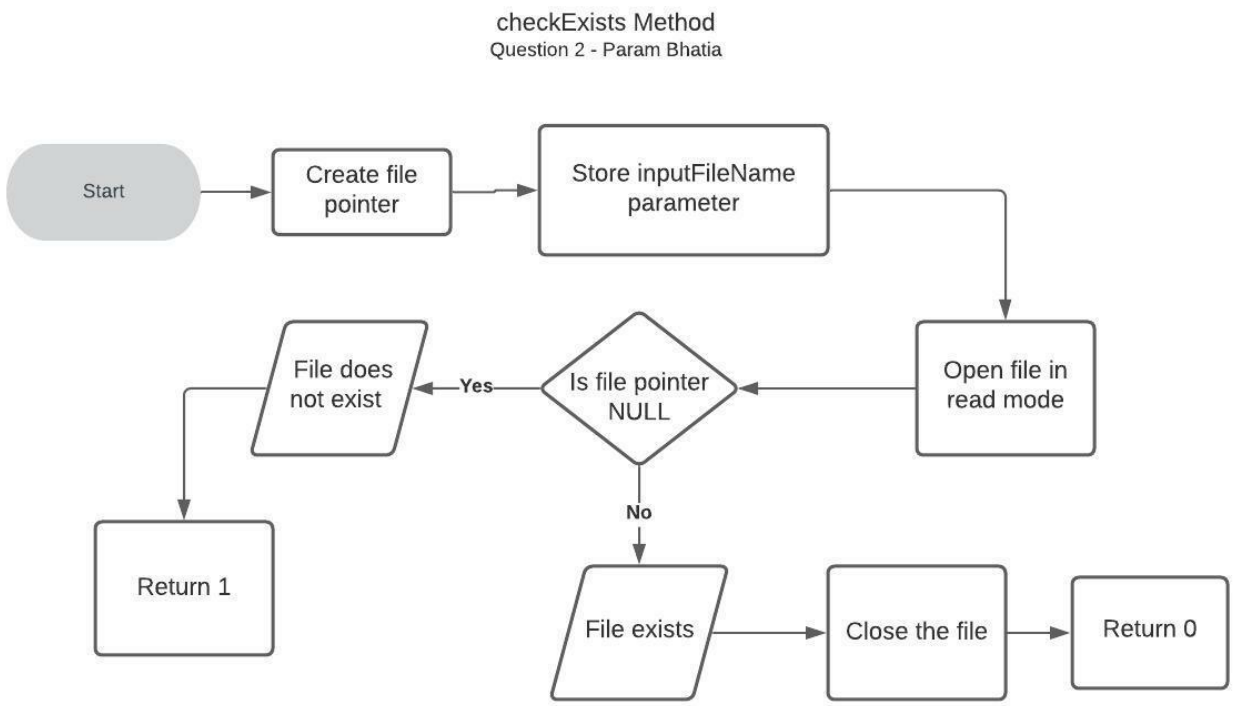
The method starts by defining an integer variable ‘validInput’ and initialising it to the value 1. This variable will be used as a method of deciding whether the file name is a valid file name. Then the method initialises a character array called ‘text’ and sets it equal to “.txt”. This will be used to concatenate the file name with a .txt at the end to ensure that the file is of a text format. The while loop ensures that the method will only be exited successfully when a valid input is entered, i.e. when validInput = 0. Inside the loop, the program prompts user to enter a file name with a set of regulations. This name is then read using scanf() function, and it is read into the parameter value.

An interesting feature I added is that the user may not want to enter a file name and they want to leave the program. They can do this easily by simply typing in “exit”. The program uses the strcmp() function in order to compare the string read from user’s input with the string “exit”. If they both are the same, i.e. if the user entered exit, the strcmp() function returns a value of 0. This will exit the method with a value of 1, which indicates an unsuccessful method call.

If the user does not enter “exit”, then the user input is concatenated with “.txt”. Then it is passed as an argument to the checkExists() function. The integer validInput is then set to the return value of checkExists(inp). If it is a file that exists, then checkExists will return 0, indicating a success, so the current method will leave the loop, and exits the method with value 0. If the file does not exist, checkExists will return 1, and the current method will loop again.

checkExists();

Here is a flowchart implementation of the method:



Let us go through the code:

```c
int checkExists(char inputFileName[MAXLENGTH]){
    FILE *fptr;     //creating a file pointer
    fptr = fopen(inputFileName, "r");       //simply opening file in 'read' mode will allow us to know if the file exists.
    if(fptr != NULL)
    {
        fclose(fptr);       //close file so it can be edited from other function too.
        printf("This files exists!\n");     //provide update to user regarding status of the program
        return 0;           //0 is the C standard to represent success.
    }
    else
    {
        printf("File does not exist, please try again\n");  //provide update to user regarding status of the program
        printf("Maybe you have spelt the name incorrectly, or the file does not exist!");   //provide user with error reasoning
        return 1;           //1 is the C standard to represent fail
    }
}
```

The method begins by declaring a file pointer "*fptr", and this pointer is set to the value of the fopen() function. The input parameter of the checkExists method is used as the name of the file to be opened. This is because the parameter is a char array

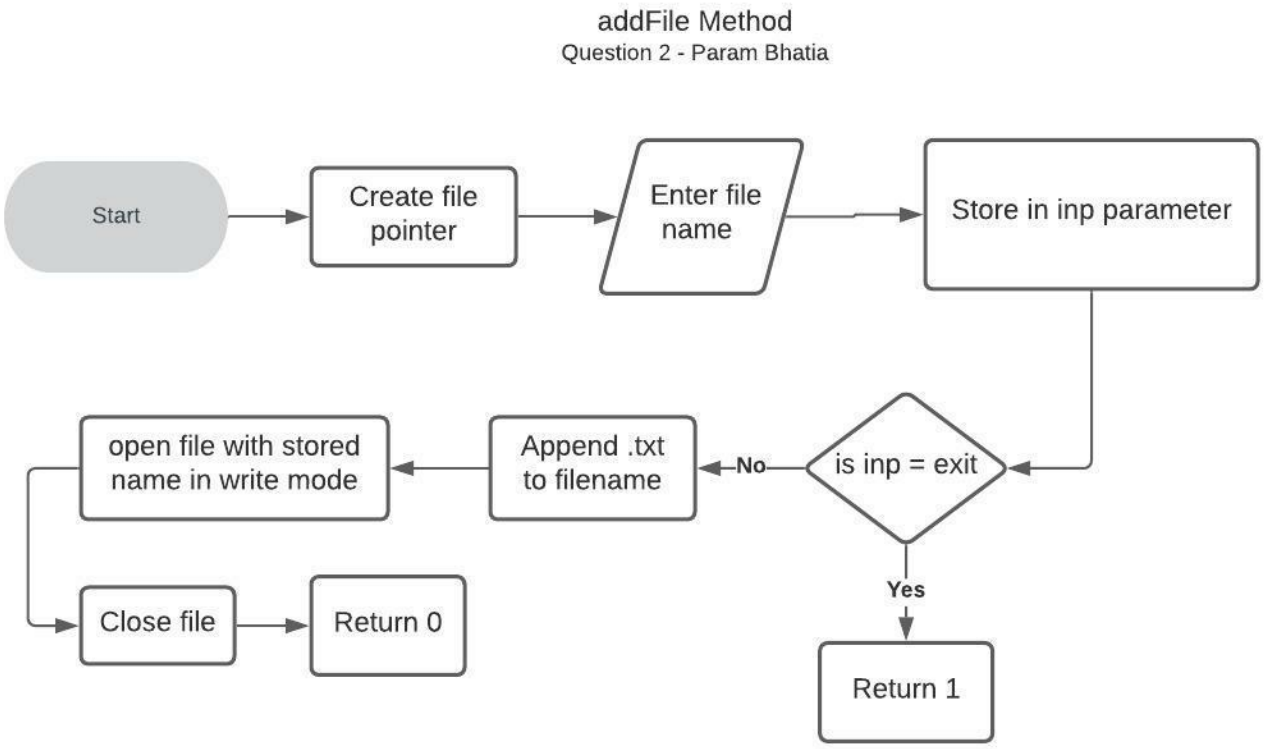
which has already been defined by a name provided by the user. Open the file in read mode, not in write as we are checking if the file exists, and write mode will create a file if none exist with that name. In read mode, if a file does not exist with that name, the file pointer value will be equal to NULL.

If the file pointer is not NULL, the function prints to the user that such a file exists. This file is then closed, so that other functions can edit and read this file. Then return from the method with a return value of 0, indicating that the function has been completed successfully.

If the file pointer is NULL, the function prints to the user that no such file exists. It will also print reasons for why the file may not exist. In this case the method is exited with a return value of 1, indicating that the function has not been completed successfully.

addFile();

Here is a flowchart implementation of the method:



Let us go through the code:

```c
int addFile(char (inp)[MAXLENGTH]){
    FILE *fptr;

    char text[] = ".txt";       //used to concatenate file name with .txt since the program is meant for text files.

    //the following print statements explain to the user how to input a file name.
    printf("Enter file to be created\n");
    printf("Please note: File Names cannot have spaces. If the name has multiple words, separate using hyphens\n");
    printf("Please note: Do not add .txt to the end of the file name\n");
    printf("Please note: If you choose to leave program, please type exit\n");

    scanf("%s", inp);   //input as only one word, as spaces in file names can cause errors in external software, eg shell scripts.

    if (strcmp("exit", inp) == 0)   //checks if user input value is the same as "exit"
    {
        //this statement checks whether the user has entered exit, in order to leave the program from the loop.
        return 1;
    }
    printf("The file name is %s\n", strcat(inp,text));  //provide user with update on what is happening in the program

    fptr = fopen(inp, "w");         //use write mode to create file
    fclose(fptr);                   //close file so other functions can edit
    return 0;                       //return with success value of 0
}
```

The function begins by declaring a file pointer “*fptr”, as well as a char array with a defined value equal to “.txt”. This will be used to concatenate the file name with a .txt at the end to ensure that the file is of a text format. The while loop ensures that the method will only be exited successfully when a valid input is entered, i.e. when validInput = 0. Inside the loop, the program prompts user to enter a file name with a set of regulations. This name is then read using scanf() function, and it is read into the parameter value.

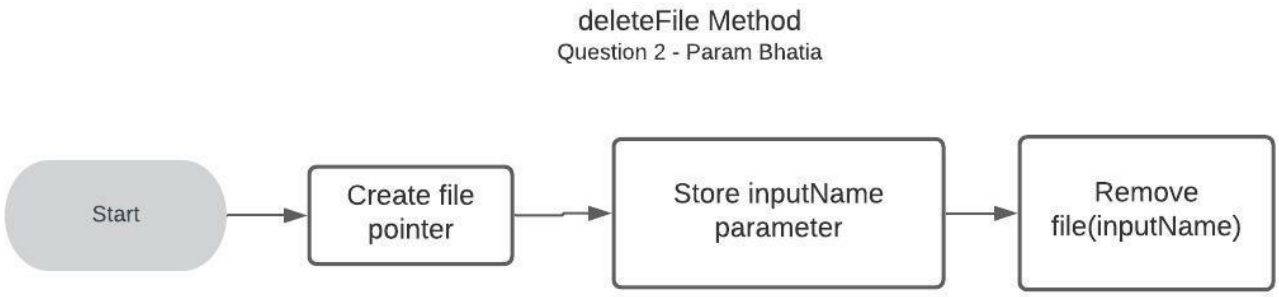
An interesting feature I added is that the user may not want to enter a file name and they want to leave the program. They can do this easily by simply typing in “exit”. The program uses the strcmp() function in order to compare the string read from user’s input with the string “exit”. If they both are the same, i.e. if the user entered exit, the strcmp() function returns a value of 0. This will exit the method with a value of 1, which indicates an unsuccessful method call.

If the user does not enter “exit”, then the user input is concatenated with “.txt”. Then a file with this name is opened in write mode. This file is then subsequently closed, so that other functions can read and edit this file.

The reason why addFile is not passed through inputFile() method unlike other methods, is that there is no need for file validation in this method, since we are not checking for a file, but instead creating one.

deleteFile();

Here is a flowchart implementation of the method:



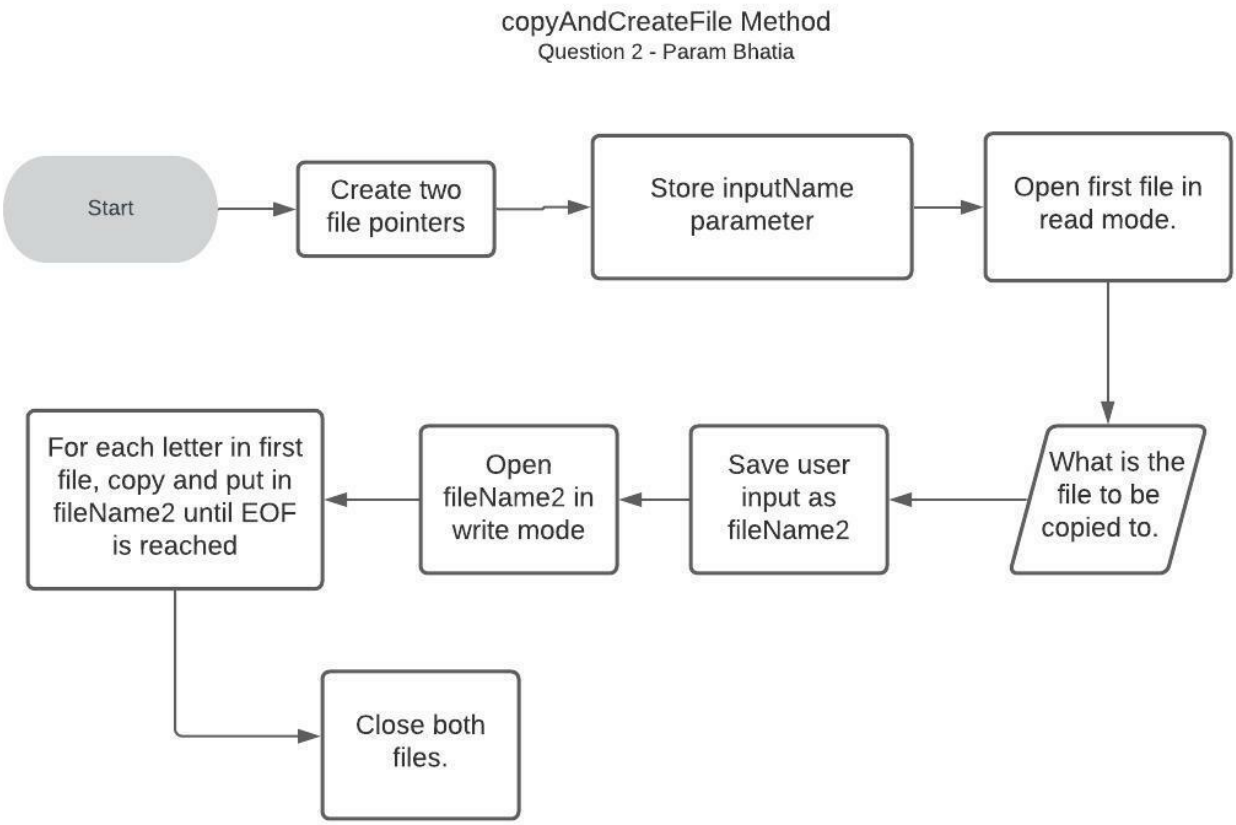
Let us go through the code:

```
void deleteFile(char (inputName)[MAXLENGTH]){
    remove(inputName);
    printf("The file has been deleted\n");  //Updates user with what is happening.
}
```

This is a very straightforward function. It takes the parameter as a file name, and uses the remove() function to delete the file with the name in parameter. It will then update the user that the file has been deleted.

copyAndCreateFile();

Here is a flowchart implementation of the method:



Let us go through the code:

```
FILE *fptr1, *fptr2;        //declaring two file pointers, one for each file.
char text[] = ".txt";
char input2[MAXLENGTH], letter, temp;

fptr1 = fopen(inputName, "r");      //opens first file in read mode

printf("The content of this file is as follows:\n");    //update user with the progress of operation.

temp = getc(fptr1);     //temporary char variable to print out contents of file.

//this loop will print contents of file 1, using while loop until the end of file is reached.
while (temp != EOF)
{
    putchar(temp);      //used to print char type to stdout
    temp = getc(fptr1); //getc is used instead of fgetc because it is much more efficient
}
rewind(fptr1);          //sets the file stream of file 1 to the beginning.
```

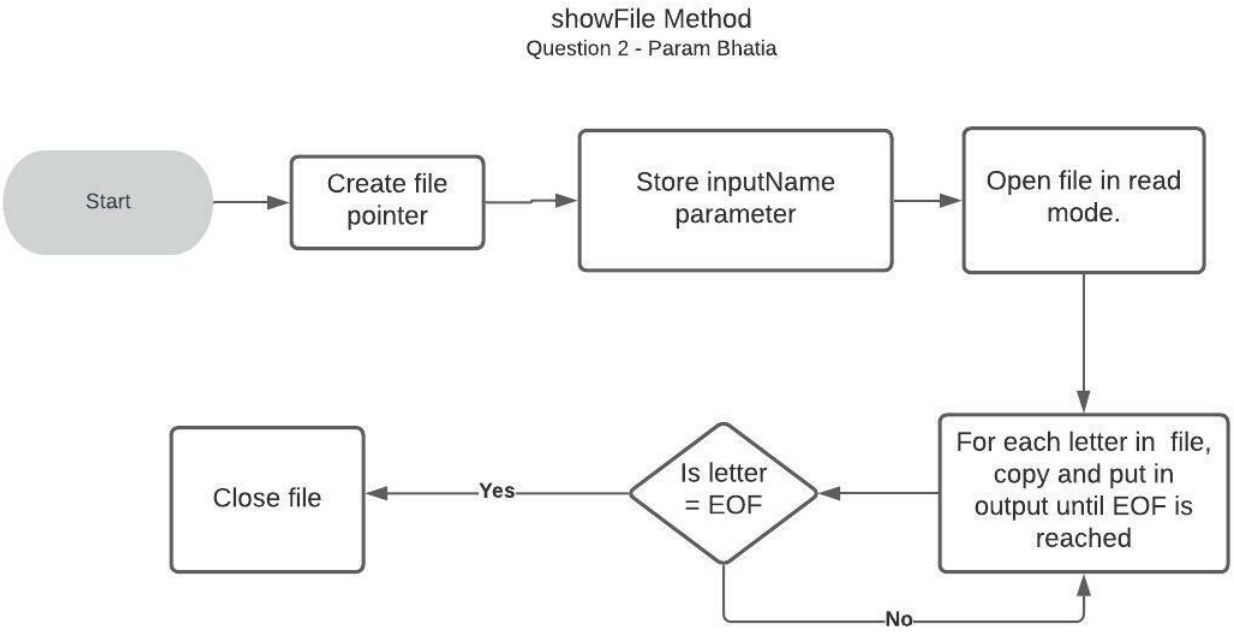
Firstly, the program declares two file pointers, one for the file to copy the contents from, and one to copy the contents into. Then a char array with value ".txt" is also initialised. The program opens the file, using the parameter value, in read mode. The function goes through the text file letter by letter, using the variable 'temp'. Temp uses getc(fptr1), which gets a letter and then increments to the next letter. In between this, the function uses the putchar() function to put the temp value in stdout. After the whole file has been printed to the user, they are prompted to enter a file name for the new file to be created. This name is then read and the file is created, just like in the addFile() function. This new file is then opened in write mode.

```
fptr2 = fopen(input2, "w"); //open file in write mode, in order to create the file first, and also write to it.

letter = getc(fptr1);
while (letter != EOF)
{
    //takes content of file 1 letter by letter, and puts it in file 2.
    putc(letter, fptr2);
    letter = getc(fptr1);
}

printf("The content has been copied\n");        //update user with progress of operation

fclose(fptr1);
fclose(fptr2);
changeLog(input2, "created with content from file listed above");   //update change log
```

Then, the function uses the variable letter to read through each character of the first file until it reaches the end-of-file. Letter uses the getc function in order to read characters from the original file, and uses putc function to paste the same characters until end-of-file is reached. Print to the user when completed. Then the changeLog function is called from here because a new file has been created.

showFile();

Here is a flowchart implementation of the method:



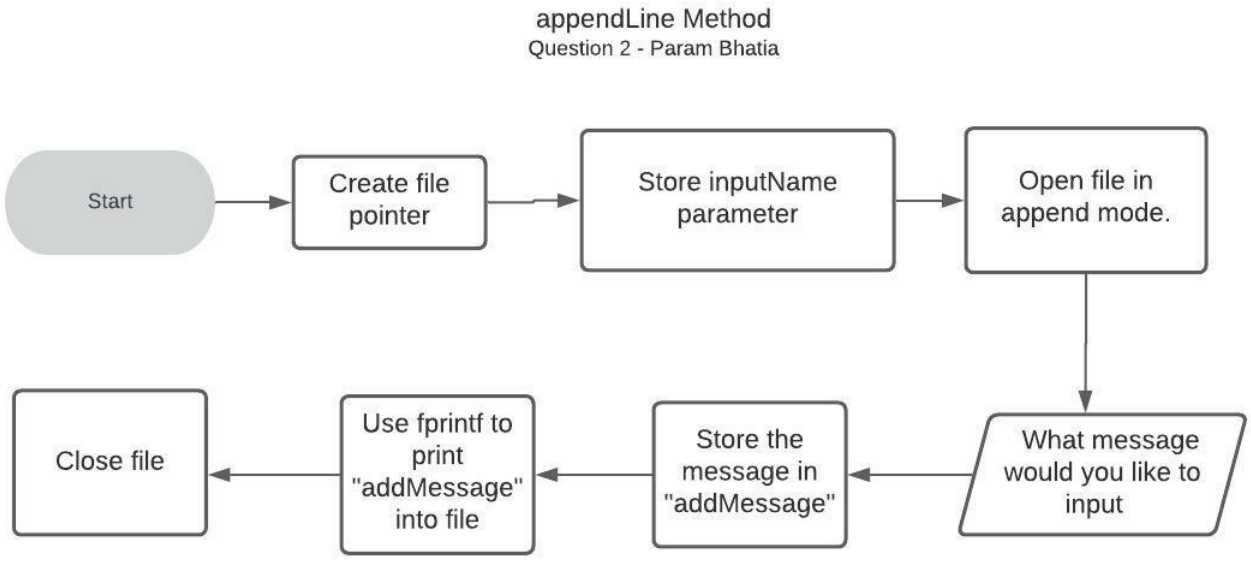
Let us go through the code:

```
void showFile(char (inputName)[MAXLENGTH]){
    FILE *fptr;     //declare file pointer
    char letter;

    printf("The contents of the file are as follows:\n");       //update user with what is
    fptr = fopen(inputName, "r");       //opening the file in read mode, because there is
    letter = getc(fptr);                //using getc is more efficient than fgetc
    while (letter != EOF)
    {
        //takes content of file letter by letter, and puts it in stdout for user to read
        putchar(letter);
        letter = getc(fptr);    //will iterate to the next letter
    }

    fclose(fptr);   //close the file so it can be edited elsewhere in the program.
}
```

The code is very similar to the copyAndCreateFile function. It declares a character variable ‘letter’ and letter uses getc to iterate through the file character-by-character, and uses putchar function to output these characters into the stdout stream.

appendLine();

Here is a flowchart implementation of the method:



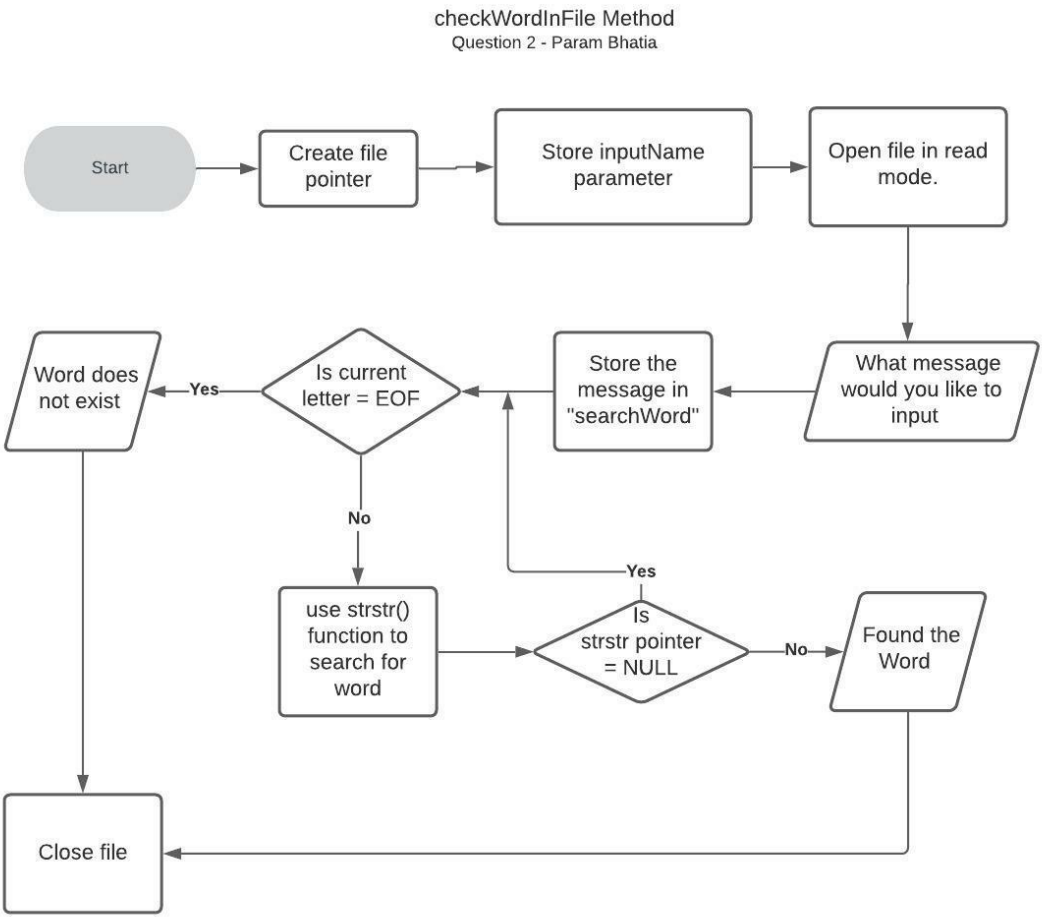
Let us go through the code:

```
void appendLine(char (inputName)[MAXLENGTH]){
    FILE *fptr;     //declaring file pointer
    char letter;
    char addMessage[40] = "";   //initialise char array to be appended

    getchar();      //use to remove trailing new line character(\n) from buffer
    printf("Enter message to be appended\n");
    fgets(addMessage, 40, stdin);   //using fgets allows user to enter data with spaces into the text file
    fptr = fopen(inputName, "a");   //open the file in append mode, in order to add to the existing data

    fprintf(fptr, "%s", addMessage);        //use  fprintf as a printf function, but printing to the file stream.
    fclose(fptr);
}
```

This function firstly declares a file pointer “*fptr”, and also an empty char array of a defined length 40. Then the function uses getchar(). This is because we have previously used scanf() function, and scanf() does not read the trailing newline (\n), it just leaves it in the buffer. However, we are now going to use fgets(), which is a function that stops reading as soon as it reaches EOF or a newline character. Since scanf had left it in the buffer, fgets will read ‘\n’ as the first input and stop. This is why getchar() is used to absorb the \n’. Next , fgets is used to read user input, since we will allow whitespaces in the line of text to be appended. After this, the function uses fprintf() in order to print the data into the file, and since the file is in append mode, the data entered will be at the end of the file, and not overwriting the file.

checkWordInFile();

Here is an implementation of the program in a flow chart:



Let us go through the code:

```
void checkWordInFile(char (inputName)[MAXLENGTH]){
    int maxLineSize = 50;
    int isFound = 1;
    char line[maxLineSize];
    FILE *fptr;     //declare file pointer

    fptr = fopen(inputName, "r");   //open file in read mode
    char searchWord[10];        //max size of word to be searched is 10

    printf("Enter word to be searched\n");  //prompt user to enter word
    scanf("%s", searchWord);
    getchar();  //since scanf is used and fgets is used next, there is a
```

Here the program declares an int called maxLineSize of value 50, and declares a char array with size defined integer variable. Then a file pointer is declared and the file is opened in read mode. Prompts user to enter a word, and user input is read through scanf(), since only 1 word needs to be read. Getchar() function is then used because

we will now use the fgets() function to search the file.

```
while (fgets(line, maxLineSize, fptr) != NULL)
{
    //iterate through the text file line-by-line
  char *pointer = strstr(line,searchWord);      ///using strstr() allows the program to compare strings
  if (pointer != NULL)
  {
      //strtstr return a pointer, and if the word does not exist the pointer is null.
      isFound = 0;      //0 represents success by C standards
      printf("This word does exist in the file\n");     //inform user about update with operation
      break;        //exit the loop once the word is found, in order to make the program time-efficient
  }
}
if (isFound == 1) {
  printf("Your word was not found in the file\n");  //inform user about update with operation
}
fclose(fptr);
```

The program uses a while loop in order to search the file line-by-line. The fgets() function is used to read each line until EOF is reached, and that is when the fgets() value will be equal to NULL. For each line, the strstr() function is used to check if the specified word exists in that line. If it does not exist, then the pointer of the strstr() function will be NULL. If the pointer is not null, then set isFound to 0, in order to leave the loop. Also print to the user that the word exists in the file. If the pointer is NULL after the while loop has been executed, then print to the user that the word does not exist in the file.

Other Functions:

All other functions are very straightforward to understand, and there is a concise and comprehensive set of comments in the program to understand each function. There is no need to go through those function in this specification as it would be redundant, and all the information about the functions already exist as comments in the program.

Justifying Design Decision

The flow charts above have helped me to make my own design decisions throughout the program. These decisions have been justified in the list below:

- My first design decision was to use **scanf()** to read input file names, and not fgets(). This is because scanf() will read only one word, whereas fgets() will be able to read multiple words. Since we are asking the user for file names, I made the decision to **not allow spaces in file names**. This is because we want the program to be user friendly, and having spaces in file names can cause problems in external software. The program should create a file which is **robust** and **does not fail** when used by other software. Having spaces creates problems because it is not supported by all operating systems and/or command line applications, as well as some poorly written shell scripts, as is made clear here**[5]**. Instead it is recommended that users use the camelCase method of naming files, or replace the spaces with hyphens, in order to replicate the effect of having white-spaces.

-Since the program is made to be user friendly, users are informed that when creating new files, **'.txt'** will be added to the **end of the file name automatically**, and that the user does not need to do so. This is because if users had to enter .txt after each file name there would be two problems. Firstly, the program would have to check that there is indeed a '.txt' value at the end of the file name. This will make the program more t**ime and memory intensive**, since it will need to iterate through the last 4 characters of each file added. This can significantly **increase the time** taken to create a file, if tested over a large number of files. Secondly, the user would always have the burden of remembering to add '.txt', and instead, the program has been made so that it reminds the user not to add the '.txt', making the program **easier to use** as they can simply just enter the file name they want.

- When the user has to enter a file name that already exists, in order to check whether that file exists, I have found while testing that it can be a little

problematic. This may be because the user has forgotten the name of the file, or how to spell it, or many other reasons, and they will be forced to explicitly close the program, i.e. by closing their terminal that the program is open in. This can be tedious and time-consuming, so instead I have added a feature when the user finds themselves in a similar position, they are told they can simply type in **"exit"** and the program will close. This is a useful decision to prevent users from being stuck in a loop, and makes the program more **time-efficient**, as users will not need to close the whole terminal.

- The maximum number of characters allowed in a line is **set to 60**. This is shown in the appendLine() and insertLine() functions. The reason behind this is backed by UX research **[6]** and it shows that having between 50-60 characters in a line is optimal in terms of **legibility**, while also keeping the users attention. It is also shown in several papers that the optimal line length in reading should not exceed 70 characters per line **[7]**, since it can put the users off reading the file. This is why I chose 60 as the limit, since it is the **average** of multiple sources put together.

- In this program, I have switched between using **fgets()** and **scanf()**, both valid methods of storing user input. However, I have used each one for a significant purpose. When I need to read only a **single word**, e.g. for the file name, or a line number, I use scanf(). I don't use scanf to read user input when multiple words are possible. This is because scanf() stops reading when it reaches a white-space character, which is problematic. In order to tackle this, I use fgets(), which allowed me to read user **input with spaces**. However, when using fgets(), I have noticed that scanf() has been used in the code previously. This can be a problem because scanf() leaves the **newline** character of an input in the buffer, and fgets() will stop reading the input once it comes across a newline character. This problem can lead to fgets() not reading the user input, but instead reading the newline character. I solved this problem by using the **getchar()** function before using the fgets() function **[8]**, which allowed me to remove the newline character from the buffer, before implementing fgets().

- Another design decision that I implemented is the format to call the **changeLog()** function. It will have two parameters. Firstly, a memory address of a character array, which will point to the **file name** being operated on currently. Secondly, another character array of a fixed length will be passed to the changeLog(). This will contain data on **what has happened to the file**. For example, if the addFile() function was used, with the 'input' array storing the file name, then the call to changeLog will be as follows after the addFile() function:

changeLog(input, "has been created")

This function call is efficient because the operation is passed in as an argument, so the changeLog function does not need to search for which operation was conducted and then add to the log. This is because the change Log is called after every function, with different arguments, so regardless of the operation, the log will always contain all the operations conducted on the file.

- All my operations are **modularised**, i.e they all have their own functions, which is also a design decision **[9]**. It is done in this way to improve code **maintainability** and **readability**. This is because if there is an unexpected error in one of the function, we can easily locate the fault by searching in the function for that operation. Similarly, if a user wants to read the source code, they can easily see that it is split into manageable chunks of code, with concise commenting, allowing the user to follow the logic very efficiently.

- In my code, I have chosen to use the **getc()** method instead of the **fgetc()**, when iterating through a file character by character. The reason for this is that getc() is implemented by **macro** definitions, while fgetc() is implemented by a **function**. This key difference allows the program to be much more **time-efficient**, as calls to a function take longer than implementing a macro definition. This will lead to a large difference in time used between the two function, because on a text file with a large amount of characters, it will be more time-efficient to iteratively call getc() compared to fgetc().

- When iterating through a file character by character, and printing those characters to the user, e.g. in showFile() function, I have used the **putchar()** function instead of the **printf()** function. This is because putchar() has a very simple functionality, to put characters on the screen, which allows the compiler to print all the characters quickly. However for printf(), it is used to print various formats, such as int, long, char etc. This means the compiler will first have to check that we have given the "%c" format specifier, and then print the character form. From this comparison, it is clear that using putchar() makes the program significantly more **time-efficient**, especially for larger files.

- I use a while loop in the main method, which allows users to use the program again, and **conduct several operations** without having to recompile the file. This is a key design decision, since generally users will want to use a text editor for more than one function. For example if we want to read the contents of a text file, and then delete the file afterwards, in a normal command line text editor, you would not have to recompile. This is why my program allows users to conduct **as many operations** as they like. All the operations will be updated in the change log, even if the program is only compiled once.

- The maximum length of a file name is set to **20 characters**, and there is a mathematical reason behind it. I accessed my local files, and found the average number of characters in my file names. This average came to be ~8 characters long. However, very few of my files had multiple words in a camelCase format, so I needed to account for that. Since the average word in English is ~5 characters long, it is a safe assumption that the user file name is less than 20 characters long. This value is defined in the **macro** 'MAXLENGTH' in the header section of the file.

- Throughout the code, the **return value** for a successful function call is 0, and for a failed function call 1. This is accordance with the **GNU C Library**, where it is clearly defined that a success value in C is represented with a 0, and vice-versa.

Justifying my additional operations:

My first additional feature is the ability to **search if a given word exists in a text file** or not. I believe this is a very useful tool to have in a text editor, with several different applications in the real world. Firstly, it is well known that for job openings with several hundred applicants, it is common for hiring managers to scan an applicant's CV/Resume/Cover Letter to search for certain words. For example, if there is a job opening for a 'Machine Learning Scientist", then the hiring manager will likely search through the applicant-provided text files and search for any mentions of 'Python", "Automation", and "Experience". This will allow the hiring manager to screen out applicant who did not not talk about either of the topics mentioned above, since it is a highly competitive role.

My second additional feature is a **word counter**, which is a necessity in all text editors in the modern day and age. This has several different uses, such as if a student is writing an essay on their computer, and they need to have the minimum amount of words, the word counter can show exactly how many words there are in the file. Another common use would be in social media content, where the text may be used to present an idea to a wide audience, however, social media is very fast paced, so the information would need to be short. The word counter can be used to see how many words there are in the social media information text, and the user will be able to keep it under their limit by keeping a note of the word count.

Test Specification:

In this section, we will be testing the input validation when the program asks the user to enter a number corresponding to an operation.

Positive scenarios:
These are examples where the program runs normally without any error messages.

Scenario Types	Input Value	Expected Outcome	Actual Outcome
Numbers between 0-11	0	Call relevant function	Calls the relevant function.
	1	Call relevant function	Calls the relevant function.
	2	Call relevant function	Calls the relevant function.
	3	Call relevant function	Calls the relevant function.
	4	Call relevant function	Calls the relevant function.
	5	Call relevant function	Calls the relevant function.
	6	Call relevant function	Calls the relevant function.
	7	Call relevant function	Calls the relevant function.
	8	Call relevant function	Calls the relevant function.
	9	Call relevant function	Calls the relevant function.
	10	Call relevant function	Calls the relevant function.
	11	Call relevant function	Calls the relevant function.

Negative Scenarios:
These are scenarios where the user has entered an invalid input

Value of Input	Input values	Expected Outcome	Actual outcome
Non-integer	3.4	Error Message indicating it is not a valid number	Error Message indicating it is not a valid number
	-2.3	Error Message indicating it is not a valid number	Error Message indicating it is not a valid number
	31.3	Error Message indicating it is not a valid number	Error Message indicating it is not a valid number

Non-number	Ab	Error Message indicating it is not a valid number	Error Message indicating it is not a valid number
	CDGE	Error Message indicating it is not a valid number	Error Message indicating it is not a valid number

Boundary Scenarios:
Here we test to see if the calculator works as expected near its boundary limit

Value of input	**Input values**	**Expected Outcome**	**Actual outcome**
Greater than 11	12	Error Message indicating number should be less than or equal to 11	Error Message indicating number should be less than or equal to 11
	23	Error Message indicating number should be less than or equal to 11	Error Message indicating number should be less than or equal to 11
	22.5	Error Message indicating number should be less than or equal to 11	Error Message indicating number should be less than or equal to 11
Less than 0	-1	Error Message indicating number should be less than or equal to 11	Error Message indicating number should be less than or equal to 11
	-100	Error Message indicating number should be less than or equal to 11	Error Message indicating number should be less than or equal to 11

As shown by these examples of testing, the program is working as expected, and does not fail in negative test scenarios or boundary test scenarios. No errors exist that cannot be handled by the program, and all errors provide user with a message on how to fix the input.

The program has been tested on Linux and Mac OS, and works exactly as expected.

References:

In this write-up, I have used a number of resources of different types, and a lot of them are peer-reviewed and used as academic literature.

[1] Gnu.org. 1987. *Exit Status (The GNU C Library)*. [online] Available at: <https://www.gnu.org/software/libc/manual/html_node/Exit-Status.html> [Accessed 24 January 2022].

[2] Kernighan, B. and Ritchie, D., 2011. *The C programming language*. Englewood Cliffs, N.J.: Prentice-Hall.

[3] Fraenkel, A., 2000. *Abstract set theory*. Burlington: Elsevier Science.

[4] Dot Net Tutorials. 2022. *Static vs Dynamic Array in C/C++*. [online] Available at: <https://dotnettutorials.net/lesson/static-vs-dynamic-array/> [Accessed 24 January 2022].

[5] Scott, J., 2022. *Library Guides: DataShare: ISU's Open Research Data Repository: File Name Recommendations*. [online] Instr.iastate.libguides.com. Available at: <https://instr.iastate.libguides.com/datashare/filenames> [Accessed 25 January 2022].

[6] Holst, C., 2022. *Readability: the Optimal Line Length — Articles — Baymard Institute*. [online] Baymard Institute. Available at: <https://baymard.com/blog/line-length-readability> [Accessed 24 January 2022].

[7] Nanavati, A., 2005. Optimal Line Length in Reading--A Literature Review. *ERIC*,.

[8] Stevens, W. and Rago, S., 2014. *Advanced programming in the UNIX environment*. Upper Saddle River, NJ: Addison-Wesley.

[9] Schildt, H., 1992. *Turbo C*. Berkeley: McGraw-Hill.