Table of Contents

- tl;dr
- Background
- Specification
- Walkthrough
- Usage
- Testing
  - Correctness
  - Style
- Staff’s Solution
- Hints

Caesar

tl;dr

Implement a program that encrypts messages using Caesar’s cipher, per the below.

```
$ ./caesar 13
plaintext:  HELLO
ciphertext: URYYB
```

Background

Supposedly, Caesar (yes, that Caesar) used to "encrypt" (i.e., conceal in a reversible way) confidential messages by shifting each letter therein by some number of places. For instance, he might write A as B, B as C, C as D, …, and, wrapping around alphabetically, Z as A. And so, to say HELLO to someone, Caesar might write IFMMP. Upon receiving such messages from Caesar, recipients would have to "decrypt" them by shifting letters in the opposite direction by the same number of places.

The secrecy of this "cryptosystem" relied on only Caesar and the recipients knowing a secret, the number of places by which Caesar had shifted his letters (e.g., 1). Not particularly secure by modern standards, but, hey, if you’re perhaps the first in the world to do it, pretty secure!

Unencrypted text is generally called *plaintext*. Encrypted text is generally called *ciphertext*. And the secret used is called a *key*.

Table 1. Encrypting HELLO with a key of 1 yields IFMMP.

plaintext	H	E	L	L	O
+ key	1	1	1	1	1
= ciphertext	I	F	M	M	P

More generally, Caesar's algorithm (i.e., cipher) encrypts messages by "rotating" each letter by k positions. More formally, if p is some plaintext (i.e., an unencrypted message), p_i is the i^{th} character in p, and k is a secret key (i.e., a non-negative integer), then each letter, c_i, in the ciphertext, c, is computed as

$$c_i = (p_i + k) \bmod 26$$

wherein $\bmod 26$ here means "remainder when dividing by 26." This formula perhaps makes the cipher seem more complicated than it is, but it's really just a concise way of expressing the algorithm precisely.

Specification

Design and implement a program, `caesar`, that encrypts messages using Caesar's cipher.

- Implement your program in a file called `caesar.c` in a directory called `caesar`.
- Your program must accept a single command-line argument, a non-negative integer. Let's call it *k* for the sake of discussion.
- If your program is executed without any command-line arguments or with more than one command-line argument, your program should print an error message of your choice (with `printf`) and return from `main` a value of `1` (which tends to signify an error) immediately.
- You can assume that, if a user does provide a command-line argument, it will be a non-negative integer (e.g., `1`). No need to check that it's indeed numeric.
- Do not assume that *k* will be less than or equal to 26. Your program should work for all non-negative integral values of *k* less than 2^{31} - 26. In other words, you don't need to worry if your program eventually breaks if the user chooses a value for *k* that's too big or almost too big to fit in an `int`. (Recall that an `int` can overflow.) But, even if *k* is greater than 26, alphabetical characters in your program's input should remain alphabetical characters in your program's output. For instance, if *k* is

27, `A` should not become `[` even though `[` is 27 positions away from `A` in ASCII, per asciichart.com (http://www.asciichart.com/); `A` should become `B`, since `B` is 27 positions away from `A`, provided you wrap around from `Z` to `A`.

- Your program must output `plaintext:` (without a newline) and then prompt the user for a `string` of plaintext (using `get_string`).
- Your program must output `ciphertext:` (without a newline) followed by the plaintext’s corresponding ciphertext, with each alphabetical character in the plaintext "rotated" by *k* positions; non-alphabetical characters should be outputted unchanged.
- Your program must preserve case: capitalized letters, though rotated, must remain capitalized letters; lowercase letters, though rotated, must remain lowercase letters.
- After outputting ciphertext, you should print a newline. Your program should then exit by returning `0` from `main`.

Walkthrough

Usage

Your program should behave per the examples below. Assumed that the underlined text is what some user has typed.

```
$ ./caesar 1
plaintext:  HELLO
ciphertext: IFMMP
```

```
$ ./caesar 13
plaintext:  hello, world
ciphertext: uryyb, jbeyq
```

```
$ ./caesar 13
plaintext:  be sure to drink your Ovaltine
ciphertext: or fher gb qevax lbhe Binygvar
```

```
$ ./caesar
Usage: ./caesar k
```

```
$ ./caesar 1 2 3 4 5
Usage: ./caesar k
```

Testing

Correctness

```
check50 cs50/2018/x/caesar
```

Style

```
style50 caesar.c
```

Staff’s Solution

```
~cs50/pset2/caesar
```

Hints

This program needs to accept a command-line argument, *k*, so you'll want to declare `main` with:

```
int main(int argc, string argv[])
```

Recall that `argv` is an "array" of strings. You can think of an array as row of gym lockers, inside each of which is some value (and maybe some socks). In this case, inside each such locker is a `string`. To open (i.e., "index into") the first locker, you use syntax like `argv[0]`, since arrays are "zero-indexed." To open the next locker, you use syntax like `argv[1]`. And so on. Of course, if there are `n` lockers, you'd better stop opening lockers once you get to `argv[n - 1]`, since `argv[n]` doesn't exist! (That or it belongs to someone else, in which case you still shouldn't open it.)

And so you can access *k* with code like

```
string k = argv[1];
```

assuming it's actually there! Recall that `argc` is an `int` that equals the number of strings that are in `argv`, so you'd best check the value of `argc` before opening a locker that might not exist! Ideally, `argc` will be `2`. Why? Well, recall that inside of `argv[0]`, by default, is a program's own name. So `argc` will always be at least `1`. But for this program you want the user to provide a command-line argument, `k`, in which case `argc` should be `2`. Of course, if the user provides more than one command-line argument at the prompt, `argc` could be greater than `2`, in which case, again, your program should print an error and return `1`.

Now, just because the user types an integer at the prompt, that doesn't mean their input will be automatically stored in an `int`. Au contraire, it will be stored as a `string` that just so happens to look like an `int`! And so you'll need to convert that `string` to an actual `int`. As luck would have it, a function, `atoi` (https://reference.cs50.net/stdlib/atoi), exists for exactly that purposes. Here's how you might use it:

```
int k = atoi(argv[1]);
```

Notice, this time, we've declared `k` as an actual `int` so that you can actually do some arithmetic with it.

Because `atoi` is declared in `stdlib.h`, you'll want to `#include` that header file atop your own code. (Technically, your code will compile without it there, since we already `#include` it in `cs50.h`. But best not to trust another library to `#include` header files you know you need.)

Okay, so once you've got `k` stored as an `int`, you'll need to ask the user for some plaintext. Odds are CS50's own `get_string` can help you with that.

Once you have both `k` and some plaintext, `p`, it's time to encrypt the latter with the former. Recall that you can iterate over the characters in a `string`, printing each one at a time, with code like the below:

```
for (int i = 0, n = strlen(p); i < n; i++)
{
    printf("%c", p[i]);
}
```

In other words, just as `argv` is an array of strings, so is a `string` an array of chars. And so you can use square brackets to access individual characters in strings just as you can individual strings in `argv`. Neat, eh? Of course, printing each of the characters in a string one at a time isn't exactly cryptography. Well, maybe technically if *k* is 0. But the above should help you help Caesar implement his cipher!

Incidentally, you'll need to `#include` yet another header file in order to use `strlen` (https://reference.cs50.net/string/strlen).

Besides `atoi`, you might find some handy functions documented at reference.cs50.net (https://reference.cs50.net/) under **ctype.h** and **stdlib.h**. For instance, `isalpha` might prove helpful when iterating over plaintext's characters.

And, with regard to wrapping around from `Z` to `A` (or `z` to `a`), don't forget about `%`, C's modulo operator. You might also want to check out http://asciichart.com/ (http://asciichart.com/), which reveals the ASCII codes for more than just alphabetical characters, just in case you find yourself printing some characters accidentally.