hw01: Caesar Cypher

Focus:

- vectors
- strings
- functions
- Standard I/O
- Clean code!

Task:

- We are given a file of text that has been encrypted. For simplicity, we will assume the name is "encrypted.txt". That will save you having to ask the user.
- Luckily we know what algorithm was used, the classic Caesar cypher. Each lowercase alphabetic character has be modified by “rotating” it through the alphabet some number of steps.
- For example, if the number of steps was three and the original letter was 'b' then the encrypted file would instead have a 'e' in the same place, since 'e' is three letters past 'b' in the alphabet.
- Clearly we have to address the problem of what to do when the letter to be encrypted is too close to the end of the alphabet. For example, what should we do if we have to encrypt the letter 'y', again with the number of steps specified as three. The first step takes us to 'z'. What about the second step? We wrap around, back to 'a'. The third and final step then is easy, it will take us on to 'b'.
- Your program will read an encrypted file. You are to decrypt it.
    - The first item in the file will be an int, the number of steps that was used in the rotation.
    - The following lines will contain the text that you are to decode. Only lowercase alphabetic characters have been encrypted. The rest, e.g. uppercase, digits and punctuation all appear in the encrypted file the same way they did in the original.
- Oh, one other thing. To make the encoded file more difficult, they thought it would be clever to <u>reverse the lines in the file</u>. So what should be the first line shows up last, etc.
- An example. Suppose this is the file you were asked to decrypt:

```
5
Asi ymj rtrjwfymjx tzylwfgj.
```

```
Aqq rnrxd bjwj ymj gtwtlwtajx
Dni ldwj fsi ldrgqj ns ymj bfgj.
Tbfx gwnqqnl fsi ymj xqnymjd ytajx
```

- That says that the original file was encoded with a Caesar cypher using a rotation of five. Remember that the lines have also been reversed meaning that the first line of text we see is the encoded version of what was the last line of code in the original file, etc. When we decrypt it, we should get:

```
Twas brillig and the slithey toves
Did gyre and gymble in the wabe.
All mimsy were the borogroves
And the momerathes outgrabe.
```

Further Constraints

Functions

- A good program will have the task nicely broken up into functions. Certainly the solution to this assignment *could be* written with all of the code in main and take up barely 20 lines. But our goal throughout the semester is to make a practice of good programming style and that means, among other things, of making good use of functions.
- For this assignment we will actually tell you a couple of functions to provide. Usually we would leave it up to you to come up with them. After all, programming is supposed to be fun and it is less fun when someone is telling you what to do every step of the way.
- First, write a function that takes an encrypted character and returns the corresponding decrypted character. Note that only lowercase characters will result in a change. For all other characters, the function will return the input value. (Note that the function's return type will need to be `char`.)
- Second, write a function that, using the above function, will modify an encrypted string changing it into it into it unencrypted form. Note that unlike in some languages strings in C++ are *mutable*, meaning that we can directly change their contents. This function changes its input, it does not return anything, i.e. it has a void return type.

- Neither of the above functions prints anything. That will be the task of main, or another function if you see some merit in further breaking up the program.
- If you have not covered functions and parameter passing in your lecture yet, you can still take a first pass at solving this assignment putting everything in main, and then when that material is covered, add it to your solution. Once you know how, the changes shouldn't take long to add to a correctly working program. You can read up on functions and parameter passing in the lecture notes.

Submit

- Submit a single source file, hw01.cpp.

Things that might help

- In C++ and related languages, a character has the type `char`. A *character literal* is represented by the character between *single* quotes , e.g. `'m'`, as opposed to a *string literal* which is the string surrounded by double quotes, e.g. `“this is a string literal”`.
- We can compare characters with the usual operators `<`, `>`, `==`, etc. Worthy of note is that all of the lowercase characters are in order, so that `'a'` is less than `'b'`, etc.
- Characters are actually represented as a kind of integer, so we can also do arithmetic with them. We can add an integer to a character and get another character. For example, the expression `'m' + 3` has the same value as the character `'p'`.
- The spec says to read the input from a file. If you haven't covered files yet, you can still get started by reading the input from standard input, i.e. from `cin`. Note that the lecture notes on how to open / close a file and read from it can be found at http://cis.poly.edu/jsterling/cs2124/LectureNotes/01.Intro.html#fileio.
- Since you have to process each line as a whole, you will want to read the file line by line. The getline function is ideal for that. You can find it discussed in the lecture notes for file I/O, mentioned in the bullet above.
- One thing that may trip you up a little is that when you have read the integer at the beginning of the file that represents the rotation distance, your "read pointer" will be sitting immediately after the number and just before the newline character at the end of the line. Before you can read the next line, you need to gobble up that

newline character. One way to get rid of it is just to call getline an "extra" time, right after you read the integer and before you read the rest of the lines.

Good Code

- Reminder that this is really about writing good code. That means, your program needs:
- Good comments.
  - *Every* program file should have a comment at the top to identify the file, the author and, briefly, the purpose of the code in the file.
  - Every function definition should have a comment identifying its purpose, the purpose of its parameters and that of its return value. Unlike Python docstrings, this comment is just *before* the function.
  - Any block of code, e.g. loop, should have a comment to explain enough of what it does so that the reader could skip reading the code in that block.
- Good functions. Well, we addressed that above in the requirements. Normally, thinking of good functions would be a key part of your task.
- Good naming. Both variables and functions should have good names.
  - A function's name should say clearly what it does.
  - A variable's name should say clearly what it holds.
  - In general, single letter names are unacceptable.
    - Yes, we do often use the letters `i` and `j` as loop index variables, at least if the loop is reasonably short and their scope is limited by the loop itself. Even then, a more meaningful name helps clarify at a glance the intent of the code. An index variable called `row` might work better in some cases.
  - Similarly, variables with names like `temp` don't help the reader.
- Good format.
  - It should be easy to see the *structure* of your program even if it was held far enough away that you could not read the code itself. Unlike Python, C++ does not require indentation. But good indentation makes code easier to read.
  - Line length. Long lines have two problems. First, unless you are reading the code on a device with a wide enough window,

    there is going to be annoying line wrap. And secondly, it just makes it harder to see the logic that must be embedded in there. A good rule of thumb is to assume your program will eventually get printed and due to whatever limitations, anything over 80 characters in a line will result in line wrap.
  - Whitespace. It usually makes code more readable if you put whitespace around your operators. Especially when there is a lot going on on the line. A simple example would be that `cin >> word`, is more readable than `cin>>word`
  - Some people will insist that the placement of an opening curly brace for a block is extremely important. { Wars have been waged over such! } I have my preferences, but I won't force you to share them. Just be consistent.
- Global variables. **Don't.** No program that you will write for this course will need global variables. Do not use them. Global *constants*, on the other hand are highly recommended.