Caesar

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*.

To be clear, then, here's how 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 formally, 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

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

wherein `% 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. Indeed, for the sake of discussion, think of A (or a) as 0, B (or b) as 1, …, H (or h) as 7, I (or i) as 8, …, and Z (or z) as 25. Suppose that Caesar just wants to say Hi to someone confidentially using, this time, a key, k, of 3. And so his plaintext, p, is Hi, in which case his plaintext's first character, p_0, is H (aka 7), and his plaintext's second character, p_1, is i (aka 8). His ciphertext's first character, c_0, is thus K, and his ciphertext's second character, c_1, is thus L. Can you see why?

Let's write a program called `caesar` that enables you to encrypt messages using Caesar's cipher. At the time the user executes the program, they should decide, by providing a command-line argument, on what the key should be in the secret message they'll provide at runtime. We shouldn't necessarily assume that the user's key is going to be a number; though you may assume that, if it is a number, it will be a positive integer.

Here are a few examples of how the program might work. For example, if the user inputs a key of `1` and a plaintext of `HELLO`:

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

Here's how the program might work if the user provides a key of `13` and a plaintext of `hello, world`:

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

Notice that neither the comma nor the space were “shifted” by the cipher. Only rotate alphabetical characters!

How about one more? Here’s how the program might work if the user provides a key of `13` again, with a more complex plaintext:

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

▶ **Why?**

Notice that the case of the original message has been preserved. Lowercase letters remain lowercase, and uppercase letters remain uppercase.

And what if a user doesn’t cooperate?

```
$ ./caesar HELLO
Usage: ./caesar key
```

Or really doesn’t cooperate?

```
$ ./caesar
Usage: ./caesar key
```

Or even…

```
$ ./caesar 1 2 3
Usage: ./caesar key
```

▶ **Try It**

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.
- If any of the characters of the command-line argument is not a decimal digit, your program should print the message `Usage: ./caesar key` and return from `main` a value of `1`.
- 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 http://www.asciichart.com/[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`.

How to begin? Let's approach this problem one step at a time.

Pseudocode

First, write some pseudocode that implements this program, even if not (yet!) sure how to write it in code. There's no one right way to write pseudocode, but short English sentences suffice. Recall how we wrote pseudocode for finding Mike Smith. Odds are your pseudocode will use (or imply using!) one or more functions, conditions, Boolean expressions, loops, and/or variables.

▶ **Spoiler**

Counting Command-Line Arguments

Whatever your pseudocode, let's first write only the C code that checks whether the program was run with a single command-line argument before adding additional functionality.

Specifically, modify `caesar.c` in such a way that: if the user provides exactly one command-line argument, it prints `Success` ; if the user provides no command-line arguments, or two or more, it prints `Usage: ./caesar key` . Remember, since this key is coming from the command line at runtime, and not via `get_string` , we don't have an opportunity to re-prompt the user. The behavior of the resulting program should be like the below.

```
$ ./caesar 20
Success
```

or

```
$ ./caesar
Usage: ./caesar key
```

or

```
$ ./caesar 1 2 3
Usage: ./caesar key
```

▶ **Hints**

Accessing the Key

Now that your program is (hopefully!) accepting input as prescribed, it's time for another step.

Recall that in our program, we must defend against users who technically provide a single command-line argument (the key), but provide something that isn't actually an integer, for example:

```
$ ./caesar xyz
```

Before we start to analyze the key for validity, though, let's make sure we can actually read it. Further modify `caesar.c` such that it not only checks that the user has provided just one command-line argument, but after verifying that, prints out that single command-line argument. So, for example, the behavior might look like this:

```
$ ./caesar 20
Success
20
```

▼ **Hints**

Recall that `argc` and `argv` give you information about what was provided at the command line.

Recall that `argv` is an array of strings.

Recall that with `printf` we can print a string using `%s` as the placeholder.

Recall that computer scientists like counting starting from 0.

Recall that we can access individual elements of an array, such as `argv` using square brackets, for example: `argv[0]` .

Validating the Key

Now that you know how to read the key, let's analyze it. Modify `caesar.c` such that instead of printing out the command-line argument provided, your program instead checks to make sure that each character of that command line argument is a decimal digit (i.e., `0` , `1` , `2` , etc.) and, if any of them are not, terminates after printing the message `Usage: ./caesar key` . But if the argument consists solely of digit characters, you should convert that string (recall that `argv` is an array of strings, even if those strings happen to look like numbers) to an actual integer, and print out the *integer*, as via `%i` with `printf` . So, for example, the behavior might look like this:

```
$ ./caesar 20
Success
20
```

or

```
$ ./caesar 20x
Usage: ./caesar key
```

▼ **Hints**

Recall that `argv` is an array of strings.

Recall that a string, meanwhile, is just an array of `char` s.

Recall that the `string.h` header file contains a number of useful functions that work with strings.

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.

Recall that we can `return` nonzero values from `main` to indicate that our program did not finish successfully.

Recall that with `printf` we can print an integer using `%i` as the placeholder.

Recall that the `atoi` function converts a string that looks like a number into that number.

Peeking Underneath the Hood

As human beings it's easy for us to intuitively understand the formula described above, inasmuch as we can say "H + 1 = I". But can a computer understand that same logic? Let's find out. For now, we're going to temporarily ignore the key the user provided and instead prompt the user for a secret message and attempt to shift all of its characters by just 1.

Extend the functionality of `caesar.c` such that, after validating the key, we prompt the user for a string and then shift all of its characters by 1, printing out the result. We can also at this point probably remove the line of code we wrote earlier that prints `Success` . All told, this might result in this behavior:

```
$ ./caesar 1
plaintext:  hello
ciphertext: ifmmp
```

▶ **Hints**

Your Turn

Now it's time to tie everything together! Instead of shifting the characters by 1, modify `caesar.c` to instead shift them by the actual key value. And be sure to preserve case! Uppercase letters should stay uppercase, lowercase letters should stay lowercase, and characters that aren't alphabetical should remain unchanged.

▼ **Hints**

Best to use the modulo (i.e., remainder) operator, `%`, to handle wraparound from Z to A! But how?

Things get weird if we try to wrap `Z` or `z` by 1 using the technique in the previous section.

Things get weird also if we try to wrap punctuation marks using that technique.

Recall that ASCII maps all printable characters to numbers.

Recall that the ASCII value of `A` is 65. The ASCII value of `a`, meanwhile, is 97.

If you're not seeing any output at all when you call `printf`, odds are it's because you're printing characters outside of the valid ASCII range from 0 to 127. Try printing characters as numbers (using `%i` instead of `%c`) at first to see what values you're printing, and make sure you're only ever trying to print valid characters!

Walkthrough

How to Test Your Code

Execute the below to evaluate the correctness of your code using `check50`. But be sure to compile and test it yourself as well!

```
check50 cs50/problems/2020/x/caesar
```

Execute the below to evaluate the style of your code using `style50`.

```
style50 caesar.c
```

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/2020/x/caesar
```