1. **Introduction / Purpose / Intent**

For this assignment I was tasked to write a program to decrypt an encrypted message, contained in a text file, to reveal the plaintext message. I was provided the encryption key that was used to encrypt the message, a detailed instruction of how the encryption was processed, a walkthrough of how decryption would occur, and a list of function declarations that would be needed in the decryption process.

The purpose of this assignment was to challenge my problem solving abilities as a programmer and to begin familiarizing myself with the C language. The scope of topics addressed in this assignment are the understanding of functions, loops, pointers, variables, variable types, bitwise operations, binary math, and troubleshooting.

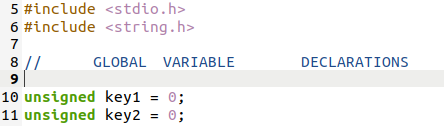
Successful completion of the assignment would be a functioning program that decrypts the encrypted message successfully, utilizing all of the functions provided in the “Hints” section of the assignment page, as they are written. Partial completion could be accomplishing the decryption task without utilizing all of the given functions, using modified versions of those functions, or using none of the functions and placing all code for the decryption routine in main(). Unsuccessful completion the assignment may be the production of some code that is unable to decrypt the message.

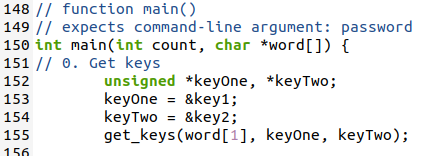
1. **Process**

The first step in this lab was to understand the process in which encryption took place. For this I opened a blank text editor in my Ubuntu VM and recorded each of the steps, in numbered order, with pseudo-code written below the overarching step that listed the potential tasks to complete for the step. This was placed in a comment block for reference and understanding how the original message was decoded. Next I recorded the steps, as written in the assignment page of canvas, for the decryption method with similar pseudo-code tasks. This list of steps and subtasks would be what I would be writing code to accomplish. Finally I copied into my text editor the provided function declarations from the ‘Hints’ section of the assignment page, along with the corresponding text as a comment above the declaration. I placed the main() function around the numbered steps and pseudo-code for the decryption process. I now had a list of smaller problems to tackle one at a time.

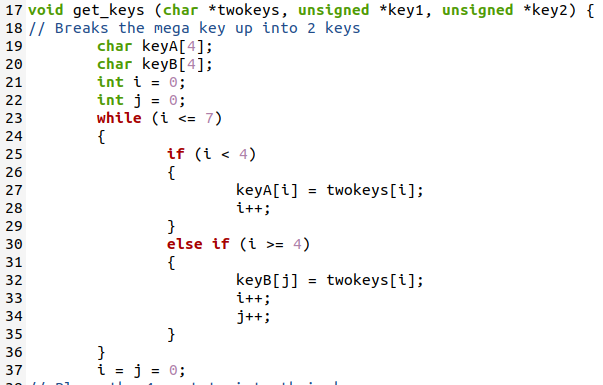
1. *Get keys;*

The first task to accomplish was to extract the 2 keys from the provided mega-key ‘password’ from input at the command line. The declaration for the function, provided by the ‘Hints’ section of the assignment, “*void get\_keys (char \*twokeys, unsigned \*key1, unsigned \*key2)*” indicates that all of the variables accepted as arguments will be pointers. To accomplish this I created and initialized 2 unsigned global variables, **key1** and **key2**, and 2 unsigned pointers to the global variables in main(), **\*keyOne** and **\*keyTwo**. I then assigned the pointers to their respective global variables with assignment statements. The pointers could be passed to the function get\_keys() along with the mega-key stored in **word**[1] of main from input.



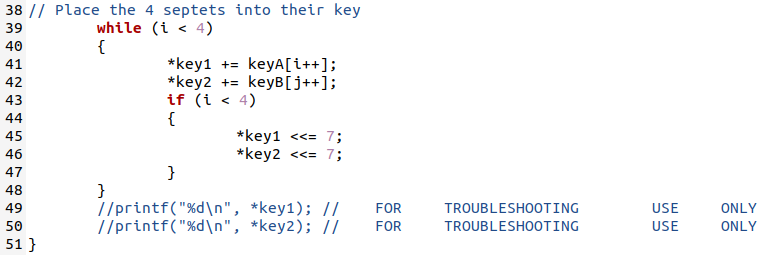


For the definition of the function get\_keys() my logic was to extract the first and last 4 characters from the **word**[1] variable into separate arrays, **keyA**[4] and **keyB**[4], so the characters “pass” were stored separately from “word”. These would become the values of the global variables **key1** and **key2**. I chose a while loop with nested if/else statements to accomplish the separation of the keys, incrementing through variable **i** to place characters 0-3 of **word**[1], “pass”, into **keyA** and incrementing through variable **j** to place characters 4-7, “word”, into **keyB**. An if statement was originally implemented for this process, however I experiences ‘stack smashing’ errors and chose to redesign this portion of the code.

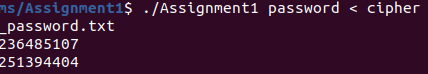


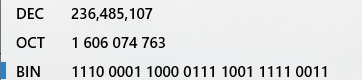
With the first sub-task of breaking up the mega-key now complete, I had to accomplish the next sub-task of placing each array value into the global variable for use outside the get\_keys() function.

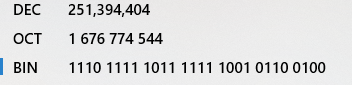
Part of this sub-task was to ensure that the last bit of each of the 4 bytes was removed, storing the 32 bits as 28 bits. Since the last bit of each character is always a 0, I accomplished this with a while loop, adding the value of the character stored in the array to the global variable and shifting left 7. The shift left 7 placed the bits in the global variable to overlap the last bit in the next character value being stored if added together. Control of the shift was given to a nested if statement that would not execute if the bits had already been shifted 3 times. This ensured that all 4 characters of the array for both keys were stored in their respective global variables.



I verified the values of my new 2 keys with troubleshooting printf() statements, as seen in the image above, and comparing these values to the expected values provided in the assignment page on canvas. The values matched and task 0 was completed successfully.

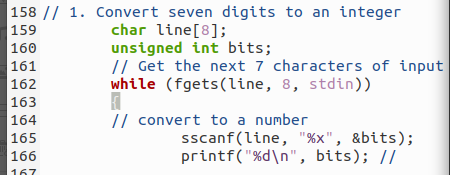




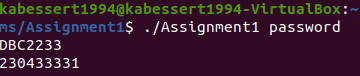


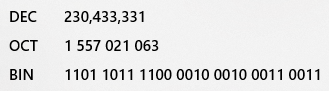
1. *Convert seven digits to an integer;*

This step was provided to me from the “Decryption” section of the assignment page on canvas. I implemented the code as written. 7 characters at a time would now be read from the coded message, translated into an integer number, and stored in the **bits** variable.



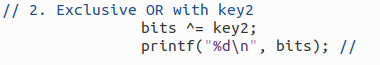
I tested the result by passing in the test case that was provided within the assignment page, “DBC2233”, and comparing the output of a printf() statement of that value against the decimal equivalent of the provided binary value. This test passed successfully and I could continue to the next task.



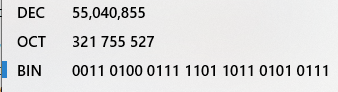


1. *Exclusive OR with key2;*

This step was a simple implementation of a line of code, using a bitwise operator. After code was implemented I tested the value of **bits** with a printf() statement to the provided expected result, which resulted in a pass and progression in the program.





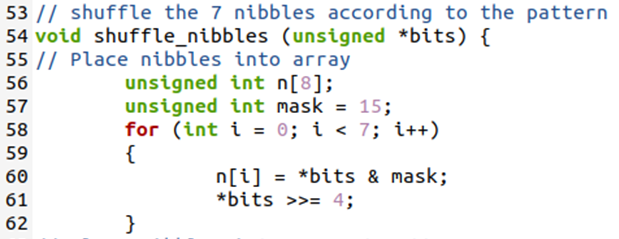


1. *Shuffle the Nibbles;*

This task was another function that was declared within the assignment text, “*void shuffle\_nibbles (unsigned \*bits)*”. I implemented the line of text in main that would call this function then returned to the declaration to begin work on the definition’s subtasks.

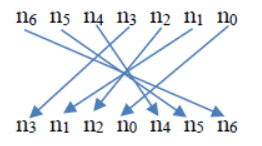


The first subtask to complete within the function definition was taking the variable passed, **bits**, and separating the nibbles for shuffling. To do this, I decided to place **bits** 4 bits at a time into an array with a mask. I declared **n[8]** to hold the nibbles and **mask**=15 for this operation. Initializing **mask** with a value of 15 translates to 1111 in binary, which will be used with the bitwise & operation to place the nibbles from **bits** into their corresponding **n[]** array. I chose to utilize a for loop, declaring **i** to step through the **n[i]** array one at a time and placing the nibbles inside. With each loop I shift **bits** to the right by 4, ensuring that each array’s nibbles are at the very beginning of the 8 bits of the **n[i]** char array. This loop is repeated 7 times, placing nibbles 1-7 in array locations n[0]-n[6] and leaving n[8] as a null byte.



With all of the nibbles filled into array locations, I utilized the diagram provided in the assignment text to determine the order with which to place the nibbles back into **bits**.

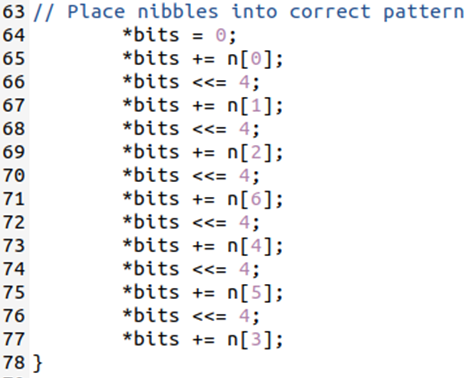
Encryption Method:



Decryption Method:



**Bits** is initialized to 0, as all of its data is stored within the **n[i]** array, and I proceed to add each nibble into **bits** in the order it should appear in the decryption method. After each nibble is added to **bits**, **bits** is shifted left 4 bits to make room for the next nibble with the exception of **n[3]**, as it is the last nibble to be added. After the completion of this subtask I had finished the shuffle\_nibbles() function and returned to main().



Once again I tested a printf() statement of the value of **bits** after the shuffle\_nibbles() call and compared the result with the expected result in the assignment text. This test proved successful and I moved onto the next task.



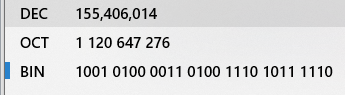


1. *Exclusive OR with key1;*

This task was another simple implementation of a line of code, using a bitwise operator. After code was implemented I tested the value of **bits** using a printf() statement to the provided expected result, which succeeded and moved on to the next task





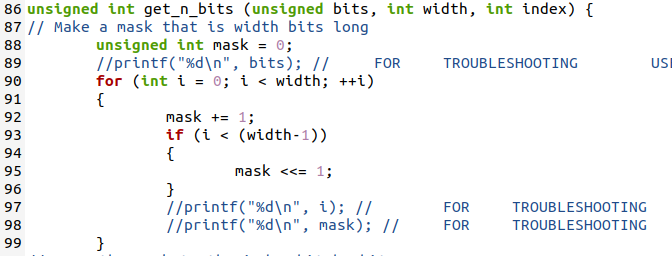
.

1. *Rotate septets;*

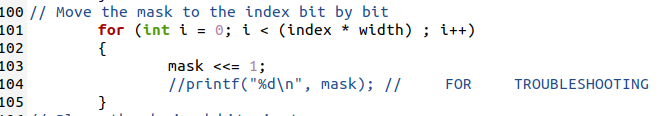
This task had multiple functions that were declared within the assignment text required for completion, “*unsigned int get\_n\_bits (unsigned bits, int width, int index)*” and “*unsigned int rotate\_left3(unsigned bits)*”. This is because in order to rotate the septets, I would first need to select the correct bits that made up the septets to be rotated. I chose to start with the get\_n\_bits() function.

***get\_n\_bits():***

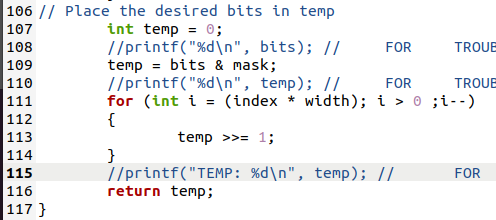
The declaration provided indicated that get\_n\_bits() accepts arguments for **bits**, which are the bits we have been manipulating up until now, along with 2 int values **width** and **index**. Because this is a selection function that returns a value of unsigned int, I decided I would need to declare a **mask** variable similarly to the one utilized in shuffle\_nibbles(). With **width** being a variable I decided to construct a for loop that would fill **mask** to whatever width would be specified. This for loop simply adds 1 to the value of mask and shifts it to the left 1, effectively multiplying it by 2. There is a nested if statement that prevents the final addition from shifting, similarly to shuffle\_nibbles(). With **mask** constructed I moved forward to the next subtask.



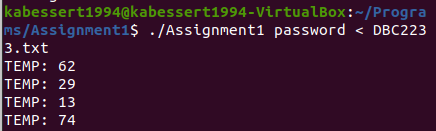
The mask needed to be shifted into the correct position to select the correct bits indicated by the **index**. A comment included on the assignment page indicated that *“// index \* width is the index of the rightmost bit to extract*” which indicated to me that **index** would only ever be 4 values within a 32 bit sequence, 0-3. Bits 0-7 are index = 0, 8-15 index = 1, 16-23 index = 2, and 24-32 index = 3. This is only true however with a **width** of 8. I constructed a for loop to place the mask in the correct position indicated by the (**width** \* **index**) product, using a left shift of that value. With mask in place, the subtask was complete and I moved on to the final subtask for the first function.

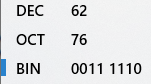


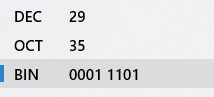
Finally, because we wish to return an unsigned int, I declared and initialized a **temp** variable and set the return statement to return **temp**. My intention is to place the bits I wish to manipulate into **temp** and return them to main() to be rotated. I use a bitwise & operation with **bits** and **mask** to obtain the septet I wish to work with and place it in **temp**. I then manipulate **temp** back to the least significant bits using a right shift of the value (**index** \* **width**). With this and the return statement, the subtask is complete along with get\_n\_bits().



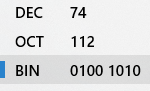
Before moving on I test my return values of **temp** with a printf() statement against my expected return value of temp, calculated on paper from the provided test case. With success I move on to the next function.





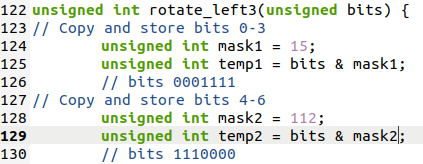




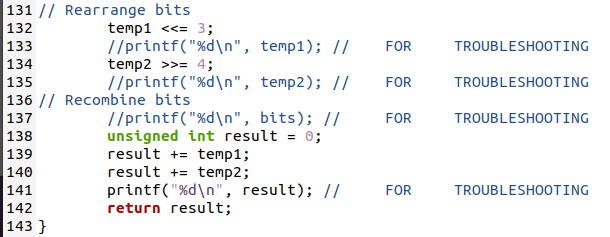


***rotate\_left3():***

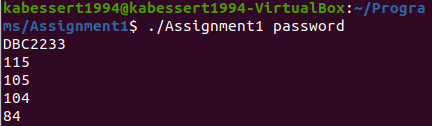
The declaration provided indicates that rotate\_left3() only takes a single argument and returns the unsigned integer. The way I plan to implement these functions will have the prior function, get\_n\_bits(), return the selected septet for manipulation as input for this function. For the first subtask of preserving the bits, I declare into existence 2 mask variables, **mask1** and **mask2**, and set their values to 15 and 112 respectively. Because the shift is going to occur inside the septet, and not an octet, I need to preserve and move the bits exactly rather than using simple rotation. I declare 2 temp variables, **temp1** and **temp2**, and use a bitwise & operation with **bits** that is being returned from the prior function call. With these bits preserved I move onto the next subtask.



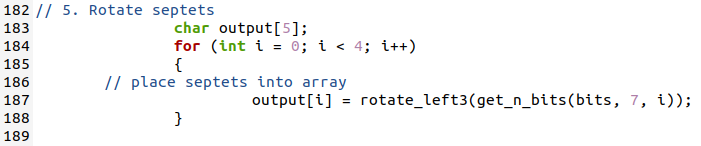
The next subtask is to rearrange and recombine these preserved bits. I used a bitwise left shift 3 on **temp1** and a bitwise right shift 4 on **temp2** to place the bits in their correct positions. With the rotation complete, I declare into existence a new variable, which will be returned to main(), **result**. I added each of the temp variables to **result**, which places them in their correct order, and completes the function.



Before proceeding I tested the expected return from the previous function, calculated on paper, to a printf() statement of **result**. Because the first septet received from get\_n\_bits() was 0011 1110 (62) the expected result would be 1110011 (115). The test passes and calculations show that the function is correctly ‘rotating’ the bits of each of the 4 septets. With this accomplished I return to main to implement the 2 functions together.

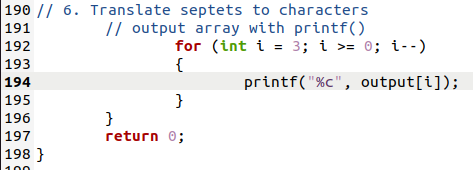


With both functions defined I only have to decide what to do with each rotated septet that is returned to main(). I decided to place each into an array **output**[] so I can use a printf() statement later to see the values translated. I chose to implement a for loop that iterates through each of the septets. The rotate\_left3() function uses the get\_n\_bits() function’s return value, the septet, as its only argument, then rotates the bits as described above, and places the result in the empty array. I then move onto the final task of the assignment and decryption process.



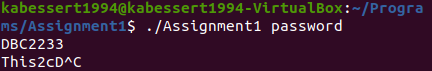
1. *Translate septets to characters;*

Because the septets are stored in an array of characters, all that needs to be done is to use a printf() statement to output them as characters. I use a for loop that automatically outputs the 4 septets one after another and then issues a “return 0;” statement to complete the program. After this is complete I compile the program and run it with the original encrypted test case text “DBC2233” to verify everything is working successfully.

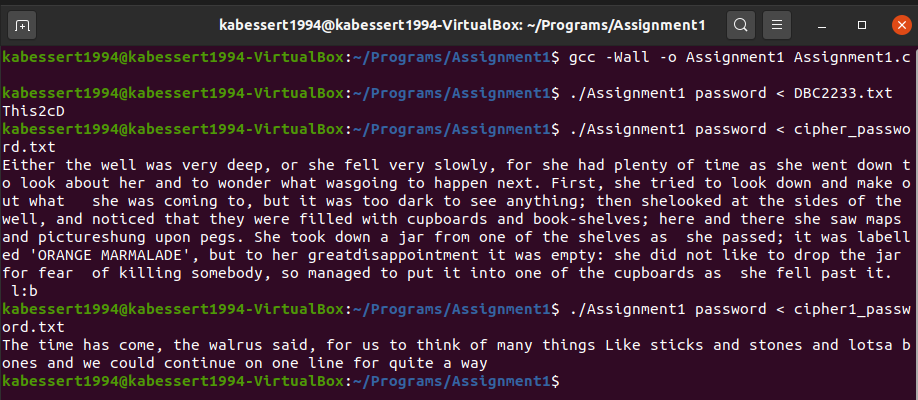




The for loop originally output the septets in the reverse order, and needed to be altered to rectify this issue, but otherwise the program is successfully unencrypting the message, but not as intended. There are additional characters printed out to the terminal after the translation that indicate the while loop in main is being executed an additional time, which indicates that there is more data present in the text file to be translated. I attempted to execute the program as intended, with no input file, and received the same results.



Knowing this anomaly, and with successful decryption of the test case, I attempted to decrypt the encrypted message that was provided within “cipher\_password.txt” with the password “password”, which decrypted successfully as seen below. Without conducting any cracking, I simply attempted to decrypt the additional encrypted message provided within “cipher1\_password.txt” using the same password, which seemingly succeeded as well. The additional characters output by the additional execution of the while loop are still present, and will be addressed in follow on testing. The program is successful in decryption and the assignment task is complete.



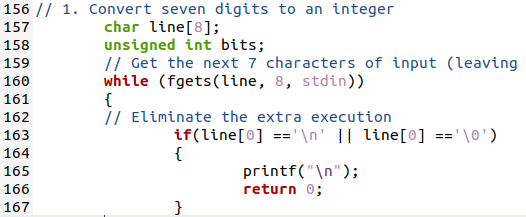
1. **Testing**

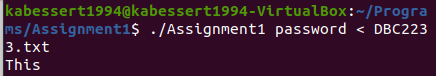
All throughout the coding process many “printf();” statements were used to output variables in various locations including before, in, and after loops and before, in, and after function calls. These statements were utilized during the writing of code to troubleshoot incorrect loop/function logic, incorrect usage of pointers or variables, and unexpected values. This method of testing is what fundamentally shaped the design of the final code. Their utilization can be seen in screenshots of the “**2. Process**”portion of this report. These statements have been removed for the final submitted version of the code, but have been retained in the working copy.

The testing conducted during the writing and troubleshooting of non-working code has been recorded in the “**2. Process**”portion of this report, with provided screenshots. Iterations of the functions and code that were ultimately removed or unused will be detailed here. The first revision of the program, which was essentially a mix of pseudo-code and code to debug, did not utilize pointers or global variables. Because of this, I was utilizing 2 altered versions of the get\_keys() function that returned a value for each of the 2 keys declared in main. These functions were effective but proved to be difficult to work with as the assignment progressed, and were scrapped to attempt to implement the function get\_keys() as provided by the assignment page. During the period of time that I was utilizing 2 functions that returned values for the individual keys, I had altered each of the provided functions that utilized pointers. Again, this method was scrapped and I returned to utilize the provided declarations as intended.

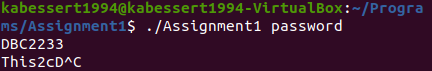
The program was non-functioning, but written, at the beginning of writing this report, returning only gargled characters. However, during the slow and methodical method taken to record the process, multiple major errors were discovered. One of which was in the get\_n\_bits() function, which at the time successfully stored the desired value from **bits** into **temp** with a bitwise & **mask**, but returned that value without shifting it to the right over to the least significant bits. That means that for iterations 2-4 of the while loop in main, the calling function rotate\_left3() was working with a septet of all 0s at index 0 within the 32 bit returned value. This error was caught and rectified during the writing of this report.

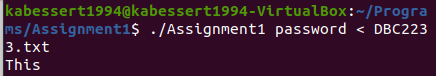
The issue still exists of the additional execution of the while loop in main(), which produces 4 characters at the end of decryption because of an additional whitespace character being read into stdin and being decrypted. This anomaly was resolved with an if statement nested immediately inside the while loop in main() that checks if the first character of the **line**[] array is a whitespace character.



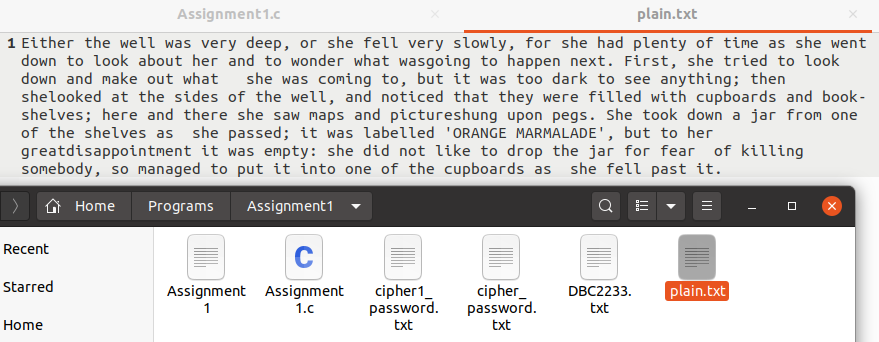
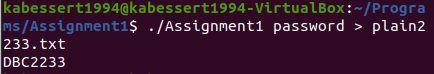


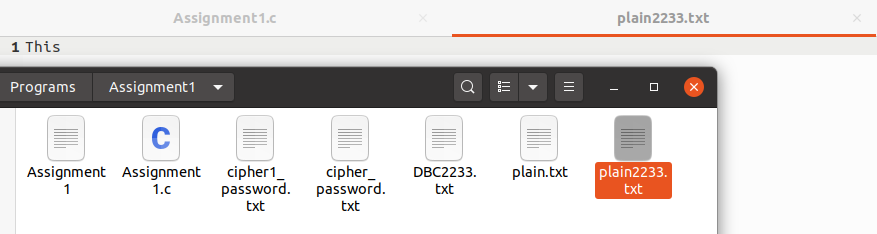
As part of the assignment the program must be capable of receiving input from the command line or from file. This has been demonstrated satisfactorily by the methods utilized in testing at this point in the report, and has been recorded in the following screenshots.



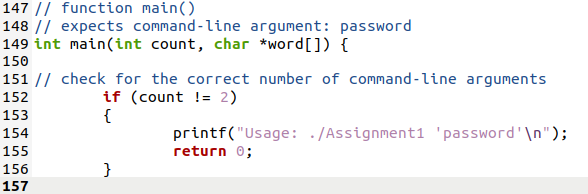


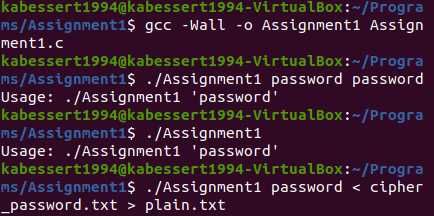
Another requirement of operation of the program is the ability to redirect output to a file, which has not been demonstrated. Below are examples of both taking input from a file to be output to a different file and taking input from the command line to be output to a file. All 4 methods of operation are successfully implemented as intended.



In regards to input, no checks currently are being made for the number of arguments accepted by the program. I implemented a simple if statement at the beginning of the execution of main() to correct this deficiency. The code was again compiled and tested successfully, still allowing for input and output to be redirected.





Finally, no test for the size of the password being passed in with the program call is being conducted in main() prior to the decryption process. With the current definition of get\_keys(), only passwords of 8 characters are accepted or utilized and any additional characters will be discarded. This is limiting, but accomplishes the intent of the assignment. No further changes to the program were made to rectify this issue.

1. **Results**

The results of this assignment are a success in that the test case and both of the provided encrypted messages are successfully decrypted by the program. In addition to successful decryption of the provided messages, all forms of required input and output operations outline in the “Running the Decryption Program” portion of the assignment page are available, accomplished and tested. All of the function declarations provided by the “Hints” section of the assignment page on canvas were utilized, as written, with one exception.

The decode\_28bits() function was declared, but not defined throughout the writing of the program, and was ultimately removed from the final version submitted with this report. With additional time the function could be successfully implemented by placing the bulk of function calls from main into the unused function to perform the decryption routine. This would require severe overhaul of the current program, specifically with output of the decrypted plaintext, and would require additional testing to ensure proper implementation. With a limit of time I have decided to forgo this process and preserve the program as is in its functioning state.

1. **Conclusions**

Based on the results and intent of this assignment I conclude that I have successfully accomplished the assignment to the best of my ability. Having started this assignment with a loose understanding of the C language and the belief that I would be unable to accomplish the assignment in any significant manner, I have gone above and beyond my previously perceived ability. I have succeeded in working through the difficult task of comprehending the assignment by starting, in a methodical manner, with breaking down each task to be accomplished into the smallest steps of what is being asked of me as a programmer. The large scope of the assignment was intimidating to me at first, but through breaking down each step into the smallest sub-steps of pseudo-code I was able to understand what was being asked of me.

Another large takeaway from this process was the utilization of breaks, where most of my success in troubleshooting a large problem was found by stopping work for the day and deciding to pick it up tomorrow. During these breaks I would process the situation in the back of my mind, unintentionally, and usually find something I’d like to try the next time I sat down to work again. This allowed me to tackle the large issues I ran into with a new aspect and a refreshed mind, and lead to the successful completion of the assignment.

My final large take-away from this assignment was the reliance on other students for simple checking of comprehension and logic. Many smaller issues were resolved simply by verbalizing what trouble I was experiencing to other students working on the assignment as well. This was especially true when I was attempting to decrypt ‘password’ as the test case instead of ‘DBC2233’ from a simple misunderstanding. My actions were corrected by a fellow student and I could resume testing of the program. It seems that as the quarter progressed more and more students are discovering this as well, which increased the availability of another perspective.

This assignment has successfully ‘knocked the rust off’ my programing ability from the break over the summer, and has introduced to me the level of attention that must be provided to large assignments like this. I feel prepared to tackle the oncoming assignments and labs for the remainder of the quarter after the completion of this assignment, as well as a reinforced confidence in my overall ability as a programmer.

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.

[C tutorial, Tutorialspoint](https://www.tutorialspoint.com/cprogramming/index.htm)

Keegan Giles assisted with troubleshooting logic of functions and the order of shuffle\_Nibbles() on 10 Oct 2020

Jared Larson assisted with troubleshooting of the additional execution of the while loop in main on 13 Oct 2020