# Problem Set 2: Crypto

This is CS50. Harvard University. Fall 2014.

## Table of Contents

## Objectives

- Become better acquainted with functions and libraries.
- Dabble in cryptography.

## Recommended Reading

- Pages 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

This course's philosophy on academic honesty is best stated as "be reasonable." The course recognizes that interactions with classmates and others can facilitate mastery of the course’s material. However, there remains a line between enlisting the help of another and submitting the work of another. This policy characterizes both sides of that line.

The essence of all work that you submit to this course must be your own. Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you. Generally speaking, when asking for help, you may show your code to others, but you may not view theirs, so long as you and they respect this policy’s other constraints. Collaboration on the course’s final project is permitted to the extent prescribed by its specification.

Below are rules of thumb that (inexhaustively) characterize acts that the course considers reasonable and not reasonable. If in doubt as to whether some act is reasonable, do not commit it until you solicit and receive approval in writing from the course’s heads. Acts considered not reasonable by the course are handled harshly.

## Reasonable

- Communicating with classmates about problem sets' problems in English (or some other spoken language).
- Discussing the course’s material with others in order to understand it better.
- Helping a classmate identify a bug in his or her code at Office Hours, elsewhere, or even online, as by viewing, compiling, or running his or her code, even on your own computer.
- Incorporating snippets of code that you find online or elsewhere into your own code, provided that those snippets are not themselves solutions to assigned problems and that you cite the snippets' origins.
- Reviewing past semesters' quizzes and solutions thereto.
- Sending or showing code that you’ve written to someone, possibly a classmate, so that he or she might help you identify and fix a bug.
- Sharing snippets of your own code online so that others might help you identify and fix a bug.
- Turning to the web or elsewhere for instruction beyond the course’s own, for references, and for solutions to technical difficulties, but not for outright solutions to problem set’s problems or your own final project.

- Whiteboarding solutions to problem sets with others using diagrams or pseudocode but not actual code.
- Working with (and even paying) a tutor to help you with the course, provided the tutor does not do your work for you.

## Not Reasonable

- Accessing a solution in CS50 Vault to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem set's problem before (re-)submitting your own.
- Decompiling, deobfuscating, or disassembling the staff's solutions to problem sets.
- Failing to cite (as with comments) the origins of code or techniques that you discover outside of the course's own lessons and integrate into your own work, even while respecting this policy's other constraints.
- Giving or showing to a classmate a solution to a problem set's problem when it is he or she, and not you, who is struggling to solve it.
- Looking at another individual's work during a quiz.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problem sets to individuals who might take this course in the future.
- Searching for, soliciting, or viewing a quiz's questions or answers prior to taking the quiz.
- Searching for or soliciting outright solutions to problem sets online or elsewhere.
- Splitting a problem set's workload with another individual and combining your work.
- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.
- Submitting the same or similar work to this course that you have submitted or will submit to another.
- Submitting work to this course that you intend to use outside of the course (e.g., for a job) without prior approval from the course's heads.
- Using resources during a quiz beyond those explicitly allowed in the quiz's instructions.

- Viewing another's solution to a problem set's problem and basing your own solution on it.

## Getting Ready

First, curl up with these shorts on loops, functions, Caesar's cipher, and command-line arguments:

https://www.youtube.com/watch?v=HHmiHx7GGLE
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 `make` (or, really, `clang`)?
- Why is Caesar's cipher not very secure?
- What's a function?
- Why bother writing functions when you can just copy and paste code as needed?

Next, take a self-paced tour through Week 2's examples, the source code for which can be found at http://cdn.cs50.net/2014/fall/lectures/2/m/src2m/ and http://cdn.cs50.net/2014/fall/lectures/2/w/src2w/, fast-forwarding through any programs with which you're already comfortable:

https://www.youtube.com/watch?v=9zoRoz8Pq4E&list=PLhQjrBD2T380bVx_CQ2EXtlqZh0frwOsn
Finally, take a closer look at command-line arguments by way of these additional examples from Week 3, the source code for which can be found at http://cdn.cs50.net/2014/fall/lectures/3/m/src3m/:

https://www.youtube.com/watch?v=1VbHJz2L6dM

## Getting Started

Alright, here we go again!

Open a terminal window if not open already (as via **Menu > Accessories > gedit** or via **Menu > Accessories > Terminal**). Then execute

```
update50
```

to make sure your appliance is up-to-date. In general, if you run into errors like "No such file or directory" with the staff's solutions, best to re-run `update50`, just to make sure you have the latest updates.

Next, 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! Keep in mind 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.

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

## Initializing

Alright, let's get more comfortable with `string`.

Write, in a file called `initials.c`, a program that prompts a user for their name (using `GetString` to obtain their name as a `string`) and then outputs their initials in uppercase with no spaces or periods, followed by a newline (`\n`) and nothing more. You may assume that the user's input will contain only letters (uppercase and/or lowercase) plus single spaces between words. Folks like `Joseph Gordon-Levitt`, `Conan O'Brien`, and `David J. Malan` won't be using your program.

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

```
jharvard@appliance (~/Dropbox/pset2): ./initials
Zamyla Chan
ZC
jharvard@appliance (~/Dropbox/pset2): ./initials
robert thomas bowden
RTB
```

If you'd like to check the correctness of your program with `check50`, you may execute the below.

```
check50 2014.fall.pset2.initials initials.c
```

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

```
~cs50/pset2/initials
```

## 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 (cf. 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.

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 (cf. 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[1].

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* for the sake of discussion. If your program is executed without any command-line arguments or with more than one command-line argument, your

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

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`, as via the statement below:

```
return 0;
```

If you don't explicitly return an `int` from within `main`, `0` is actually returned for you automatically. (Indeed, per its "return type," `main` does need to return an `int`. But more on that another time.) Now that you're returning `1` explicitly to signify errors, it's best to return `0` (by convention) explicitly to signify success. Whereas `0` generally represents success, any non-`0` `int` generally represents an error. That way, you can represent (gasp) upwards of four billion errors (since an `int` is generally 32 bits)!

Anyhow, even though 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`[2], 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 (even though the staff's solution does catch such); `atoi` will just return `0` in such cases.

[2] https://reference.cs50.net/stdlib.h/atoi

Because `atoi` is declared in `stdlib.h`, you'll want to `#include` that header file atop your own code. (Technically, your code will compile without it there, since we already `#include` it in `cs50.h`. But best not to trust another library to `#include` header files you know you need.)

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 characters 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 chars. 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`[3].

And Zamyla has some tips for you as well:

https://www.youtube.com/watch?v=V6IDxl-3WAA
So that we can automate some tests of your code, your program must behave per the below. Assumed that the underlined 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!
```

Besides `atoi`, you might find some handy functions documented at https://reference.cs50.net/ under **ctype.h** and **stdlib.h**. For instance, `isdigit` sounds

[3] https://reference.cs50.net/string.h/strlen

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 check the correctness of your program with `check50`, you may execute the below.

```
check50 2014.fall.pset2.caesar caesar.c
```

And 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
```

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

## Parlez-vous français?

Well that last cipher was hardly secure. Fortunately, per Nate’s short on Vigenère’s cipher[4], there’s a more sophisticated algorithm out there. Suffice it to say it’s French, per http://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher. Though 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.

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 $p$.

[4] https://cs50.harvard.edu/shorts/3

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`.

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*.

And here's Zamyla again with some tips:

https://www.youtube.com/watch?v=Uma2HZMPm2M
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 check the correctness of your program with `check50`, you may execute the below.

```
check50 2014.fall.pset2.vigenere vigenere.c
```

And 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

### Step 1 of 2

- When ready to submit, open up Chrome *inside* of the appliance (not on your own computer) and visit cs50.edx.org/submit[5], logging in if prompted.
- Click **Submit** toward the window's top-left corner.
- Under **Problem Set 2** on the screen that appears, click **Upload New Submission**.
- On the screen that appears, click **Add files…**. A window entitled **Open Files** should appear.
- Navigate your way to `initials.c`, as by clicking **jharvard**, then double-clicking **Dropbox**, then double-clicking **pset2**. Once you find `initials.c`, click it once to select it, then click **Open**.
- Click **Add files…** again, and a window entitled **Open Files** should appear again.

[5] http://cs50.edx.org/submit

- Navigate your way to `caesar.c` as before. Click it once to select it, then click **Open**.
- Navigate your way to `vigenere.c` as before. Click it once to select it, then click **Open**.
- Click **Start upload** to upload all of your files at once to CS50's servers.
- On the screen that appears, you should see a window with **No File Selected**. If you move your mouse toward the window's lefthand side, you should see a list of the files you uploaded. Click each to confirm the contents of each. (No need to click any other buttons or icons.) If confident that you submitted the files you intended, consider your source code submitted! If you'd like to re-submit different (or modified) files, simply return to cs50.edx.org/submit[6] and repeat these steps. You may re-submit as many times as you'd like; we'll grade your most recent submission, so long as it's before the deadline.

## Step 2 of 2

Head to http://cs50.edx.org/2015/psets/2/ where a short form awaits. Once you have submitted that form (as well as your source code), you are done!

This was Problem Set 2.

[6] http://cs50.edx.org/submit