

CS132 Computer Organisation and Architecture

Coursework 2

Selin Kayay

February 4, 2021

Contents

Exercise 1	3
1.1 Problem Statement:	3
1.2 Analysis and Algorithm	3
1.3 Solution	5
1.4 Testing	8
1.5 Optimization Routes	10
Exercise 2	11
2.1 Introduction	11
2.1.1 Research	11
2.1.2 Purpose	12
2.1.3 Scope	12
2.1.4 Assumptions	12
2.2 Requirement Analysis	13
2.2.1 Functional Requirements	13
2.2.2 Non-Functional Requirements	14
2.3 Design	15
2.3.1 User Interface : Feature #1	15
2.3.2 File Operations : Feature #2	17
2.3.3 Line Operations : Feature #3	19
2.3.4 Change Log : Feature #4	21
2.3.5 Spelling Checker : Feature #5	21
2.4 Implementation	24
2.4.1 The Key Input Processing Component	25
2.4.2 The Display Component	30
2.4.3 File I/O Component	37
2.4.4 The Editing Buffer	38
2.4.5 Editing/Control Component	48
2.4.6 Change Log	54
2.4.7 Spelling Checker	55
2.5 Testing	63
2.5.1 Feature #1	63
2.5.2 Feature #2	64
2.5.3 Feature #3	65
2.5.4 Feature #4	66
2.5.5 Feature #5	66
2.6 Evaluation	67
2.7 Conclusions & Optimizations	70
2.8 Appendix	71
2.9 References	75

Exercise 1

1.1 Problem Statement:

Find the sum of all possible positive, non-zero integers such that their digits add up to a given sum. E.g. for a given sum of 3, the following set of integers can be produced

$$\{3, 12, 21, 111\}$$

which sums up to the value 147.

1.2 Analysis and Algorithm

Brute force approach:

This problem could be visualized as a tree such that it's easier to come up with a working algorithm.

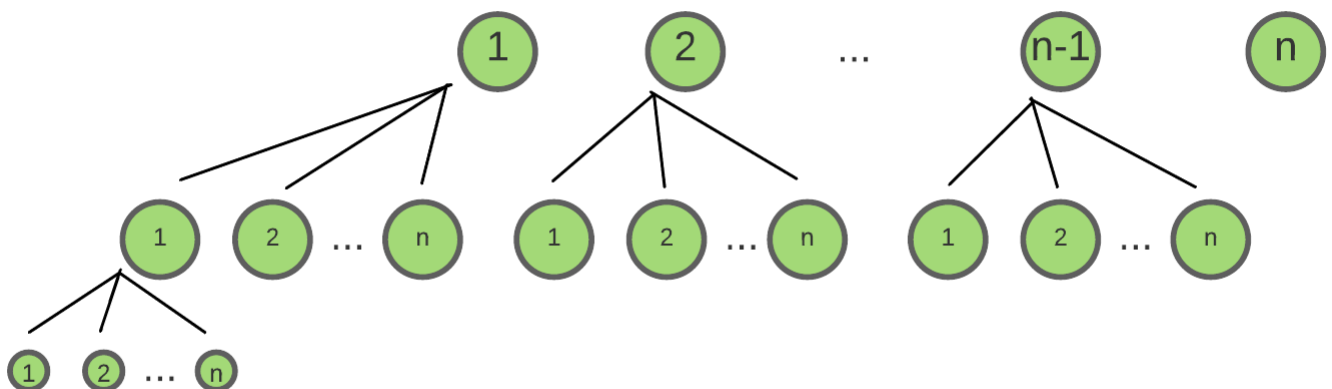


Figure 1: Recursive tree representation of the solution.

As seen above, for a given sum there will be 1 to n nodes where $n \leq 9$ and this will continue recursively and solutions will be found when the current total that is made up of the digits in the tree result in a difference of 0 with the given sum.

This concludes bits of information that will ultimately draw the algorithm required:

Choice	The decision space of this solution lies in between 1 and 9 for each digit in the result combination string.
Constraints	The sum of the combination strings must be less than or equal to the given sum.
Goal	Produce digit combinations for each n, where n is the combination length ranging from 1 to the value of the given sum.

Figure 2: The algorithm requirements.

Further Constraints for Optimization

This function calls itself for every possible digit to be considered. Once the created combination of digits is the same as the given sum and length of digits, then the result is added to a variable, `finalsum`.

If this recursive function was called for every n , such that every number of digits up to the size of the given sum is considered. The solution would be complete.

However, the approach above iterates through every possible digit even if the sum of those digits would be greater than the given sum. This is unnecessary and can be removed with a simple checking if statement such that checking happens as so:

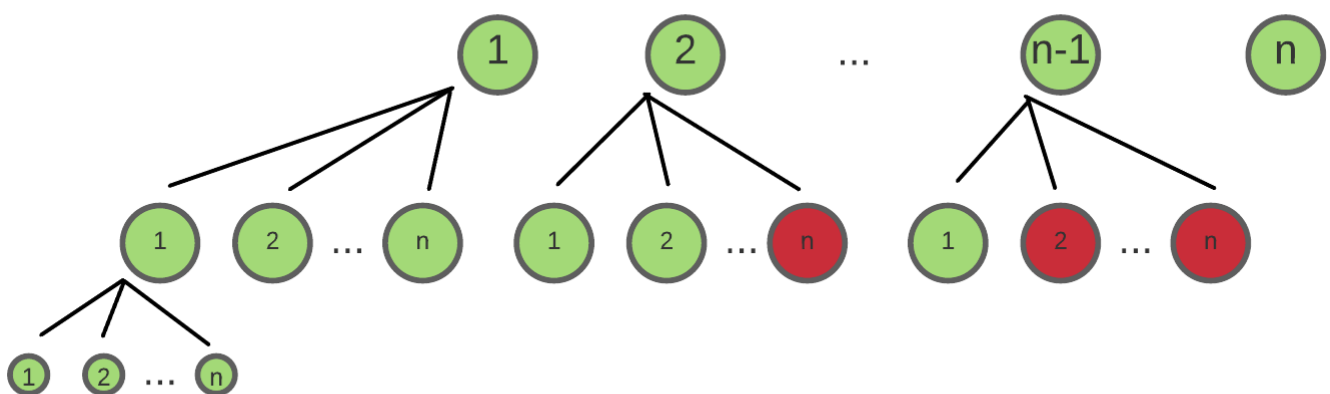


Figure 3: Recursive tree representation of the solution.

This means, as the possible 'root nodes' are considered, if the 'child nodes' being tried out in the combination sum up with the root node to yield a value greater than the given sum, then iterating further will be futile.

Time Complexity Analysis:

Considering every possible value and disregarding constraints considers 9 iterations (1-9) for every digit in the combination of digits. This is a complexity of $O(9^n)$ for every n (length of digit combination).

With this, the current time complexity can be reduced from $O(9^n)$ to $O(m^n)$ which isn't the optimal solution but reduces the number of iterations overall, as long as the given sum isn't ≥ 9 then the reduced complexity remains, otherwise nothing changes. m is number of considered digits, dependent on the given sum.

Template Solution

Algorithm 1 Algorithm

```
Require:  $n = 1$ 
Require:  $sum > 0$ 
Require:  $index = 1$ 
Require:  $result[]$ 
0: procedure FINDDIGITS( $result[], index, n, sum$ )
0:   if  $sum \geq 0 \ \&\& \ index < n$  then {// CASE 1}
0:      $char \ digit = '1';$ 
0:     while  $digit \leq '9'$  do
0:        $result[index] = digit;$ 
0:        $findDigits(result, index + 1, n, sum - (int)digit);$ 
0:     end while
0:   end if
0:   if  $index == n \ \&\& \ sum == 0$  then {// CASE 2}
0:      $finalsum += (int)result;$ 
0:   end if
0: end procedure=0
```

To avoid redundancy of text, this algorithm will be explained further, for the commented cases, in the section below.

1.3 Solution

The below is the wrapper for the recursive function. For every n , length of the sought combination of digits, this function will iterate over the set decision space until some combination of strings that fit the constraints of the problem are found.

Case #1 is where the solution is not yet found and case #2 maps to a found solution, hence the base case. Another identified edge case was pointed out on page 4 (Figure 3), where combinations being made exceed the given sum. This edge case results in the current stack frame being popped and the solution backtracking to the previous stack in the decision space to keep seeking further solutions.

This edge case will occur within Case #1.

```
void findDigits(char result[], int index, int n, int sum)
{
    /** Case 1 **/
    if (index < n && sum >= 0)
    {
        ...
    }

    /** Case 2 **/
    else if (index == n && sum == 0) {
        ...
    }
}
```

Case #1

```
char digit = '1';

/** Loop through the decision space 1 to 9 */
while (digit <= '9')
{
    /** Append a digit */
    result[index] = digit;

    /** Get the int value of the digit */
    int intdigit = (digit - '0');

    /** Skip useless iterations, backtrack to last
     * stack frame */
    if ( (sum-intdigit) < 0) break;

    findDigits(result, index + 1, n, sum - intdigit);
    digit++;
}
```

This case is when a solution string has not yet been found. Firstly, a choice is made from the decision space. This choice is appended to the `result` string. The edge case constraint is checked, does the sum of the digits in `result` exceed the given sum? If so the stack frame is popped and the algorithm backtracks. Otherwise, a recursive call is made in order to prompt the algorithm to make yet another choice from the decision space.

For each stack frame the entire decision space will be searched, unless the constraint is met. The constraint will be met depending on the given sum. Smaller given sums will meet the edge case more hence runtime will be faster.

Case #2

```
/** Get the integer value of the result
 * string */
long ans;
sscanf(result, "%ld", &ans);

/** Add it to the finalsum, f(n) */
finalsum+=ans;
```

This case is when a solution is found, also known as the base case of the recursive algorithm. If a solution string is found, this string needs to be casted to an integer type and added to a `finalsum` variable. The `finalsum` variable might need to store very large integer values as the given sum increases. Therefore it is of type `unsigned long`; this allows for very large positive numbers to be stored.

`findDigits` doesn't return anything. It only looks for possible solutions and sums them into a global variable `finalsum`.

Main flow

```
int main()
{
    int n = 1;      /** Number of digits **/
    char result[n+1]; /** Combination strings **/

    printf("\nExercise 1 Calculator for f(n)\n");

    /** Prompt user: to get a given sum value
     * in between 1 and 19 **/
    int sum = prompt();

    /** For n digits that range from 1 to the
     * value of the given sum, call findDigits **/
    while (n<=sum){
        findDigits(result, 0, n, sum);
        n++;
    }

    printf("f(n)=%ld\n", finalsum);
}
```

Briefly:

1. Initialize the parameters passed to findDigits.
2. Prompt the user to provide a given sum.
3. Iterate over n and call findDigits.
4. Print out the output of the function.

Inputs and Error Handling

Lastly, the inputs to the program need to be constrained. Since a set range wasn't requested in the problem statement. The range of valid inputs has been set in the range 1 to 19 inclusive. This simply due to the memory requirements to store the results for larger inputs being a lot more demanding than unsigned long types. The range could be expanded upon request from the specification.

Unexpected inputs such as characters, negative integers, and positive integers out of range are rejected by the program. The user is re-prompted whenever such a case occurs. Unexpected inputs are flushed out of the input buffer with getchar.

```
int prompt()
{
    int sum;
    do {
        printf("Enter n (1 to 19): \n");

        /** Get user input, if not of expected type
         * consume standard output with getchar **/
        int errorCode = scanf("%d", &sum);
```

```

    if (errorCode!=1) {
        printf("Invalid input. Please enter an integer from 1 to 19.\n");
        while(getchar() != '\n') {
            continue;
        }
    }
} while (sum<1 || sum>19);

return sum;
}

```

1.4 Testing

The below are a sample of the tests conducted for this program. All of the 5 below encapsulate all the expected valid and invalid input types and shows how the program deals with them. Testing has been conducted throughout development and these are the results of the final outputs of the program before the end product.

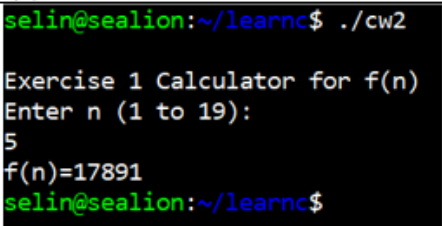
Test	When a user inputs the given example (n=5) the result is the same as given.
Input	n = 5
Expected Behaviour	f(5) = 17891
Resulting Behaviour	 <pre> selin@sealion:~/learnnc\$./cw2 Exercise 1 Calculator for f(n) Enter n (1 to 19): 5 f(n)=17891 selin@sealion:~/learnnc\$ </pre>

Figure 4: Test 1

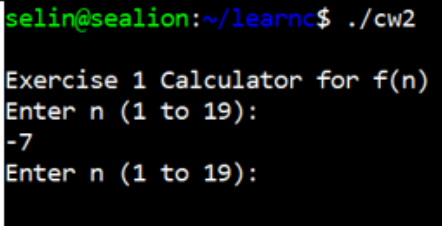
Test	When the user inputs a negative n, it is rejected.
Input	n = -7
Expected Behaviour	The user will be re-prompted to enter a valid input.
Resulting Behaviour	 <pre> selin@sealion:~/learnnc\$./cw2 Exercise 1 Calculator for f(n) Enter n (1 to 19): -7 Enter n (1 to 19): </pre>

Figure 5: Test 2

Test	When the user inputs (a) character(s), it is rejected.
Input	n = "hello"
Expected Behaviour	An error message, "Invalid Input..." will be printed and the user will be re-prompted.
Resulting Behaviour	<pre>selin@sealion:~/learnc\$./cw2 Exercise 1 Calculator for f(n) Enter n (1 to 19): hello Invalid input. Please enter an integer from 1 to 19. Enter n (1 to 19):</pre>

Figure 6: Test 3

Test	When the user inputs a value of n, out of the given range, it is rejected.
Input	n = 20
Expected Behaviour	The user is re-prompted.
Resulting Behaviour	<pre>selin@sealion:~/learnc\$./cw2 Exercise 1 Calculator for f(n) Enter n (1 to 19): 20 Enter n (1 to 19):</pre>

Figure 7: Test 4

Test	When the user inputs any valid integer, followed by some invalid characters, the valid number is processed as expected. (scanf reads stdin byte by byte)
Input	n = 5"hello"
Expected Behaviour	f(n) = 17891
Resulting Behaviour	<pre>selin@sealion:~/learnc\$./cw2 Exercise 1 Calculator for f(n) Enter n (1 to 19): 5hello f(n)=17891 selin@sealion:~/learnc\$</pre>

Figure 8: Test 5

1.5 Optimization Routes

DFS with proper Backtracking:

It can be seen that the solution above has a best case of $O(m^n)$ where $1 \leq m < 9$ and a worst case of $O(9^n)$ for every n . How can this be optimized?

With the tree visualization, it's clear to see that this hypothetical tree could be traversed depth first, starting with the largest possible value of n for every root node and backtracking to $n=1$. This will essentially create all possible solutions for each root node at once. This should decrease the time complexity by a large factor.

The code above will call the recursive function for every number of digits up to the given sum and the entire 'tree' will be iterated for solutions of that length. This isn't very efficient since previous iterations are re-calculated. A dynamic approach could be taken where previous solutions could be backtracked with an implementation of some bottom-up or top-down approach. [1]

Dynamic Programming is one of the elegant algorithm design standards and is powerful tool which yields classic algorithms for a variety of combinatorial optimization problems [2]. Some well-devised algorithm design standards - such as Divide and Conquer, Greedy Methods, Graph Exploration, etc, yield classic algorithms for a range of optimization problems. Such solutions usually create a time-memory trade off either due to the growing size of the call stack during execution or intentional uses of techniques like memoization.

In this context, the trade off could be used in the sense that instead of scrapping the solutions for each n digit combination of `result []`, it could be stored. For example, for some digit from the decision space explore all possible solutions starting with that digit for n digits such that there aren't n iterations of the entire tree for every length of combination. This would grow the size of the stack frames in memory quite quickly, especially considering this wouldn't be an implementation of tail-recursion. However, larger ranges of inputs could be worked with in shorter amounts of time when the number of iterations are cut down.

Exercise 2

2.1 Introduction

The goal of this document is to describe, present, and explain the process of creation of a UNIX-based command-line editor.

A 'command-line' refers to a 'text-based interface for a computer'. The main purpose of the command-line is to allow the user of a computer system to navigate and manage computer files. A command-line text editor is a software program that helps the user in their management of these files. To be more specific, in this context one would care about what 'text' is and how it will be possible to allow users to access this specific format of text in the form of files; with capabilities like displaying them and manipulating their contents. (see 2.8)

Background

Currently, some of the popular command-line editors, according to Google's search engine, are vim, GNU emacs, GNU nano, etc. The reasoning behind these options are often the features that these editors provide over others such as pico (the parent of GNU nano). Some of the most useful functionalities of a command-line editor included, but not limited to, extend from find and replace, language-specific syntax highlighting, writing on multiple lines at once to auto-correcting (spell/syntax checking).

In this document, the final product will likely be similar to the editors currently in the market and most likely quite lacking in terms of features; however, this editor could be a template for future iterations.

A final question to answer is: why write this editor in C? (see 2.8)

2.1.1 Research

The command line interface (CLI), used mainly for navigation of the computer's file system, will always require a good text editor. One of the commonly preferred editors is vi; which has been around almost since UNIX began. Since, an improved version, vim, has been put on the market. The main differentiator is the feature-rich nature of the editor; such as, multiple windows, syntax highlighting, HTML support, etc. [3]

Most UNIX editors don't really differ and the creativity that can be leveraged from a CLI is simply limited with the available resources. However, considering the recent advancements in the fields of Artificial Intelligence (AI) and Machine Learning (ML), specific domains like Natural Language Processing (NLP) look promising for both the future of a smarter CLI as well as text editors to follow up. Recent advancements and notable research are currently in progress; e.g. IBM's project CLAI [4]. With the emergence of large-scale networked terminals, there is room for improvement in the CLI experience. Capabilities include automation of generic tasks and natural language support. This could include, retrieving the actual command line syntax for a natural language question such as "How do I compress a file?"; this is a prime example of the power of NLP.

A notable use of similar capabilities, integrated with CLI text editors has been the GNEWSYS-Emacs ontology editor [5]. This text editor, provides RDF editing in plain text from the CLI. Such a project further pushes the possibility of semantics on the command-line.

Towards the main point of this document, the future of the CLI and respective text editors lie in the hands of, mainly NLP and ML. Natural language on the command line could be extended to editor commands. Future work could skew towards different types of inputs to command-line editors such as speech. Since the data being worked with is text, features such as spelling checkers, auto-correct, auto-syntax, and other similar text tools will be the most useful considering the users point of view.

The editor produced in this document will implement a simple spelling checker. While AI, ML, semantic language models, and ontologies are out of the scope and too complex for the given time; instead, basic dictionary lookups with some efficient data structure will suffice.

2.1.2 Purpose

This editor, like many-other UNIX command-line editors, must provide a command-line user interface and allow the user to display and operate on files in their current working directory. This will be further decomposed in the Scope as well as Requirement Analysis.

2.1.3 Scope

The scope of this editor can be explained with a division into 3 overlapping sections.

1. File operations: Must provide tools for the user to operate on files in their current working directory with operations such as creating, copying, deleting, saving, and displaying them on a user-interface.
2. Line operations: Given a file, or not, must allow the user to edit a buffer real-time with operations such as inserting, deleting, and showing lines as the user writes them on the buffer.
3. General Requirements: It must be possible for the user to be able to view an automatically created change log for each file they edit and save with this editor. As an extra feature, it will be possible for the user to run a spelling checking feature on the files.

2.1.4 Assumptions

- Assuming that the program specification did not require a user interface with a menu and real-time editing; adding such a feature has been considered as an 'addition'. This means that **the user interface is created as part b of Exercise 2**. This goes along with the other additional feature, the spell checker.
- The implemented spell checker only detects English words. Hence some assumptions are made when picking 'valid' words. One of these are that the longest possible word is of length 45 and there are 26 possible nodes for every letter of a word.

2.2 Requirement Analysis

2.2.1 Functional Requirements

Feature number	Specification	Specific objectives
1	The editor must provide a user interface such that the user is able to display, modify, navigate through files, and exit the interface when done.	<ul style="list-style-type: none">a) It should be possible to enter raw input to the user interface at real-time.b) It should be possible to process these key inputs and control the editor.c) It must be possible for the user to move their cursor to navigate the interface in the form of scrolling horizontal and vertically.d) The editor must present a user interface with a status bar that shows the number of lines read to the editor buffer.
2	The editor must allow the user to operate on files in their current working directory with capabilities such as creating, displaying, copying, saving, and deleting them.	<ul style="list-style-type: none">a) The editor must load files into a temporary buffer to be displayed on the interface. Given no file, an empty buffer must be provided to allow the creation of a file.b) It must be possible to refresh the buffer after every modification.c) It should be possible to display, save, copy, and delete files through control inputs to the interface.
3	The editor must allow line operations on files with specifics such as inserting, deleting, displaying, and appending of lines.	<ul style="list-style-type: none">a) It must be possible to insert characters, new lines, and append lines to a file.b) It must be possible to delete characters as well as entire lines from a file.
4	The editor must create a change log for each file edited with the editor.	<ul style="list-style-type: none">a) The change log must contain all versions of that file, after it is saved.b) The change log must contain the total number of lines written for each version saved and timestamps.c) The change log must be accessible through a command line flag.
5	The editor must provide a spell check feature.	<ul style="list-style-type: none">a) Spell checking should be applied through a control key.b) The spell checker must highlight misspelled words on the buffer.

Figure 9: Specific functional requirements and some objectives.

2.2.2 Non-Functional Requirements

- **Reliability:** The user must be able to use the editor end-to-end with no sudden, unknown, crashes. If errors occur, they must be caught, and appropriate messages must let the user know of what they did and hence the program was unable to behave as expected.
- **Portability:** The editor must be compliant to POSIX standards. It is not expected for this editor to be portable to other systems such as Windows due to the low-level systems programming involved that is specific to the interface of POSIX compliant systems.
- **Usability:** The editor must provide appropriate information to the user, presented on the editor user interface throughout the process. The control keys available for use must be specified and should 'make sense'. Any command-line flags must be explained for ease of use. In addition, at exit the initial state of the command-line must be restored.
- **Maintainability:** The developed code must be sectioned, documented, compliant to the traditional way of structuring C code. This will, iteratively, be tested with Code Quality Monitoring Software to obtain metrics on the maintainability factor of the end product.

2.3 Design

This section will iterate over each functional feature in Figure 3 and explain the design of the specific objectives given. This is the high-level structure of the program.

(See 2.8 for justification of design decisions)

2.3.1 User Interface : Feature #1

Feature 1.a: Interacting with the user interface in real-time

This refers to processing every byte that the user inputs to the program immediately and not preceding an Enter key. In more technical terms, the mode of input is byte by byte rather than line by line (followed by a newline character).

In POSIX systems there are two main modes of input, canonical and non-canonical. (see 2.8). The desired mode is non-canonical.

By default, the command-line is canonical. To enable non-canonical mode on the UNIX terminal, a header file called `termios` can be used. `termios` is a structure which essentially provides a general interface to asynchronous communication devices.

Using a `termios` structure to edit the mode of input of the terminal is done through specific "flags" that are implemented as bitmasks. These can be "disabled" or "enabled" for each flag field for an instance of the `termios` structure in the editor program. It happens that the non-canonical mode of input is simply enabled by disabling the default canonical mode of input. (see 2.8 for more information on modes of input and the `termios` structure)

Feature 1.b: Input and Processing

After enabling raw input how will the user be able to "interact" with this interface?

The program will be required to read the standard input stream until there is an input. According to the GNU C Library docs [6] it is possible to make use of functions such as `read()` and `write()` in order to read and write to file descriptors. File descriptors provide a primitive, low-level interface to input and output operations in order to communicate with a device, pipe, or socket.[7].

The `read()` function returns error codes hence it is possible to check `stdin` iteratively until `read()` returns a successful code.

After a byte is read it must be differentiated and then passed to some other function to process the input specific to the interface being designed.

How will bytes be differentiated?

- Escape/Control inputs
- Any other input like upper and lower characters as well as ASCII number codes.

Escape/Control inputs refer to the ESC key. ESC keys are usually the first byte of any ANSI escape code; these escape codes are used as commands in the terminal for the control of i.e. cursor location, font styling, etc... [8] (see 2.8 for ANSI codes used and their meanings)

Tracking of the cursor for navigation and future line operations will depend on these control sequences that could be written to standard output through the `write()` function introduced earlier. (see 2.8 for a table of ANSI sequences to be used for the navigation of the user interface).

CTRL-Q	Quit the editor.
CTRL-X	Display the help message.
CTRL-S	Save the current buffer.
CTRL-K	Delete the currently selected line.
CTRL-H	Delete a character.
CTRL-F	Run a spell check.
CTRL-C	Copy the current file.
CTRL-D	Delete the current file.

Figure 10: The possible control keys for the editor, mapping to various functions.

It will be possible to use the arrow keys for movement in all directions.

This table represents how the user would navigate and work with the system. These keys must be processed real-time and the results must be observed immediately.

Feature 1.c: Status and Message bars: 'Drawing' the user interface

The end result must be something like the design below.



Figure 11: Initial terminal design.

In order to create such a screen for the user, the same approach of dealing with ANSI escape codes [8] can be used.

There are specific codes to clear the screen, invert terminal color for a status bar, and deal with the cursor positions. Briefly, these codes must be written to standard output in order such that the visual above can be achieved.

It will be required to get the window size of the terminal for the screen of the user. This is because it must be possible to keep track of the cursor and draw the status bar according to this size. According to the GNU C Library docs [6] , this is possible through the use of the `ioctl()` function and a structure called `winsize`. (see 2.8 for more on this)

2.3.2 File Operations : Feature #2

This feature will aim to provide the created user interface with file operation capabilities. To be specific, the editor will be able to load in files to the interface from current working directory or just open the interface, modify it and be able to save it as a file. There will be some additional features like copying and deleting files available through both the control keys from the interface.

Feature 2.a: The editor buffer

A 'buffer' is a memory area, allocated for a specific, temporary purpose. In this case, the editor will be required to allocate a buffer for the user to store text data in. Why create a buffer and not just interact with the already existing stdin and stdout?

The need for a buffer

Without a buffer, the other option is to directly `write()` to stdout. This won't work 'well' because this program strives to provide a user interface; as discussed, in order to create this interface ANSI codes will be used and these will be written in order to the interface. If these codes as well as the input of the user was written directly one by one, not only would there be a lot of calls to `write()` but also there would be a visible 'flickering effect' as the function writes.

With this, an informed decision can be made to provide a buffer to write to temporarily, and then simply write the buffer to the screen once.

Creating a buffer

A buffer is simply an allocated memory location. This can be created as a `struct` in order to store the pointer to the location as well as the size of the buffer; which is required when writing the buffer to stdout.

Feature 2.b: Refreshing the screen for an interactive user interface

The objective to provide the user with an interactive screen that gives immediate feedback will be fulfilled with this feature. Essentially, the editor program's user interface consists of the call to two main functions. These will be to refresh the screen and process a key input. This means that the editor will simply process user inputs and display them by refreshing the screen.

Assuming that at this stage the buffer contains all the data required to create the screen the refresh screen method will effectively just write this to standard output. Hence, if it is desired to have an interactive screen as the user writes, it is necessary to write to this buffer; shortly after the write just call this refresh function.

Feature 2.c: Writing to the buffer and Displaying the buffer

Row structures to organize lines of text

The buffer displayed on the screen can be visualized as a chessboard. The cursor is the moving piece. Hence, it makes sense to create some row structures to store in the buffer such that every line the cursor moves to is an organized line of data stored in the buffer in a specific

location.

This row structure is basically a dynamic string, hence the data required to be held is:

- The size of the row
- Pointer to the row

With this, every row is a dynamic string and these rows can be appended to the buffer, for the stored strings to be displayed.

Writing to this buffer consists of copying the required data to where the buffer is located in memory and updating the size of the buffer. These operations will require some common functions that deal with memory management and copying to memory like `malloc`, `realloc`, `memcpy`, `free` etc... As already discussed displaying this is a simple `write()` to `stdout`.

At this stage, the buffer will provide for the 'canvas' where the user will be able to enter data (how this data will be entered and what it will be will be discussed further in Feature #3 below.) This deals with the displaying of files.

Saving an edited buffer or prompting for a filename will also be required as a part of this feature's functionality.

These features can be provided as so:

- A buffer will be deemed as 'edited' if throughout the program the user inputs a character (not a control character).
- If a buffer has been edited, upon quitting the user will be prompted to save and if the buffer has no name a name must be requested.
- In order to save, the buffer contents will overwrite the specified file; if a file didn't exist previously it will first be created with the given name.

File operations in C

Specific to this feature a couple common file operation functions will be required. C provides functions to create, open, read/write to files. Files are accessed through file descriptor or pointers [7]. Writing and reading can be done through these values with functions i.e. `getLine`, `write`, `read`, `(f)getc`, `(f)putc`, etc...

With this being said, copying and deleting files is done through such functions.

Copying

- Given a source and destination filename, file descriptors/pointers are created with the `(f)open` functions. The source will be read, destination will be written to.
- Iterate over the source file and put every byte read into the destination file with `(f)putc` and `(f)getc`.
- Clean up file pointers and handle for file errors.

Deleting files can be as quick as a simple call to the function `remove()` given the filename of the file to remove.

Again, the assumption that all these operations are being done in the current working directory remains.

2.3.3 Line Operations : Feature #3

This feature will provide the user with the ability to 'manipulate' and actually edit the files.

Feature 3.a: Inserting characters and new lines to specific positions

As discussed previously, the cursor can be manipulated with ANSI escape codes. The cursor positions can be stored as coordinate values of the entire screen. Hence, with access to the cursor coordinates, data can be inserted and deleted from where the user points to with their cursor.

When the user inputs a non-escape sequence character at a specific cursor position the behaviour of the screen will differ based on the position the user requested to insert at:

- If the current position is somewhere in between the beginning and end of a row string then insert where the cursor points and move all the data to the right by 1.
- If the current position is at the tail of a row, just insert the character with a newline character following it.

If the user inputs a newline:

- If the current position is somewhere in between the beginning and end of a row string, then move all the data on the right of the cursor one row down.
- If the current position is at the head of a row, move the entire row one row down.
- If the current position is at the tail of a row, just move one row down.

When newline characters are used to create a new row, a new row structure must be created and appended to the buffer.

These behaviours are the common behaviours that popular text editors exhibit. Such a feature will satisfy the set objective of inserting to the editor at specific positions.

Appending lines

The specification explicitly suggests an 'append' operation where some text can be appended to the end of line. This operation is clearly distinct from the insert line operation at specific positions.

It makes sense for this operation to be available from the command line as an argument. This way the specification is simply met in both fields.

The command line can receive arguments at the main method arguments `argc` and `argv[]`. An appropriate command line flag can be specified for the append line feature, along with the other features (discussed previously) such as the copy files and delete files features. Additionally, it makes sense to provide a `-help` flag to explain to the user the usage of these flags when required.

```

1 Usage editor [OPTION] [FILE, ...]
2
3 Option                                Meaning
4 --delete <filename>                  Delete a specified file.
5 --append <filename> <string>        Append to a specified file.
6 --show-change-log <filename>        Show the change log/versions of a specified file.
7 --copy <source> <dest>              Copy a source file to a specified destination file.

```

Figure 12: A file to be loaded when the `-help` flag is sought. Explains the usage of the command line arguments available to the editor.

Feature 3.b: Deleting characters and entire lines at specific positions

Deleting at specific positions also entails the use of the cursor coordinates. The behaviour will differ though:

- If the user presses delete at the head of a row, data stored in the entire current row must be moved up by 1. The rest of the rows preceding them must also follow and move up by 1.
- If the user presses delete in the middle of a row the character can be overwritten with the rest of the row overlapping it. If the user presses delete at the tail of a line the newline character will overlap it.

Additionally, it should be possible to delete entire rows at once. This can be provided through control keys. In this editor, the CTRL-K key is reserved for deleting entire rows.

Effectively, given a specific row, by the cursor, the contents of the entire line are wiped and the rows preceding the deleted one all move up by 1. This brings by ease of use for the user since it is very quick and easy to delete multiple lines at once through a single key input.

2.3.4 Change Log : Feature #4

The change log is essentially a version tracker for each file edited with this editor. The required functionality of this feature will be to map each file to a log file, within this log file each saved version of the edited file will be appended along with the number of lines written after each modification.

Access to this feature will be provided both through the command line as well as a menu option through the main user interface.

Creating the log files

After every save of a file with this editor, the editor should make a call to some other function that works to create a log file for the current buffer, if doesn't already exist, and append the contents of the buffer to the log file along with information as to how many lines were written each time.

This could look something like so:

```
1 #####
2 first line
3
4 !
5 Total lines written: 1
6 #####
7 #####
8 first line
9 second line
10
11 !
12 Total lines written: 2
13 #####
```

Figure 13: Change Log Design

This will provide the user with versions of edited files and will allow them to go back and be able to view them.

2.3.5 Spelling Checker : Feature #5

A spell checking feature can differentiate and add complexity to this editor. This feature would need to check each word of the editing buffer, when requested through control keys, against a dictionary. Where words are not found, they must be classified as 'misspelled'. Then, these words must be highlighted on the terminal screen for the user to see in order to meet all the functional requirements. This feature will mainly overlap with the Display component. However, overall it needs to be tested manually as an individual component.

Feature#5.a: Checking for Spelling Errors

1. When requested by the user to apply a spell check, iterate over all the current rows in the buffer and send them to a spell checking feature one by one.
2. The spell checking feature must first load a dictionary into **an efficient data structure**.
3. After loading, each given row must be iterated over and checked against the data structure.
4. If a word is not found in the data structure, it is misspelled.

5. For each misspelled word, its start and end index in the passed row must be returned.

The chosen data structure

In order to lookup words against a dictionary, a dictionary must first be loaded into memory such that traversal or lookup is efficient throughout its use. Two common structures for this feature are hash tables and Tries [cite]. In terms of search operations, both structures will support $O(n)$ time operations on average where n is the length of the searched keyword. However, Tries are in fact usually faster than hash maps. This is because Tries represent every node in order with an array of pointers, this makes every node uniquely retrievable hence there is no need for something like a hash function and neither is there a risk of collision. (see 2.8 for more information on the advantages and disadvantages of Tries)

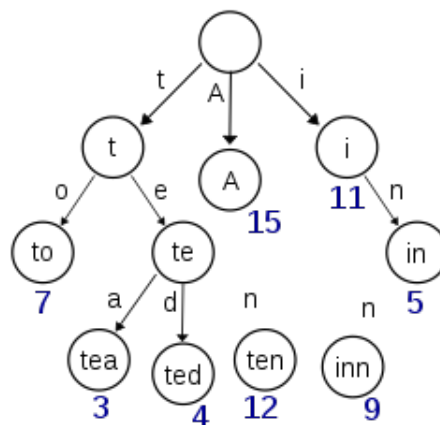


Figure 14: A trie tree example.

This feature will make use of a Trie structure for searching. As seen above, the Trie will store a bunch of strings from a dictionary where each node will have two main components; an array of pointers to the entire alphabet and a value. This value can be a Boolean value indicating whether the end of a word is reached or not. After the entire given dictionary populates the Trie, for given words the Trie can be traversed. If during traversal, a pointer to a NULL node is encountered; this means that the word must be misspelled as it doesn't exist in the dictionary. [9]

After finding this word, its start and end index on the given line of text must be stored and returned to the editor. This can be an array of structures where each structure has two integer fields of `start` and `end`. This is required because the editor must simply know where in the row to highlight instead of the intrinsic value of what to highlight.

Highlighting Misspelled Words

Terminal text can be highlighted with ANSI escape codes. If a text is highlighted, it needs to stay highlighted until the user makes a change to the word. In order to do this words in every row need to be assigned a binary value such as `NORMAL` or `MISSPELLED`. This way, when the spell checker returns the range in which the misspelled words lie, the row can be traversed and these types can be assigned to the words. With this type distinction, words that are `MISSPELLED` can be inverted/highlighted on the terminal when the rows are being drawn to the screen in the `Display` component.

High-Level System Design

Below is a diagram that maps the design into specific components. These components are separate sections within the program, each containing multiple functions to perform tasks with a specific goal.

- Feature #1 : Key input processing and Display Component
- Feature #2 : File I/O Component
- Feature #3 : Editing/Control Component

Features #4 and #5 are additional and they don't directly map to the high-level system design. Their implementations will be explained further in section 2.4.

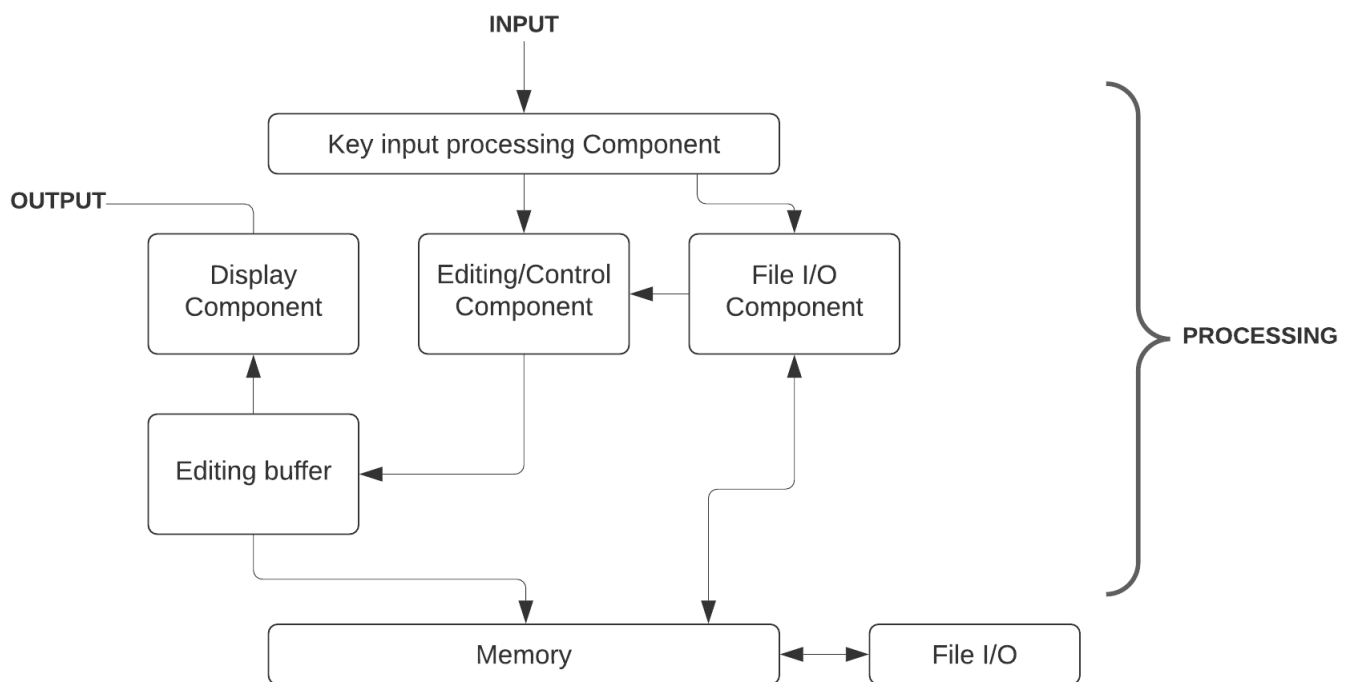


Figure 15: The high level component design of the editor.

The diagram above presents the final architecture of the editing program.

- Key input processing component: must read user input and process them by calling the appropriate functions. These functions will be a part of the Editing/Control or File I/O component.
- The editing/control component: must deal with the user's modifications to the editing buffer such as inserting and deleting data.
- The file I/O component: must deal with file operations such as saving, loading from memory, deleting, and copying.
- The editing buffer: must temporarily store the body of text that can be modified, displayed and later saved to a file.
- The display component: must render and display the buffer on the terminal along with a user menu.

2.4 Implementation

This section will decompose each component of the high level design in Figure 8. To implement each component, the functions required are listed below.

Below is the system implementation diagram where all of the main components of the editor can be seen. The green highlighted components will be a part of Feature #1 and will be implemented now.

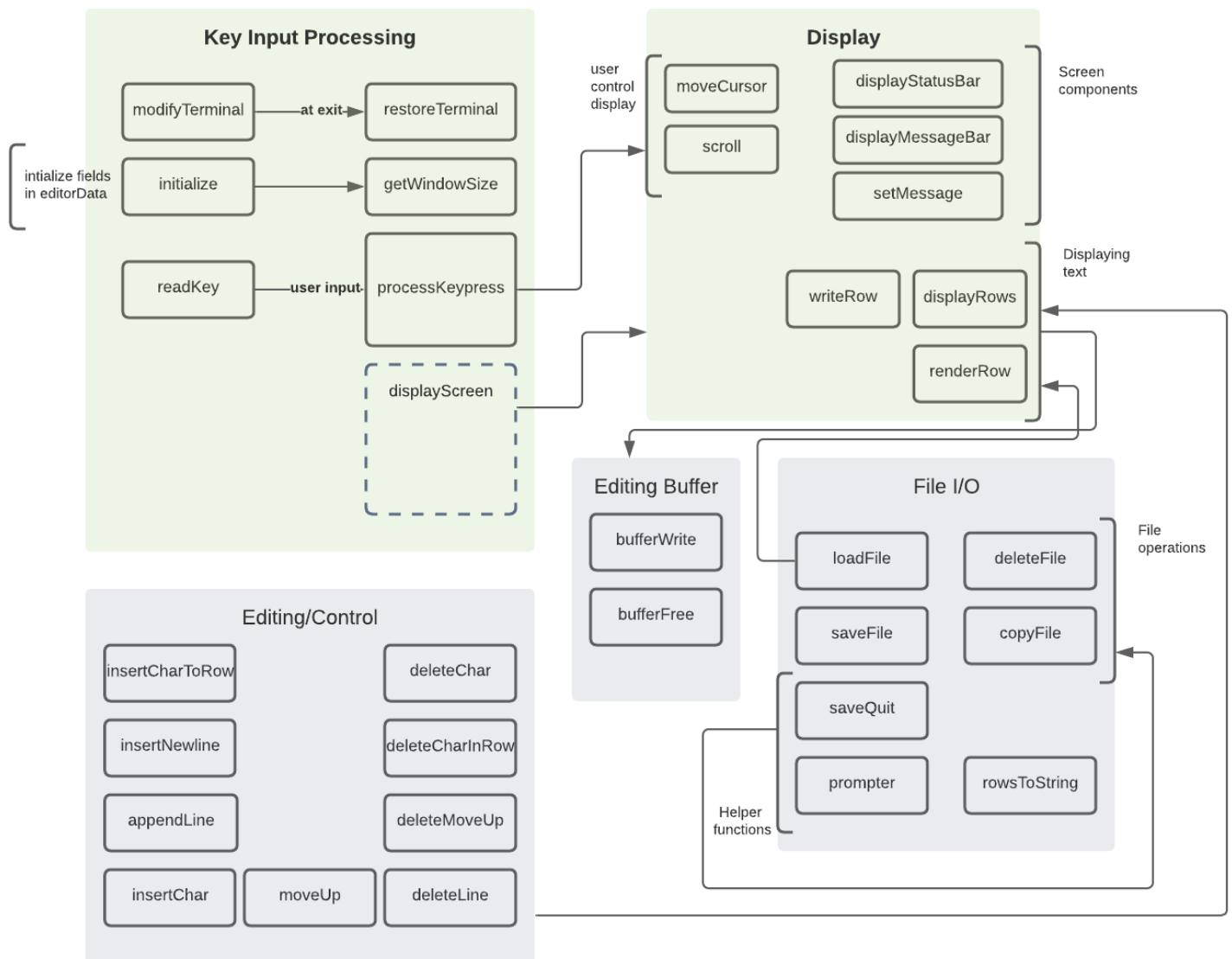


Figure 16: Functions within the Feature #1 components and how they communicate.

FEATURE # 1:

2.4.1 The Key Input Processing Component

Feature #1.a

Functions	Purpose
modifyTerminal()	Setup the terminal mode of input and call restoreTerminal() at exit.
restoreTerminal()	Restore the initial attributes of the terminal.
die()	Clean the terminal screen and display error messages when called.
readKey()	Read the standard input for any user inputs. When a byte is read, return to processKeypress()
processKeypress()	Given a user input, call the appropriate functions in the editor to fulfill user requests.

Figure 17: Functions involved in the Key Input Processing Component.

Setting up the terminal mode of input

It has been deduced in the design section that the desired mode of input was 'non-canonical' 2.3.1. This can be setup in C through systems programming and access to terminal properties. This involves the use of the `termios` header.

Modify Terminal

1. Keep a global `termios` structure.
2. Modify the terminal for usage only throughout the program, such that at exit original terminal properties are restored.
3. Test by printing every key input to the terminal; the goal is to view the key presses immediately as they come in.

See 2.8 for more on the usage of `termios` and its programmatic interface.

The `termios` header will be kept in a global structure in order to keep the code structured and well maintained. Further on, this structure will also store data like cursor coordinate variables, number of rows, window size coordinates, etc.

```
struct editorData {  
    struct termios terminal;  
    ...  
};
```

```
struct editorData E;
```

This structure needs to be populated with data about the initial state of the terminal and then manipulated to change the mode of input to suit the editor objectives. This is done through the programmatic interface to `termios`: two functions `tcgetattr` and `tcsetattr` to get and set terminal attributes.

```
void modifyTerminal() {
```

```

// Get the terminal's initial attributes
if (tcgetattr(STDIN_FILENO, &E.terminal) == -1) die("tcgetattr");
atexit(restoreTerminal); // at exit restore initial attributes

/* Modify the terminal attributes to change the mode of input*/
struct termios newterminal = E.terminal;
newterminal.c_iflag &= ~(BRKINT | ICRNL | IXON); // input flags
newterminal.c_lflag &= ~(ECHO | ICANON | ISIG); // local flags
newterminal.c_cc[VMIN] = 0; // control characters
newterminal.c_cc[VTIME] = 1;
if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &newterminal) == -1)
    die("tcsetattr");
}

```

Firstly, the `terminal` is populated with initial attributes. Then, at exit, another function that restores initial attributes will be called. An instance of the `terminal` is created, `newterminal`. This new `termios` structure is modified to enter non-canonical mode and is set to the current terminal.

The modification flags:

- The input flag field: turns off ICRNL (UNIX terminal translates carriage returns into newlines automatically), IXON (XON and XOFF which are the CTRL-S and CTRL-Q signals that pause and resume data transmission), and BRKINT (sends break conditions and when disabled will allow CTRL signals to function). [10], [11]
- The local flags: turn off ECHO on the terminal line when the user inputs control sequences, ICANON turns off canonical mode of input, and ISIG turns off process terminating sequences like CTRL-Z and CTRL-C since CTRL-Q will be used instead to quit. [10]
- The control characters: alter the granularity of non-canonical mode. In this case, MIN is 0 and TIME is 1 hence read this means that as soon as an input is available read will immediately return it. If no input is available before TIME expires, read returns a value of zero. [12]

Restore Terminal

```

void restoreTerminal() {
    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &E.terminal) == -1)
        die("tcsetattr");
}

```

This function will be called at exit, set in `modifyTerminal()`, such that the originally kept attributes of the terminal are set back. The TCSAFLUSH option specifies when to apply the change. This means that pending output is written to terminal and unread input is discarded. [6]

Error Handling: the `die` function

Finally, for some error handling an extra function to write error messages and exit cleanly conforms to the maintainable and reliable non functional requirements.

```
void die(const char *s) {
    write(STDOUT_FILENO, CLEAR_SCREEN);
    write(STDOUT_FILENO, HOME_CURSOR);
    perror(s);
    write(STDOUT_FILENO, "\r\n", 3);
    exit(1);
}
```

The write functions command to clear the screen and move the cursor to home position and return the error message followed by a carriage return and newline.

These commands are ANSI escape sequences in the form of definitions:

```
#define HOME_CURSOR "\x1b[H", 3
#define CLEAR_SCREEN "\x1b[2J", 4
```

ANSI escape sequences [8] as well as other control sequences will be used throughout this feature, when building the user interface.

This program now will accept immediate input. For this to happen, key inputs must be read from standard input and a key processor function needs to be defined.

Read Keyboard Input

The program must read in inputs from standard input and map them to the specific features defined for this editor, e.g. CTRL-Q to quit. This will be implemented and will be tested as the first milestone.

- Wait for user input from standard input.
- When user input is read, classify control characters from non-control characters and map accordingly.
- Return these keys to a processor function that fulfills the Processing Component design by directing the editor according to user input. E.g. CTRL-Q quits the program, arrow keys move the cursor.

Checking standard input until a byte is read:

```
while ((nread = read(STDIN_FILENO, &c, 1)) != 1) {
    if (nread == -1 && errno != EAGAIN) die("read");
}
```

This loop utilizes the read function; and the loop continues until the function returns success such that a byte has been inputted to the program. If read returns an error message die is called for proper error handling.

After a single byte is read, this will either be an ESC character or not. If the byte is ESC, this will need to be mapped to enums. This way, keys such as the arrow keys can be used throughout the program without writing down escape sequences to do so.

If an ESC is read, the remaining characters of the escape sequence must be read from standard input to map these.

Check 2.8 for what these codes are.

```
if (c == ESC) {
    char seq[3]; // space to read the rest of the control key

    /** Try to read 2 bytes, individually, into seq**/
    if (read(STDIN_FILENO, &seq[0], 1) != 1) return ESC;
    if (read(STDIN_FILENO, &seq[1], 1) != 1) return ESC;

    /** Deal with two types of sequences e.g. ^[0~ and ^[A **/
    if (seq[0] == '[') {
        if (seq[1] >= '0' && seq[1] <= '9') {
            if (read(STDIN_FILENO, &seq[2], 1) != 1) return ESC;
            if (seq[2] == '~') {
                switch (seq[1]) {
                    /** LINUX/RXVT CONSOLE **/
                    case '3': return DEL_KEY;
                    case '5': return PAGE_UP;
                    case '6': return PAGE_DOWN;

                }
            }
        } else {
            switch (seq[1]) {
                /** FREEBSD CONSOLE **/
                case 'A': return ARROW_UP;
                case 'B': return ARROW_DOWN;
                case 'C': return ARROW_RIGHT;
                case 'D': return ARROW_LEFT;

            }
        }
    }

    return ESC;
}
```

As seen above, the sequences are mapped to some comprehensible and easy to write values for maintainability. These are enum values within the program (1000..).

```
enum editorKey {  
    ARROW_LEFT = 1000,  
    ARROW_RIGHT,  
    ARROW_UP,  
    ARROW_DOWN,  
    DEL_KEY,  
    PAGE_UP,  
    PAGE_DOWN  
};
```

Any control sequences that aren't these values are returned as escape characters. If the first read byte was not an ESC character then simply the character is returned.

Feature #1.b

Process Key Inputs

This is function will take the classified keys from the function that reads and maps and simply process them to drive the different components of the editor.

```
void processKeypress() {  
    int c = readKey();  
  
    switch (c) {  
        case CTRL_KEY('q'):  
            exit(0); //exit the program  
            break;  
        case ESC:  
            break;  
        case DEL_KEY:  
            // delete characters  
            break;  
        case ARROW_UP:  
        case ARROW_DOWN:  
        case ARROW_LEFT:  
        case ARROW_RIGHT:  
            // move the cursor  
            break;  
        default:  
            // insert characters  
            break;  
        ...  
    }  
}
```

It's clear to see that this is the interface between the user and the program. As read inputs come in from the user, the requests of the user will call the appropriate functions and will be reflected on the editor screen.

See 2.5.1 where the raw input and CTRL-Q to quit functionality will be tested.

2.4.2 The Display Component

Feature #1.c & d

Functions	Purpose
getWindowSize()	Get the user's terminal window size.
moveCursor()	Move the cursor when requested through arrow keys.
scroll()	Move the display screen up or down according to cursor position.
displayStatusBar()	Display a bar above the message bar to contain the following information: number of lines read into the editing buffer and filename.
displayMessageBar()	Display information about the editor and inform the user.
setMessage()	Set a message on the message bar for 5 seconds.

Figure 18: Functions involved in the Display Component.

Get the Window-size of the user's Terminal

Window sizes are kept in the kernel and updated when the size of the console changes from monitor to monitor. Such data is stored in headers as structures (sys/ioctl). `ioctl` is for input output control; through specific arguments (TIOCGWINSZ) the structure that stores window properties can be accessed (winsize). [13]

```
void getWindowSize() {
    struct winsize win;

    if(ioctl(STDOUT_FILENO, TIOCGWINSZ, &win) == -1 || win.ws_col == 0) {
        /** Error Handling **/
        die("ioctl error");
    } else{
        /** The winsize struct will be filled with info about terminal width
            and height **/
        E.screencols=win.ws_col; // store in editorData
        E.screenrows=win.ws_row;
    }
}
```

The row and column size of the terminal will be useful in navigating the screen and constraining the cursor movement, hence will be used in the functions below i.e. `moveCursor` and `scroll`. This data will be stored in `editorData`.

Moving the Cursor

The cursor position on the terminal screen will be kept as coordinate variables in the `editorData` structure:

```
struct editorData {
    struct termios terminal; /* The terminal attributes */
    int cx, cy; /* Cursor coordinates */
}
```

```

    int screenrows, screencols; /* Window size of terminal */
    ...
};

```

The cursor can be moved through ANSI escape code commands to the terminal. Hence, the coordinates as well as the position of the cursor can be initialized:

```

void initialize() {
    E.cx = 0;
    E.cy = 0;
    getWindowSize();
    ...
}

```

The initialize method can initialize the data held in the `editorData` structure in the beginning of every execution of the editor program.

```

#define INIT_CURSOR "\x1b[%d;%dH", (E.cy) + 1, (E.rx) + 1

```

The coordinates on the screen must be (1,1) when initialized to the top left corner and can be commanded with the escape sequence seen above.

The movement of the cursor will depend on the direction of the arrow key pressed. `processKeyPress` from the Key input processing component will make a call to `moveCursor` with the arrow key as an argument. With this the cursor variables can be updated and printed on the screen.

```

void moveCursor(int key) {
    switch (key) {
        case ARROW_LEFT:
            if (E.cx != 0) { // if not trying to go left out of the editor window
                E.cx--; // move left
                break;
            }
        case ARROW_RIGHT:
            E.cx++; // move right
            break;
    }
    case ARROW_UP:
        /** Don't move further up than page limit 1,1, */
        if (E.cy != 0) {
            E.cy--; // move up
        }
        break;
    case ARROW_DOWN:
        E.cy++; // move down
        break;
    }
}

```

This function updates the cursor variables based on arrow input. Currently, the cursor is constrained from moving left too much and moving up too much. This means it must not go

out of the window. However, in the upcoming sections the program will get more complex and further additions will be required:

- Constraining the right and down positions: As text will be displayed on the screen, the cursor will need to be restrained within the bounds of the length of the text (horizontally and vertically).
- When the length of a loaded file exceeds the size of the user's terminal window, as the cursor moves down or to the right the screen will need to scroll.

Scrolling

Scrolling will basically be keeping an offset value (in the `editorData` structure) of the cursor position such that when the user requests to go out of the offset value the page will scroll by setting the cursor position to the offset. This will depend on the window's size as well.

- Scrolling up: If the cursor's y coordinate is less than the offset this means that the user is requesting to go up hence the offset is set to the cursor position.
- Scrolling down: If the cursor's y position exceeds the visible window, scroll down one by one by setting the offset to the difference between the y coordinate and the screen plus 1.
- Scrolling left: If the cursor's x coordinate is less than the offset set the offset to the x coordinate.
- Scrolling right: If the cursor's x coordinate exceeds the window, set the offset to the very right of the screen plus 1.

```
void scroll() {
    /** Scroll Up */
    if (E.cy < E.rowoff) {
        /** If the cursor is above the visible window */
        E.rowoff = E.cy; // scroll up by setting current display index to
            that index
    }
    /** Scroll Down */
    else if (E.cy >= (E.rowoff + E.screenrows) ) {
        /** If the cursor is below the visible window */
        E.rowoff = E.cy - E.screenrows + 1; // scroll down
    }
    /** Scroll left */
    if (E.cx < E.coloff) {
        E.coloff = E.cx;
    }
    /** Scroll right */
    else if (E.cx >= E.coloff + E.screencols) {
        E.coloff = E.cx - E.screencols + 1;
    }
}
```

Upon every refresh of the screen, the cursor must be printed such that the screen can be simulated to 'scroll'.

For this the INIT_CURSOR definition is written to the screen with every refresh.

Display a Status Bar

A status bar, as seen on Figure 5 in the design section, is a row on the screen with inverted colors; displaying information such as number of lines read to the editing buffer, name of the file loaded, etc...

Graphic rendition and inversion of colors can be done with escape codes [8].

```
void displayStatusBar() {
    write(STDOUT_FILENO, "\x1b[7m", 4); // GRAPHIC RENDITION
    ...
}
```

Hence, a bar with inverted colors to the usual terminal screen will be written to standard output. The main objective was to write status messages to this bar.

- Display number of lines loaded to buffer: In the upcoming sections when the editing/-control component is being built, a field can be kept in editorData such that the number of lines read from a file and written to the buffer can be tracked. This value will need to be printed on the status bar.
- Display filename: Filenames can also be kept in editorData when filenames are given as command line arguments.
- Display the line number the user is on: Write the cursor's y coordinate on the status bar.

```
/** Initialize space for status bar message strings */
char status[80];
char tracker[80];

/** Display <name of file> - <length of file> */
int len = snprintf(status, sizeof(status), "[ %s - READ %d LINES ]",
E.filename ? E.filename : "[No Name]", E.numrows); // if there is no
    filename, write No Name.

/** Display the current line the user is on */
int trackerlen = snprintf(tracker, sizeof(tracker), "LINE %d \t", E.cy + 1);
```

Then, these messages to be written across the status bar on standard output.

- Display status on the left hand side of the status bar.
- Display tracker on the opposite, right hand side.

```

/** Append the status messages to the editing buffer */

write(STDOUT_FILENO, status, len); // append status message to bar

/** Append spaces until the end of the window screen to show the white
    background */
while (len < E.screencols) {
    if ( E.screencols - len == trackerlen) {
        /** At the right end of the status bar display the tracker */
        write(STDOUT_FILENO, tracker, trackerlen);
        break;
    } else {
        write(STDOUT_FILENO, " ", 1);
        len++;
    }
}
write(STDOUT_FILENO, "\x1b[m", 3); // turn off color inversion

```

The code above writes the status directly on the left hand side of the bar. Then, moves to the right hand side in a while loop. Until the loop gets to the point on the screen where writing the tracker message would be on the end of the bar, spaces are written to show the white bar all the way.

Display Messages

A message bar will be written under the status bar. This is for:

- Displaying help messages: e.g. "Press CTRL-Q to quit"
- Displaying messages after user actions: e.g. "x bytes written to disk" after saving.
- Saving before quitting message or enter filename if no name: e.g. "Are you sure you want to quit without saving?" to prompt users.

```

void displayMessageBar() {
    write(STDOUT_FILENO, ERASE_IN_LINE);
    int msglen = strlen(E.statusmsg);

    /** If message exceeds window length, don't display the extra characters
        */
    if (msglen > E.screencols) msglen = E.screencols;

    /** If message is less than 5 seconds old display it */
    if (time(NULL) - E.statusmsg_time < 5)
        abAppend(ab, E.statusmsg, msglen);
}

```

The status message is kept in the editorData structure and will be set in another function setMessage so that it's possible to set temporary messages throughout the program, upon request.

Upon a message being set, the length of the string is consolidated such that it's not too long and exceeds screen length. Then, a timer is set for 5 seconds and the message is displayed until the 5 seconds is up.

Set Messages

This is a variadic function, like `printf` in the C library [6] is. This means that given the type of the first argument to this function, multiple more can be passed of the same type. This way it's possible to use the message bar when prompting the user and displaying more than a single string on the bar.

```
void setMessage(const char *fmt, ...) {  
    va_list ap;  
    va_start(ap, fmt);  
    vsnprintf(E.statusmsg, sizeof(E.statusmsg), fmt, ap);  
    va_end(ap);  
    E.statusmsg_time = time(NULL);  
}
```

`va_list` is...

Following Figure 5, the initial design, the end product looks as so:



Figure 19: The terminal with a status and message bar.

Feature #1 Outcomes

At this stage, the terminal accepts immediate input, reads keys, processes them and displays a screen with a status and message bar. This can be tested so far with three key functionalities:

- Moving the arrows to see the cursor move.
- Move out of the current window to see if the cursor scrolls the screen (horizontally and vertically)
- CTRL-Q to quit immediately.

In order to get such a program the main flow will be as so:

```
int main(int argc, char *argv[]) {
    modifyTerminal(); // change terminal mode
    initialize(); // initialize editorData
    setMessage("Ctrl-Q = QUIT | Ctrl-X = HELP | Ctrl-S = SAVE | Ctrl-F =
        SPELLCHECK");

    while (1) {
        displayScreen(); // will be updated in the future
        processKeypress();
    }

    return 0;
}
```

displayScreen() displays all the components by writing them to standard output one by one.

```
void displayScreen() {
    scroll();

    write(STDOUT_FILENO, HIDE_CURSOR);
    write(STDOUT_FILENO, HOME_CURSOR); // cursor to home 0,0

    displayStatusBar();
    displayMessageBar();

    char buf[32];
    snprintf(buf, sizeof(buf), INIT_CURSOR); // print cursor to restricted
        position
    write(STDOUT_FILENO, buf, strlen(buf));
    write(STDOUT_FILENO, SHOW_CURSOR); // show it
}
```

FEATURE #2

2.4.3 File I/O Component

This component will first implement the editing buffer. Then, starting with displaying, a file will be loaded and displayed on this buffer. Finally, it will be possible to save the buffer as a file, copy files, and delete them.

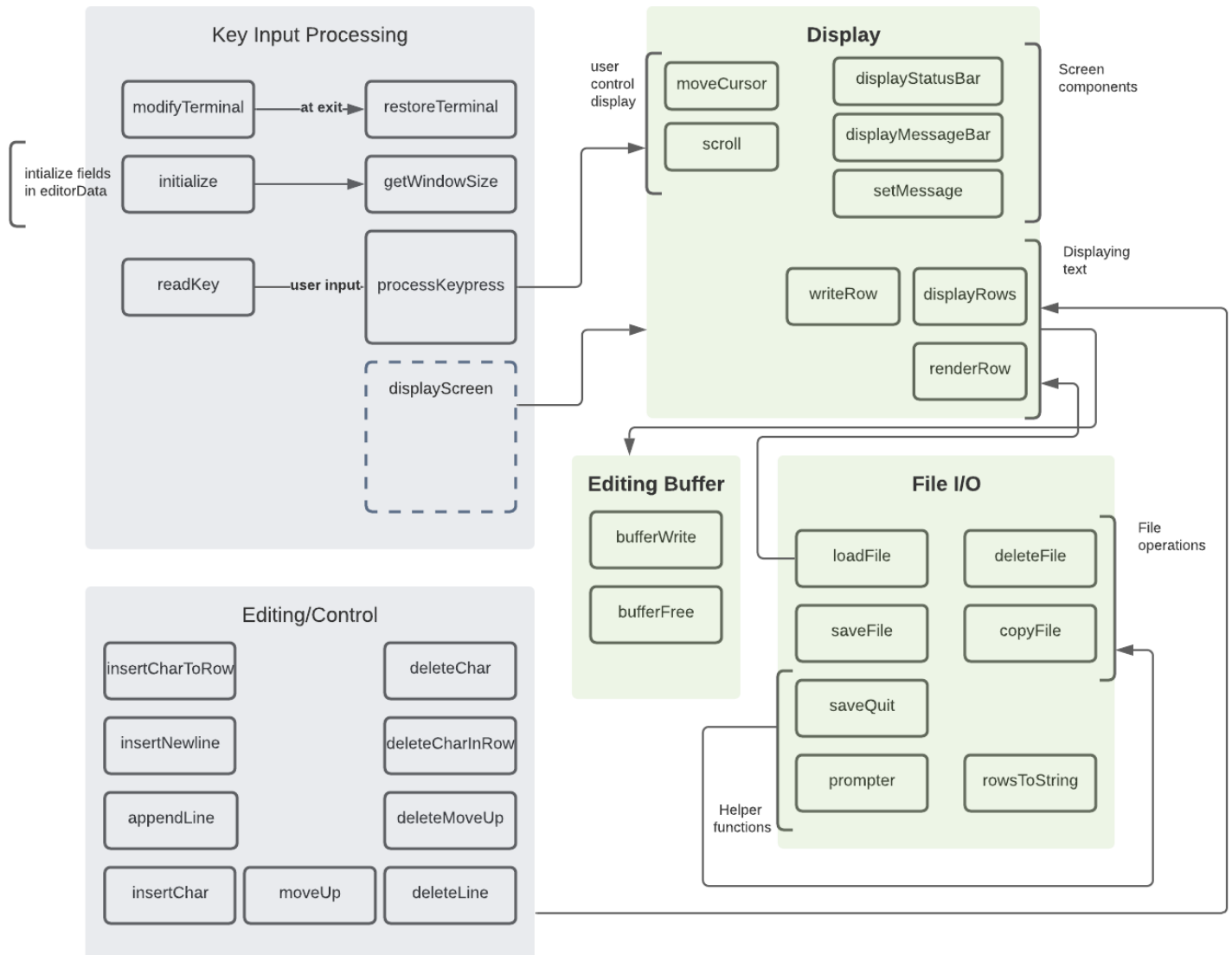


Figure 20: Feature #2 components and how they communicate.

2.4.4 The Editing Buffer

Feature #2.a & b

Functions	Purpose
bufferWrite()	Given a string of some length, write it to the editing buffer.
bufferFree()	Free the entire buffer.
loadFile()	Given a file, writes each row to a row structure with the writeRow function.
writeRow()	Writes the file rows to the row structure and renders each with renderRow
renderRow()	Deals with inconsistent tabs before displaying the files.
displayRows()	Writes each rendered row onto the editing buffer with bufferWrite.

Figure 21: Functions required to create and display a buffer.

A buffer is a temporary storage location. This is where the user will make their modifications, and see it being displayed real-time. A buffer can later be saved into a file.

```
typedef struct editorBuffer {  
    char *b;  
    int len;  
} ebuffer;
```

This structure is essentially just a dynamic string. It stores a character array and its length. It needs to be possible to write to and free this buffer; such that, instead of writing the different components that will be displayed on the terminal screen directly to standard output, it will be written to the buffer. Then the buffer will be written to standard output only once.

Write to a buffer

```
/** Given a string of some length, write it to the editing buffer  
    structure. */  
void bufferWrite(struct editorBuffer *ab, const char *s, int len) {  
    /** Reallocate space for the string to be written */  
    char *new = realloc(ab->b, ab->len + len);  
  
    if (new == NULL) return;  
    /** If reallocation was successful, copy the string to the buffer array  
        */  
    memcpy(&new[ab->len], s, len); // copies s to the buffer, starting from  
        the new pointer.  
    ab->b = new;  
    ab->len += len;  
}
```

Everytime writeBuffer is called, the buffer's memory location is expanded by the length of the string being written to it. Given that reallocation is successful, the string is copied to this new memory location (appended to the buffer) and the length and buffer pointer are updated. The buffer will contain everything that will be written to the display screen. Hence, the

contents of the buffer will be displayed as is until the user requests a modification. When this happens, the buffer will likely change and will be displayed again with new contents. For this, the buffer must be freed after every display.

```
void abFree(struct editorBuffer *ab) {  
    free(ab->b);  
}
```

This frees the buffer pointer. Next time the buffer is requested to be written to, reallocation will happen.

Loading files to the buffer

In order to display rows of data on the screen, the editing buffer needs a way of storing rows in an organized manner. This way, when the user requests modifications, the rows are easily accessible through something like arrays. Hence, we can create an array of row structures (Similar to the concept of an array of objects). This array can be created by making an array of rows in editorData.

```
/** Holds each row of a read file */  
typedef struct rows {  
    int size; /* Size of chars */  
    char *chars; /* String of data, a single row */  
    int rsize; /* Size of rendered row */  
    char *render; /* The rendered string of data */  
} rows;
```

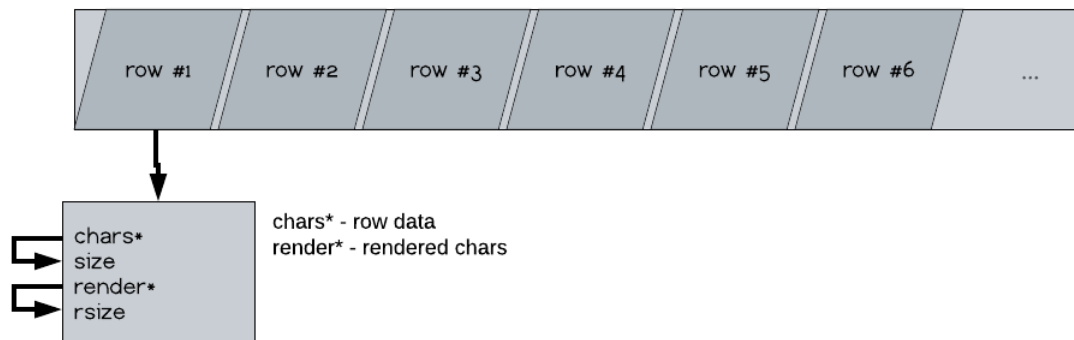


Figure 22: An array of row structures to be stored in the buffer.

It needs to be possible to write to the row structure as well as render the written rows to be displayed on the terminal screen. Rendering means dealing with tabs and spacing. The code below will open a requested file and send each line read to another function `writeRow()` where the line read from the file will be written to a row structure.

The below code will load in a file to the row structure, and keep a count of the number of lines read:

```
char *line = NULL; /* temporary string to hold lines in the file */  
size_t linecap = 0;
```

```

ssize_t linelen;

/** Until the end of the file, read each line and append to
 * rows */
while ( (linelen = getline(&line, &linecap, fp)) != -1 ) {
    /** Remove empty lines in between read lines */
    if (linelen > 0 && (line[linelen - 1] == '\n'
        || line[linelen - 1] == '\r'))
        linelen--;
    writeRow(E.numrows, line, linelen);
}

```

To write lines read from the loaded file into the rows structure a couple of things will need to be done.

- Create another row in the rows structure array by reallocating space.
- Fill in the row structure with the passed in line, and update size of the line.
- Render each row to be displayed with appropriate spacing.
- Keep a record of number of lines read by incrementing a global variable every time a call is made to writeRow.

```

void writeRow(int index, char *line, size_t len) {
    /** Extend size of the rows string array by 1 for every row read */
    E.row = realloc(E.row, sizeof(rows) * (E.numrows + 1));

    /** Fill in the row structure for the current row in file */
    E.row[index].size = len; // set size of row in file
    E.row[index].chars = malloc(len + 1); // allocate memory to the row array
    memcpy(E.row[index].chars, line, len); // copy the line into the row array

    /** Initialize rsize and render and then make a call to render the
     * currently read line */
    E.row[index].rsize = 0;
    E.row[index].render = NULL;
    renderRow(&E.row[index]);

    /** Keep a record of the number of lines read, display on status bar */
    E.numrows++;
}

```

Rendering Rows

The rows that are being written to the row structures must be rendered before being displayed. This is because of the tab characters. Instead of each tab, we place 8 spaces in between words. A tab could be a different number of columns depending on your environment, but a space is always one column - this makes the display consistent.

- Count the total number of "\t" characters in a given row.

- Allocate space for: the number of tabs * 8 in order to place 8 spaces instead of each tab.
- iterate through the row and when a tab character is encountered place 8 spaces instead of it; otherwise just assign the usual characters to the render field.
- Set the null character at the end of each rendered row.

```

void renderRow(rows *row) {
    int tabs = 0;
    /** count number of tabs in the row string**/
    for (int i = 0; i < row->size; i++)
        if (row->chars[i] == '\t') tabs++;

    free(row->render);
    /** Allocate memory to rendered row with size of text + 8 characters for
        each tab in row **/
    row->render = malloc(row->size + 1 + tabs*(TABS - 1));

    /** Iterate over the row and place spaces instead of tabs, otherwise copy
        the row as is **/
    int idx = 0;
    for (int j = 0; j < row->size; j++) {
        if (row->chars[j] == '\t') {
            // append space if there is a tab encountered, until a tab stop,
            // which is 8 characters later
            row->render[idx++] = ' ';
            while (idx % TABS != 0) row->render[idx++] = ' ';
        } else {
            row->render[idx++] = row->chars[j];
        }
    }

    /** set the null character and size of the rendered row **/
    row->render[idx] = '\0';
    row->rsize = idx;
}

```

While a file is being loaded, if `renderRow` is called for each call to `writeRow`, then with this the rows structure would be filled with data ready to be displayed on the terminal screen.

Display Rows

Rows must be displayed such that only the visible screen is filled. This must make use of the offset value set during scrolling in order to decide which row in the rows structure will be displayed at what time. The following function will need to consider the actions below:

- Iterate through the total length of the screen (provided by `getWindowSize`)
- Taking into account the offset row value, write the row in the structure that is at an appropriate row index.
- Constraints: don't display row data out of the window scope, display only up to window scope. When the user tries to scroll, `displayRows` will display that data since it will

now be within the row offset.

```
void displayRows(struct editorBuffer *ab) {
    /** Iterate through all the window screen rows **/
    for (int y = 0; y < E.screenrows; y++) {
        /** index of line to be displayed on screen, takes into account
         * moving out of visible editor window **/
        int filerow = y + E.rowoff;

        if (filerow < E.numrows) {
            /** Make sure current row is not past the total number of
             * rows in file **/
            int len;
            /** If user tries to display past end of line, display nothing **/
            if ((len = E.row[filerow].rsize - E.coloff) < 0) len = 0;
            /** if user tries to display out of window scope, display last
             * possible line and don't go out of scope **/
            if (len > E.screencols) len = E.screencols;
            /** append rendered row to buffer to be displayed **/
            bufferWrite(ab, &E.row[filerow].render[E.coloff], len);
        }

        bufferWrite(ab, ERASE_IN_LINE);
        bufferWrite(ab, "\r\n", 2); // append a return and end of line
    }
}
```

writeRow displayRows, renderRow will be a part of the Display Component. Therefore, this function can be called from displayScreen upon every user modification. This creates an interactive real-time screen where with every user input, the rows are updated, rendered, and re displayed on the screen. It will be possible to test this once row/line operations are introduced in Feature #3.

See 2.5.2 to see the testing for the implemented functionalities.

Feature #2.c

Functions	Purpose
saveFile()	Saves the current buffer to a given filename.
rowsToString()	Converts the contents of the rows structure into a single string.
saveQuit()	Prompts the user when they try to quit without saving a modified buffer.
prompter()	Prompts the user on the message bar for some requested information.
copyFile()	Copies a source file to a destination file.
deleteFile()	Deletes a given file from the file system.

Figure 23: Functions required for file operations.

Save Files

In order to save files, the contents of the buffer must be converted into a string. Then this can be written to a given filename. This can be done by creating a string of length of all the rows combined and then iterating over the rows to copy each row to this string.

First, get the length of all rows combined, then allocate some space for this string and create a pointer to it:

```
int totlen = 0;
/** Get the length of all the rows lengths
 * combined **/
for (int i = 0; i < E.numrows; i++)
    totlen += E.row[i].size + 1;
*length = totlen;

/** Allocate space to store the string in **/
char *string = malloc(totlen);
char *p = string; // a pointer to the string
```

Then, the entire rows structure can be iterated. Each row will be copied to the location of the string and the pointer to this string will be 1 plus the size of the copied row. This way, all the rows are iterated and the end product is one long string ready to be written to a file in a single stream.

```
for (int j = 0; j < E.numrows; j++) {
    /** Iterate over the rows and copy every row to the
     * allocated string location **/
    memcpy(p, E.row[j].chars, E.row[j].size);
    /** move pointer to the start of next row **/
    p += E.row[j].size;
    *p = '\n';
    p++;
}
```

To open a file and write to it, first it must be checked whether the user opened this editor with a specific file to begin with or is trying to save a previously empty buffer with no name.

If it's the latter, the user must be prompted to give this buffer a filename. Filenames are restricted, such that no control keys can be a part of them. Upon pressing enter or carriage return, the filename will be accepted.

Allocate some initial space for the user input.

```
/** Allocate some space for the user input **/
int bufsize = 128; // initial size, can be expanded
char *usrinput = malloc(bufsize);
int size = 0;
usrinput[0] = '\0';
...
return usrinput;
}
```

Now prompt the user until they press Enter:

```
while (1) {
    /** Display the prompt on the message bar **/
    setMessage("Enter: %s", usrinput);
    displayScreen();

    int c = readKey(); // read the user keys
    /** Enter key/return ends the prompt **/
    if (c == '\r') {
        break;
    /** Allow the user to delete **/
    if (c == DEL_KEY || c == CTRL_KEY('h') || c == BACKSPACE)
        if (size != 0) usrinput[--size] = '\0';
    /** If not a control key, take it **/
    if (!(iscntrl(c))) {
        if (size == bufsize - 1){
            /** Reallocate if user exceeds initial size **/
            bufsize *= 2;
            usrinput = realloc(usrinput, bufsize);
        }
        usrinput[size++] = c;
    }
}
}
```

Save File:

```
void saveFile() {
    /** Check if the file has a name, if not prompt **/
    if (E.filename == NULL) E.filename = prompter();

    /** Convert the rows structure, and take a single string **/
    int len;
    char *buf = rowsToString(&len);

    /** Open the file, if doesn't exist create it. **/
    int fd = open(E.filename, O_RDWR | O_CREAT, 0644);
    if (fd != -1) {
        /** If the file opens **/
        if (write(fd, buf, len) == len) {
            /** Write to the file and clean up **/
            close(fd);
            free(buf);
            setMessage("Saved successfully.");
            return;
        }
    }
    /** File didn't open **/
    free(buf);
    setMessage("Error:", strerror(errno));
}
```

```
}
```

Another issue that may arise during the program is if the user tried to quit a modified buffer. It wouldn't make a very reliable editor if there wasn't any checking done for this case. Hence, if the user makes a single modification to an open buffer, a boolean variable e.g. `modified` will be `True`. Then, if the key input processor detects a request to CTRL-Q, this boolean variable can be consolidated before fulfilling the request.

- If the user makes any modification to an open buffer, set `modified` to `True`.
- Throughout the program, if the user saves the buffer, set `modified` to `False`.
- If `modified` is true and the user tries to quit, call `saveQuit` to prompt a save.

```
void saveQuit() {  
    setMessage("Do you want to save before you quit? y/n");  
    displayScreen();  
    int c = readKey();  
    if (c == 'y' || c == 'Y') saveFile();  
}
```

This will ask the user if they wish to save before quitting, the answer y/n determines the route taken.

Copy Files

```
void copyFile(char* source, char* dest) {  
    /** Check if source and dest are the same files **/  
    if(strcmp(source,dest)==0) die("Cannot copy the same files.");  
  
    /** Open source to read and dest to write to **/  
    FILE *sourcef = fopen(source, "r");  
    FILE *destf = fopen(dest, "w");  
    if (!sourcef || !destf) {  
        die("Files failed to open");  
    }  
  
    /** Copy every character one by one to dest until  
    * the end of the source file **/  
    int c;  
    while((c = fgetc(sourcef)) != EOF) fputc(c, destf);  
  
    /** Clean up file pointers **/  
    fclose(sourcef);  
    fclose(destf);  
    char message[80] = "Operation successful.\r\n";  
    write(STDOUT_FILENO, message, strlen(message));  
}
```

- Given two filenames, source and destination, check if they are not the same. If not, open source to read and destination to write.

- Copy everything in source to destination.
- Clean up file pointers and display a message to the user.

Delete Files

```
void deleteFile(const char *filename){
    if ((remove(filename)) == -1) die("Couldn't remove file.");
    char message[80] = "Operation successful.\r\n";
    write(STDOUT_FILENO, message, strlen(message));
}
```

Given a filename, make a call to remove to permanently remove this file from the file system.

Feature #2 Outcomes

1. Now the editor has an editing buffer and can display rendered text from loaded files.
 2. Additionally, the required file operations have been made available; the user can save/load a buffer, delete, and copy files.
 3. The flickering effect, in Feature #1, has been eliminated with the editing buffer. Now displayScreen makes a single write operation.
-

```
void displayScreen() {
    scroll();

    /** Initialize the editing buffer */
    struct editorBuffer ab = ABUF_INIT;
    /** Hide the cursor and initialize position */
    bufferWrite(&ab, HIDE_CURSOR);
    bufferWrite(&ab, HOME_CURSOR);
    /** Write the modified rows to the buffer and
     * draw the screen with status/message bars */
    displayRows(&ab);
    displayStatusBar(&ab);
    displayMessageBar(&ab);
    /** Print the hidden cursor then show it */
    char buf[32];
    snprintf(buf, sizeof(buf), INIT_CURSOR); // print cursor to restricted
        position
    bufferWrite(&ab, buf, strlen(buf));
    bufferWrite(&ab, SHOW_CURSOR); // show it

    /** Write the buffer to standard output and free buffer */
    write(STDOUT_FILENO, ab.b, ab.len);
    abFree(&ab);
}
```

FEATURE #3

Feature #3 will allow actual modifications to files and the editing buffer. This implementation will provide functions like insertion and deletion.

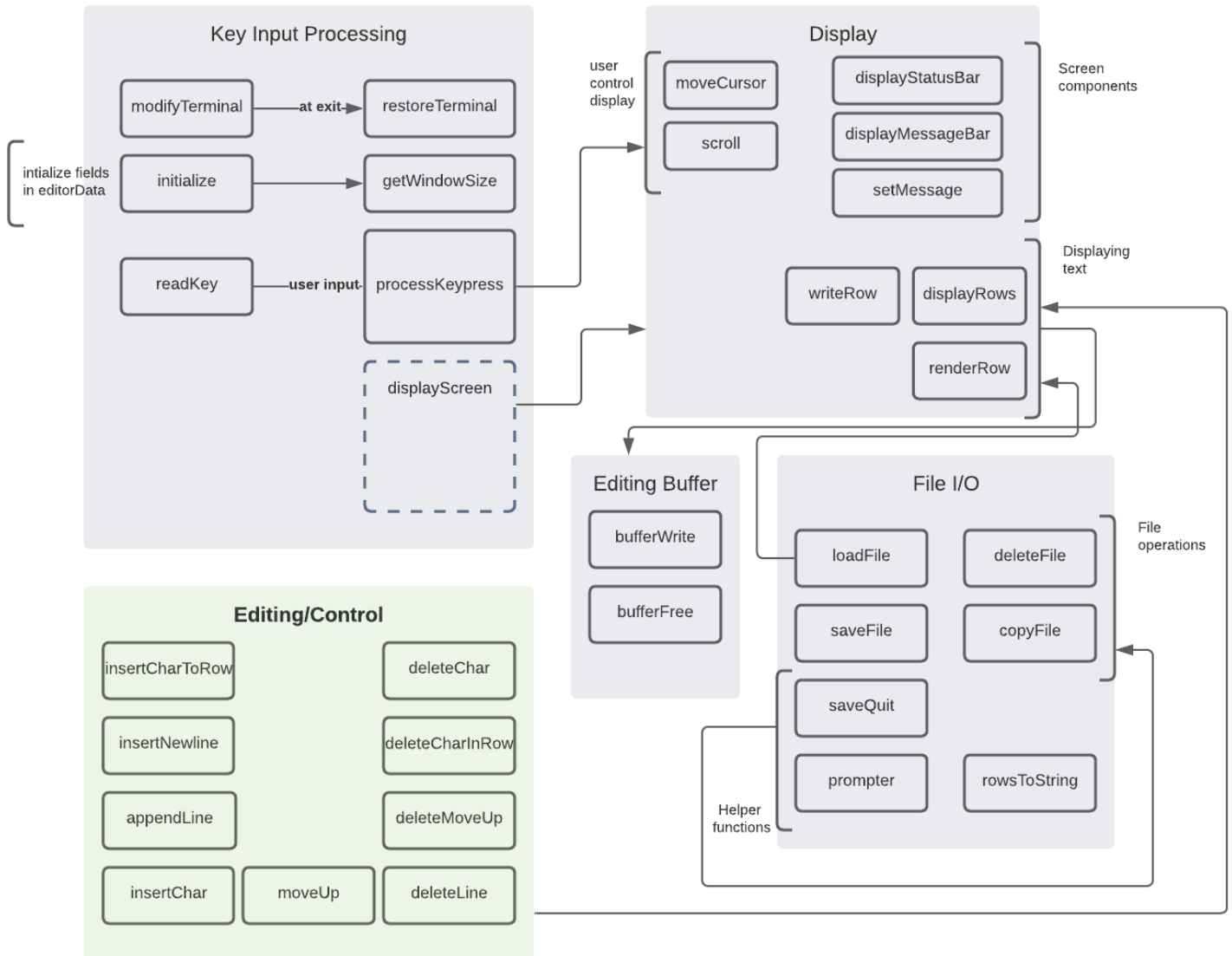


Figure 24: Functions required to implement Feature #3a

2.4.5 Editing/Control Component

Feature # 3.a

Functions	Purpose
insertChar()	Inserts a user input character to a specific position, indicated by the screen cursor.
insertCharToRow()	Inserts the character from insertChar() to the rows structure.
insertNewline()	Inserts a newline character and deals with displaying it.
appendLine()	Appends a string to a given file from the command line.

Figure 25: Functions required to implement Feature #3a

Inserting characters

Whenever there is a new user input from the Key input Processing component, a newly inserted character must be written to the row structures. This way it can be written to the buffer per usual and displayed on the screen immediately.

- When the user inputs a character, check if a new row must be created and appended to the rows array.
- Otherwise, just insert the character to the existing rows in the rows array.

```
void insertChar(int c) {  
    /** Set modified to true hence, the user must  
    * be prompted to save the buffer before quitting **/  
    E.modified = true;  
    if (E.cy == E.numrows)  
        /** If the user inserts a character to a  
        * newline, add a newline to rows **/  
        writeRow(E.numrows, "", 0);  
    insertCharToRow(&E.row[E.cy], c); // insert character to rows  
}
```

insertCharToRow will need to insert the character to the cursor's position.

- Allocate 1 more byte in the current row for the new character.
- Move all the characters to the right of the cursor across by one to make space for the new character.
- Insert the character, update the row structure; then, render the entire row before displaying.

```
void insertCharToRow(rows *row, int c) {  
    /** Reallocate memory to rows for the new byte **/  
    row->chars = realloc(row->chars, row->size + 1);  
  
    /** Move the data to the cursor's right by 1 **/  
    memmove(&row->chars[E.cx + 1], &row->chars[E.cx], row->size - E.cx + 1);  
    /** Insert the character in the cursor's position  
    * and render the row **/  
}
```

```

row->chars[E.cx] = c;
row->size++;
renderRow(row);
E.cx++; /** move cursor to the right hence next insert
        wont overwrite */
}

```

Insert newline characters

Newline characters refer to "\n". These need to be considered with more care since there are more than one behaviours they could result in, depending on the position of the cursor when they are inserted.

1. If the cursor is at the head of a row, insert a new row to the current cursor position and move one row down.
2. If the cursor is anywhere else, write all the characters to the right of the cursor to one row below and update the current row's size. Then move down.

1.

```

void insertNewline() {
    if (E.cx == 0) {
        /** At the head of a row */
        writeRow(E.cy, "", 0); // insert a new row
    } else {
        ...
    }
    /** Move the cursor to the head of the bottom row */
    E.cy++;
    E.cx=0;
}

```

2.

```

/** Anywhere else within a row */
rows *row = &E.row[E.cy]; // get the row structure

/** Write the contents of the current row
 * to one row down, starting from current position
 * of cursor */
writeRow(E.cy + 1, &row->chars[E.cx], row->size - E.cx);
/** Update the current row and render it*/
row = &E.row[E.cy];
row->size = E.cx;
row->chars[row->size] = '\0';
renderRow(row);

```

Append to a file

Appending is a distinct operation from inserting to specific positions, as given in the specification. Hence, it makes sense to simply make this available through command line arguments. Given a filename and a string, this can be appended to the given file with a command as such:

```
./editor -append <filename> <string>
```

```
void appendLine(char *filename, char *s) {  
    /** Open the given file **/  
    FILE *fp = fopen(filename, "a");  
    if (fp==NULL) die("fopen failed");  
  
    /** Append the string and close **/  
    fprintf(fp, "%s\n", s);  
    fclose(fp);  
}
```

Feature # 3.b

Functions	Purpose
deleteChar()	Deletes a specific character that the cursor points to when requested.
deleteCharinRow()	Deletes the character from the rows structure and renders the row.
deleteMoveUp()	Upon deletes at the head of a row, moves the rows up by 1.
moveUp()	Moves rows up by 1.
deleteLine()	Deletes an entire line with user input (CTRL-K) and moves all following lines up.

Figure 26: Functions required to implement Feature #3b

Deleting characters

1. If the user requests to delete a character in the middle of a row, just delete that character and modify the current row.
2. If the user requests to delete a character at the head of a row, move the contents of the current row up along with the cursor. If the row above already has content, the current row must be appended to it. This must be followed up by every row following the current row moving up by 1.

1.

```
void deleteChar() {
    /** Check if cursor is out of bounds **/
    if ( (E.cy == E.numrows) || (E.cx == 0 && E.cy == 0) )
        return;

    /** Get current row **/
    rows *row = &E.row[E.cy];
    if ( E.cx > 0 ) {
        /** If the cursor is in the middle of some
         * row just delete the character**/
        deleteCharinRow(row);
    } else {
        ...
    }
}
```

Deleting a character in a row is similar to inserting one. The text to the right of the cursor will overlap the current position of the cursor. This will 'delete' by overwriting the current byte. After doing this the cursor moves to the left by 1 to simulate a deletion behaviour.

```
void deleteCharinRow(rows *row) {
    /** Check whether the cursor is out of bounds **/
    if (E.cx-1 < 0 || E.cx-1 >= row->size) return;

    /** Overlap the text to the right of the cursor
     * with the current position in order to overwrite
     * the byte to be deleted. **/
    memmove(&row->chars[E.cx - 1], &row->chars[E.cx],
        row->size - (E.cx - 1));

    /** Update the row structure, and render **/
    row->size--;
    renderRow(row);
    E.cx--; // move the cursor up
    E.modified = true;
}
```

2.

```
...
/** If the cursor is trying to delete at the
 * head of a row, move the row up **/
deleteMoveUp(&E.row[E.cy - 1], row->chars, row->size);
```

Deleting at the head must append the contents to the previous row and move all rows up by 1. To do this the cursor and the row structures must be manipulated:

- Set the cursor the the end of the previous row.

- Reallocate memory to the previous row and append all of the current row to it.
- Move up every other row following the deleted one.
- Update the row structures and render to display.

```
void deleteMoveUp(rows *row, char *s, size_t len) {
    /** Set the cursor the end of the previous line **/
    E.cx = E.row[E.cy-1].size;

    /** Reallocate memory for the previous row to include
     * the characters being appended from current row **/
    row->chars = realloc(row->chars, row->size + len + 1);

    /** Copy the characters in the current row to the end
     * of the previous row **/
    memcpy(&row->chars[row->size], s, len);

    /** Update the current row and render it **/
    row->size += len;
    row->chars[row->size] = '\0';
    renderRow(row);

    /** Move all the following rows up by 1 **/
    moveUp(E.cy, &E.row[E.cy]);
    E.cy--;
}
```

Moving up every following line will simply require to free the current row and move the following row up to it.

```
void moveUp() {
    /** Get current row **/
    rows *row = &E.row[E.cy];

    /** free the current row **/
    free(row->render);
    free(row->chars);

    /** Move the rows below up by 1 **/
    memmove(&E.row[E.cy], &E.row[E.cy + 1], sizeof(rows) * (E.numrows - E.cy
        - 1));
    E.numrows--; // decrease total number of rows
}
```

Delete Lines

- When the user presses CTRL-K, call `deleteLine`
- Set the cursor position to the end of the currently selected call.
- Call `deleteCharinRow` until the contents of the entire row is erased.

- Move all following rows up by 1.

```
void deleteLine() {
    /** Get current row **/
    rows *row = &E.row[E.cy];

    /** Set the cursor to the size of the row and call
     * deleteCharInRow until the entire row is deleted **/
    E.cx = row->size;
    while(row->size>0 && E.cy<E.numrows) deleteCharinRow(row);
    /** Move up all following rows **/
    if (E.cy<E.numrows) {
        moveUp(E.cy, row);
    }
}
```

Feature #3 Outcomes

1. Now the editor can support line operations such as inserting, deleting.
2. The user can append to specific files from the command line through command line flags.

The flow of main can be seen below.

```
int main(int argc, char *argv[]) {
    modifyTerminal();
    initialize();
    args(argc, argv);

    setMessage("Ctrl-Q = QUIT | Ctrl-X = HELP | Ctrl-S = SAVE | Ctrl-F =
        SPELLCHECK | Ctrl-C = COPY FILE | Ctrl-D = DELETE FILE");

    /** Editor screen flow **/
    while (1) {
        displayScreen();
        processKeypress();
    }

    return 0;
}
```

FEATURE #4 and #5

2.4.6 Change Log

The change logs will be created automatically and appended to after every save of a file with the editor. In order to achieve this, for an open file in the editor, the first save of this file will create a log file with the same name as the saved file. Then the contents of the buffer will be saved to both the file and the log of that file. This file will be formatted to ease view and will provide a timestamp as well as number of lines written at the time.

```
char* changeLogFilename(char *filename) {
    char *logfile = strdup(filename);
    int index;
    char *p;
    if(strstr(logfile, ".")!=NULL){
        p = strchr(logfile, '.');
        index = (int) (p - logfile);
    }
    /** Get rid of .txt extension and concatenate
     * the .log extension then return **/
    logfile[strlen(filename) - index] = '\0';
    strcat(logfile, ".log");
    return logfile;
}
```

```
for (int i=0;i<BAR_LENGTH;i++)
    fputs("####", fp);
    NEWLINE

    /** Append a timestamp for every log save **/
    fputs("Time of save: ", fp);
    fputs(ctime(&t), fp);

    /** Append the current file to the log with,
     * even amounts of space in between **/
    for (int i=1; i<=(SPACE*2)-1;i++) {
        if (i % SPACE == 0)
            fputs(buf, fp);
        else
            NEWLINE
    }

    /** Append the number of lines written **/
    fputs("Total lines written: ", fp);
    sprintf(rows, "%d", E.numrows);
    fputs(rows, fp);
    NEWLINE
```

NEWLINE is a definition that appends a newline \n character to the log file. The change log is accessible for every file that was saved with the editor through a command line flag "-log <original_filename>".

2.4.7 Spelling Checker

This feature runs a spell checker upon a CTRL-F request and highlights the misspelled words on the screen.

Feature #5.a

Functions	Purpose
spellCheck()	Iterates over the editing buffer rows and sends each to the spellChecker function. Renders and highlights detected misspelled words in each row.
loadDictionary()	Loads a dictionary into memory as a Trie structure.
unloadDictionary()	Calls unload and checks for errors.
load()	Traverses through and fills in a Trie of size as the dictionary.
unload()	Frees the Trie nodeBucket.
spellChecker()	Given a row of text, returns number of misspelled words, using check.
getMisspellings()	Returns a 'misspellings' structure which stores the start and end index of the occurrence of a misspelling in a given row.
check()	Traverses the Trie to check whether the given word exists in the dictionary in memory.
highlightWords()	Sets misspelled words with a start and end index to the enum type "MISPELLED".

Figure 27: Functions required to implement Feature #5

Running a Spell Check

When the user presses CTRL-F, the Key Processor Component will make a call to spellCheck. This function first loads the a Trie dictionary into memory; then, iterates over each row in the editing buffer and calls spellChecker. spellChecker, is a function from the speller header, given a row of text finds all the misspelled words in the row and returns the number of misspellings. For each row that these misspellings are found for, the misspelled words are iterated over and their start and end indexes are retrieved and set to the editor's namespace as E.start and E.end. These will be used in the next feature objective, highlighting the words.

Loading and Unloading the Dictionary

```
int loadDictionary() {
    bool loaded = load(DICTIONARY);
    /** Check for errors **/
    if (!loaded) {
        printf("Could not load %s.\n", DICTIONARY);
        return 1;
    }
    return 0;
}
```

The load function is exposed from the dictionary header. This function takes a dictionary of some size allocates a node bucket of the size of this dictionary. With the use of a pointer, the given dictionary buffer is traversed and the Trie is filled in. As discussed in design, traversal

starts with the root node and each node has a value and n pointers to n letters in the alphabet. In this case, 26 pointers from each node to each letter in the English alphabet. At the end of a given word, the node is given a value of `true`. This means a word has been created. Then the traversal node goes back to root and iterates over the entire dictionary buffer.

Unloading consists of freeing the allocated node bucket.

The code for load and unload can be found in the `dictionary.c` file.

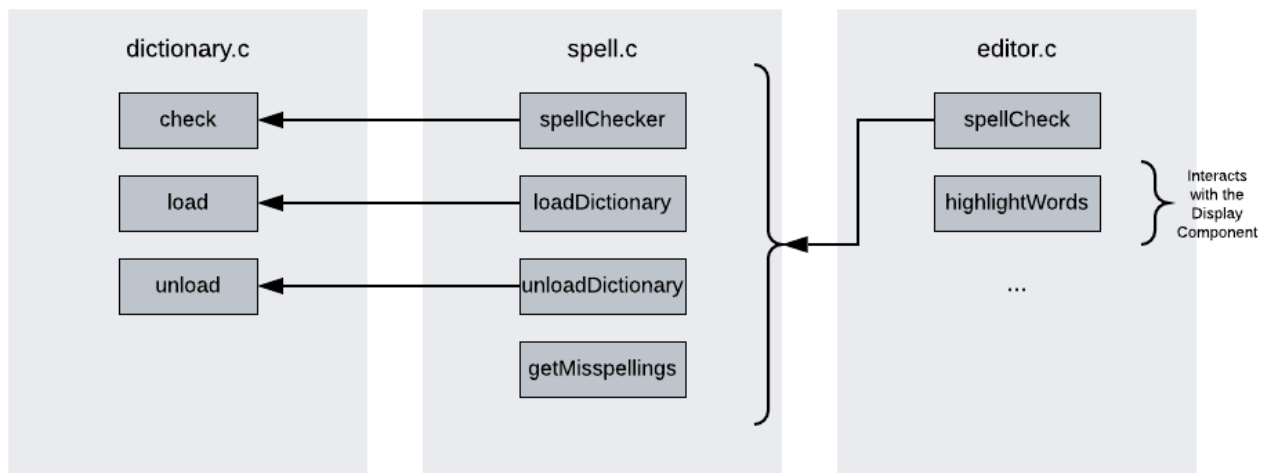


Figure 28: The files and functions involved in the Spelling Checking component

```

void spellCheck() {
    loadDictionary();

    for (int i=0; i<E.numrows; i++) {
        /** Send each row of the buffer to the spell checker function**/
        int miss = spellChecker(E.row[i].chars, E.row[i].size,0);

        for (int j=0; j<miss;j++) {
            /** For each misspelled word in the row, gets its start and
             * end index on the row **/
            E.highlight = true;
            E.start = getMisspellings(j).start;
            E.end = getMisspellings(j).end;
            ...
        }
    }
}

```

The `spellChecker` is the function that prompts the loading of the Trie and traversing the tree for each word in the passed row.

- Iterate over each word in the given row of text.
- Find acceptable words that can be checked against the dictionary.
- If `check` returns false, fill in the `misspellings` structure and append it to an array of structures.

- Return the number of misspellings at the end of the row.

Search the Trie

```
int spellChecker(const char* text, int len, int misspellings) {
    int index = 0, words = 0, cursor = 0;
    char word[LENGTH+1];

    /** Iterate over the given text */
    for (int i = 0; i < len; i++) {
        ...
        /** FIND 'ACCEPTABLE WORDS' FIRST, code below */
        ...
    } else if (index > 0) {
        /** For an acceptable word, check
         * its spelling against the dictionary */
        word[index] = '\0';
        words++;

        if (!check(word)) {
            /** Reallocate space for the array if it is
             * exceeded. */
            if (misspellings-1==INITIAL_SIZE) {
                struct misspelling *miswords = realloc(miswords,
                    (sizeof(spell) * INITIAL_SIZE*2));
            }
            miswords[misspellings].start = (cursor-index);
            miswords[misspellings].end = cursor;
            misspellings++;
        }
        index = 0;
    }
    cursor++;
}

/** Cleaning up */
return misspellings;
}
```

The check function traverses the Trie, given a word.

- Given a word, the traversal node is set to root.
- For every character of the given word, the pointer to the correct letter in the Trie is checked.
- If the pointer points to a NULL node for the next letter, the word is not in the dictionary, otherwise the traversal continues until the end of the word and the value of the node is returned.
- The returned value of the node will be true if the checked word is a word in the dictionary, otherwise not.

```

bool check(const char* word) {
    /** Create a traverse pointer and points to root */
    node* trav = NULL;
    trav = root;

    int i = 0;
    /** Iterate over every char in the given word */
    while(word[i] != '\0')
    {
        /** Make all letters lower case */
        char c = ( isalpha(word[i])) ? tolower(word[i]) : word[i];

        /** Letters */
        else if(isalpha(c))
        {
            if(trav->children[c - 'a'] == NULL)
                return false;
            trav = trav->children[c - 'a'];
        }
        i++;
    }

    /** Return the value of the node if
     * none of the nodes were NULL */
    return trav->is_word;
}

```

Another case is written for checking words with apostrophes (see the `dictionary.c` file). Additionally the structure of the Trie, usage of the nodes, and the behaviour of the traversal can be visualized as seen below.

The figure below aims to show an example traversal of the Trie structure. Things to note are that each node has an array of pointers and the values of the nodes can be true, false, or simply NULL. NULL nodes mean that the word's prefix is not even in the dictionary while false means that the word is not valid but the prefix exists.

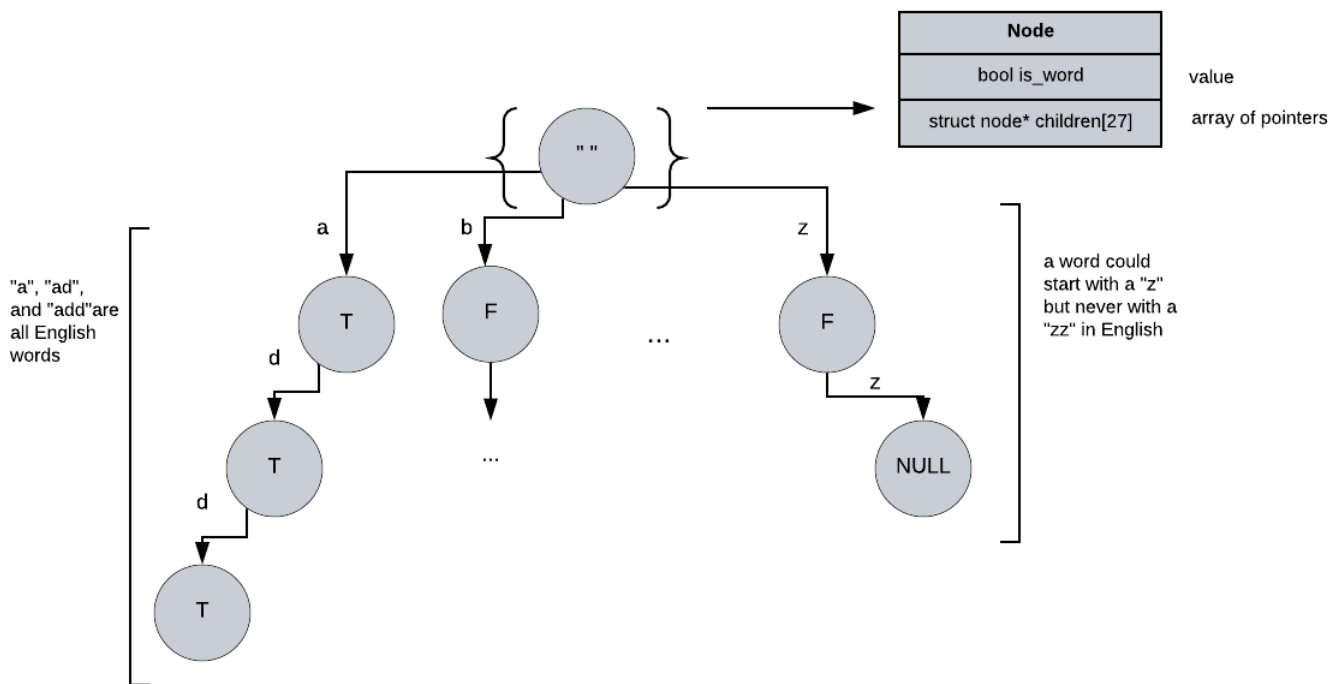


Figure 29: Trie Traversal example

Acceptable Words

As discussed in Assumptions (see 2.1.4) the only considered language here is English. With this there is a limit to the length of words from 1 to maximum of 45, according to the Oxford English Dictionary /citedictionary. Some words will also need to be simply ignored; these are words with numbers in them. Accepted words, on the other hand, are those with letters or apostrophes under the length of 45.

```

/** Find acceptable words that can be checked */
if (isalpha(text[i]) || (text[i] == '\\' && index > 0)) {
    /** Accept usual letters and apostrophes,
     * append to the word string */

    word[index] = text[i];
    index++;

    /** If the word exceeds the maximum
     * (45 in English), skip it */
    if (index > LENGTH) {
        while (isalpha(text[i])) i++;
        index = 0;
    }
}

} else if (isdigit(text[i])) {
    /** Skip words with digits */
    while (isalnum(text[i])) i++;
    index = 0;
}

```

...

The misspellings structure

This structure is filled in for each detected misspelled word in a given row. The start and end index of a word in a row is filled in and this structure is 'appended' to an array of these structures. This array is accessible from another function `getMisspellings`. This getter function is exposed to the editor script through the `speller` header file. Given the index of the misspelling, returns the `misspellings` structure. The intrinsic value of the misspelling, the word itself, doesn't matter. Only the range of the word must be used to highlight the screen.

```
/** info about detected misspelled words */  
struct misspelling {  
    int start,end;  
}spell;
```

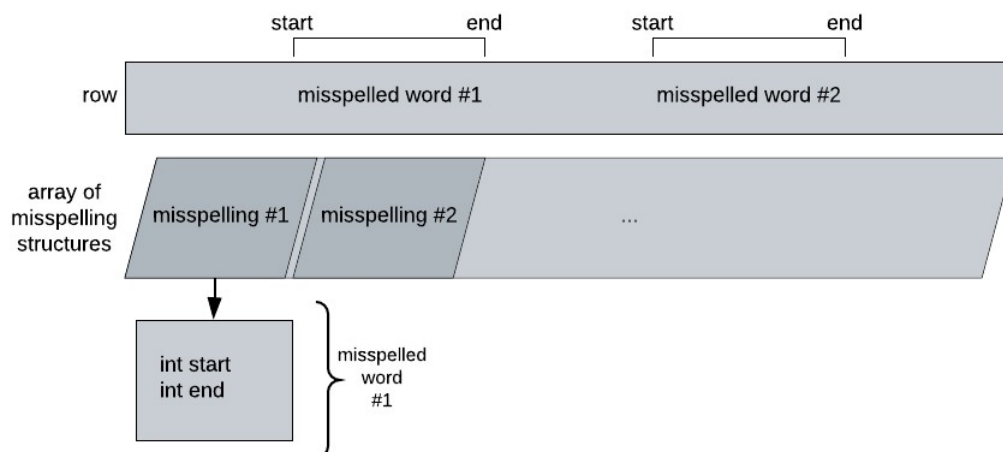


Figure 30: Diagram of the array of misspellings structures.

```
struct misspelling getMisspellings(int index) {  
    return miswords[index];  
}  
/** Array of misspelling structures */  
struct misspelling *miswords;
```

The array of misspelling structures is initially set to 10 for each row, when the dictionary is first loaded, reallocation is considered when the limit is reached (see the `spellChecker` code above). The size of the array is doubled each time the limit is reached. This introduces stability and the code is not at risk of accessing random memory locations that it shouldn't be reading and writing to.

Feature #5.b: Highlighting Misspelled Words on the Display

This objective needs to be fulfilled so that for some highlighted misspelled words in a row, they must remain highlighted until that row is edited. In which case, the entire row is 'un-highlighted'.

To achieve this, the range of the misspelled word in the row needs to be labelled differently from the rest of the row. This way, `displayRows` can be updated to go through each index of a row and check its type before writing to the editing buffer. This type could be an enum binary value; e.g. 0 and 1 to represent a normal word vs a misspelled word. Then, each row structure could possess an extra integer array field. This array would store the type value of each byte in the row.

```
/** Used for the spell checker **/  
enum wordType {  
    NORMAL = 0,  
    MISPELLED  
};  
  
/** Holds each row of a read file **/  
typedef struct rows {  
    unsigned char *hl;  
    ...  
} rows;
```



Figure 31: Example of how highlighting works with word types.

At first, each row is a string of 0s, after `spellCheck` is ran, for each misspelled word found in the row `highlightWords` will be called such that that range of that word in the row is set to all 1s. At the end of the spell check, `displayRows` will run and invert the color of all ranges that are typed as 1s. For example, in the figure above it can be seen that the misspelled version of "right", "rihgt", is given a string of 1s. This will highlight the misspelling with ANSI escape codes in the Display component.

```
void highlightWords(rows *row) {  
    /** Allocate space to the highlighting field **/  
    row->hl = realloc(row->hl, row->rsz);  
  
    if (!E.highlight)  
        /** If not in highlight mode, set to NORMAL **/  
        memset(row->hl, NORMAL, row->rsz);  
  
    for (int i = 0; i < row->rsz; i++) {  
        /** Iterate over the entire row and where  
         * the misspelled word starts and ends, set
```

```
    * to MISPELLED **/  
    if (i >= E.start && i <= E.end) {  
        row->hl[i] = MISPELLED;  
    }  
}  
}
```

This concludes this feature, having met both the objectives for Feature #5. Now all components can be tested individually and evaluated against the preset functional requirements/objectives.

2.5 Testing

In order to test this program, a manual approach will be taken and each component/unit will be tested individually to meet the preset functional objectives. The Unit testing approach has been taken due to the nature of the program. The goal is to validate the solution against the specification and refer to the design in the process [14], [15].

It's important to note that testing has been carried out iteratively throughout the development process for each developed component. The test cases described and presented in this section are a small sample that are indicative of an exhaustive list of the total number of tests carried out.

Since the editor is dynamic and changes display based on user input, it would be useful to test all these cases and record videos of each.

Please access this link to see the resulting behavior of each test case described below:

<https://youtu.be/-pAq43psZnk>

Each section in the video above is mapped to the test case tables below. The video's purpose is to showcase the resulting behavior given the test cases and inputs.

2.5.1 Feature #1

TEST #1 & #2 : Key Input Processing Component

TEST 1:

Test	Input is accepted to the program immediately after pressing the keys.
Input	"abcdef"
Expected Behaviour	All of the keys above seen in standard output.

TEST 2:

Test	Exit the program with control keys.
Input	CTRL-Q
Expected Behaviour	The program exits, clears terminal and homes the cursor to initial position.

Figure 32: Test 1: Raw Input

HORIZONTAL SCROLLING

Test	Load a file with a long row of text and scroll to the right to view it all.
Input	Arrow keys left & right.
Expected Behaviour	Display the rest of the characters as the user scrolls.

VERTICAL SCROLLING

Test	Load a long file of text and scroll to the bottom to view it all.
Input	Arrow keys up & down.
Expected Behaviour	Display the rest of the characters as the user scrolls.

Figure 33: Test 2: Navigation

TEST #3 : Display Component

TEST 1

Test	Open the editor program.
Input	None.
Expected Behaviour	Empty buffer. The second bar from the bottom is inverted in color and displays the number of lines read. (Should be 0 right now)

TEST 2

Test	Open the editor program.
Input	Non-control keys.
Expected Behaviour	Empty buffer. The bottom bar displays some message for 5 seconds.

Figure 34: Test 3: Display

2.5.2 Feature #2

TEST #4 : File I/O Component

Test	Load a file to the editing buffer.
Input	The editor's source code file.
Expected Behaviour	Display the code in the buffer.

SAVE FILES

Test	Open an empty buffer. Modify it. Save the buffer to a file in current working directory.
Input	CTRL-S, (conditional) filename.txt
Expected Behaviour	If the file already existed in cwd, the message bar will say "Saved successfully". If not, the message bar will prompt the user with "Enter filename:" and take a name before saving successfully.

COPY TEST 1

Test	Open a file on the editor. Copy its contents to another.
Input	CTRL-C, file.txt, file2.txt
Expected Behaviour	The message bar will prompt the user for a destination filename with "Enter filename:". Then display, "file.txt was successfully saved to file2.txt."

COPY TEST 2

Test	Open a file on the editor. Copy its contents to another.
Input	CTRL-C, file.txt, file.txt
Expected Behaviour	Same as above, however, the message bar should display an error message "Cannot copy a file to itself."

DELETE FILES

Test	Open a file on the editor. Delete the file.
Input	CTRL-D
Expected Behaviour	The editor closes and the user is back on the terminal. The file is no longer in their current directory.

Figure 35: Test 4: Files

2.5.3 Feature #3

TEST #5 : Editing/Control Component

TEST #5 EDITING/CONTROL

Test	Open an empty buffer. Insert characters to the buffer.
Input	"hello world!"
Expected Behaviour	See each character right after it is pressed. End result is "hello world!".

NEWLINE TEST 1

Test	Open a buffer. Write a row of text. Insert a newline at the tail of the row.
Input	"hello world!", Enter
Expected Behaviour	The cursor moves to the bottom line and a new row is created in the buffer.

NEWLINE TEST 2

Test	Open a buffer. Write a row of text. Insert a newline in the middle of the row.
Input	"hello world!", Arrow left to split the text, Enter
Expected Behaviour	Move all the characters to the right of the cursor, when user presses Enter, to the bottom row.

NEWLINE TEST 3

Test	Open a buffer. Write a row of text. Insert a newline at the head of the row.
Input	"hello world!", Arrow left to the head, Enter
Expected Behaviour	Moves the entire row of text to the bottom row.

APPEND LINES : TEST 1

Test	Append a string of text to an existing file through the command line.
Input	./editor --append file.txt "string of text"
Expected Behaviour	A message is written to stdout: "Operation successful!". file.txt then contains the contents "a string of text" at the end of the file.

APPEND LINES : TEST 2

Test	Append a string of text to a new file through the command line.
Input	./editor --append new.txt "string of text"
Expected Behaviour	A message is written to stdout: "Operation successful!". new.txt then contains the contents "a string of text" at the end of the file.

Figure 36: Test 5: Editing

DELETE LINES

Test	Open a file with the editor. Delete an entire line of text.
Input	CTRL-K
Expected Behaviour	Deletes the selected line, with the cursor, and moves the following lines up by 1 row.

DELETE CHARACTERS : TEST 1

Test	Open a file with the editor. Delete a character from the middle or tail of a row of text.
Input	Arrow keys, (Backspace/Delete/CTRL-H).
Expected Behaviour	The character to the left of the cursor gets deleted and the cursor moves to the left.

DELETE CHARACTERS : TEST 2

Test	Open a file with the editor. Delete a character from the head of a row of text.
Input	Arrow keys, (Backspace/Delete/CTRL-H).
Expected Behaviour	The row, along with all the following rows, move up by 1. Unless it is the top of the file.

Figure 37: Test 5 continued: Editing

2.5.4 Feature #4

TEST #6 : Change Log

Test	Open a file with the editor, Make two different modifications and save after each one. Then view the change log for that file.
Input	Modification, CTRL-S, CTRL-Q, [./editor --log file.txt]
Expected Behaviour	The change log must contain both entries, newest at the bottom, along with timestamps at the time of save and the number of lines written.

Figure 38: Test 6: Change Log

2.5.5 Feature #5

TEST #7 : Spelling Checker

TEST 1

Test	Open the editor, copy in some text with misspelled words, run the spell checker.
Input	CTRL-F
Expected Behaviour	The misspelled words are highlighted on the screen with a message on the message bar "The misspelled words are highlighted. Found x."

TEST 2

Test	Open the editor, copy in the same text with the misspelled words fixed, run the spell checker.
Input	CTRL-F
Expected Behaviour	Nothing should be highlighted if the misspellings are fixed for sure. The message bar will display "".

TEST 3

Test	Open the editor, copy in some text with misspelled words, run the spell checker, then edit the row with the misspelled words.
Input	CTRL-F, Modifications
Expected Behaviour	The highlights are removed from the edited row.

Figure 39: Test 7: Spelling Checker

2.6 Evaluation

Functional Evaluation

There were 5 specific functional features to be implemented, which the Design and Implementation section explicitly stated. The program requirements have been decomposed and iterated over many times throughout the development of this editor. This emphasized the specification and improved on the design each time. This section will evaluate each section and how its design meets the preset functional requirements.

Feature #1

This feature focused mainly on the user and the ease of use of this editor. A 'user interface' was designed; such that, the user was given specific controls to interact with the editor. User inputs were taken real-time and processed immediately to present the requested behavior, e.g. CTRL-F to spell check or CTRL-Q to quit. The entire terminal was reserved for the editor screen and the user could view a status and message bar as specified as the last objective of this feature. The information that the user has access to was the name of the loaded file, length of that file, and the line that the user's cursor was pointing to at any point in time. The message bar would let the user know about some outcome of the available functions or prompt the user to request information such as a filename to create a new file. Lastly, the user was given the ability to easily navigate the interface with arrow keys; where the loaded file exceeded the terminal screen, the user was able to scroll both vertically and horizontally to view the exceeding information.

Feature #2

This feature provided the ability to manipulate files in the user's current working directory. It has been made possible to load files into a temporary editing buffer. In order to move the contents of a modified buffer to the file system, it was possible to save files. Two specific features from the specification were copying and deleting files. Both have been implemented as controls to the user CTRL-C and CTRL-D.

Feature #3

This feature completed the main text editor such that now the user could load files, navigate through them, and make modifications. Modification controls in this feature were the insertion and deletion of characters from the editing buffer. While insertion is simply any letter from a user's keyboard, the behavior of newlines and backspaces on the buffer rows were complex to implement and think through. Both keys have been made available and tested thoroughly, with some representative sample of tests given in the Testing section above. Finally, for easy deletion of text, a control was introduced (CTRL-K) to delete entire lines at once and move all following rows up by one. This was inspired by GNU nano, a UNIX command-line editor.

Feature #4

This feature was a part of the specification, as a general operation. (Another general operation, display number of lines read from a file, was provided in Feature #1 - on the status bar) The change log worked to append every version of a file to a single log file. This log file, not only included the file contents between versions but also a timestamp of save and the number of

lines written at the time. This log could be viewed for every file through a simple flag on the command line '-log', followed by the name of the file the log belongs to.

Feature #5

The final feature, one of the additional ones, provided a spell check to the edited files. The user could simply load a file to the editor buffer and check its spelling, real-time, during modification. This feature was a control, CTRL-F, and found every misspelled word by checking each row up against a dictionary in memory. This feature made use of Tries, also known as prefix trees, to do so. The misspelled words were made apparent to the user through highlights on the terminal buffer. The highlights would only disappear if the user edited the row they were on.

This concludes all the functional requirements and their evaluation. Each sentence was an explanation of how the specific objectives in Figure 9 were met. In conclusion, the full scope, file, line, and the general operations of a change log and spelling checker were met and thoroughly tested.

Final section is the non-functional requirements.

Non-Functional Evaluation

Reliability

During the final testing, there were no noted reports of crashing in any of the implemented components. Each component was already tested during development and its inputs and outputs were iterated over throughout. Any input errors from the user were handled with the implemented message bar in the Display component, if not any File I/O or standard input errors were handled with an appropriate function that exits the program safely and displays an appropriate error message to inform the user. In any case, at exit, the program always restores the initial properties of the user's terminal.

Portability

The program made use of low level systems programming and some headers that are specific to be used in a UNIX-like system. The editor is portable to any UNIX-like terminal and the key input processing component takes care of multiple consoles like LINUX/RXVT, FREEBSD, etc.. The editor is POSIX compliant.

Usability

The user is presented with a set of control keys, all explained in both the message bar of the editor screen as well as through a -help flag. The commands make sense such that CTRL-Q maps to quit, CTRL-D maps to deleting a file, while CTRL-C maps to copying one... The provided status and message bar overall fulfill the criteria. Navigation is simple through the universal arrow keys and for any long files, the user can scroll both vertically and horizontally, following their cursor.

Maintainability

The code for this editor was developed in separate, testable components. It is easy to add new features and make use of the key components like Display and Key Input Processing. The program was developed in a robust environment with the use of code metrics as feedback and

a proper Software Code Quality checker to check the overall security and complexity of the code. Any additional features that weren't directly mapped to the main components of the editor were included as header files and separate source code files. Specifically, the spelling checker and its dictionary are compiled with the editor if the spelling feature is to be used.

2.7 Conclusions & Optimizations

In conclusion of this document, to build on the basic text editor, two main optimization points were identified for the general and additional features. These were mainly based on previous research as well as the implementation of the design to the solution.

#1 Optimizing the Change Log

The change log currently has a one to one relationship with its parent text file. Upon saving every version of a text file, the log is appended to with that version. While this can be useful to look at problems will arise as the log file grows and it will be impossible for the user to easily view versions or retrieve them back.

With that being said the main points of optimization would be ease of use:

1. Make it possible to retrieve versions of the file given some unique ID, like timestamps.
2. Introduce a concept of 'expiration', such that versions older than a set threshold are discarded to make room for newer ones.
3. Introduce a Find feature to the editor in general. This would specifically be useful when the user looks at the history of their files.

These features would overall improve the user experience.

#2 Spelling Recommendations

Having implemented a spelling checker; it follows to ask, can the misspelled words also correct themselves? This refers to auto correct. Currently, editors like vim provide spelling correction in the form of recommendation files. This means, for a detected misspelling, when the user selects this word, vim presents the user with a list of close words to the misspelling. [3] The user can then select one of these to replace that word with. While this isn't bad, it could be optimized.

1. Provide a 'personal dictionary' to each user. This could map frequent misspellings to the words that the user corrected them to upon using the vim-like recommendations file.
2. Moving forward, as discussed in the Research section in the Introduction, as the command line advances it could be plausible to have access to cloud language models and reach Word Processor like capabilities on the command line. [4], [16]

2.8 Appendix

Context of 'text'

Go back to 2.1

'Text' in this context refers, strictly, to ASCII text (American Standard Code for Information Interchange), which is a character encoding standard. To make it possible for users to access these specific text files to display and manipulate this text editor, it cannot be ignored that most popular text editors (GNU nano and vim, to name some) follow a similar route of providing a buffer for the user to temporarily work with. This buffer refers to an allocated memory location for the user throughout the running process of the editor. It is possible to read existing files to a buffer, or simply provide an empty buffer; in both cases, the user can insert and delete text real time and when complete the user is able to save or just quit through control keys.

Systems level programming in C

Go back to 2.1

First of all, almost all of the UNIX kernel code was written in C in order to move the system to a higher level than assembly. One of the main reasons lies in the fact that in order to do some systems low level programming and work with the command-line interface we must make use of certain structures that are readily provided structures in C, such as termios.

Justification of Design Choice

Go back to 2.3

Routes to specification refinement: There are many ways to implement this editor such that it fits the given specification. Two plausible routes to take would be:

1. Allow input to the program only through the use of flags from the usual terminal. An example usage would be a command along the lines of `./editor -filename test.txt -insert "Hello World" 6` to insert the string Hello World to line 6 of test.txt in the current working directory presumably.
2. Provide an editing interface where the inputs of the user are displayed and taken real time to provide for both reading and writing to files. Include flags and arguments to the editor for uses such as deletion, copying of files and perhaps the change log display: e.g. `./editor -delete test.txt` or `./editor -change-log test.txt`. While these can be available options through the usual command line it could also be made available through a menu provided from the editor interface also.

It is clear to see why route 2 is much better than route 1. First of all, route 1 would be very difficult to use. For example, if the user wanted to insert a certain line to a file this cannot be done simultaneously with viewing the file and seeing your change be made real time. The editing of files will take a lot more effort and it's almost impossible to write programs with ease. Route 2 is essentially, what any other command-line editor in the market, such as vim and GNU nano, is...While both routes fit the specification it's the job of the programmer to think about ease of use and pick the best implementation even if it wasn't clearly specified. We don't wish to make the use of the end product extremely difficult and time inefficient just

for the sake of meeting the specification in which case the product actually holds no value.

POSIX modes of input

Go back to 2.3.1

POSIX systems support two basic modes of input: canonical and noncanonical. In canonical input processing mode, terminal input is processed in lines terminated by newline character. This means that no input can be read until an entire line has been typed by the user.

On the other hand, in non-canonical input processing mode, characters are not grouped into lines. This means that they are read byte by byte, immediately after being read. Granularity with which bytes are read are controlled by two setting parameters called MIN and TIME.

A termios structure looks as so:

```
#include <termios.h>

struct termios {
    tcflag_t c_iflag; /* input modes */
    tcflag_t c_oflag; /* output modes */
    tcflag_t c_cflag; /* control modes */
    tcflag_t c_lflag; /* local modes */
    cc_t c_cc[NCCS]; /* special characters */
};
```

Each “flag” field contains a number of flags (implemented as a bitmask) that can be individually enabled or disabled. The choice of canonical or non-canonical input is controlled by the ICANON flag in the `c_lflag` member of `struct termios`. This flag, along with many others e.g. ECHO may be turned off through the use of `~/tildes` followed by the flag name. This stands for a bitwise negation of the flag; hence, E.g. `~(ECHO)` means ‘do not ECHO’.

```
struct_name.c_lflag &= ~(ECHO | ICANON);
```

The primary programmatic interface to termios consists of two functions:

```
int tcgetattr(int fd, struct termios *termios_p);
int tcsetattr(int fd, int optional_actions, const struct termios
               *termios_p);
```

These functions are used to get and set parameters associated with the terminal. Upon specifying the file descriptor, `fd`, e.g. as `stdin` attributes of the current state of the terminal could be read into a global `termios` structure. Then an instance of this structure can be modified through the program in order to switch to non-canonical mode of input throughout the execution of the editor and upon exit the original `termios` structure can be set again to restore the previous terminal mode.

ANSI Escape Codes

Go back to 2.3.1

```
vt sequences:
<esc>[1~      - Home           <esc>[16~     -
<esc>[2~      - Insert        <esc>[17~     - F6
<esc>[3~      - Delete        <esc>[18~     - F7
<esc>[4~      - End            <esc>[19~     - F8
<esc>[5~      - PgUp          <esc>[20~     - F9
<esc>[6~      - PgDn          <esc>[21~     - F10
<esc>[7~      - Home           <esc>[22~     -
<esc>[8~      - End            <esc>[23~     - F11
<esc>[9~      -                <esc>[24~     - F12
<esc>[10~     - F0             <esc>[25~     - F13
<esc>[11~     - F1             <esc>[26~     - F14
<esc>[12~     - F2             <esc>[27~     -
<esc>[13~     - F3             <esc>[28~     - F15
<esc>[14~     - F4             <esc>[29~     - F16
<esc>[15~     - F5             <esc>[30~     -

xterm sequences:
<esc>[A       - Up             <esc>[K       -
<esc>[B       - Down           <esc>[L       -
<esc>[C       - Right          <esc>[M       -
<esc>[D       - Left           <esc>[N       -
<esc>[E       -                <esc>[O       -
<esc>[F       - End            <esc>[1P      - F1
<esc>[G       - Keypad 5       <esc>[1Q      - F2
<esc>[H       - Home           <esc>[1R      - F3
<esc>[I       -                <esc>[1S      - F4
<esc>[J       -                <esc>[T       -
```

Figure 40: ANSI escape sequences for terminal input.

The winsize structure

Go back to 2.3.2

The size of the terminal window can be accessed through the use of `ioctl` and the `winsize` structure. `ioctl` stands for i/o control, it is a system call for device-specific I/O operations. The function takes a parameter that specifies the request code; in this case, the request code is `TIOCGWINSZ`. This gets the window size (GWINSZ) using `ioctl` (IOC). E.g. Define a `winsize` structure and call it `ws` then, `ioctl(STDIN_FILENO, TIOCGWINSZ, &ws)` would populate the `ws` structure with data about the current terminal window where `ws_row` and `ws_col` would specify the total rows and columns currently in view. These two pieces of information could also be stored in the global structure about the editor where the cursor positions are stored.

Tries

Go back to 2.3.5

The data structure that the dictionary is loaded into must be selected wisely for this use case. Two common options are hash tables and Tries. Both are somewhat reminiscent of each other but differ where hash tables, first of all, use hash functions as well as arrays combined with linked lists. Trie trees also use arrays, but combines them with an array of pointers from each node to its child nodes. Trie trees represents nodes in order and are uniquely retrievable. This makes it so that not only do Trie trees don't require hash functions, they also don't have any collisions to deal with. [17]

A downside of Trie trees is the space complexity due to the unnecessary storage of null pointers to child nodes if they are not being used. E.g. there is no prefix 'blc' in the English language however that the l node must have a pointer to the null c child node.

However, there is a space-time trade off here considering that the insert, search, and delete operations of this tree are achieved in $O(n)$ complexity where n is the length of the string being processed. The worst case scenario is simply a word of length 45, which is the longest word in English. Where this worst case scenario takes place, the biggest worry would be space complexity.

Overall, Tries are really useful for applications such as spelling checkers. This is because of their structure. One can quickly look up prefixes of keys, enumerate all entries with a given prefix, etc. The overall linked structure provides for efficient traversals with unique order for all words of a dictionary.

2.9 References

References

- [1] Combinatorial problems. <https://www.techiedelight.com/find-all-n-digit-numbers-given-sum-digits/>. Accessed: 2021-01-25.
- [2] Biswajit Bhowmik. Dynamic programming – its principles, applications, strengths, and limitations. *International Journal of Engineering Science and Technology*, 2, 09 2010.
- [3] Steven Oualline. *Vi Improved*. New Riders Publishing, USA, 2001.
- [4] Mayank Agarwal, Jorge Barroso, Tathagata Chakraborti, Eli Dow, Kshitij Fadnis, Borja Godoy, and Kartik Talamadupula. Clai: A platform for ai skills on the command line. 05 2020.
- [5] S Divya, Alpesh Gajbe, Rajiv Nair, Ganesh Gajre, and Nagarjuna Gadiraju. Gnowsyst-mode in emacs for collaborative construction of knowledge networks in plain text. 10 2009.
- [6] The gnu c library. https://ftp.gnu.org/old-gnu/Manuals/glibc-2.2.3/html_node/libc_toc.html. Accessed: 2021-01-25.
- [7] Streams and file descriptors. https://www.gnu.org/software/libc/manual/html_node/Streams-and-File-Descriptors.html. Accessed: 2021-01-25.
- [8] Ansi escape codes. https://en.wikipedia.org/wiki/ANSI_escape_code. Accessed: 2021-01-25.
- [9] Trie dictionary. <https://github.com/Gaivile/spell-checker/>. Accessed: 2021-01-25.
- [10] Termios. <https://man7.org/linux/man-pages/man3/termios.3.html>. Accessed: 2021-01-25.
- [11] Kilo text editor. <https://github.com/antirez/kilo/>. Accessed: 2021-01-25.
- [12] Canonical input. https://www.gnu.org/software/libc/manual/html_node/Noncanonical-Input.html. Accessed: 2021-01-25.
- [13] ioctl. https://man7.org/linux/man-pages/man4/tty_ioctl.4.html. Accessed: 2021-01-25.
- [14] Golara Garousi, Vahid Garousi, Mahmoud Moussavi, Guenther Ruhe, and Brian Smith. Evaluating usage and quality of technical software documentation: An empirical study. *ACM International Conference Proceeding Series*, pages 24–35, 04 2013.
- [15] Ian Sommerville. *Engineering Software Products : an Introduction to Modern Software Engineering*. Pearson Education, 2020.
- [16] Alexander Bergmayr, Manuel Wimmer, and Michael Grossniklaus. Cloud modeling languages by example. *Proceedings - IEEE 7th International Conference on Service-Oriented Computing and Applications, SOCA 2014*, pages 137–146, 11 2014.
- [17] H. Shang and T. H. Merrettal. Tries for approximate string matching. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):540–547, 1996.