There were many obstacles that I overcame in this project. The first and foremost obstacle was ensuring proper control flow when building the *match* C-string. Originally, I tried to use 5 nested for loops to develop and test the *match* C-string in the function *decrypt* and then transitioned to 2 for loops and two while loops. It took me a while to realize that no amount of *break* statements would solve the mysterious increment errors I was seeing. So, after reading a bit online about *break* and *goto* statements I saw a lot of people recommend the use of helper functions as an alternative. it felt great to delete all the ugly, nested loops and once I implemented the function *build\_match*, my code cleared up considerably. I still ran into a few errors, but they were significantly easier to track down and fix. The second biggest obstacle I had was that my *build\_match* function was only reading the first line of a given *ciphertext*. I was confused as to why this was happening since my *while* loop in *decrypt* seemed to be operating normally. I finally tracked down the error to an incrementing variable not resetting after the innermost while loop in decrypt exited. When I fixed this, my program immediately righted itself. I faced many other problems, mostly concerning boolean logic and developing a proper match, but I felt that those were easier for me to solve since the issues themselves were more obvious. I learned that readability in code is so important for debugging purposes and comments not only help other people see what my code is doing but also help me when I’m trying to track down an error.

Pseudocode for decrypt function:

1. Pass given ciphertext and crib strings into the function decrypt.
2. If the length of the crib is zero or over 80 characters, return false.
3. Create a C-string called key of the max size of crib and, repeatedly:
   1. copy the corresponding character in crib to key if the character in crib is alpha or non-repeating non-alpha and increment the position of key
   2. When finished, terminate the C-string with the null byte.
4. Create a 2D character array to hold all of ciphertext (“rows” = lines, columns = characters) and initialize to null. Then, repeatedly:
   1. copy each character of ciphertext into a temp variable. Increment the position of the temp variable.
   2. If the character is a newline or the last char in ciphertext, transfer hold into the array and increment the variable designating rows.
5. Repeatedly, traverse through the rows of the array.
   1. Repeatedly, traverse through the characters of the current row in array.
      1. \*\*Call build\_match function that returns a boolean value depending on if text starting with the current character in the row of the array matches the crib, per specifications.
      2. If the bool value is false, move to the next character. Repeat until a match is found or we traverse the entire row.
         1. If a match is found, exit both while loops. If we traverse the entire row, increment to the next row of the array.
6. If a match is still not found after traversing the entirety of the ciphertext, return false.
7. If a match is found, create a C-string with a size equal to the max size of ciphertext. Then, repeatedly:
   1. Copy characters of ciphertext into the C-string if they do not match any of the characters in the match C-string. Copy non-alpha characters as they are into the C-string.
      1. Avoid printing duplicates by using a break statement.
      2. Use an incrementor to make sure that the character does not match anything in match.
   2. If a letter in ciphertext matches a letter in match (not case sensitive), copy an uppercase version of the corresponding element to match in key.
      1. Avoid printing duplicates by using a break statement.
   3. Make sure to terminate the C-string with a null byte.
8. Print the newly made C-string and return true.

Pseudocode for build\_match Function:

1. Assign the C-string match to null.
2. If the position of the row in the array is zero, the first element of match gets the first element of the row.
3. Repeatedly,
   1. Traverse each line of the array character by character.
      1. If the length of the match is less than the length of the key and the character is alpha or non-repeating non-alpha, match gets the current character in the row. Increment the position of match and the array.
      2. If the character is a non-alpha repeating, do not copy into match and move to the next character in the row.
      3. If the length of the match is less than the length of the key and its the last iteration of the loop, return false.
      4. If the lengths of match and key are equal, break.
   2. If lengths of match and key are not equal, return false.
   3. Otherwise, repeatedly:
      1. Navigate through the characters of match.
         1. If the character is non-alpha, the corresponding character must also be non-alpha.
            1. If this condition fails, return false.
         2. If the character is alpha, repeatedly:
            1. Navigate through the characters of match to check if the current character (outer for loop) in match is equal to any characters in match (not case sensitive). If the characters equal each other, their corresponding characters in key must be the same.
            2. Also check if the current character (outer for loop) in key is equal to any characters in key (not case sensitive). If the characters equal each other, their corresponding characters in match must be the same.
            3. If these corresponding characters are different, return false.
         3. If the crib was not just non-alpha characters or the next character in the array is a character (thus meaning the match was just a segment of a word), return false.
         4. Otherwise, return true.

Test Data:

1. runtest("Hirdd ejsy zu drvtry od.\nO'z fodvtrry.\n", "my secret"); *//double word match and double word crib, tests if non-alpha chars are treated correctly. returns true.*
2. runtest("Hirdd ejsy zu drvtry od.\nO'z fodvtrry.\n", "shadow"); *//tests if analysis of potential match given correct length is logical; also tests that a match cannot be a segment of a word.*
3. runtest("The abcde\nbrown fox", "quick"); *//should match either abcde or brown.*
4. runtest("Zysqjs zbguncyqzo jdsbyo eybmnu bg Wqzsvbbf.\nUnysqx eybmgxrsuu ymtbyu kcq Jicjjsy.\nNbuj sajysts rcvsyqr qgx sajysts zbgusykqjcks nbucjcbgu bg xcuzmuucbg wbymtu.\nZU 31 cu zdqrrsgecge!", "conspiracy theory"); *//handling of large strings, multiple lines.*
5. runtest("Kpio't dmpbl-boe-ebhhfs opwfm", "s cloak and"); *//tests if match can span many “words” that are really separated by non-alpha chars.*
6. runtest("DiebjiggK, zyxZYXzyx--Abca abCa bdefg## $$hidbijk6437 wvuWVUwvu\n\n8 9\n", "hush-hush until January 20, 2021"); *//tests if crib length was adjusted and that program prints out non-alpha characters as seen in ciphertext.*
7. runtest("bwra wmwt\nqeirtk spst\n", "alan turing"); *//match cannot span multiple lines.*
8. runtest("fox quiz", "8888"); *// tests that a match with all non-alpha chars is invalid. This means that a crib with all non-alpha chars will automatically be invalid because it has no valid match.*

My program handles all of these test cases correctly.