This is CS50x

OpenCourseWare

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.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Problem Set 2

For this problem set, you'll use CS50 IDE, a cloud-based programming environment.

What to Do

1. Go to ide.cs50.io (https://ide.cs50.io) and click "Sign in with GitHub" to access your CS50 IDE.
2. Submit Readability
3. Submit one of:
   - Caesar if feeling less comfortable
   - Substitution if feeling more comfortable

If you submit both Caesar and Substitution, we'll record the higher of your two scores.

When to Do It

By 11:59pm on 31 December 2020.

Advice

- Try out any of David's programs from class via Week 2's source code.
- If you see any errors when compiling your code with `make`, focus first on fixing the very first error you see, scrolling up as needed. If unsure what it means, try asking `help50` for help. For instance, if trying to compile `hello`, and

  ```
  make readability
  ```

  is yielding errors, try running

  ```
  help50 make readability
  ```

  instead!

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.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Readability

Implement a program that computes the approximate grade level needed to comprehend some text, per the below.

```
$ ./readability
Text: Congratulations! Today is your day. You're off to Great Places! You're off and away!
Grade 3
```

Reading Levels

According to Scholastic (https://www.scholastic.com/teachers/teaching-tools/collections/guided-reading-book-lists-for-every-level.html), E.B. White's "Charlotte's Web" is between a second and fourth grade reading level, and Lois Lowry's "The Giver" is between an eighth grade reading level and a twelfth grade reading level. What does it mean, though, for a book to be at a "fourth grade reading level"?

Well, in many cases, a human expert might read a book and make a decision on the grade for which they think the book is most appropriate. But you could also imagine an algorithm attempting to figure out what the reading level of a text is.

So what sorts of traits are characteristic of higher reading levels? Well, longer words probably correlate with higher reading levels. Likewise, longer sentences probably correlate with higher reading levels, too. A number of "readability tests" have been developed over the years, to give a formulaic process for computing the reading level of a text.

One such readability test is the Coleman-Liau index. The Coleman-Liau index of a text is designed to output what (U.S.) grade level is needed to understand the text. The formula is:

```
index = 0.0588 * L - 0.296 * S - 15.8
```

Here, `L` is the average number of letters per 100 words in the text, and `S` is the average number of sentences per 100 words in the text.

Let's write a program called `readability` that takes a text and determines its reading level. For example, if user types in a line from Dr. Seuss:

```
$ ./readability
Text: Congratulations! Today is your day. You're off to Great Places! You're off and away!
Grade 3
```

The text the user inputted has 65 letters, 4 sentences, and 14 words. 65 letters per 14 words is an average of about 464.29 letters per 100 words. And 4 sentences per 14 words is an average of about 28.57 sentences per 100 words. Plugged into the Coleman-Liau formula, and rounded to the nearest whole number, we get an answer of 3: so this passage is at a third grade reading level.

Let's try another one:

```
$ ./readability
Text: Harry Potter was a highly unusual boy in many ways. For one thing, he hated the summer holidays more than any other time of
Grade 5
```

This text has 214 letters, 4 sentences, and 56 words. That comes out to about 382.14 letters per 100 words, and 7.14 sentences per 100 words. Plugged into the Coleman-Liau formula, we get a fifth grade reading level.

As the average number of letters and words per sentence increases, the Coleman-Liau index gives the text a higher reading level. If you were to take this paragraph, for instance, which has longer words and sentences than either of the prior two examples, the formula would give the text an eleventh grade reading level.

```
$ ./readability
Text: As the average number of letters and words per sentence increases, the Coleman-Liau index gives the text a higher reading l
Grade 11
```

Specification

Design and implement a program, `readability`, that computes the Coleman-Liau index of the text.

- Implement your program in a file called `readability.c` in a directory called `readability`.
- Your program must prompt the user for a `string` of text (using `get_string`).
- Your program should count the number of letters, words, and sentences in the text. You may assume that a letter is any lowercase character from `a` to `z` or any uppercase character from `A` to `Z`, any sequence of characters separated by spaces should count as a word, and that any occurrence of a period, exclamation point, or question mark indicates the end of a sentence.
- Your program should print as output `"Grade X"` where `X` is the grade level computed by the Coleman-Liau formula, rounded to the nearest integer.
- If the resulting index number is 16 or higher (equivalent to or greater than a senior undergraduate reading level), your program should output `"Grade 16+"` instead of giving the exact index number. If the index number is less than 1, your program should output `"Before Grade 1"`.

Getting Started

Log into CS50 IDE (https://ide.cs50.io/) and then, in a terminal window, execute each of the below.

- Execute `cd` to ensure that you're in `~/` (i.e., your home directory).
- Execute `mkdir readability` to make (i.e., create) a directory called `readability` in your home directory.
- Execute `cd readability` to change into (i.e., open) your new `readability` directory.
- Execute `open readability.c` to create and open in the editor an empty file called `readability.c` in your `readability` directory.

Getting User Input

Let's first write some C code that just gets some text input from the user, and prints it back out. Specifically, write code in `readability.c` such that when the user runs the program, they are prompted with `"Text: "` to enter some text.

The behavior of the resulting program should be like the below.

```
$ ./readability
Text: In my younger and more vulnerable years my father gave me some advice that I've been turning over in my mind ever since.
In my younger and more vulnerable years my father gave me some advice that I've been turning over in my mind ever since.
```

Letters

Now that you've collected input from the user, let's begin to analyze that input by first counting the number of letters that show up in the text. Modify `readability.c` so that, instead of printing out the literal text itself, it instead prints out a count of the number of letters in the text.

The behavior of the resulting program should be like the below.

```
$ ./readability
Text: Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do: once or twice she
235 letter(s)
```

Letters can be any uppercase or lowercase alphabetic characters, but shouldn't include any punctuation, digits, or other symbols.

You can reference https://man.cs50.io/ (https://man.cs50.io/) for standard library functions that may help you here! You may also find that writing a separate function, like `count_letters`, may be useful to keep your code organized.

Words

The Coleman-Liau index cares not only about the number of letters, but also the number of words in a sentence. For the purpose of this problem, we'll consider any sequence of characters separated by a space to be a word (so a hyphenated word like `"sister-in-law"` should be considered one word, not three).

Modify `readability.c` so that, in addition to printing out the number of letters in the text, also prints out the number of words in the text.

You may assume that a sentence will not start or end with a space, and you may assume that a sentence will not have multiple spaces in a row.

The behavior of the resulting program should be like the below.

```
$ ./readability
Text: It was a bright cold day in April, and the clocks were striking thirteen. Winston Smith, his chin nuzzled into his breast i
250 letter(s)
55 word(s)
```

Sentences

The last piece of information that the Coleman-Liau formula cares about, in addition to the number of letters and words, is the number of sentences. Determining the number of sentences can be surprisingly trickly. You might first imagine that a sentence is just any sequence of characters that ends with a period, but of course sentences could end with an exclamation point or a question mark as well. But of course, not all periods necessarily mean the sentence is over. For instance, consider the sentence below.

```
Mr. and Mrs. Dursley, of number four Privet Drive, were proud to say that they were perfectly normal, thank you very much.
```

This is just a single sentence, but there are three periods! For this problem, we'll ask you to ignore that subtlety: you should consider any sequence of characters that ends with a `.` or a `!` or a `?` to be a sentence (so for the above "sentence", you may count that as three sentences). In practice, sentence boundary detection needs to be a little more intelligent to handle these cases, but we'll not worry about that for now.

Modify `readability.c` so that it also now prints out the number of sentences in the text.

The behavior of the resulting program should be like the below.

```
$ ./readability
Text: When he was nearly thirteen, my brother Jem got his arm badly broken at the elbow. When it healed, and Jem's fears of never
295 letter(s)
70 word(s)
3 sentence(s)
```

Putting it All Together

Now it's time to put all the pieces together! Recall that the Coleman-Liau index is computed using the formula:

```
index = 0.0588 * L - 0.296 * S - 15.8
```

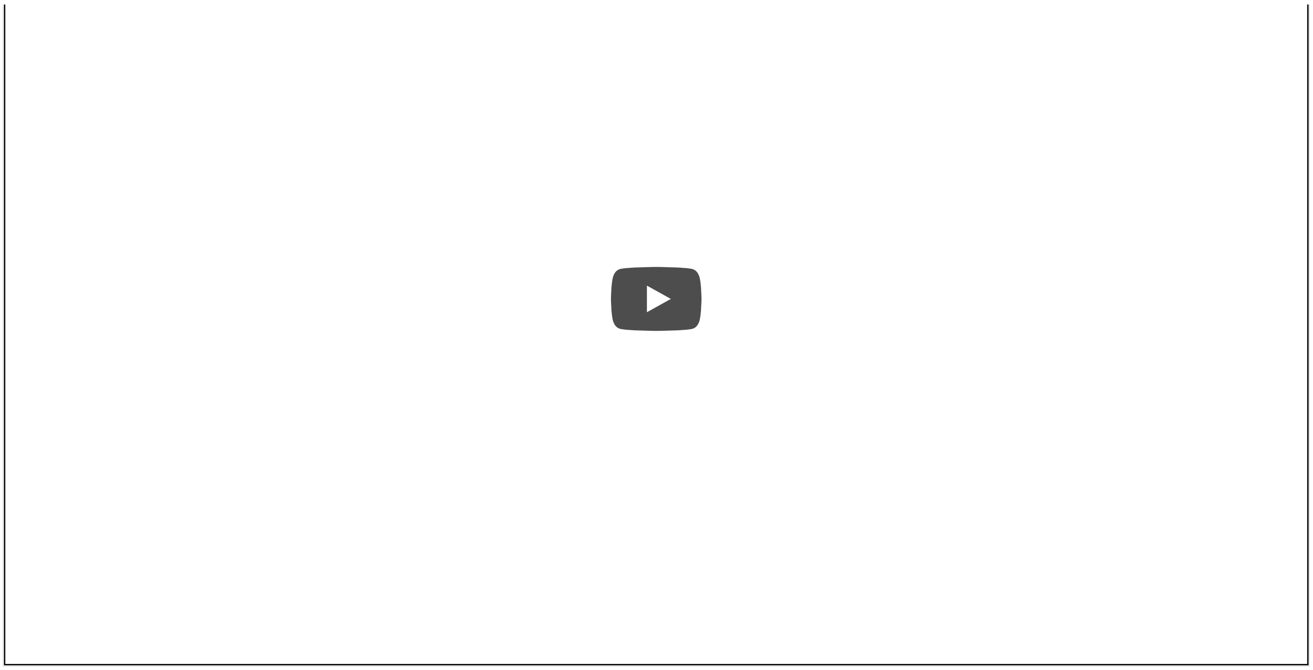
where `L` is the average number of letters per 100 words in the text, and `S` is the average number of sentences per 100 words in the text.

Modify `readability.c` so that instead of outputting the number of letters, words, and sentences, it instead outputs the grade level as given by the Coleman-Liau index (e.g. `"Grade 2"` or `"Grade 8"`). Be sure to round the resulting index number to the nearest whole number!

If the resulting index number is 16 or higher (equivalent to or greater than a senior undergraduate reading level), your program should output `"Grade 16+"` instead of giving the exact index number. If the index number is less than 1, your program should output `"Before Grade 1"`.

▶ **Hints**

Walkthrough



Note that while the walkthrough illustrates that words may be separated by more than one space, you may assume, per the specifications above, that no sentences will contain more than one space in a row.

How to Test Your Code

Try running your program on the following texts.

- `One fish. Two fish. Red fish. Blue fish.` (Before Grade 1)
- `Would you like them here or there? I would not like them here or there. I would not like them anywhere.` (Grade 2)
- `Congratulations! Today is your day. You're off to Great Places! You're off and away!` (Grade 3)
- `Harry Potter was a highly unusual boy in many ways. For one thing, he hated the summer holidays more than any other time of year. For another, he really wanted to do his homework, but was forced to do it in secret, in the dead of the night. And he also happened to be a wizard.` (Grade 5)
- `In my younger and more vulnerable years my father gave me some advice that I've been turning over in my mind ever since.` (Grade 7)
- `Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do: once or twice she had peeped into the book her sister was reading, but it had no pictures or conversations in it, "and what is the use of a book," thought Alice "without pictures or conversation?"` (Grade 8)
- `When he was nearly thirteen, my brother Jem got his arm badly broken at the elbow. When it healed, and Jem's fears of never being able to play football were assuaged, he was seldom self-conscious about his injury. His left arm was somewhat shorter than his right; when he stood or walked, the back of his hand was at right angles to his body, his thumb parallel to his thigh.` (Grade 8)
- `There are more things in Heaven and Earth, Horatio, than are dreamt of in your philosophy.` (Grade 9)
- `It was a bright cold day in April, and the clocks were striking thirteen. Winston Smith, his chin nuzzled into his breast in an effort to escape the vile wind, slipped quickly through the glass doors of Victory Mansions, though not quickly enough to prevent a swirl of gritty dust from entering along with him.` (Grade 10)
- `A large class of computational problems involve the determination of properties of graphs, digraphs, integers, arrays of integers, finite families of finite sets, boolean formulas and elements of other countable domains.` (Grade 16+)

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/readability
```

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

```
style50 readability.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/readability
```

This is CS50x

OpenCourseWare

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.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

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
```

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

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 (https://cdn.cs50.net/2018/fall/lectures/0/lecture0.pdf). 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**

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**

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
```

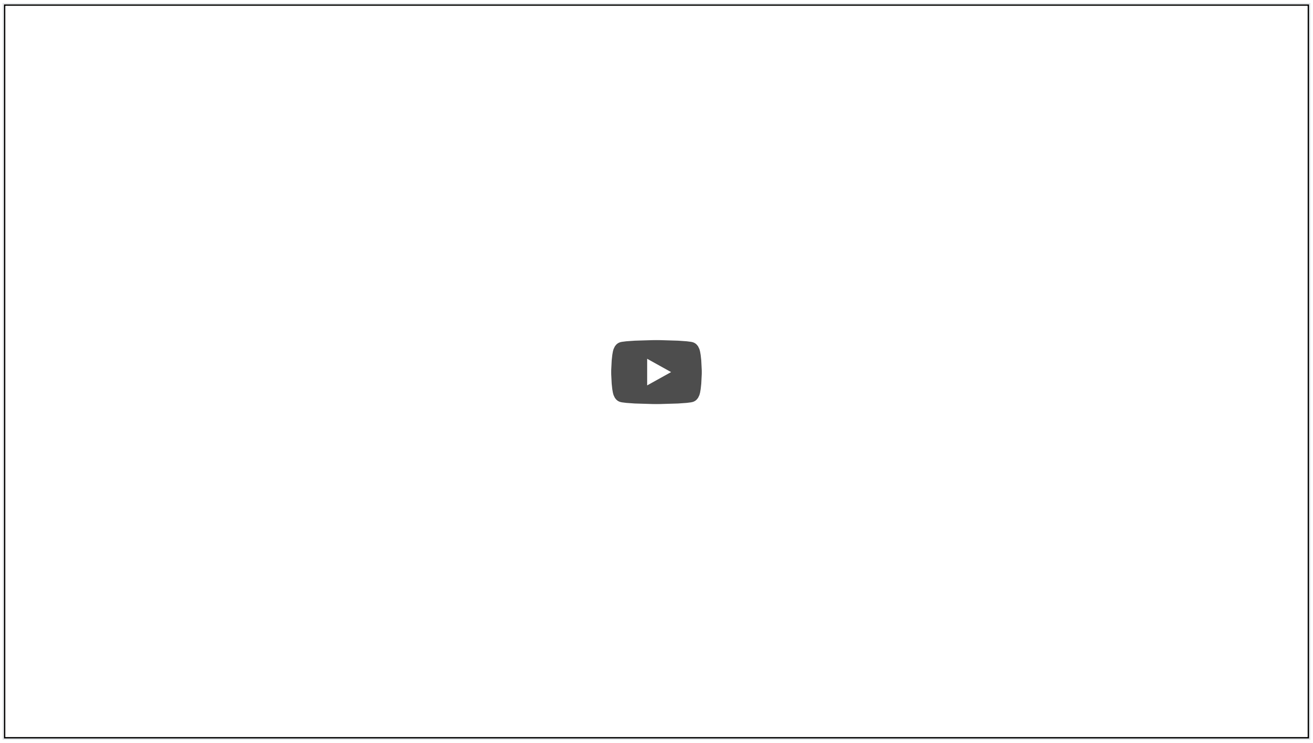
▶ **Hints**

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**

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
```

This is CS50x

OpenCourseWare

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.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Substitution

Implement a program that implements a substitution cipher, per the below.

```
$ ./substitution JTREKYAVOGDXPSNCUIZLFBMWHQ
plaintext:  HELLO
ciphertext: VKXXN
```

Background

In a substitution cipher, we “encrypt” (i.e., conceal in a reversible way) a message by replacing every letter with another letter. To do so, we use a *key*: in this case, a mapping of each of the letters of the alphabet to the letter it should correspond to when we encrypt it. To “decrypt” the message, the receiver of the message would need to know the key, so that they can reverse the process: translating the encrypt text (generally called *ciphertext*) back into the original message (generally called *plaintext*).

A key, for example, might be the string `NQXPOMAFTRHLZGECYJIUWSKDVB` . This 26-character key means that `A` (the first letter of the alphabet) should be converted into `N` (the first character of the key), `B` (the second letter of the alphabet) should be converted into `Q` (the second character of the key), and so forth.

A message like `HELLO` , then, would be encrypted as `FOLLE` , replacing each of the letters according to the mapping determined by the key.

Let’s write a program called `substitution` that enables you to encrypt messages using a substitution 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.

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

```
$ ./substitution YTNSHKVEFXRBAUQZCLWDMIPGJO
plaintext:  HELLO
ciphertext: EHBBQ
```

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

```
$ ./substitution VCHPRZGJNTLSKFBDQWAXEUYMOI
plaintext:  hello, world
ciphertext: jrssb, ybwsp
```

Notice that neither the comma nor the space were substituted by the cipher. Only substitute alphabetical characters! Notice, too, that the case of the original message has been preserved. Lowercase letters remain lowercase, and uppercase letters remain uppercase.

Whether the characters in the key itself are uppercase or lowercase doesn’t matter. A key of `VCHPRZGJNTLSKFBDQWAXEUYMOI` is functionally identical to a key of `vchprzgjntlskfbdqwaxeuymoi` (as is, for that matter, `VcHpRzGjNtLsKfBdQwAxEuYmOi`).

And what if a user doesn’t provide a valid key?

```
$ ./substitution ABC
Key must contain 26 characters.
```

Or really doesn't cooperate?

```
$ ./substitution
Usage: ./substitution key
```

Or even...

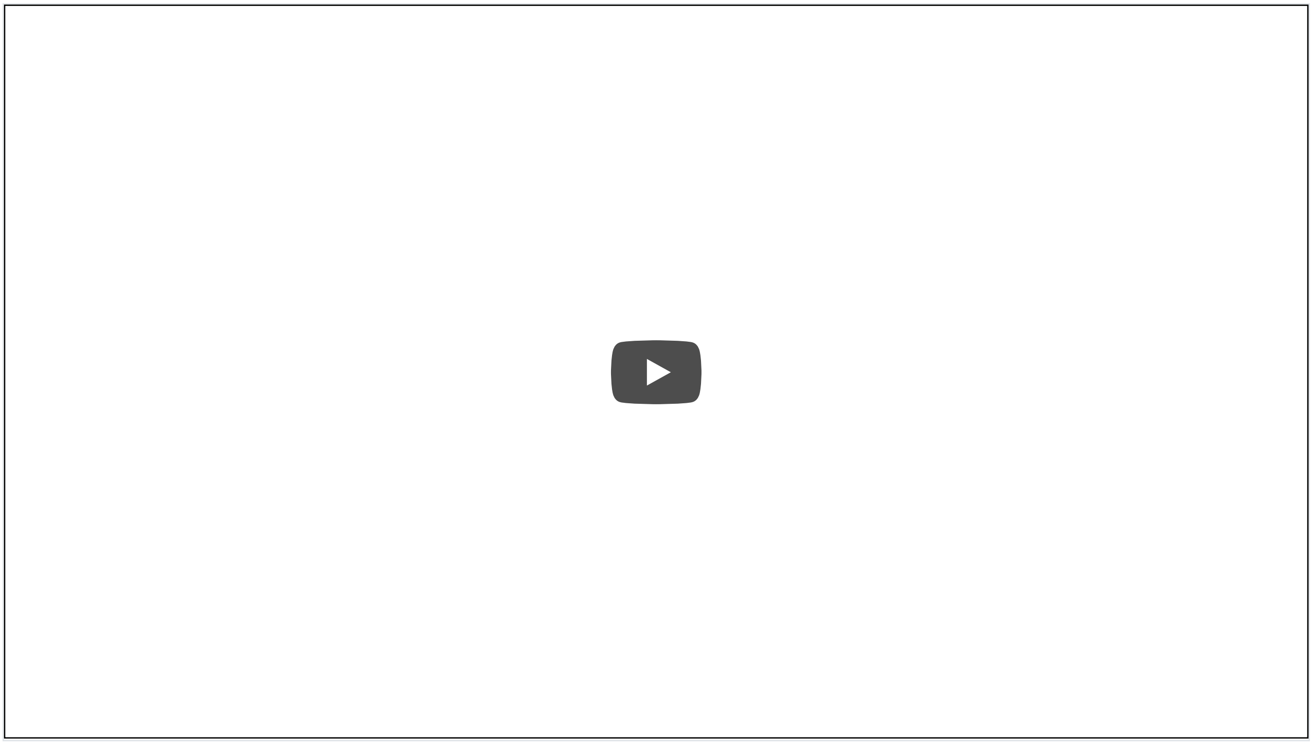
```
$ ./substitution 1 2 3
Usage: ./substitution key
```

Specification

Design and implement a program, `substitution`, that encrypts messages using a substitution cipher.

- Implement your program in a file called `substitution.c` in a directory called `substitution.
- Your program must accept a single command-line argument, the key to use for the substitution. The key itself should be case-insensitive, so whether any character in the key is uppercase or lowercase should not affect the behavior of your program.
- 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 the key is invalid (as by not containing 26 characters, containing any character that is not an alphabetic character, or not containing each letter exactly once), your program should print an error message of your choice (with `printf`) and return from `main` a value of `1` immediately.
- 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 substituted for the corresponding character in the ciphertext; non-alphabetical characters should be outputted unchanged.
- Your program must preserve case: capitalized letters must remain capitalized letters; lowercase letters must remain lowercase letters.
- After outputting ciphertext, you should print a newline. Your program should then exit by returning `0` from `main`.

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/substitution
```

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

```
style50 substitution.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/substitution
```