Problem Set 2: Crypto

due by noon on Thu 9/27

Per the directions at this document's end, submitting this problem set involves submitting source code via `submit50` as well as filling out a Web-based form, which may take a few minutes, so best not to wait until the very last minute, lest you spend a late day unnecessarily.

Goals.

- Better acquaint you with functions and libraries.
- Allow you to dabble in cryptography.

Recommended Reading.

- Sections 11 – 14 and 39 of `http://www.howstuffworks.com/c.htm`.
- Chapters 6, 7, 10, 17, 19, 21, 22, 30, and 32 of *Absolute Beginner's Guide to C*.
- Chapters 7, 8, and 10 of *Programming in C*.

Academic Honesty.

All work that you do toward fulfillment of this course's expectations must be your own unless collaboration is explicitly allowed in writing by the course's instructor. Collaboration in the completion of problem sets is not permitted unless otherwise stated by some problem set's specification.

Viewing or copying another individual's work (even if left by a printer, stored in an executable directory, or posted online) or lifting material from a book, website, or other source—even in part—and presenting it as your own constitutes academic dishonesty, as does showing or giving your work, even in part, to another student or soliciting the work of another individual. Similarly is dual submission academic dishonesty: you may not submit the same or similar work to this course that you have submitted or will submit to another. Nor may you provide or make available solutions to problem sets to individuals who take or may take this course in the future. Moreover, submission of any work that you intend to use outside of the course (*e.g.*, for a job) must be approved by the course's instructor or preceptor.

You are welcome to discuss the course's material with others in order to better understand it. You may even discuss problem sets with classmates, but you may not share code. In other words, you may communicate with classmates in English, but you may not communicate in, say, C. If in doubt as to the appropriateness of some discussion, contact the course's instructor or preceptor.

You may turn to the Web for instruction beyond the course's lectures and sections, for references, and for solutions to technical difficulties, but not for outright solutions to problems on problem sets or your own final project. However, failure to cite (as with comments) the origin of any code or technique that you do discover outside of the course's lectures and sections (even while respecting these constraints) and then integrate into your own work may be considered academic dishonesty.

All forms of academic dishonesty are dealt with harshly. If the course refers some matter to the Administrative Board and the outcome for some student is *Admonish*, *Probation*, *Requirement to Withdraw*, or *Recommendation to Dismiss*, the course reserves the right to impose local sanctions on top of that outcome for that student that may include, but not be limited to, a failing grade for work submitted or for the course itself.

Fine Print.

Your work on this problem set will be evaluated along four axes primarily.

Scope. To what extent does your code implement the features required by our specification?
Correctness. To what extent is your code consistent with our specifications and free of bugs?
Design. To what extent is your code written well (*i.e.*, clearly, efficiently, elegantly, and/or logically)?
Style. To what extent is your code readable (*i.e.*, commented and indented with variables aptly named)?

All students, whether taking the course Pass/Fail or for a letter grade, must ordinarily submit this and all other problem sets to be eligible for a passing grade (*i.e.*, Pass or A to D–) unless granted an exception in writing by the course's instructor or preceptor. No more than one late day may be spent on this, or any other, problem set.

Help!

☐ Surf on over to `cs50.net/discuss` and log in if prompted. Then take a look around!

Henceforth, consider CS50 Discuss *the* place to turn to anytime you have questions. Not only can you post questions of your own, you can also search for or browse answers to questions already asked by others. When posting, just be sure to provide as much detail as possible so that we can assist!

It is expected, of course, that you respect the course's policies on academic honesty. Posting snippets of code about which you have questions is generally fine. Posting entire programs, even if broken, is definitely not. If in doubt, simply flag your discussion as "private to staff," particularly if you need to show us most or all of your code. But the more questions you ask publicly, the more others will benefit as well!

Certainly don't hesitate to post a question because you think that it's "dumb." It is not!

Sanity Check.

☐ You should already have the CS50 Appliance installed, per Problem Set 1. But be sure that you have version 17 (and not 17a, 3-15, or even earlier). To check which version you have, look in the appliance's bottom-right corner, where you should see **17-#**, whereby **#** represents the "release" of version 17 that you have. If you see some number other than **17** (or no number at all), head to `cs50.net/appliance` for instructions on how to upgrade to version 17.

Once sure that you have version 17, update to the latest release by opening a Terminal window and executing the command below (which may take a minute or more).

```
sudo yum -y update
```

If the command appears to fail, restart the appliance, then try again. If it still appears to fail, restart your computer, then try again. If it *still* appears to fail, let us know at `cs50.net/discuss`, and we'll lend a hand!

Tips.

☐ Realize that the CS50 Appliance is a computer, albeit a virtual one. For better or for worse (mostly worse), computers don't like to be forcibly shut down or otherwise interrupted while in the middle of something. Do take care, then, not to quit VMware (or VirtualBox), shutdown your own computer, or even close your laptop's lid while the appliance is in the middle of something (*e.g.*, downloading or installing updates, submitting your work*, etc.*) Best to wait until the appliance isn't doing anything important, then shut it down (as via the green icon in the appliance's bottom-right corner). Bad things can happen, too, if your own computer runs out of disk space, so beware downloading big files on your own computer if you know you're low on disk space while the appliance is running.

☐ When running, the CS50 Appliance "borrows" some of your computer's own RAM and CPU cycles, which can slow down programs on your computer and vice versa. For maximum performance, try to launch VMware (or VirtualBox) and the appliance before launching other programs on your computer, and try to minimize the number of programs running on your computer while the appliance is running.

With that said, if you have lots of RAM (*e.g.*, 4GB) and lots of CPU cycles (*e.g.*, 2GHz), you might not need to give any of this a second thought!

A Section of Questions.

You're welcome to dive into these questions on your own, but know that they'll also be explored in section!

☐ Head to

`https://www.cs50.net/shorts/`

and watch the shorts on loops, scope, and the Caesar Cipher. Be sure you're reasonably comfortable answering the below when it comes time to submit this problem set's form!

- ☐ How does a while loop differ from a do-while loop? When is the latter particularly useful?
- ☐ What does `undeclared identifier` usually indicate if outputted by `clang`?
- ☐ Why is the Caesar Cipher not very secure?

And unrelated to those shorts!

- ☐ What's a function?
- ☐ Why bother writing functions when you can just copy and paste code as needed?

☐ Back when MySpace was cool, it was all the rage to TyPe LiKe ThIs. Maybe it still is? I'm not really sure. In any case, using the CS50 Appliance, CS50 Run (`cs50.net/run`), or CS50 Spaces (`cs50.net/spaces`), write a program that prompts the user for a message, and then outputs the message with its first letter capitalized, with all other letters in alternating case, as per the sample output below, wherein boldfaced text represents some user's input.[12] For simplicity, you may assume that the user will only input lowercase letters and spaces. Consider this problem an opportunity to practice; you won't be asked to submit this program.

```
jharvard@run.cs50.net (~): ./a.out
thanks for the add
ThAnKs FoR tHe AdD
```

[1] CS50 Spaces is essentially a fancier version of CS50 Run, complete with code-sharing capabilities and a chat room.

[2] Note that CS50 Run and CS50 Spaces have what appears to be a Terminal window, but you can't actually type commands (*e.g.*, `clang`) in it. That black panel is instead there so that you can see what CS50 Run and CS50 Spaces are doing underneath the hood when you click the ▸ button at top-left. However, if your program prompts for user input (as with `GetString`), you can type user input in that black panel.

☐ Recall the following song from childhood. (Mine, at least.)

```
This old man, he played one
He played knick-knack on my thumb
Knick-knack paddywhack, give your dog a bone
This old man came rolling home

This old man, he played two
He played knick-knack on my shoe
Knick-knack paddywhack, give your dog a bone
This old man came rolling home

This old man, he played three
He played knick-knack on my knee
Knick-knack paddywhack, give your dog a bone
This old man came rolling home

This old man, he played four
He played knick-knack on my door
Knick-knack paddywhack, give your dog a bone
This old man came rolling home

This old man, he played five
He played knick-knack on my hive
Knick-knack paddywhack, give your dog a bone
This old man came rolling home

This old man, he played six
He played knick-knack on my sticks
Knick-knack paddywhack, give your dog a bone
This old man came rolling home

This old man, he played seven
He played knick-knack up in heaven
Knick-knack paddywhack, give your dog a bone
This old man came rolling home

This old man, he played eight
He played knick-knack on my gate
Knick-knack paddywhack, give your dog a bone
This old man came rolling home

This old man, he played nine
He played knick-knack on my spine
Knick-knack paddywhack, give your dog a bone
This old man came rolling home

This old man, he played ten
He played knick-knack once again
Knick-knack paddywhack, give your dog a bone
This old man came rolling home
```

Oddly enough, the lyrics to this song don't seem to be standardized. In fact, if you'd like to be overwhelmed with variations, search for some with Google. And then stop procrastinating.

Using the CS50 Appliance, CS50 Run (`cs50.net/run`), or CS50 Spaces (`cs50.net/spaces`), write a program that verbatim, the above version of "This Old Man." Your version should be capitalized and spelled exactly as ours is.

Notice, though, the repetition in this song's verses. Perhaps you could leverage a loop that iterates from 1 to 10 (or 0 to 9) to generate them? Though they do vary somewhat, so you might need some conditions? Seems you could even implement a couple of functions that take, as input, an integer and return, as output, a string? Or maybe you could store all those strings in arrays? Hm. So many possibilities! Consider this problem an opportunity to practice; you won't be asked to submit this program.

Getting Started.

☐ Alright, here we go again!

Open a terminal window if not open already (whether by opening gedit via **Menu > Programming > gedit** or by opening Terminal itself via **Menu > Programming > Terminal**). Then execute

```
mkdir ~/Dropbox/pset2
```

at your prompt in order to make a directory called `pset2` in your `Dropbox` directory. Take care not to overlook the space between `mkdir` and `~/Dropbox/pset2` or any other character for that matter! Recall that `~` denotes your home directory, `~/Dropbox` denotes a directory called `Dropbox` therein, and `~/Dropbox/pset2` denotes a directory called `pset2` within `~/Dropbox`.

Now execute

```
cd ~/Dropbox/pset2
```

to move yourself into (*i.e.*, open) that directory. Your prompt should now resemble the below.

```
jharvard@appliance (~/Dropbox/pset2):
```

If not, retrace your steps and see if you can determine where you went wrong. You can actually execute

```
history
```

at the prompt to see your last several commands in chronological order if you'd like to do some sleuthing. You can also scroll through the same one line at a time by hitting your keyboard's up and down arrows; hit Enter to re-execute any command that you'd like. If still unsure how to fix, remember that `cs50.net/discuss` is your friend!

All of the work that you do for this problem set must ultimately reside in your `pset2` directory for submission.

Hail, Caesar!

☐ Recall from David DiCiurcio's short that Caesar's cipher encrypts (*i.e.*, scrambles in a reversible way) messages by "rotating" each letter by *k* positions, wrapping around from 'Z' to 'A' as needed:

`http://en.wikipedia.org/wiki/Caesar_cipher`

In other words, 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:

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

This formula perhaps makes the cipher seem more complicated than it is, but it's really just a nice way of expressing the algorithm precisely and concisely. And computer scientists love precision and, er, concision.[3]

For example, suppose that the secret key, *k*, is 13 and that the plaintext, *p*, is "Be sure to drink your Ovaltine!" Let's encrypt that *p* with that *k* in order to get the ciphertext, *c*, by rotating each of the letters in *p* by 13 places, whereby:

```
Be sure to drink your Ovaltine!
```

becomes:

```
Or fher gb qevax lbhe Binygvar!
```

We've deliberately printed the above in a monospaced font so that all of the letters line up nicely. Notice how `O` (the first letter in the ciphertext) is 13 letters away from `B` (the first letter in the plaintext). Similarly is `r` (the second letter in the ciphertext) 13 letters away from `e` (the second letter in the plaintext). Meanwhile, `f` (the third letter in the ciphertext) is 13 letters away from `s` (the third letter in the plaintext), though we had to wrap around from `z` to `a` to get there. And so on. Not the most secure cipher, to be sure, but fun to implement!

Incidentally, a Caesar cipher with a key of 13 is generally called ROT13:

`http://en.wikipedia.org/wiki/ROT13`

In the real world, though, it's probably best to use ROT26, which is believed to be twice as secure.[4]

Anyhow, your next goal is to write, in `caesar.c`, a program that encrypts messages using Caesar's cipher. Your program must accept a single command-line argument: a non-negative integer. Let's call it *k*. If your program is executed without any command-line arguments or with

[3] Okay, fine, conciseness. So much for parallelism.

[4] `http://www.urbandictionary.com/define.php?term=ROT26`

more than one command-line argument, your program should yell at the user and return a value of `1` (which tends to signify an error) immediately as via the statement below:

```
return 1;
```

Otherwise, your program must proceed to prompt the user for a string of plaintext and then output that text with each alphabetical character "rotated" by *k* positions; non-alphabetical characters should be outputted unchanged. After outputting this ciphertext, your program should exit, with `main` returning `0`.

Although there exist only 26 letters in the English alphabet, you may 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`. Now, 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; `A` should become `B`, since 27 modulo 26 is 1, as a computer scientists might say. In other words, values like *k* = 1 and *k* = 27 are effectively equivalent.

Your program must preserve case: capitalized letters, though rotated, must remain capitalized letters; lowercase letters, though rotated, must remain lowercase letters.

Where to begin? Well, this program needs to accept a command-line argument, *k*, so this time 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 it's time for some yelling.

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`, 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. Ah, much better. Incidentally, you can assume that the user will only type integers at the prompt. You don't have to worry about them typing, say, `foo`, just to be difficult; `atoi` will just return `0` in such cases. Incidentally, you'll need to `#include` a header file other than `cs50.h` and `stdio.h` in order to use of `atoi` without getting yelled at by `clang`. We leave it to you to figure out which one![5]

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 `GetString` can help you with that.

Once you have both *k* and some plaintext, it's time to encrypt the latter with the former. Recall that you can iterate over the `char`s 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 `char`s. 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! For Caesar!

Incidentally, you'll need to `#include` yet another header file in order to use `strlen`.[6]

So that we can automate some tests of your code, your program must behave per the below. Assumed that the boldfaced text is what some user has typed.

```
jharvard@appliance (~/Dropbox/pset2): ./caesar 13
Be sure to drink your Ovaltine!
Or fher gb qevax lbhe Binygvar!
```

[5] https://www.cs50.net/resources/cppreference.com/stdstring/atoi.html
[6] https://www.cs50.net/resources/cppreference.com/stdstring/strlen.html

Besides `atoi`, you might find some handy functions documented at:

`http://www.cs50.net/resources/cppreference.com/stdstring/`

For instance, `isdigit` sounds interesting. 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://asciitable.com/`, which reveals the ASCII codes for more than just alphabetical characters, just in case you find yourself printing some characters accidentally.

If you'd like to play with the staff's own implementation of `caesar` in the appliance, you may execute the below.

`~cs50/pset2/caesar`

☐ `uggc://jjj.lbhghor.pbz/jngpu?i=bUt5FWLEUN0`

Parlez-vous français?

☐ Well that last cipher was hardly secure. Fortunately, per Week 3's first lecture, there's a more sophisticated algorithm out there: Vigenère's. It is, of course, French:[7]

`http://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher`

Vigenère's cipher improves upon Caesar's by encrypting messages using a sequence of keys (or, put another way, a keyword). In other words, if p is some plaintext and k is a keyword (*i.e.*, an alphbetical string, whereby `A` and `a` represent 0, while `Z` and `z` represent 25), then each letter, c_i, in the ciphertext, c, is computed as:

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

Note this cipher's use of k_j as opposed to just k. And recall that, if k is shorter than p, then the letters in k must be reused cyclically as many times as it takes to encrypt p.

Your final challenge this week is to write, in `vigenere.c`, a program that encrypts messages using Vigenère's cipher. This program must accept a single command-line argument: a keyword, k, composed entirely of alphabetical characters. If your program is executed without any command-line arguments, with more than one command-line argument, or with one command-line argument that contains any non-alphabetical character, your program should complain and exit immediately, with `main` returning `1` (thereby signifying an error that our own tests can detect). Otherwise, your program must proceed to prompt the user for a string of plaintext, p, which it must then encrypt according to Vigenère's cipher with k, ultimately printing the result and exiting, with `main` returning `0`.

[7] Do not be mislead by the article's discussion of a *tabula recta*. Each c_i can be computed with relatively simple arithmetic! You do not need a two-dimensional array.

As for the characters in *k*, you must treat `A` and a as 0, `B` and `b` as 1, ... , and `Z` and `z` as 25. In addition, your program must only apply Vigenère's cipher to a character in *p* if that character is a letter. All other characters (numbers, symbols, spaces, punctuation marks, *etc.*) must be outputted unchanged. Moreover, if your code is about to apply the j^{th} character of *k* to the i^{th} character of *p*, but the latter proves to be a non-alphabetical character, you must wait to apply that j^{th} character of *k* to the next alphabetical character in *p*; you must not yet advance to the next character in *k*. Finally, your program must preserve the case of each letter in *p*.

Not sure where to begin? As luck would have it, this program's pretty similar to `caesar`! Only this time, you need to decide which character in *k* to use as you iterate from character to character in *p*.

So that we can automate some tests of your code, your program must behave per the below; highlighted in bold are some sample inputs.

```
jharvard@appliance (~/Dropbox/pset2): ./vigenere bacon
Meet me at the park at eleven am
Negh zf av huf pcfx bt gzrwep oz
```

How to test your program, besides predicting what it should output, given some input? Well, recall that we're nice people. And so we've written a program called `devigenere` that also takes one and only one command-line argument (a keyword) but whose job is to take ciphertext as input and produce plaintext as output.

To use our program, execute

```
~cs50/pset2/devigenere k
```

at your prompt, where `k` is some keyword. Presumably you'll want to paste your program's output as input to our program; be sure, of course, to use the same key. Note that you do not need to implement `devigenere` yourself, only `vigenere`.

If you'd like to play with the staff's own implementation of `vigenere` in the appliance, you may execute the below.

```
~cs50/pset2/vigenere
```

How to Submit.

In order to submit this problem set, you must first execute a command in the appliance and then submit a (brief) form online.

☐ Open a terminal window (as via **Menu > Programming > Terminal** or within gedit) then execute

```
sudo yum -y update
```

to ensure you have the latest release of the appliance. Then execute:

```
cd ~/Dropbox/pset2
```

And then execute:

```
ls
```

At a minimum, you should see `caesar.c` and `vigenere.c`. If not, odds are you skipped some step(s) earlier! If you do see those files, you are ready to submit your source code to us. Execute:

```
submit50 ~/Dropbox/pset2
```

and follow the on-screen instructions. That command will essentially upload your entire `~/Dropbox/pset2` directory to CS50's servers, where your TF will be able to access it. The command will inform you whether your submission was successful or not. And you may inspect your submission at `cs50.net/submit`.

You may re-submit as many times as you'd like; we'll grade your most recent submission. But take care not to submit after the problem set's deadline, lest you spend a late day unnecessarily or risk rejection entirely.

If you run into any trouble at all, let us know via `cs50.net/discuss` and we'll try to assist! Just take care to seek help well before the problem set's deadline, as we can't always reply right away!

☐ Head to the URL below where a short form awaits:

```
https://www.cs50.net/psets/2/
```

Once you have submitted that form (as well as your source code), you are done!

This was Problem Set 2.