Ceboyrz Frg 2: Pelcgb

qhr ol 7:00cz ba Sev 9/25

Or fher gung lbhe pbqr vf gubebhtuyl pbzzragrq
gb fhpu na rkgrag gung yvarf' shapgvbanyvgl vf nccnerag sebz pbzzragf nybar.

Tbnyf.

- Orggre npdhnvag lbh jvgu shapgvbaf naq yvoenevrf.
- Nyybj lbh gb qnooyr va pelcgbtencul.

Erpbzzraqrq Ernqvat.

- Frpgvbaf 11 – 14 naq 39 bs `uggc://jjj.ubjfghssjbexf.pbz/p.ugz`.
- Puncgref 6, 7, 10, 17, 19, 21, 22, 30, naq 32 bs *Nofbyhgr Ortvaare'f Thvqr gb P*.
- Puncgref 7, 8, naq 10 bs *Cebtenzzvat va P*.

;-)

Problem Set 2: Crypto

due by 7:00pm on Fri 9/25

Be sure that your code is thoroughly commented
to such an extent that lines' functionality is apparent from comments alone.

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 (*e.g.*, by some problem set or the final project). Viewing or copying another individual's work (even if left by a printer, stored in an executable directory, or accidentally shared in the course's virtual terminal room) 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.

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 staff.

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 staff.

You may even 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.

Grades.

Your work on this problem set will be evaluated along three primary axes.

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)?

Getting Started.

☐ For this problem set, we're again going to have you use `nice.fas.harvard.edu`. SSH to `nice.fas.harvard.edu`, create a directory called `pset2` in your `~/cs50/` directory, and then navigate your way to that directory. (Remember how?) All of the work that you do for this problem set must ultimately reside in this directory for submission.

Your prompt should now resemble the below.

```
username@nice (~/cs50/pset2):
```

☐ Surf on over to the course's website and follow the link to the course's **Bulletin Board**; you may be prompted to log in. Take a look around!

Henceforth, consider the course's bulletin board *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.

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 email `help@cs50.net` with your question instead, 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!

Lest you feel uncomfortable posting, know that students' posts to the course's bulletin board are anonymized. Only the staff, not fellow students, will know who you are!

Let's Warm Up with a Song.

☐ 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.

Your first challenge this week is to write, in `oldman.c`, a program that prints, verbatim, the above version of "This Old Man."

Notice, though, the repetition in this song's verses. Clearly your program should make use of some sort of loop to generate repeated lyrics. And perhaps there's some way to store all those numbers in an array for easy access. Or perhaps you'd prefer to use some conditions, perhaps `switch`. Hmm...

There are, as you may be increasingly aware, many ways to solve problems like this one. Pick an approach, implement it, test it, then go back and see if you can improve it before moving on! Ultimately, not only should your code be correct (*i.e.*, work right), it should also manifest good design and good style.

Style is easy. Ask yourself questions like these: Is my code well commented, without being excessively so? Is my code "pretty-printed" (*i.e.*, consistently indented and no wider than 80 characters across)? Are my variables aptly named?

As for design, ask yourself questions like these: Is my code straightforward to read? Am I wasting CPU cycles unnecessarily? Is my code more complicated than it need be to get this job done?

Consider yourself done with this problem when you feel there's no more room for improvement!

If you'd like to play with the staff's own implementation of `oldman` on `nice.fas.harvard.edu`, you may execute the below.

```
~cs50/pub/solutions/pset2/oldman
```

Hail, Caesar!

☐ Recall that Caesar's cipher encrypts 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 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.[1]

Your next challenge this week 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, *k*. 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 characters other than '0' through '9', your program should complain and exit immediately, with `main` returning any non-zero `int` (thereby signifying an error that our own tests can detect). 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

[1] Okay, fine, conciseness. Ruin the parallelism why don't you.

than 2^{31}. 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. 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.

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

```
username@nice (~/cs50/pset2): caesar 13
Be sure to drink your Ovaltine!
Or fher gb qevax lbhe Binygvar!
```

So that you don't reinvent the wheel, do scour

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

for any functions that might be of assistance to you. As will be often the case, there is more than one way to solve the problem at hand. But know that `isdigit` and `atoi` might prove particularly good friends.[2] And best not to forget about an operator like `%`. You might also want to check out `http://asciitable.com/` itself.

If you'd like to play with the staff's own implementation of `caesar` on `nice.fas.harvard.edu`, you may execute the below.

`~cs50/pub/solutions/pset2/caesar`

[2] Realize that, even if a user types a number as a command-line argument, it gets passed to `main` as a `string` (in the array generally called `argv`)! So, odds are, you'll want to convert *k* from a `string` to an `int` somehow.

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. Oh yes, it's French:[3]

`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 any non-zero `int` (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*.

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

```
username@nice (~/cs50/pset2): vigenere FOOBAR
HELLO, WORLD
MSZMO, NTFZE
```

[3] 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.

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/pub/tests/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.

If you'd like to play with the staff's own implementation of `vigenere` on `nice.fas.harvard.edu`, you may execute the below.

```
~cs50/pub/solutions/pset2/vigenere
```

Submitting Your Work.

☐ Ensure that your work is in `~/cs50/pset2/`. Submit your work by executing the command below.

```
cs50submit pset2
```

Thereafter, follow any on-screen instructions until you receive visual confirmation of your work's successful submission. You will also receive a "receipt" via email to your FAS account, which you should retain until term's end. You may re-submit as many times as you'd like; each resubmission will overwrite any previous submission. But take care not to re-submit after the problem set's deadline, as only your latest submission's timestamp is retained.