# Caesar Cypher

For this problem, you'll 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 instead. 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`. Make sense?

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, 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
```

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, providing a command-line argument that isn't a number? The program should remind the user how to use the program:

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

Or really doesn't cooperate, providing no command-line argument at all? The program should remind the user how to use the program:

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

Or really, really doesn't cooperate, providing more than one command-line argument? The program should remind the user how to use the program:

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

# 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 https://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:` (with two spaces but without a newline) and then prompt the user for a `string` of plaintext (using `get_string`).
- Your program must output `ciphertext:` (with one space but 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`.