This is CS50

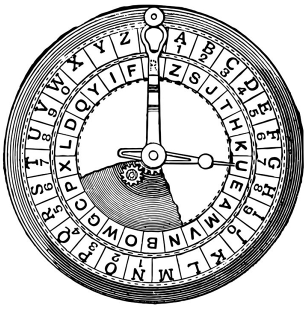
CS50's Introduction to Computer Science

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://www.reddit.com/user/davidjmalan)
(https://www.threads.net/@davidjmalan) (https://twitter.com/davidjmalan)

Caesar



Problem to Solve

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?

In a file called `caesar.c` in a folder called `caesar`, write a program 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.

Demo

```
$ ./caesar
```

Recorded with **asciinema**

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 asciitable.com (https://www.asciitable.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`.

Advice

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

Pseudocode

First write, try to write a `main` function in `caesar.c` that implements the program using just pseudocode, even if not (yet!) sure how to write it in actual code.

▼ **Hint**

There's more than one way to do this, so here's just one!

```
int main(int argc, string argv[])
{
    // Make sure program was run with just one command-line argument

    // Make sure every character in argv[1] is a digit

    // Convert argv[1] from a `string` to an `int`

    // Prompt user for plaintext

    // For each character in the plaintext:

        // Rotate the character if it's a letter
}
```

It's okay to edit your own pseudocode after seeing ours here, but don't simply copy/paste ours into your own!

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 `main` in `caesar.c` in such a way that, if the user provides no command-line arguments, or two or more, the function prints `"Usage: ./caesar key\n"` and then returns `1`, effectively exiting the program. If the user provides exactly one command-line argument, the program should print nothing and simply return `0`. The program should thus behave per the below.

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

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

```
$ ./caesar 1
```

▶ **Hints**

Checking the Key

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

Add to `caesar.c`, below `main`, a function called, e.g., `only_digits` that takes a `string` as an argument and returns `true` if that `string` contains only digits, `0` through `9`, else it returns `false`. Be sure to add the function's prototype above `main` as well.

▼ **Hints**

- Odds are you'll want a prototype like:

  ```
  bool only_digits(string s);
  ```

  And be sure to include `cs50.h` atop your file, so that the compiler recognizes `string` (and `bool`).
- Recall that a `string` is just an array of `char`s.
- Recall that `strlen`, declared in `string.h`, calculates the length of a `string`.
- You might find `isdigit`, declared in `ctype.h`, to be helpful, per manual.cs50.io (https://manual.cs50.io/). But note that it only checks one `char` at a time!

Then modify `main` in such a way that it calls `only_digits` on `argv[1]`. If that function returns `false`, then `main` should print `"Usage: ./caesar key\n"` and return `1`. Else `main` should simply return `0`. The program should thus behave per the below:

```
$ ./caesar 42
```

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

Using the Key

Now modify `main` in such a way that it converts `argv[1]` to an `int`. You might find `atoi`, declared in `stdlib.h`, to be helpful, per [manual.cs50.io (https://manual.cs50.io/)](https://manual.cs50.io/). And then use `get_string` to prompt the user for some plaintext with `"plaintext: "`.

Then, implement a function called, e.g., `rotate`, that takes a `char` as input and also an `int`, and rotates that `char` by that many positions if it's a letter (i.e., alphabetical), wrapping around from `Z` to `A` (and from `z` to `a`) as needed. If the `char` is not a letter, the function should instead return the same `char` unchanged.

▼ Hints

- Odds are you'll want a prototype like:

  ```
  char rotate(char c, int n);
  ```

  A function call like

  ```
  rotate('A', 1)
  ```

  or even

  ```
  rotate('A', 27)
  ```

  should thus return `'B'`. And a function call like

  ```
  rotate('!', 13)
  ```

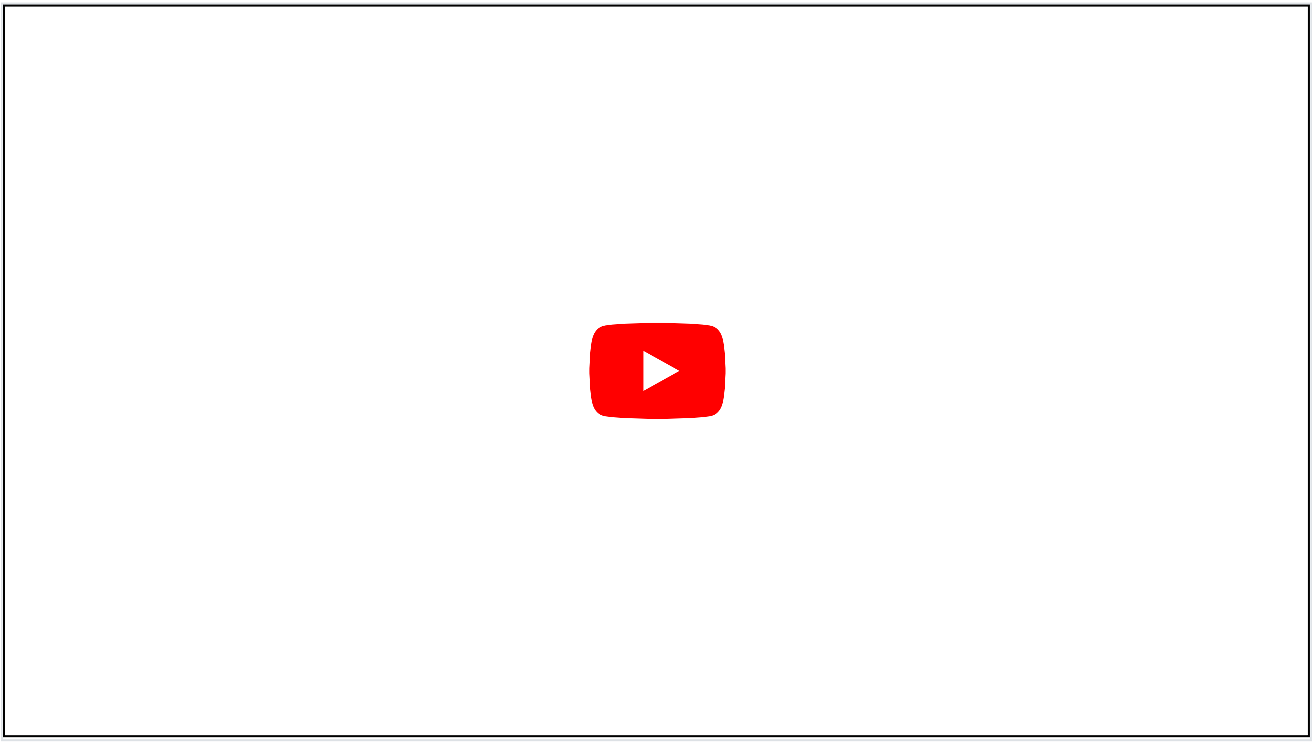
  should return `'!'`.
- Recall that you can explicitly "cast" a `char` to an `int` with `(int)`, and an `int` to a `char` with `(char)`. Or you can do so implicitly by simply treating one as the other.
- Odds are you'll want to subtract the ASCII value of `'A'` from any uppercase letters, so as to treat `'A'` as `0`, `'B'` as `1`, and so forth, while performing arithmetic. And then add it back when done with the same.
- Odds are you'll want to subtract the ASCII value of `'a'` from any lowercase letters, so as to treat `'a'` as `0`, `'b'` as `1`, and so forth, while performing arithmetic. And then add it back when done with the same.
- You might find some other functions declared in `ctype.h` to be helpful, per [manual.cs50.io (https://manual.cs50.io/)](https://manual.cs50.io/).
- Odds are you'll find `%` helpful when "wrapping around" arithmetically from a value like `25` to `0`.

Then modify `main` in such a way that it prints `"ciphertext: "` and then iterates over every `char` in the user's plaintext, calling `rotate` on each, and printing the return value thereof.

▼ Hints

  - Recall that `printf` can print a `char` using `%c`.
  - 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 temporarily as numbers (using `%i` instead of `%c`) to see what values you're printing!

Walkthrough



How to Test

Correctness

In your terminal, execute the below to check your work's correctness.

```
check50 cs50/problems/2024/x/caesar
```

▶ **How to Use `debug50`**

Style

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

```
style50 caesar.c
```

How to Submit

In your terminal, execute the below to submit your work.

```
submit50 cs50/problems/2024/x/caesar
```