# Vigenère

Next

## Ooh, la la!

Vigenère's cipher improves upon Caesar's cipher by encrypting messages using a *sequence* of keys (or, put another way, a *keyword*).

In other words, if *p* is some plaintext and *k* is a keyword (i.e., an alphabetical string, whereby A (or a) represents 0, B (or b) represents 1, C (or c) represents 2, …, and Z (or z) represents 25), then each letter, $c_i$, in the ciphertext, *c*, is computed as:

$c_i = (p_i + k_j) \% 26$

Note this cipher's use of $k_j$ as opposed to just *k*. And if *k* is shorter than *p*, then the letters in *k* must be reused cyclically as many times as it takes to encrypt *p*.

In other words, if Vigenère himself wanted to say HELLO to someone confidentially, using a keyword of, say, ABC, he would encrypt the H with a key of 0 (i.e., A), the E with a key of 1 (i.e., B), and the first L with a key of 2 (i.e., C), at which point he'd be out of letters in the keyword, and so he'd reuse (part of) it to encrypt the second L with a key of 0 (i.e., A) again, and the O with a key of 1 (i.e., B) again. And so he'd write HELLO as HFNLP, per the below:

| plaintext | H | E | L | L | O |
|---|---|---|---|---|---|
| + key | A | B | C | A | B |
| (shift value) | 0 | 1 | 2 | 0 | 1 |
| = ciphertext | H | F | N | L | P |

Let's now write a program called `vigenere` that enables you to encrypt messages using Vigenère's cipher. At the time the user executes the program, they should decide, by providing a command-line argument, on what the keyword should be for the secret message they'll provide at runtime.

Here are a few examples of how the program might work.

```
$ ./vigenere bacon
plaintext:  Meet me at the park at eleven am
ciphertext: Negh zf av huf pcfx bt gzrwep oz
```

or for when the user provides a keyword that is not fully alphabetic:

```
$ ./vigenere 13
Usage: ./vigenere keyword
```

or for when they don't provide a keyword at all:

```
$ ./vigenere
Usage: ./vigenere keyword
```

or for when they provide too many keywords:

```
$ ./vigenere bacon and eggs
Usage: ./vigenere keyword
```

Try It

How to begin? Let's start with something familiar.

Next

## Déjà vu

As you may have gleaned already, the basic idea for this cipher is strikingly similar to the idea underlying Caesar's cipher. As such, our code from Caesar seems like a good place to begin, so feel free to start by replacing the entire contents of `vigenere.c`, at right, with your solution to `caesar.c`.

One difference between Caesar's and Vigenère's ciphers is that the key for Vigenère's cipher is a series of letters, rather than a number. So let's make sure that the user actually gave us a keyword! Modify the check you implemented in Caesar to instead ensure every character of the keyword is alphabetic, rather than a digit. If any of them isn't, print `Usage: ./vigenere keyword` and return a non-zero value as we did before. If they are all alphabetic, after checking you should print `Success` and then, `return 0;` immediately (for now), since our enciphering code is not quite ready to work just yet, so we won't have our program execute it.

Sample behavior:

```
$ ./vigenere alpha
Success
```

or

```
$ ./vigenere 123
Usage: ./vigenere keyword
```

**Hints**

- Recall that the `string.h` header file contains a number of useful functions that work with strings. See CS50 Reference's menu for some!
- Recall that we can use a loop to iterate over each character of a string if we know its length.
- Recall that the `ctype.h` header file contains a number of useful functions that tell us things about characters. See CS50 Reference's menu for some!

Next

## Getting the shift value

Let's for now assume that the user is providing single-character keywords. Can we convert that character into the correct shift value? Let's do so by writing a *function*.

Near the top of your file, below the `#include` lines, let's *declare* the *prototype* for a new function whose purpose is to do just that. It will take a single character as input, and it will output the shift value for that character.

```
int shift(char c);
```

Now we've declared a function called `shift` that takes a single character ( `c` ) as input, and will output an integer.

Now, down below the closing curly brace of `main` , let's give ourselves a place to *define* (i.e., implement) this new function.

```
int shift(char c)
{
   // TODO
}
```

In place of that `TODO` is where we'll do the work of converting that character to its positional integer value (so, again, `A` or `a` would be 0, `B` or `b` would be 1, `Z` or `z` would be 25, etc.)

To test this out, delete the line where you printed `"Success"` (but leave the `return 0;` for now), and in place of the just-deleted line, add the below lines to test whether your code works.

```
int key = shift(argv[1][0]);
printf("%i\n", key);
```

Your program should print a 0 if run with the keyword `A` or `a` . Try running the program with other capital and lowercase letters as the keyword. Is the behavior what you expect?

**Hints**

- Functions have inputs and outputs.

- When we *declare* a function, we need to provide its return type, name, and an argument list, each of which also has a type.
- When we *use* or *call* a function, we just plug in appropriate values in the argument list, and assign the output of the function to a variable that corresponds to the function's return type.
- If `argv[1]` is a string, then `argv[1][0]` is just the first character of that string.
- Recall that the `ctype.h` header file contains a number of useful functions that tell us things about characters.
- The ASCII value of `A` is 65. The ASCII value of `a` is 97.
- The ASCII value of `B` is 66. The ASCII value of `b` is 98. See a potential pattern emerging?

Next

## One-character keywords

Time to get back to using that enciphering code you wrote before! You may have noticed that if your keyword *k* consists of exactly one letter (say, `H` or `h`), Vigenère's cipher effectively becomes a Caesar cipher (of, in this example, 7). Let's for now indeed assume the user's keyword will just be a single letter. Use your newly-written `shift` function to calculate the shift value for the letter they provided, assign the return value of that function to an integer variable `key`, and use `key` exactly as you did in Caesar's cipher! It should suffice, in fact, to simply delete the recently-added `printf` and the `return 0;` line now, letting the program finally proceed to your previously-written Caesar cipher code!

```
$ ./vigenere A
plaintext:  hello
ciphertext: hello
```

or

```
$ ./vigenere b
plaintext:  HELLO
ciphertext: IFMMP
```

or

```
$ ./vigenere C
```

```
plaintext:  HeLlO
ciphertext: JgNnQ
```

Hints

If some of your variables in your Caesar solution don't match what they've been called so far in this lab, just edit the names of things so they do match!

Next

## Final Steps

Now it's your turn to take things across the finish line by implementing the remaining functionality in `vigenere.c`. Remember that the user's keyword will probably consist of multiple letters, so you may need to calculate a new shift value for each letter of the plaintext; you may then want to move your `shift` function into your loop somehow.

Remember also that every time you encipher a character, you need to move to the next letter of *k*, the keyword (and wrap around to the beginning of the keyword if you exhaust all of its characters). But if you don't encipher a character (e.g., a space or a punctuation mark), don't advance to the next character of *k*!

And as before, be sure to preserve case, but do so only based on the case of the original message. Whether or not a letter in the keyword is capitalized should have no bearing on whether a letter in the ciphertext is!

Hints

- You'll probably need one counter, `i` for iterating over the plaintext and one counter, `j` for iterating over the keyword.
- You'll probably find it easiest to control the keyword counter yourself, rather than relying on the `for` loop you're using to iterate over the plaintext!
- If the length of the keyword is, say, 4 characters, then the last character of that keyword can be found at `keyword[3]`. Then, for the next character you encipher, you'll want to use `keyword[0]`.

Next

## How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks ( * ) instead of the actual characters in your password.

```
submit50 cs50/problems/2019/x/vigenere
```

You can then go to https://cs50.me/cs50x to view your current scores!