CSC 211 Lab 3

Caesar cipher

Background

You are going to write a (very simple) encryption program, based on an ancient form of cryptography called a **Caesar cipher**, credited to Julius Caesar.

Also known as a **shift cipher**, the idea is to rotate letters around the alphabet by a certain amount; creating a substitute message. This encrypted message can be decrypted by rotating letters in the opposite direction by the same amount.

Consider a Caesar cipher that rotates the letters by 3 positions:

```
Plain:    ABCDEFGHIJKLMNOPQRSTUVWXYZ
Cipher:   XYZABCDEFGHIJKLMNOPQRSTUVW
```

So we could encrypt a message as follows:

```
Plaintext:  THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG
Ciphertext: QEB NRFZH YOLTK CLU GRJMP LSBO QEB IXWV ALD
```

Mathematically, we can describe encryption and decryption as functions:

$E_n(x) = (x + n) mod L$

and

$D_n(x) = (x - n) mod L$

where E_n encrypts with key n, D_n decrypts with key n, x is a letter in the message, and L is the length of the alphabet.

For our purposes, we can perform the rotation of each `char` by simply adding the key to it (for encryption) and subtracting the key (for decryption). Why does this work? Because unsigned integer addition *wraps around*, and a `char` is just a 1-byte unsigned integer. Remember the ASCII table? We're just changing the value of each `char` in a reversible way.

Getting Started

- Get into your Docker development environment and *change into the data directory*.
- Download the lab framework with `git clone https://github.com/csc211/lab3`
- Type `cd lab3` followed by `ls`
- You will see this `Readme.md` along with a C++ source file (`caesar.cpp`) and a compile script `compile`
- You will also see two input files: one very short one called `short.txt` and a longer one called `shakespeare.txt`.
- You can now compile the lab by typing `./compile` and it will produce an executable program called `caesar`. However, it won't do anything yet!

Requirements

- I have provided an empty `main()` function.
- I have also provided function declarations `char encrypt(char, int)` and `char decrypt(char, int)`.
- You must write the implementation of `encrypt()` and `decrypt()`, as well as handling command-line arguments and file I/O.
- Your program must take exactly **four** command-line arguments.
- The first is a string, either "-e" or "-d" indicating whether to call `encrypt()` or `decrypt()` respectively.
- The second is a number representing the key to be used for the encryption or decryption, so you'll need to use `atoi()` to turn it into an integer.
- The third is a filename to read from.
- The fourth is a filename to write to.
- Your program must read from the first file, determine whether to encrypt or decrypt the input, and write it out to the second file.

Hints

- It's much easier to keep track of input/output streams if you don't try to read and write at the same time. I recommend reading your input into a `string`, processing that `string`, and then writing the rotated (encrypted or decrypted) result to the output file.
- You have seen (and used) an idiom for line-at-a-time input. You've also seen (in class) character-at-a-time input. I strongly recommend reading your input a character at a time, so that you don't miss newline characters.
- Remember that you've seen two ways to convert from a string to an integer:
  - `atoi()`
  - `>>` from an `istringstream`

- Be careful when testing your code. Properly implemented, this program will **overwrite** the output file. So, if you want to test encryption and decryption, you'll need to be careful with filenames. Try something like:

```
./caesar -e 3 shakespeare.txt encrypted.txt
./caesar -d 3 encrypted.txt shake2.txt
```

- Now, how do you make sure that the decrypted file is the same as the original? Of course, you can open it in Atom and read it. But there is also the ever-useful unix `diff` command. You'd run this in the Docker command line just like you'd run your `caesar` program:

```
diff shakespeare.txt shake2.txt
```

If it doesn't produce any output, then your program works. If you see a whole bunch of output, then the files are different. You would be wise to test on *small* inputs (like `short.txt`).

Submitting

You will submit your code on Mimir.

Idioms

Suppose we have a `string` called `result`, and we want to write it to a file whose name is in the `string outfilename`. We can do the following (note that the primary difference from *reading* from a file is that we are using an `ofstream` instead of an `ifstream`):

```
ofstream outfile(outfilename);
if (!outfile.fail()){
    outfile << result;
    outfile.close();
} else {
    cerr << "Could not open file " << outfilename << endl;
    exit(EXIT_FAILURE);
}
```