1. **Introduction / Purpose / Intent**

For this assignment I was tasked to write a program in C that suggests spelling corrections to entered English words. This input is compared to a list of over 105,000 words in a data file that is supplied by the assignment. If the input word is not found within the word list, the program is to suggest alternative spellings. These alternative spellings are required to be the same length as the entered word, and have the lowest hamming distance from the entered word. This hamming distance is defined as the number of characters that differ between the two words, for example span and spam have a hamming distance of 1 due to the differing last character.

There is a list of assumptions that I can make while coding the program. The program will not need to handle words with a length greater than 40 characters, to include the null character. If the input character has multiple suggested results of the same hamming distance, the program need only to report the first 5 in alphabetical order. Finally the words list contains less than 110,000 words, which should be read from the file into an array of 110,000 strings.

The program must check that the wordlist is provided as an argument when the program was called, and if it was not the program will terminate and provide an error message. The program must open the file to read the words and add them into an array. There is one word per line in the provided word list, all words are in lower case, and each line has a newline character ‘\n’ at the end that will need to be removed. The program will prompt the user for an input word to check, using fgets to read each word entered from the keyboard by the user. The input word will be converted to lower case and either confirmed that it exists in the word list or recommend an alternate spelling for the word as described above. The program will continue prompting for inputs until the user inputs a word of length zero. The program must utilize functions for loading the word list from the supplied data file and determining the hamming distance between two words.

For full points the program must perform as specified above as well as conform to specific coding standards. These standards include the use of meaningful names that give readers of the raw code a clue as to the purpose of the thing being named; variable, function, etc. Additionally while writing the program I must avoid the repeated use of numeric constants, and for any numeric constants used in the program the value must be assigned to a variable and then use that variable instead. The use of comments at the start of the program to identify the purpose of the program, the author, and the date it was written and modified. The use of comments are required at the start of each function to describe the purpose of the procedure and the purpose of each parameter to the procedure, as well as a description of any return values if any exist. The use of comments is required at the start of each section of the program to explain what the following portion of the program does. Finally, the use of consistent indentation is required.

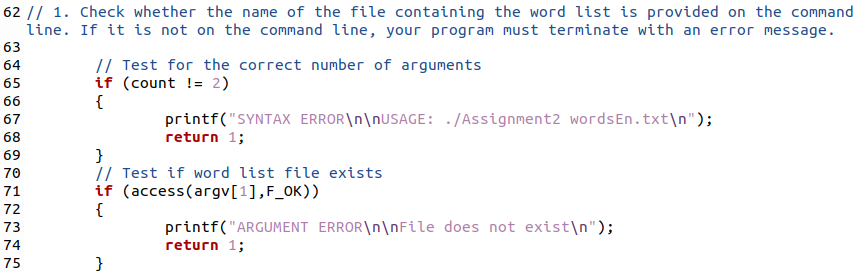
There exist 2 files that were provided by the assignment page on canvas that will be utilized by the program. The first, wordlist.txt, contains roughly 200 words and is useful during the programming portion of the assignment for on the fly verification of execution of code. The other, wordsEn.txt is the larger file that contains 110,000 English words and is intended for use in the final revision of the program.

1. **Process**

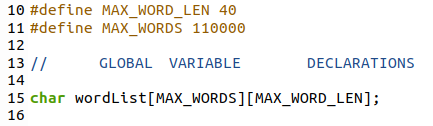
The first step in this lab was to break up the lab into multiple smaller tasks, then tackle each of those tasks by creating additional sub-tasks or steps for completion within the overarching task. To start this I copied in all of the requirements into a blank Assignment2.c file as comments, constructing a small shell for the program. The comments were placed in the general location that they would be relevant while coding, and the shell included a main() function, an area for function declarations, and a header comment to immediately accomplish one of the required coding standards. After this I began to write pseudo-code to identify the major tasks to be accomplished. This resulted in 3 tasks; load the file into an array, prompt the user for input, and compare input to the data held in the array. With this small bit of pseudo-code complete I turned my attention to the first task of reading in the file to the array.

* *Loading the File*

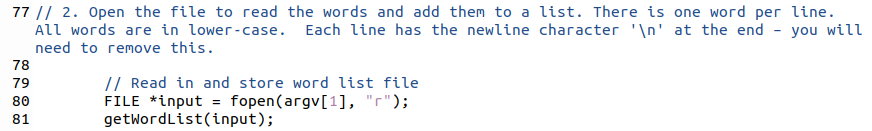
Initially I tackled the 2 required tasks of checking for the existence of the word list file as an argument and in memory. These were placed at the beginning of main and were accomplished with simple if statements. The test for the existence of the file in memory required use of the access() function, a function that I had not previously used. Essentially it does everything that is required by the task to verify its existence, and was utilized as an argument in the if statement. With these tasks complete and tested functional, I turned my attention using the file in memory.

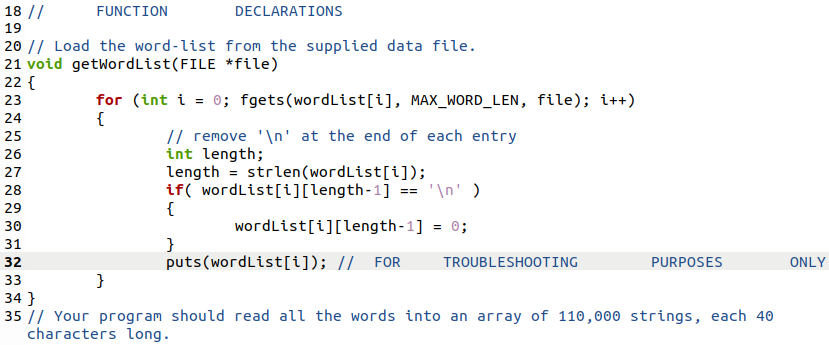


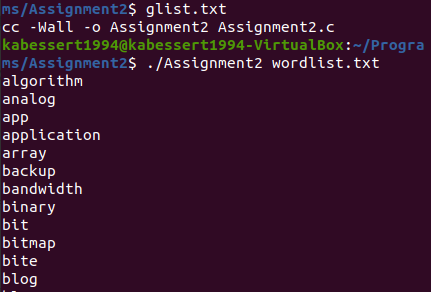
Before I could do anything with the file, I needed to establish the array of strings that would hold the data. This step was provided to me by the assumption section of the assignment page on canvas, including 2 define statements. I chose to implement the word list as a global variable, as it would be used inside of main() and functions. This utilization does not contradict any requirements of the assignment page.



Opening the file and the utilization of fgets() was provided by the “What your program must do” section of the assignment page. It is important to note that initially I coded the reading in of the file directly into main, and later moved it into a function as I reviewed the requirements of the lab, so the screenshots below may not exactly match what the content of the program looked like during the writing process. I constructed a simple for loop that utilized fgets() as the testing condition, this way as long as there was content to read in the file the loop would continue. The loop would increment the index variable and place the word read in into the global **wordlist**[] array. This method tested successfully, however did not remove the newline character at the end of each line. This functionality would be added later, after the loop was moved to the function. With a nested if statement that utilized the strlen() function. With this, the first task was complete and I moved on.

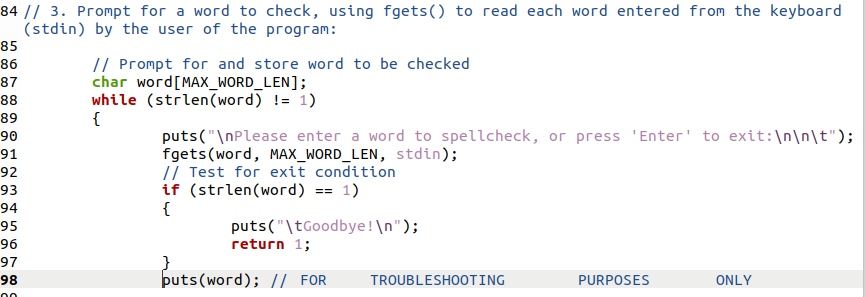


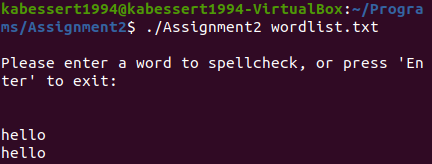


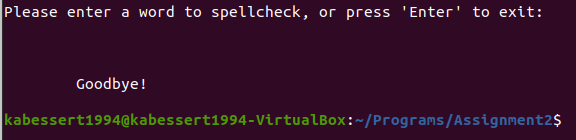


* *Prompt the User for Input*

This next task was similarly small, utilizing the fgets() function to receive input from stdin. This input was stored in a string variable **word**. I constructed a while statement around the function call that tested the length of **word** with the use of strlen(). This would later become the exit condition required. I utilized a puts() statement for testing, which displayed the successful storage of the input word. Testing of the exit condition was also successful. If blank input was placed into the program, the exit condition would execute as designed. This is contrary to what the assignment document listed, as the assignment lists the condition of a string length of 0, however it is my belief that the intent of the exit condition is met, and the detection of a single character is for the null character “\0” that must exist in all strings. Once this was complete I could move on to the next major task.



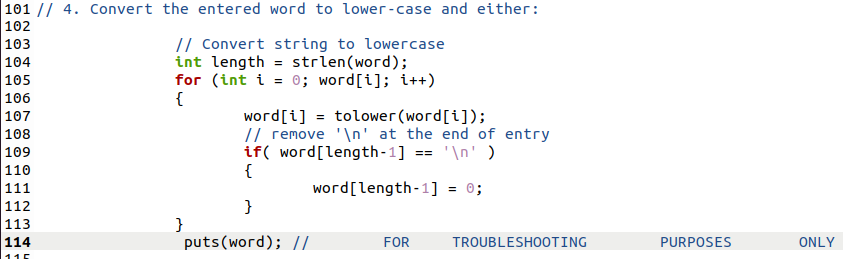


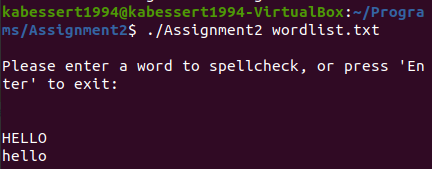


* *Manipulate the Input*

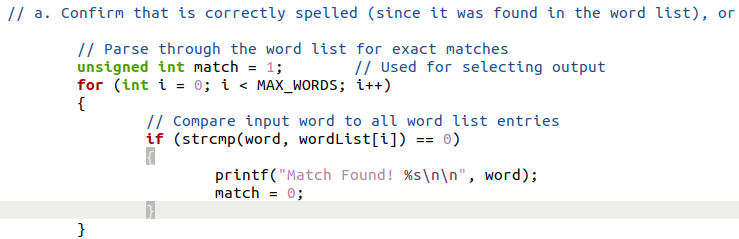
This was the largest of the 3 major tasks to accomplish, so I decided to break this task up into smaller subtasks; to include converting the input to lower case, checking if the input word matched any in the database, and finally checking the hamming distance of the input word to those found in the database. During this portion of the coding process I utilized the provided wordlist.txt file for quick and efficient testing.

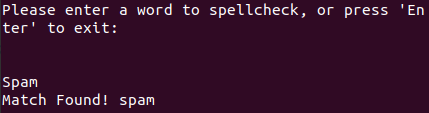
The first subtask was a simple implementation of a loop that utilized the tolower() function. This needed to be implemented to ensure that any input placed in the command line would be properly compared to the format the words were stored in the 2 word list files. I utilized a for loop that would index through each letter of the input word, utilize that letter as input in the tolower() function, and placed the returned value back into the word. This completed the first subtask, and I was now ready to tackle the real task of the program.



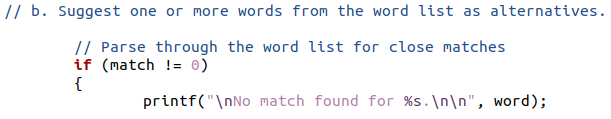


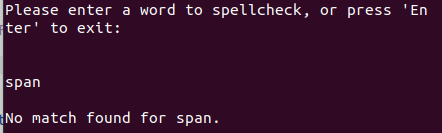
Checking the new version of the input word against the word list provided as an argument required a little bit of planning. I implemented a for loop that would index through the array that held the wordlist entries and a nested if statement for comparing the altered input to each entry using strcmp(). If the strcmp found a match the if statement would execute and print to the screen that a match had been found, and the word was spelled correctly.



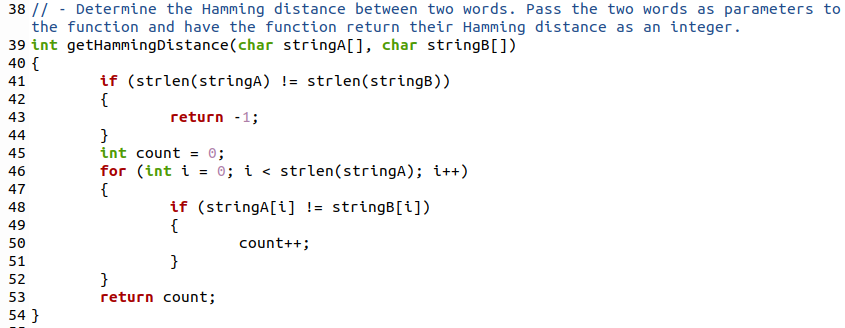


A more difficult of a task to accomplish would be to output when a match wasn’t found. This would have to exist outside of the for loop that iterates through the word list, otherwise it would be output after each word was unmatched. I decided to utilize another if statement after the database check, as this would be the start the task to dive into the hamming distance. For now I simply focused on how to display to the program user that no match was found. I settled on creating a new variable, prior to the loop that checks for the input to exists in the word list array, that would be set to 0 upon a match found. This became the testing condition for my if statement. As long as no match had been found, the **match** variable would still be 1, and as long as it was anything but a 0 the if statement would be executed.





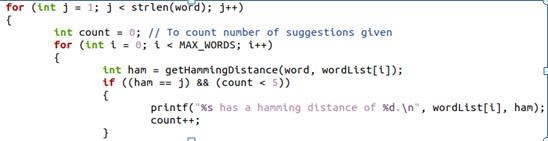
Inside this if statement is where I would craft the looping logic behind the inevitable function call to calculate hamming distance. I decided to first construct the hamming distance calculating function so I could have something to effectively test. This function would be passed 2 strings, 1 of which would be the input word that had failed to match any values in the word list array and 1 of which was the loop indexed value of the word list array. The very first thing to accomplish was to test both words lengths, as instructed only words of the same length would be recommended as alternative spellings. This was completed with an if statement and the use of the strlen() function on the 2 arguments. Next I declared a **count** variable that would be incremented for every letter that was different from the 2 arguments. I constructed a for loop that would increment through each letter of the input word and use a nested if statement to compare that value to the same letter location in the wordlist argument. If there was not a match then the if statement would execute, causing the count to be incremented. Finally the function returns the value of count. Since this function would only be called if no matches were found, this is as much as the program needs to perform.

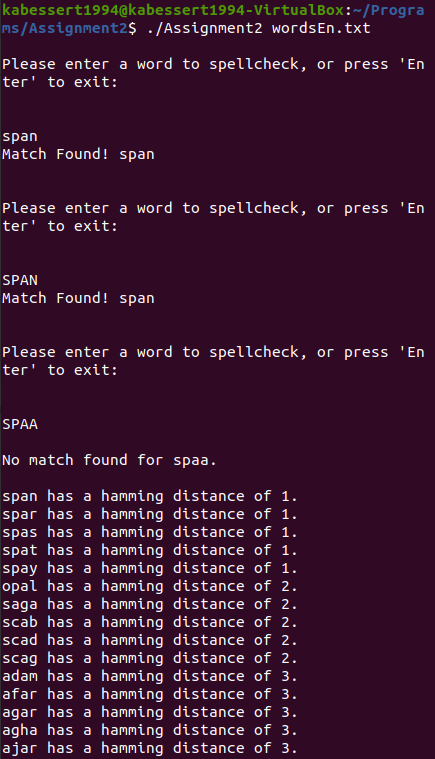


Returning to main(), I placed the call to getHammingDistance() inside the if statement that previously output that no matches were found in the word list array. I then constructed a for loop around the function call that would iterate through each of the words in the word list array and provide that value as an argument to getHammingDistance() along with our altered word that did not find any matches in the word list. This was successful, however it listed the hamming distance for every listing in the word list array that matched the same length as our input, so for “span” every 4 character word returned some hamming distance. To combat this I nested an if statement inside the for loop and declared into existence a **count** variable. This variable would be incremented after each printf() statement of the hamming distance found. The test statement for this if statement was to test the value of count under 5 and return value of getHammingDistance(), stored in the **ham** variable, was less than the length of our input variable. This ensured that the first 5 words with the same length, alphabetically, were printed to the terminal.

These could be of any hamming distance, and was not organized output. Finally, I constructed another for loop around the entirety of the code leading to the call of the function, still nested inside the if statement that tested if any matches had been found. This for loop would start at a value of 1 and increment as long as the indexed variable **j** was less than the length of the input word. This allowed me to place an additional condition in the deepest nested if statement that leads to the printing out to terminal and incrementing the **count** variable, testing the value of **ham** to equal the value of **j**. This logic ensure that a maximum of, 5 suggestions, alphabetically, per hamming distance value that is less than the length of the word are made to the user of the program. I chose to not include results that had a hamming distance of the word, as that would indicate that none of the letters provided matched. A single character was required to match to give output.

This loop logic does execute the comparison loop for each hamming value, resulting in much wasted processing power, however it successfully accomplishes what was tasked to me as a programmer despite the lack of efficiency. With the successful implementation of getHammingDistance(), and the correct formatting to the terminal, the final subtask is complete, simultaneously completing the final task and resulting in the completion of the program.



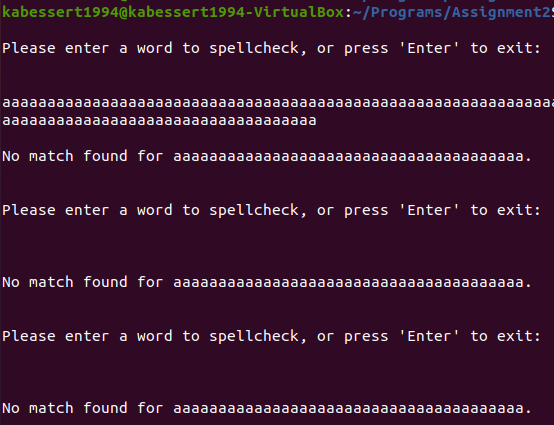


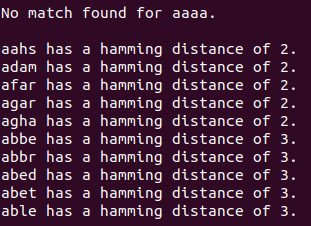
1. **Testing**

All throughout the coding process many “printf();” and “puts();” statements were used to output variables in various locations, including before and after variable assignment statements, before, in, and after loops, and in functions. These statements were used for the tracking of information and troubleshooting the overall logic and design of the program. These statements have been removed in the final submitted version of this code, although they remain in the working portion of this code.

During the programming portion of this assignment many conditions arose that required on the spot testing and troubleshooting to progress further in the assignment. These were not all recorded, however the largest of those was discussed heavily in the “Process” portion of this report. This was the task of outputting the words that had hamming distance values to the input word that was not in the word list array. While testing with the smaller of the two files there was no need to restrict the output of hamming distance as there were not many items to output. However once testing continued with the wordsEn.txt file with over 100,000 values it was clear that some restriction needed to be completed. This is when **count** was added as a procedure in the final loops to keep track of how many results had returned.

The first condition tested after completion of the program was for input that contained more than the maximum 39 character length. The value of “aaa…aaa” was tested at variable lengths. Results showed that as the string grew larger than the word variable, it would conduct the match test on all of the input in a loop, because more data existed in stdin. For my test, it ran the comparison against the word list a total of 6 times with a full string of 39 “a”s before a final check on the remaining “a”s. This last test then moved on to conducting the hamming output.. It seems that this is not harmful to the program and should be tested for as input is gathered, however with the way the program is designed, the variable input is stored in, **word**, will never exceed 40 characters. This is a problem with the way stdin functions. Because it is not harmful to the program I have decided to leave it be.

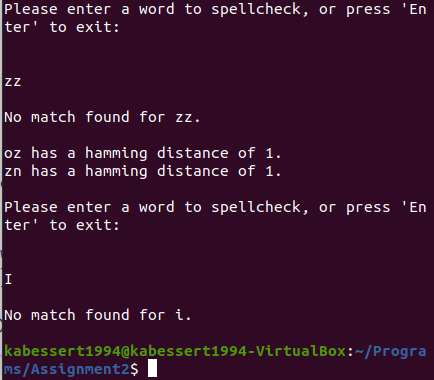




The next condition to test was of very long strings with the full wordEn.txt word list.this included strings of 20 chars, 30 chars, and 40 chars. The first test went as expected, successfully finding some matches of variable hamming distances, however the following 2 tests concluded only in returning no results. Upon investigations this is due to the availability of words with that high of character count.



Testing continued with extremely low character counts of values 3, 2, and 1. 3 and 2 length strings successfully executed as expected, however strings of a single character ended the program unintentionally. I initially thought this was because of the test I was conducting in the while loop that contained all of my procedure code “while (strlen(word) != 1)”, so I edited it attempting to allow for continued testing of strings with only a single char by changing it to “while (strlen(word) >= 1)”. This succeeded in producing the result I wanted, allowing the user of the program to enter a blank line to exit the program, and was implemented for the final revision of the code to be submitted.



1. **Results**

The results of this program are a successful in accomplishing the intent of the assignment. I have successfully written a C program that suggests spelling corrections to English words that have been input by a user. The input is compared to the data file containing over 100,000 English words, when that file is provided as an argument to the program, and if found informs the user that the spelling is correct. When the word is not found in the word list, the program suggests 5 alternative spellings to the word that are contained in the word list. These alternative spellings have the same length as the input word, and are output alphabetically and by hamming distance. I have abided by all assumptions while writing this program and have made deliberate attempts to accomplish all that was outlined within the assignment page on canvas.

The code provided for evaluation, additionally, conforms to the coding standards required for full points for this assignment. Meaningful names for functions and variables, avoiding numeric constants by utilizing variables when applicable, use of comments in the heading of the code, before functions, and in portions of code, and the use of consistent indentation has been applied. It is my belief that the provided final version of this code meets these requirements.

1. **Conclusions**

Based on the results and intent of this assignment I conclude that I have accomplished the goals of the assignment, which was to provide the programmer a task that would familiarize them with the use of files, gathering input from a user, and continue to reinforce the use of loops. The largest issue I had faced during this assignment was at the very end in which I needed to construct a way to organize the data I would output to the screen. The method I employed was not efficient in that it wastes many programming cycles to accomplish the task, however I am unsure of what I could do differently next time. I will continue to ponder on more efficient ways in which I could have accomplished outputting one hamming value at a time that does not require a reprocessing of the entirety of the wordsEn.txt data stored in the word list array.

1. **References / Acknowledgements**

C Programming Language, B. W. Kernighan & D. M. Ritchie, 2nd Edition, Prentice Hall, 1988.

C Programming: A Modern Approach, K.N. King, Norton, 2008.

[Access Function](https://pubs.opengroup.org/onlinepubs/009695399/functions/access.html)

Keegan Giles assisted in logic checking and understanding of the assignment on 21 Oct 2020