CmpE102
Spring 2017
Programming Assignment 5

Due Date: 10 April
Goal: Vigenère cipher program

Instructions:

Step 1. Creating the assembly language file

Everything should be done the same way as the previous assignment.

Step 2. The application area

In the previous programming assignment we looked at ROT13 encoding, which is an example of a simple substitution cipher called a *Caesar cipher* (because it was supposedly used by Julius Caesar 2000 years ago).

ROT13 is simply a Caesar cipher with a rotation of 13. Caesar ciphers are relatively easy to break because the code does not change the frequency distribution of letters. For example, in English the most common letter is E--around 12% of all text. By counting up the number of different letters in a ciphertext, you can make a good guess which one is E. That gives you the rotation factor and then the cipher is broken. Or you could just try all possible rotations until you find the one that works (an elementary example of "brute-force attack".)

The Vigenère cipher, invented in 1553, uses multiple Caesar ciphers so that the frequency distribution of encoded letters is spread over several different encodings. Some versions were based on something like sliding tables or concentric letter wheels. However for military use in the field it is better not to need special equipment and to have the cipher be based on something easy to remember such as a keyword.

For better frequency characteristics the keyword should not have any repeated letters. Also, if it contains the letter A the encrypted letter will be the same as the plaintext, although this is not necessarily a bad thing.

To implement this algorithm with a pencil and paper, many descriptions ask you to build a Vigenère Square. However this is not really necessary when you are using a computer to do the encoding and decoding.

Essentially the keyword is written repeatedly over and over above the plaintext. Suppose the keyword is CRYPTOGRAM.

```
CRYPTOGRAMCRYPTOGRAMCRYPTOGRAMCRYPTOGRAMCRYPTOGRAMCRYPTOGRAMCRYPTOGR
WEHAVEBEENBETRAYEDALLISDISCOVEREDFLYATONCEMEETUSBYTHEOLDTREEATNINEPM
```

Consider that the letters are numbered 0 to 25. The letter on the top determines which Caesar-cypher to use for the letter below. Thus C means shift the alphabet by 2, A means shift by 0, and so on. **In mathematical terms, we are adding the two letters together modulo 26.** (The square was used because the concept of modular arithmetic was not generally understood by soldiers in 1553.)

To decrypt the message, the same operation is performed in reverse. That is, the value of the keyword letter is subtracted rather than added.

Another example (Andrew Pennycook, *Codes and Ciphers*, MacKay, New York, 1980):

```
Key:          SUFFERINGCATSSUFFERINGCA
Plaintext:    BETHEREATMIDNIGHTTONIGHT
Ciphertext:   TYYMIIMNZOIWFAAMYXFVVMST
```

Notice that the word "night" appears twice in the message. The first time it is encoded as FAAMY and the second time as VVMJT. There is nothing to show that they came from the same word in the original.

The Vigenère cipher was long considered to be unbreakable as long as the secrecy of the keyword could be preserved. However by the middle 19th century techniques for breaking the cipher had been developed, although they were not widely known until later. In the American Civil War, Confederate forces routinely made use of Vigenère ciphers, not realizing that Union intelligence had little difficulty deciphering the messages.

Step 3. What your code should do

1. Your code should use STDIN and STDOUT for input and output. (This is the default.) Use redirection on the command line to read from a file and write to a file.
2. Your code should open a file, read it character by character and *save it into an array*. Use C I/O functions.
3. When you get to the end of the file you should encode the contents of the array with a Vigenère cipher using the keyword CRYPTOGRAM, then print it out.
4. Maintain the distinction between upper-case and lower-case letters, and do not modify non-alphabetic characters. This is not very good for the security of your message, but the result will look neater.
5. This program should use **`glibc`** functions. In addition to **`printf()`**, you may need **`getchar()`** and **`putchar()`**.
6. Assume that the input file contains just ASCII text Don't worry about what happens with non-text files.

7. Once the encoder is working, build a decoder by duplicating the code and changing the addition to a subtraction.
8. If you use **`printf()`** to output the array, remember that a null termination is required on a string.

Hints

1. There are really two different parts to this problem. One is how to do I/O, the second is how to do the encoding. Solve one problem at a time.
2. Start with how to do I/O. **`getchar()`** and **`putchar()`** are described on slide 38 of Lecture 15. The first version of the code you should write should just output the text without changing it. Remember that when passing a character on the stack, it has to be extended to 32 bits.
3. The return value will be in EAX. This is how you can tell when you're at the end of the input file.
4. The input text should be stored in an array. *This is an essential requirement for the solution.* Make sure it is clear in your comments where this happens.
5. Once you've got that figured out, you can do the encoding part. To make sure you don't mess up the part that you already have figured out, you could implement the encoding as a function call. Obviously the function will take a character as an input and return a character. Since you're writing your own function, you can pass the character in a register rather than using the standard function interface.
6. Note that the encoding only changes the values 'a' through 'z' and 'A' through 'Z'. All other characters remain unchanged.
7. The algorithm as described assumes that the letters are represented as the numbers 0 through 25, but of course the ASCII code doesn't work exactly that way.

Step 4. Turning in the assignment

Turn in the following:

1. Commented listing file.
2. Demonstration of encoding an input text file of at least 100 characters. (Grab a piece of text off a website.) Show input and output characters.
3. Run the decoder on the encoded text to show that the encoded output from the previous run can be decoded back to the original text.
4. Circle the character count to verify that your program does not lose any characters. (It's okay that the output will gradually get longer as you cycle it through multiple times.)