Assignment 1
CS 135 Review - Cryptography

Benjamin Zofcin
CS 202 - Fall 2023

1 Introduction

CS 135 (and other first year courses) establish the foundations for your critical and logical thought processes that you will apply to every problem encountered as engineers. Secondary to that, you also learned some C/C++ syntax to articulate your solutions to the aforementioned problems. The purpose of this assignment is to serve as the refresher and application of all things learned in CS 135. This assignment should be thought of as a CS 135 capstone project.

2 Background

Encryption is, and always has been, a fundamental aspect of life; without it, everyone could know everything about everyone else. Historically (over 2000 years ago now), the Romans used encryption algorithms in times of their warring and conquering periods across modern day Europe. Julius Caesar used similar algorithms in his personal correspondents to obfuscate intended messages. What is now almost a century ago, the Germans used more advanced encryption methods via the Enigma machine to transmit sensitive battle plans during World War II.
Needless to say, in the applications of modern day technology, encryption has become evermore present. The only way you could have accessed this pdf file is by logging in to the Canvas system with a user name and password. The only way you could have accessed the Canvas website is

through an encrypted TLS handshake protocol between your computer and the server where the file is hosted. None of that would have been possible if I didn't deliberately upload the files to the server in the first place (which also includes everything stated earlier). For those of us living in the information era, it is critical that you know these systems are in place to protect our information, personal assets, and identity. However, for those hoping to earn a degree in or related to computer science, engineering, or mathematics; eventually, you will need to know of more than just their existence.

3 Task

This is a multi-part assignment where you will implement two different encryption algorithms:

- Stream Ciphers: Modified Caesar Cipher (part 1)
- Textbook RSA (part 2)

Naturally, I cannot assume that you have any knowledge or background in encryption algorithms or higher level mathematics to implement either of the prior 2 algorithms. But, fret not. This writeup will walk you through everything you need to know about the mathematics, methods, and behaviors needed for the Modified Caesar Cipher and part 2 will have much of the maths already implemented. It will then be up to you to figure out how to take that information and convert it into an equivalent C/C++ style program. It should be noted, the only programming techniques needed for the program's solution are those taught to you in CS 135 (this is a review assignment after all).

4 Stream Ciphers

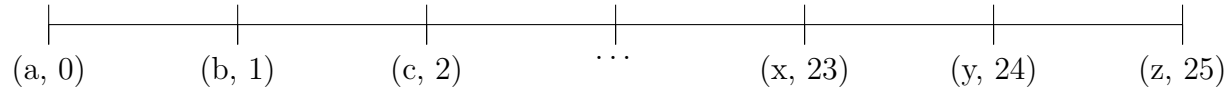
Stream Ciphers, like the ones Julius Caesar and the Germans used, augment the original message one character at a time. This augmentation can come in many forms:

1. Offsets
2. Rotations
3. Repositioning (Mixing)

In this part of the assignment we are only going to focus on offsets and rotations.

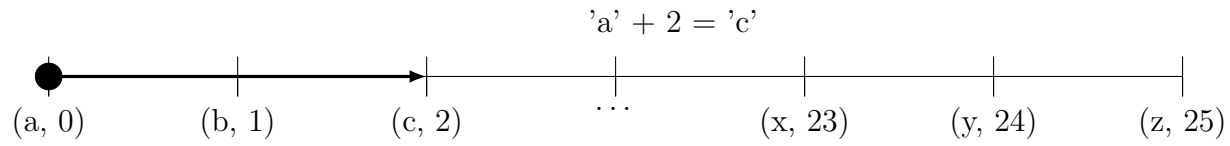
4.1 Offsets

Offsetting is the act of changing the "value" of a letter. In the English language, we have a set of letters {a,b,c, ... , x, y, z}. Assume that each one of these letters can be related to a value on a subset of the integer number line.

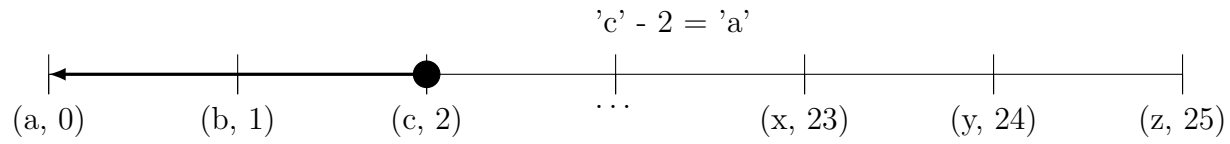


Mathematics tells us that any value with position on the number line can be moved along that line. If we wished to offset a character then we simply need to add that offset value to that character.

E.g.,



Inversely, we can also move a offset a character in the other direction by performing subtraction (negative addition). E.g.,



However, let us consider the following application of the offset function.

'y' + 2 = ?

Now, we've run into an issue. The offset has landed us beyond the boundaries of our domain.

The solution to this issue is quite simple and elegant.

4.1.1 Modular Arithmetic

Modular arithmetic is its own branch of mathematics that you will formally learn (or may already have already learned) about in any number of courses (E.g., Discrete Mathematics, Number Theory, Circuit design, Complex Algebra, etc...). We won't go into great detail on this branch of mathematics, but we will cover 2 things in this section:

1. Basic theory of counting systems
2. How to implement modular arithmetic

Basic theory of counting systems

We have been performing modular arithmetic our whole lives by simply counting on our fingers or reading an analog clock. Fundamentally, when we count numbers on our fingers, we can only count up to 10 and then start over. Therefore, we refer to our number system as "base 10". Each time the value of 10 is reached as we count, we add 1 to the next "place" in our count and reset the original "place" to 0.

$$
\begin{aligned}
& 0+1 \\
&= 1+1 \\
&= 2+1 \\
&= 3+1 \\
&= 4+1 \\
&= 5+1
\end{aligned}
\qquad
\begin{aligned}
&= 6+1 \\
&= 7+1 \\
&= 8+1 \\
&= 9+1 \\
&= 10+0
\end{aligned}
$$

These "places" are referred to as the 1's place, 10's place, 100's place, etc...
This implies that any value in the domain of base 10 integers can be mathematically represented by the following equation.

Figure 1: Series Composition

$$
\begin{aligned}
& \sum_{i=0}^{p} v[i] * 10^i, \qquad (1) \\
& p := \text{number of places} \\
& v := \text{value at place } i
\end{aligned}
$$

E.g.,

Figure 2:

$$
\begin{aligned}
v = \{9,6,7,1\} &= 9 * 10^0 + 6 * 10^1 + 7 * 10^2 + 1 * 10^3 \\
&= 9 * 1 + 6 * 10 + 7 * 100 + 1 * 1000 \\
&= 9 + 60 + 700 + 1000 \\
&= 1769
\end{aligned}
$$

We can apply this same logic to reading an analog clock.

- Seconds, base 60

  For every 60 seconds to pass, 1 minute has passed

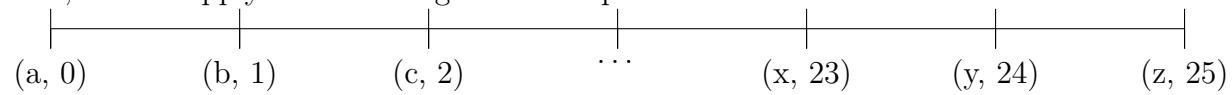
- Minutes, base 60

  For every 60 minutes to pass, 1 hour has passed

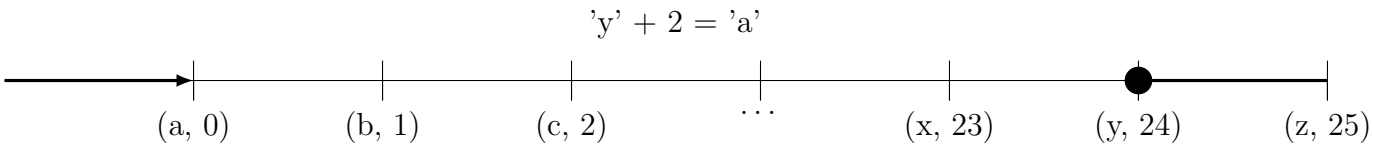
- Hours, base 12

  For every 12 hours to pass, 1 clock rotation has passed

Now, we can apply this same logic to our alphabetized number line.



We can say that our number line is base 26. Therefore, for every offset to exceed the base, the offset cycles back to the beginning (much like a clock)



We can now see that the answer to our original question is 'a'. But, why?

Implementation of Modular Arithmetic

The reason is because the remainder after division by the base is 0. Recall, that output of every division operation with a numerator(N) and denominator(D) yields a quotient(Q) and a remainder(R). In C/C++ dividing 2 integers will only give us the quotient

$$\frac{N}{D} = Q$$

If division gives us the quotient then modulus will give us the remainder.

Mathematically, this is expressed by the following equation.

$$R = N \ (mod \ D)$$
$$or$$
$$R = N \ \% \ D$$

- D is the base of our number system.
- N is the value of the character we are evaluating combined with the offset.
- R is the output character of the function.

Let's take a look at some of the examples we have done up to now.

1. 'a' + 2,

   R = 0 + 2 mod 26 = 2 mod 26 = 2 = 'c'

2. 'c' - 2,

   R = 2 - 2 mod 26 = 0 mod 26 = 0 = 'a'

3. 'y' + 2,

   R = 24 + 2 mod 26 = 26 mod 26 = 0 = 'a'

4. 'a' - 2,

$$R = 0 - 2 \bmod 26 = -2 \bmod 26 = 24 = \text{'y'}$$

4.1.2 Offset Encryption

Now, that we understand how to apply offset encryption to a single character, let's consider what it takes to apply that encryption to a word or a sentence.

Consider the phrase: "The brown cat, sleeping quietly, awoke." and an offset of 5.
What we expect is: "Ymj%twfslj%hfy1%xqjjunsl%vznjyq 1%f—tpj3".
and if we apply the offset of -5 we get the original phrase back.

While we are going to offset each character individually, we have new constraints to consider. Namely:

1. Capitol letters
2. Special characters
3. Spaces

While Caesar had an easier time expanding his base system to accommodate these constraints by adding on the additional 26 uppercase characters and a few special characters for spaces, commas, and other punctuation characters to have a base of 60; we are using computers and to do the same thing as Caesar would be inefficient and complicated. Thus, we turn to the ASCII table. No doubt, if you go to your favorite search engine and look up "ascii table" you will find something like this.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

The ascii table (UTF-8) standard represents a set of the first 128 encoded characters for most computer systems. This domain of character to value encodings contains everything we need to implement the offset encryption. Thankfully, all we have to do is use it in combination with an offset value and the domain value (number of characters, 128)

4.1.3 Code Implementation

That's enough theory. Now, let's get into writing some code to implement all of the things we've learned up to now. Along with this pdf was a tar file that contains all files you will need to complete this part of the assignment. First, we are going to look at the sequential_encryption.cpp file. This file is where you will be writing all of your code.
The first thing you may notice is all of functions that have already been declared for you. At this point you should be very well acquainted with functions, function parameters, and functions returning values from CS 135. These functions are where you will write your code. (see for more guidelines on writing the code). Throughout the remainder of this assignment writeup (in the sections where we talk about code) you will see the names of functions color coded as follows:

- Functions that you must implement are in red.
- Functions that have been implemented for you are in blue.
- Functions that have been partially implemented are in magenta and are yours to finish implementing.
- Provided variables will be highlighted in green

1. **main**:
   main has already been implemented for you and contains exactly 3 function calls. Each one of these function calls serve to test the efficacy of the functions you will write throughout this part of the assignment. You will notice that the second and third functions are already commented out. Don't uncomment them until you have finished this part of the assignment.
2. **offset_encryption**:
   In this function you will define and offset and a message to encrypt / decrypt. The offset_encryption function will then call the offset(string, int) function 2 times. The first time will encrypt the message and the second will decrypt the message. You must display the message to the terminal before and after each successive call to the offset function.
   Example output:

```
[z_xps] stream_ciphers <=> g++ stream_ciphers.cpp
[z_xps] stream_ciphers <=> ./a.out
Text : The orange cat, sleeping quietly, awoke.

Cipher: Ymj%twfslj%hfy1%xqjjunsl%vznjyq~1%f|tpj3

DeCipher: The orange cat, sleeping quietly, awoke.

[z_xps] stream_ciphers <=>
```

3. **offset(string, int)**:
   This function is where you will perform the actual character offset and return the encrypted message. You must iterate over each character individually and perform the proper maths as seen in section "Implementation of Modular Arithmetic".
   To obtain the number of characters in your string parameter use the length function from the string library
   Once a character has been offset it should be added to a the return string. After every character in the string parameter has been offset you must return the offset string back to the offset_encryption function for its print out.

   (note: the numeric value of a character can be printed by adding (int) before the character. (e.g., cout << (int)a; outputs 97))

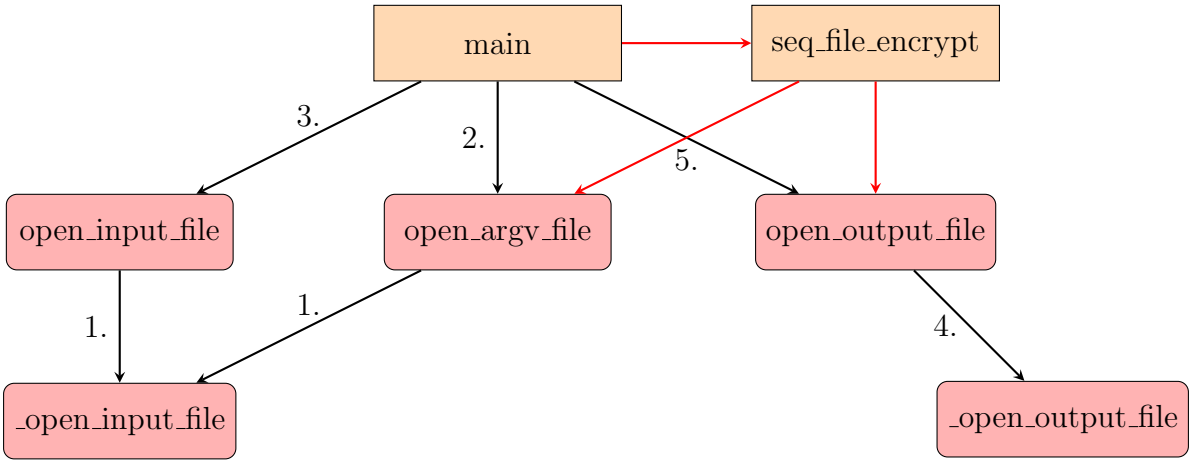
4.2 Alternative Storage

Once you get the provided offset and message from the offset_encryption function working, feel free to play around with other offset values and messages. You may find what you have coded to be full of bugs, and that's okay (it's actually expected and deliberate).

While convenient the UTF-8 ascii standard in combination with the use of cout to allow escape characters like \n, \t, \b, and ascii value 127 (delete) introduces many bugs. Thus, we must look to alternative methods for storing the data we encrypt and decrypt.

4.2.1 Input and Output Files

In CS 135 you were exposed to writing and reading data to and from files. Here you must put what you have learned to the test. There is a collection of 5 functions you will need to write. Each one is relatively small, but the progression of their function calls are linked.
When you have so many functions in a single program, it can be easy to get lost and forget which function does what and why. Therefore, you should get in the habit of making flow charts for small scale and linear programs.
The nodes are labeled with the function name, the black edges are labeled in accordance with the entries in the numbered list below, and the red edges are from the handout code.



1. _open_input_file(string)
   This is a helper function to any other function that requests a file to be opened. Given the name of a file this function will attempt to open said file an store it in an ifstream varaible. If the open function failed, then the user must log that and exit the program with a status of -1; otherwise return the ifstream varaible.
   Assume the following main function:

```
int main(int argc, char const** argv) {
  _open_input_file("incorrect_file_name.txt");
  return 0;
}
```

   The behavior of the program is as follows:

```
[z_xps] stream_ciphers <=> g++ stream_ciphers.cpp
[z_xps] stream_ciphers <=> ./a.out

ERROR: Could not open file : incorrect_file_name.txt
[z_xps] stream_ciphers <=>
```

2. open_argv_file(int argc, const char** argv)
   This function will attempt to open a file, whose name is provided before the execution of the program, and should behave as follows. Assume the following main function:

```
int main(int argc, char const** argv) {
  _open_input_file("incorrect_file_name.txt");
  return 0;
}
```

   The behavior of the program is as follows:

```
[z_xps] stream_ciphers <=> g++ stream_ciphers.cpp
[z_xps] stream_ciphers <=> ./a.out quotes.txt quotes.txt

ERROR: Incorrect usage!
./a.out <filename>
[z_xps] stream_ciphers <=> ./a.out

ERROR: Incorrect usage!
./a.out <filename>
[z_xps] stream_ciphers <=> ./a.out incorrect_file_name.txt

ERROR: Could not open file : incorrect_file_name.txt
[z_xps] stream_ciphers <=> ./a.out quotes.txt
[z_xps] stream_ciphers <=>
```

   Note: The error message generated on the second run of the program comes from _open_input_file() and not this one!

3. open_input_file()
   This function will serve as a user prompting function that requests a file name from the user at runtime. Once the user has entered a file name you must call the _open_input_file function and provide it with the aforementioned user input and return the _open_input_file return value. Assume the following main function:

```
int main(int argc, char const** argv) {
  open_input_file();
  return 0;
}
```

   The behavior of the program is as follows:

```
[z_xps] stream_ciphers <=> g++ stream_ciphers.cpp
[z_xps] stream_ciphers <=> ./a.out
Please enter the name of your input file: incorrect_file_name.txt

ERROR: Could not open file : incorrect_file_name.txt
[z_xps] stream_ciphers <=> ./a.out
Please enter the name of your input file: quotes.txt
[z_xps] stream_ciphers <=>
```

   Note: The error message generated on the first run of the function comes from the _open_input_file function.

4. _open_output_file(string)
   This is a helper function that behaves identically to the _open_input_file(string) function. Where the only difference is storing the opened file in an ofstream variable.

   Note: This function call should not generate any error messages. If it does then something larger than this function is the issue and you should seek guidance from a TA or an instructor on how to address it.

5. open_output_file()
   This function will behave identically to the open_output_file function. Where the only difference is a modification to the prompt the user sees and returning an ofstream variable instead of an ifstream.

```
int main(int argc, char const** argv) {
  open_output_file();
  return 0;
}
```

The behavior of the program is as follows:

```
[z_xps] stream_ciphers <=> g++ stream_ciphers.cpp
[z_xps] stream_ciphers <=> ./a.out
Please enter the name of your output file: test.txt
[z_xps] stream_ciphers <=> ls test.txt
test.txt
[z_xps] stream_ciphers <=>
```

Note: This function call should not generate any error messages. If it does then something larger than this function is the issue and you should seek guidance from a TA or an instructor on how to address it.

4.2.2 Parsing an Input file and Writing to and Output File

Once you get the prior 5 functions working as expected this next part should be easy. First, edit your main function so the only function call in it is the preprovided function call.
`sequential_file_encryption(open_output_file(), open_argv_file(argc, argv));`

- **sequential_file_encryption**
  Some of this function has already been implemented for you.
  First, it will prompt the user for an offset value. Then comes your job.
  This function will parse a given input file one line at a time and encrypt the line using the offset function you implemented earlier. How you obtain the data at each line is up to you. I would strongly recommend using the getline string library function.
  The string returned by offset should then be written to the output file provided in the parameter list (output).
  After each line of the input file is processed close both the input and output files.

  If you are having issues with this function here are a few debugging tips

  1. Print the data to cout instead of the output
     (remember you can encrypt data that isn't there in the first place)
  2. Don't use the quotes.txt file.
     We will use that file to grade, for sure. But, as you develope your code take it slow. Write your own file one line at a time to make sure everything is working properly.
  3. Defunctionalize your code.
     While your final submission must have all code written in the provided functions, that shouldn't stop you from writing the code in main. Then testing it and finally transferring it to the function and testing it again.

Assume the following main:

```
int main(int argc, char const** argv) {
  sequential_file_encryption(open_output_file(), open_argv_file(argc, argv));
  return 0;
}
```

The behavior of the program is as follows:

```
[z_xps] stream_ciphers <=> g++ stream_ciphers.cpp
[z_xps] stream_ciphers <=> ./a.out

ERROR: Incorrect usage!
./a.out <filename>
[z_xps] stream_ciphers <=> ./a.out quotes.txt
Please enter the name of your output file: cipher.txt
What is your shift key? 5
[z_xps] stream_ciphers <=> ./a.out cipher.txt
Please enter the name of your output file: message.txt
What is your shift key? -5
[z_xps] stream_ciphers <=>
```

What's happening here is two fold:

1. On the first run of the program. I am providing the program with the `quotes.txt` and then telling the program to create (or open) a file called `cipher.txt` to write the encrypted data to. Each line of quotes is then encrypted with a shift key of 5 and then written to cipher.
2. On the second run of the program. I am undoing all of what I just did in the first run by providing the exact same program with the `cipher.txt` file instead of the quotes file and "decrypting" it with a shift key of -5. This restores the original data to what was in quotes and writes that data to the `message.txt` file.

As part of the handout I have included the quotes, cipher, and message text file for you. Use the cipher text file to make sure your offset function is performing as expected.

4.3 Rotation Encryption

Analogous to the offset cipher encryption is the rotation based encryption schemes. Conceptually, rotation is a fairly easy subject to understand. For rotation you are given a string and a number of positions to "rotate" by.

E.g., Assume you are given the following string: "Hello" and 3 positions to rotate by. The following is the Naiive approach.

```
int main(int argc, char const *argv[]) {
  string message = "Hello";
  int r = 3;
  char c;
  int len = message.length();

  for (int i = 0; i < r; i++) {
      c = message[len - 1];

      for (int j = len - 1; j > 0; j--)
          message[j] = message[j - 1];
      message[0] = c;
  }
}
```

This approach is terrible for three big reasons.

First, it requires the string (character array) to be parsed more than once. For small strings this isn't a major issue; after the first pass your message is "oHell" then "loHel" and finally "lloHe". However, consider what would happen if your string was a million characters long and the value of r was in the hundred of thousands. That is on the order of 10^{11} (billions) of array accesses for a single line of encryption. Such approaches are never acceptable except under very strict circumstances.

Second, it cannot rotate to the left. If r is negative (as it will be in left rotation), then the for loop on line 6 of the code above will never terminate. This necessitates more code it accommodate for this fundamental flaw in rotation.

```
0  int main(int argc, char const *argv[]) {
     string message = "Hello";
2    int r = -3;
     char c;
4    int len = message.length();

6    if (r > 0) { // Rotate Right
         for (int i = 0; i < r; i++) {
8            c = message[len - 1];
             for (int j = len - 1; j > 0; j--)
10               message[j] = message[j - 1];
             message[0] = c;
12       }
     } else if (r < 0) { // Rotate Left
14       for (int i = r; i < 0; i++) {
             c = message[0];
16           for (int j = 0; j < len; j++)
                 message[j] = message[j + 1];
18           message[len - 1] = c;
         }
20   } }
```

Notice, that the structure of lines 13 to 18 are identical to the original algorithm to rotate right. Unfortunately, this is something that can't be overcome without introducing some clever index trickery.

The final reason for why this algorithm is terrible is because of its ability to allow rotations that are greater than the length of the message. Question: What is the message "Hello", after rotating by 5 or any other multiple of 5? Answer: The EXACT SAME THING. Meaning, if any application of the rotation function satisfies the following equation:

$$0 = r \ \% \ \text{message.length()}$$

Then, the output of the rotation function is the same as the input message and no work should be done other than returning the original message.

At face value rotation seems rather simple, but as you begin to think about the details the implementation complexity grows. This WILL be a common theme as you manoeuver through your engineering career. The ability to think of the details and create your own solutions is something every great engineer and scientist must master before you graduate university. Therefore, I want to you to develope your own rotation algorithm.

- **rotate(string, int)**
  The expected behavior of this function is rather simple. Given an input string and number of integer places to rotate, return the rotated string. However, we now know that the details of this will be complicated and I have some further restrictions for your implementations.

  1. The use of modulus and division operations are strictly forbidden WITHIN looping constructs.
  2. The rotator value must be adjusted to not allow excess rotations (as described in the final reason above)
  3. You may only evaluate each character of the input string ONCE (in contrast to reason 1)
  4. NO built in library functions other than length may be used.

  A good place to start this function by making a copy of the original message in another string and go from there. I also highly recommend busting out the pen and paper or dry erase board and markers to sketch out and verify your ideas before coding them. Don't develope code you aren't 100% sure will work and you will never have bugs :)

4.4 Combining Rotation and Offset

Once you have finished the rotate function and are sure it is working you can move on to the final function of this file.
Now that you have 2 methods of augmenting text data it is time to combine them. The beauty of encryption is that it is based on invertible functions. We saw this with offset where we moved a character left and inversely right on its number line and in rotation where we could rotate characters left and right within a string. Logically, we can conclude that the combination of these functions is safe to perform (so long as you remember how to undo the original manipulation).
First, I would like you to try rotating a string and then offsetting it by using the rotate and offset functions. This should give you some kind of jumbled mixture of letters that we can consider to be an encrypted message. Then I want you to reverse the offset and then reverse the rotation and you should end up with your original message. E.g.,

```
0 message = "..........";
  message = rotate(message, R);
2 message = offset(message, O);
  message = offset(message, -O);
4 message = rotate(message, -R);
```

Then I want you to do the same thing again but encrypt the message twice and then decrypt it twice with differing values. E.g.,

```
message = ".....";
message = rotate(message, R);
message = offset(message, O);
message = rotate(message, R2);
message = offset(message, O2);
message = offset(message, -O2);
message = rotate(message, -R2);
message = offset(message, -O);
message = rotate(message, -R);
```

If the final message is the same as your original than you are ready to implement the last function and get the password to part 2.

- **rotation_encryption(string)**
  This function performs the encryption or decryption protocols based on user input and will perform them for as many times as instructed by the scheme_parameters struct. You briefly learned about C/C++ structs towards the CS 135 and how they are a means of encapsulating data. I have already implemented the struct for you and all of the embedded data is there. The only thing you have to do is access it through the `es` variable. If you aren't sure how to do this, you will by then end of your lecture on classes.

```
struct scheme_parameters {
  static const int stages = 3;
  int rotate[stages] = {22, 4, 40};
  int offset[stages] = {2, 4, 8};
};
```

  The function will first prompt the user for a protocol to use; encryption (e or E) or decryption (d or D). Based on that input the function will then rotate then offset or offset then rotate `stages` many times (stages is a member of `scheme_parameters es`) where the rotate and offset parameters (R and O) are the values in the rotate and offset arrays.

Once you get the rotation_encryption(string) function working, it's time. Edit main so that the rotation_file_encryption function is the only one active. Then provide the function with the pt2Key.txt file and tell the function to decrypt (d or D).

This will give you the pass phrase to unlock the second part of this assignment. The pass phrase is something in English so that should help you to verify that it worked or not. Also keep in mind, the only one who knows the pass phrase is me and the only way to obtain it is by implementing all of this code correctly. So please, work hard to decrypt the key for yourself; because the graders will know if you didn't.

I'll see y'all in part 2.