

## The Vigenere Cipher

### I. Description of Project

- a. Project performs the *Vigenere Cipher* which is a method of encrypting alphabetic text based on substitution; the substitution comes in the form of using a *key*. Typically, one produces Vigenere encryption through looking up a table, however, this program implements math in which each character is mapped to a number like so:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2
										0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5

### II. Steps of Code Division

- a. The following will describe the steps of how the Vigenere Cipher is performed in this program and will specifically say it is written in Assembly or C.
- b. The Steps:
- note: The C language is mostly used for printf's and scanf's. Assembly language will do the rest.
  - The Menu – *in C*
    - Program will ask user to input a message or be encrypted (or decrypted) and the key
    -
  - ToUpper – *in Assembly*
    - This will change all the characters to uppercase to make the Vigenere Cipher easier
  - StrLength – *in Assembly*
    - This function checks for the string length of the message or key, which is vital to program since they must be equal in order to perform the cipher.
  - ManipKey (manipulate key) – *in Assembly*
    - This function will manipulate the key to be the same string length as the message if necessary.
  - VigenereCipherEncrypt – *in Assembly*
    - This function will take in the message & key to map them to a number and perform the substitution.
  - VigenereCipher Decrypt – *in Assembly*
    - Similar idea to the function above but a little bit different in the algorithm.

### III. Pseudocode

### IV. Source Code \*\*\*NOTE CODE IS STILL IN PROGRESS.

#### a. driver.c

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <stdint.h>
```

```
void toUpper(char* msg, int strlength);
```

```
void manipKey(char *key, int keylength, int strlength);
```

```
void vigenereCipherEncrypt(char *msg, char *key, char *newMsg);
```

```
void vigenereCipherDecrypt(char *msg, char *key, char *newMsg);
```

```

int strLength(char *msg);

void options(int x);

int main(void){
    int x = 0;

    printf("What would you like to do today?\n\n");

    printf("Please type in the corresponding number to the option, then hit SPACEBAR or ENTER.\n");

    printf("1.ENCRYPT OR DECRYPT A MESSAGE\n");

    printf("2.QUIT\n");

    options(x);

    printf("Goodbye!\n");

    }//end main-----

    void options(int x){

        scanf("%d", &x);

        if(x == 1){

            char *a;

            a = malloc(sizeof(char) * 100);

            printf("Please type a message to be encrypted OR decrypted...\n");

            printf("Then press SPACEBAR or ENTER for program to read.\n\n");

            scanf("%s",a);

            printf("Your word is: ");

            printf("%s\n\n",a);

            a = realloc(a, sizeof(a));

            printf("Please type in a 'key' you would like to use OR used.\n");

            char *key;

            //key = maalloc(sizeof(char)*sizeof(a));

            scanf("%s",key);

            printf("Your key is: ");

            printf("%s\n\n",key);

            int strlength = strLength(a);

            int keylength = strLength(key);

            printf("Converting your message & key to uppercase...\n\n");

            toUpper(key,keylength);

            toUpper(a,strlength);

```

```

printf("Now your key has been converted to all uppercase.... %s\n", key);
toUpper(a, strlen(a));
printf("Now your message has been converted to all uppercase....%s\n\n", a);
printf("Now checking to see if your key is the same length of your message...\n");
manipKey(key, keylength, strlen(a));
printf("Now converting...your key is now: %s\n\n", key);
printf("Checking if string lengths match...\n");
strlen = strlen(a);
keylength = strlen(key);
printf("The string length of your key is: %d\n", keylength);
printf("The string length of your message is: %d\n\n", strlen);
char *encryptedMsg = malloc(sizeof(char) * sizeof(a));
printf("encrypted msg %s\n", encryptedMsg);
    printf("Now perform Vigenere Cipher...\n");
    vigenereCipherEncrypt(a, key, encryptedMsg);
    printf("Your message is %s\n", encryptedMsg);
/*
int choice = 0;
printf("Last equation: Are you encrypting or decrypting?\n");
printf("Please input the number to the corresponding option.\n\n");
printf("1. ENCRYPTION\n");
printf("2. DECRYPTION\n");
scanf("%d\n", &choice);
printf("%d\n", choice);
if(scanf("%d",&choice)){
printf("%d\n", choice);
    if(choice == 1){
        printf("Now perform Vigenere Cipher...\n");
        vigenereCipherEncrypt(a, key, encryptedMsg);
        printf("Your message is %s", encryptedMsg);
    }
    else if(choice == 2){
        printf("Now perform Vigenere Cipher...\n");

```

```

    vigenereCipherDecrypt(a, key, encryptedMsg);
    printf("Your message is %s", encryptedMsg);
}
else
    printf("Wrong integer input. Program will terminate.\n");
}
else
    printf("Wrong input. Program will now terminate.\n");
*/
} //END ENCRYPT/DECRYPT
else if( x == 2 ){
}
else{
    printf("Wrong input. Program will terminate.\n");
} //end INVALID
} //end options-----

```

#### **b. functions.s**

```

.global strLength
.data
.text
strLength:
    mov  r12,r13
    sub  sp,#32
    push {lr}
    //we're passing in the array of key or the array of message
    //first we need to find the string length.
    //let r5 be our counter for our string length
    mov r5, #0 //initialize r5 = 0
    mov r6,r0 //we don't want to mess with the address of our arg, so, we'll just move it into a diff
    register.
while:
    ldrb r1,[r6] //we're loading a byte of r6, so essentially r1 = key[0]
    cmp r1, #0 // is r1 = 0? then we've reached the end of the string. you're done!
    beq done

```

```

        //else if it isn't null, then length++ ( r5 )
add r5, r5, #1

        //then we need to load the next byte.
add r6, r6, #1

        //cycle repeats till we reach null.
b while
done:
    mov r0, r5 //return the length.
pop {r7}
mov lr, r7
mov sp, r12
mov pc, lr
//*****

.global toUpper //returns char* toUpper(msg[MAX])
.data
.text
toUpper:
    mov r12, r13
    sub sp, #32
    push {lr}
    //r0 has address of key[MAX]
    //r1 has strlen
    //let r2 be your walking stick thru array.
    mov r2, #0
loop:
    cmp r1, r2 //cmp (strlen-1) with the inc, are they the same? then we reached the end, nothing more
to inc.
    beq exit
    ldrb r3, [r0, r2] //so right now we loaded into r3, max[0]

    cmp r3, #0 //another condition to check that we reached the end.
        //first test: if null, we reached the end, nothing to upper. we're done.
    beq exit

```

```
cmp r3, #'a' //second test:is r1 less than 97 'a'? then we're already uppercase.
```

```
blt exit
```

```
    //else if its not, then we're lower case. so we must sub 0x20.
```

```
sub r3, r3, #0x20
```

```
strb r3,[r0,r2] //this stores the value of r1 into address of r0, inc by r2(our walking stick)
```

```
add r2,r2,#1
```

```
b loop
```

```
    //then we need to load the next byte.
```

```
    //cycle repeats till we reach null.
```

```
exit:
```

```
pop {r7}
```

```
mov lr, r7
```

```
mov sp,r12
```

```
mov pc, lr
```

```
//*****
```

```
.global manipKey //remember manipKey(char key[MAX], int strlen, int keylength)
```

```
.data
```

```
.text
```

```
manipKey:
```

```
mov r12,r13
```

```
sub sp,#32
```

```
push {lr}
```

```
//r0 holds address of key
```

```
//r1 holds length of the key
```

```
//r2 holds the length of the message
```

```
//r4 is our walking stick thru the array
```

```
mov r4, #0
```

```
loopdeloop:
```

```
cmp r1,r2 //are they equal? then you dont need to do anywork:)
```

```
beq overit
```

```
    //else if keylength > messagelength, we got to nullify some ofit
```

```

bgt nullify

    //else, keylength < messagelength, we need to make the length the same
    //lets use r3 to hold the address of the first element array
ldrb r3,[r0,r4]

    //store the value of r3 into the array's next element thats open!
strb r3,[r0,r1]

    //now we inc the keylength since thats what we did
add r1,r1,#1
add r4,r4,#1 //now we inc our walking stick to move on to the next element
b loopdeloop

    //case where key length is greater so we need to throw away some chars
nullify:

    mov r4, r1 //let r4 be the walking stick, grab the length of key
    sub r4,r4,#1 //sub r4 - 1 bc in array we start at 0.

nullifyloop:
    cmp r1, r2 //is strlen = keylength? if it is were done.
    beq overit

    //else lets throw away some chars.
ldrb r3,[r0,r4] //load into r3 key[r4]
    mov r3, #0 //nullify that place until strlen = keylength
    strb r3,[r0,r4]
    sub r4,r4, #1 //dec to move in the array
    sub r1,r1,#1 //dec the keylength to check in the end if strlen = keylength
    b nullifyloop

overit:
pop {r7}
mov lr, r7
mov sp,r12
mov pc, lr

```

```

//*****

.global vigenereCipherEncrypt
.data
.text

vigenereCipherEncrypt://recall args: vigenereCipher(char msg[MAX], char key[MAX], char
newMsg[MAX]);

mov  r12,r13
sub  sp,#32
push {lr}


//r0 = holds address of msg[0]
//r1 = holds address of key[0]
//r2 = holds address of our new array newMsg [0] (encrypted message / or decrypted)
//use r3 as a place holder to manipulate the content of msg[]
//use r4 as a place holder to manipulate content of key[]
//r5 is walking stick (i)
//r6 is place holder for the result of the vigenere cipher
mov r5, #0 //i = 0, walking stick
repeat:
    ldrb r3,[r0,r5] //r3 gets msg[i]
    cmp r3, #0 //did you get null? we reached the end of the string:) we're done
    beq finally
    sub r3,r3, #65 // else, lets map the char to #
    ldrb r4,[r1,r5] //lets load key[i] into r4
    sub r4,r4, #65 //lets map their char to # too
    add r6,r3,r4 //lets add the mapped numbers to get our converted char
    add r6,r6,#65 //map them to ascii values
    cmp r6, #90 //check if we went out of bounds!
    bgt modulo //if it is out of bounds, we need to make it work
    //else we'll keep going
    strb r6,[r2,r5] //store value of r6 into newmsg[i]
    add r5,r5,#1 //inc the walking stick
    b repeat
    //case for when it got out of bounds

```



```

modulo:
    sub r6,r6,#26 //we sub 26 (26 chars from alphabet) to make it out of bounds
        //and get mapped to its ascii value
    strb r6,[r2,r5] //now take result and put it in newmsg[i]
    add r5,r5,#1 //inc the walking stick
    b repeat //branch to repeat.

finally:
pop {r7}
mov lr, r7
mov sp,r12
mov pc, lr

//*****
***

.global vigenereCipherDecrypt

.data

.text

vigenereCipherDecrypt:
    mov r12,r13
    sub sp,#32
    push {lr}
    mov r5, #0 //i = 0, walking stick

    repeat1:
        ldrb r3,[r0,r5] //r3 gets msg[i]
        cmp r3, #0 //did you get null? we reached the end of the string:) we're done
        beq finally1
        sub r3,r3, #65 // else, lets map the char to #
        ldrb r4,[r1,r5] //lets load key[i] into r4
        sub r4,r4, #65 //lets map their char to # too
        cmp r3,r4
        blt map // if r3 < r4, then subtraction will result in -, which cannot happen.
            //therefore go to map label.
            //PERFORMING DECRYPTION
            //else if r3> r4, we can perform subtraction
        sub r6,r3,r4 //lets add the mapped numbers to get our converted char

```

```

add r6,r6,#65 //map them to ascii values
cmp r6, #90 //check if we went out of bounds!
bgt modulo1 //if it is out of bounds, we need to make it work
//else we'll keep going
strb r6,[r2,r5] //store value of r6 into newmsg[i]
add r5,r5,#1 //inc the walking stick
b repeat1
    //case for when it got out of bounds
modulo1:
    sub r6,r6,#26 //we sub 26 (26 chars from alphabet) to make it out of bounds
        //and get mapped to its ascii value
    strb r6,[r2,r5] //now take result and put it in newmsg[i]
    add r5,r5,#1 //inc the walking stick
    b repeat1 //branch to repeat.
map://case where r3 < r4
sub r6,r4,r3 //r6 = r4-r3
add r6,r6, #10
add r6,r6,#65 //map them to ascii values
cmp r6, #90 //check if we went out of bounds!
bgt modulo1 //if it is out of bounds, we need to make it work
//else we'll keep going
strb r6,[r2,r5] //store value of r6 into newmsg[i]
add r5,r5,#1 //inc the walking stick
b repeat1
finally1:
pop {r7}
mov lr, r7
mov sp,r12
mov pc, lr

```

## **V. Tools Used**

- a.** Segger Embedded Studio

## **VI. User Instructions**

- a.** First, user will be asked on what they would like to do. They have the option to choose from performing the cipher or quitting. Otherwise, input will be taken as invalid and program will terminate.

```
Debug Terminal

Please type in the corresponding number to the option, then hit SPACEBAR or ENT
1.ENCRYPT OR DECRYPT A MESSAGE
2.QUIT
1
Please type a message to be encrypted OR decrypted...
Then press SPACEBAR or ENTER for program to read.
```

- i.
- b. All the user has to do is input their message and then input their key consecutively. Then the program will perform the Vigenere Cipher.

```
Debug Terminal

Please type in the corresponding number to the option, then hit SPACEBAR or ENT
1.ENCRYPT OR DECRYPT A MESSAGE
2.QUIT
1 ← user inputs "1" for Vigenere Cipher.
Please type a message to be encrypted OR decrypted...
Then press SPACEBAR or ENTER for program to read.
```

i.

```
Debug Terminal

2.QUIT
1
Please type a message to be encrypted OR decrypted...
Then press SPACEBAR or ENTER for program to read.

message ← user inputs "message" as their message.
Your word is: message
```

ii.

```
Debug Terminal

Please type in a 'key' you would like to use OR used.
key ← user inputs "key" as their key.
Your key is: key

Converting your message & key to uppercase...
Now your key has been converted to all uppercase.... KEY
Now your message has been converted to all uppercase....MESS

Now checking to see if your key is the same length of your message...
```

iii.

Program then proceeds to perform Vigenere cipher!

## VII. Test Results \*\*\* I STILL NEED TO FIX MY BUGS.

### a. Bugs within Output

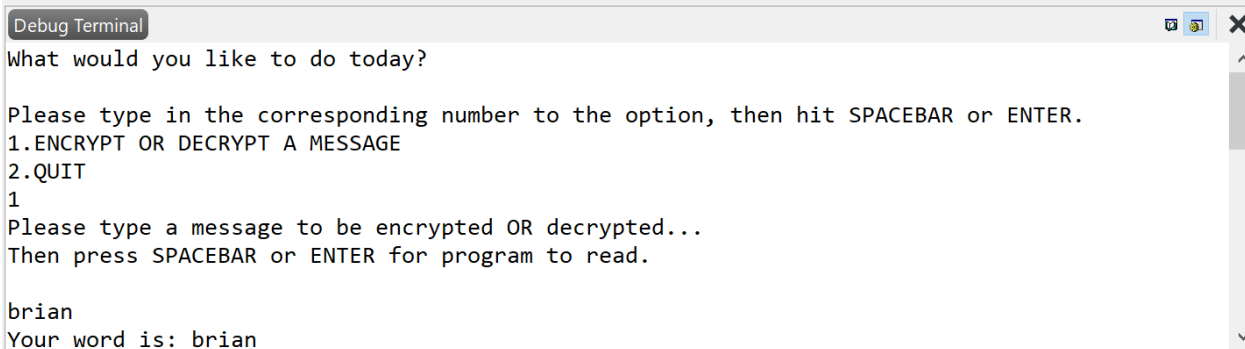
- i. A typical bug within the output of the cipher function. My function seemed to work all the way until the very end, where in certain situations (such as the varying factor of how long is the string length), the output will be either correct or relatively correct. I saw a common mistake that the output would print more than it need to, and I found out it was because of the way memory was allocated. Within my functions, I used the conditional factor that if we accessed in memory a NULL, then we've reached the end of the string so we no longer need to proceed with that function. That isn't always the case though, as seen in debugging, sometimes in memory, the content of that string was right next to a character which would cause my program to continue more than it should have. For example, I saw it next to something I printed before in C. To make my program output something relatively right, I removed a lot of print statements to make the program concise which was at the cost of pretty error checking. I realized though that I was wasting a lot of memory. I have to fix this soon by taking TA's suggestion of allocating memory and reallocating.

### b. Project Limitations

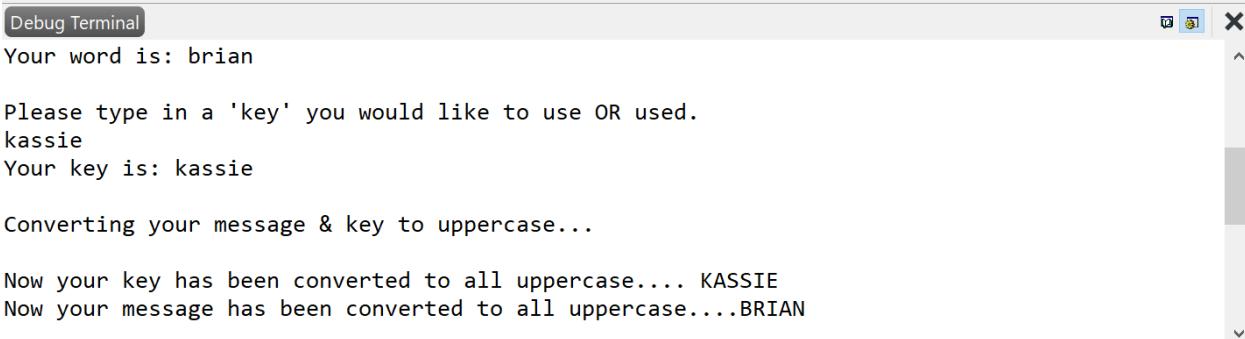
- i. A major limitation in this program is the fact that the user is unable to input messages that included spaces. This was not my intention though, as in the beginning of this journey, I was playing around in C in which scanning operation would work best for me (scanf vs fgets), however that was not the problem. It seems that using Segger Embedded Studio, hitting 'space' is equivalent to 'enter,' which indicates to the compiler that the user is done inputting. Therefore, in the user directions, I specifically printed "**Then press SPACEBAR or ENTER for program to read,**" so that the user is informed of its limitations and will not get confused.

## VIII. Demonstration / Example Outputs

a.



b.



Cortex-M3 on Simulator

```
Debug Terminal

Now checking to see if your key is the same length of your message...
Now converting...your key is now: KASSI

Checking if string lengths match...
The string length of your key is: 5
The string length of your message is: 5

Now perform Vigenere Cipher...
Your message is LRASVÍÍÍÍÍÍÍÍ
Goodbye!
```

c. *Side note: Program should have returned “LRASV.”*

**IX. Changes to make in the Future**

- a. Referring to the major limitation of not including spaces, I feel that next time I would choose another IDE that didn't cause that limitation.
- b. I've also learned that it's important to make a program efficient, and in the beginning of the journey I was so focused on making the program pretty vs. efficient. I wasting a lot of memory, declaring unnecessary variables.

**X. Conclusion**

- a. I still have bugs to fix so that the program is perfect, but the point is to essentially show that I know how to write assembly language. I feel that based on what I've learned, the challenges I've faced, the effort I put in, and how much I've reached out for help, I hope it shows that I know how to code in assembly.

**XI. Hours**

- a. Typically spent 3-4 hours each week on the project. So,  $4 * (5 \text{ reports } \underline{\text{so far}}) = \text{at most } 20 \text{ hours.}$