Ceboyrz Frg 2: Pelcgb

qhr ol 7:00cz ba Sev 9/24

Cre gur qverpgvbaf ng guvf qbphzrag'f raq, fhozvggvat guvf ceboyrz frg vaibyirf fhozvggvat fbhepr pbqr ba `pybhq.pf50.arg` nf jryy nf svyyvat bhg n Jro-onfrq sbez (gur ynggre bs juvpu jvyy or ninvynoyr nsgre yrpgher ba Jrq 9/22), juvpu znl gnxr n srj zvahgrf, fb orfg abg gb jnvg hagvy gur irel ynfg zvahgr, yrfg lbh fcraq n yngr qnl haarprffnevyl.

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 9/24

Per the directions at this document's end, submitting this problem set involves submitting source code on `cloud.cs50.net` as well as filling out a Web-based form (the latter of which will be available after lecture on Wed 9/22), which may take a few minutes, so best not to wait until the very last minute, lest you spend a late day unnecessarily.

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 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 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 course's instructor.

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 *Warn*, *Admonish*, or *Disciplinary Probation*, 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.

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)?

Help!

☐ Surf on over to

```
http://help.cs50.net/
```

and log in if prompted. Then take a look around!

Henceforth, consider `help.cs50.net` *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 flag your discussion as "private" or 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. Certainly don't hesitate to post a question because you think that it's "dumb." It is not!

Getting Started.

☐ Alright, here we go! SSH to `cloud.cs50.net`, create a directory called `pset2` in your home directory, and then navigate your way to that directory. Remember how? No? No problem. Once you've SSH'd to `cloud.cs50.net`, execute

```
mkdir ~/pset2
```

in order to make a directory called `pset2` in your home directory. Then execute

```
cd ~/pset2
```

or just

```
cd pset2
```

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

```
username@cloud (~/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 `help.cs50.net` is your new friend!

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

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." 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!

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. Rest assured that there are many ways to implement this song.

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!

Your program will need, at least, a `main` function. It's up to you to decide whether or not you want to write one or more additional functions that `main` calls. Odds are you'll find it simplest to compile your program with:

```
make oldman
```

You can then run it with:

```
./oldman
```

As this usage implies, `oldman` need not accept any command-line arguments. And so it suffices to declare `main` with

```
int
main(void)
```

without any mention of `argc` or `argv`. If you'd like to play with the staff's own implementation of `oldman` on `cloud.cs50.net`, you may execute the below.

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

Alright, off you go!

Hail, Caesar!

☐ Recall from the end of Week 2 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 ith 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.[1]

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:

```
plaintext:  Be sure to drink your Ovaltine!
ciphertext: 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!

[1] Okay, fine, conciseness. Way to ruin the parallelism.

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.[2]

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, *k*. If your program is executed without any command-line arguments or with 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, char *argv[])
```

Recall that `argv` is an "array" of strings (which are otherwise known as "char stars" for reasons we'll soon see). In fact, because `string` is just a synonym for `char *`, thanks to the CS50 Library, you could actually declare `main` with

```
int
main(int argc, string argv[])
```

if you find that syntax more clear. Either way, 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

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

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 `gcc`. We leave it to you to figure out which one![3]

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.

[3] `man atoi`

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 characters. 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* = 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`.[4]

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.

```
username@cloud (~/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:

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

For instance, `isdigit` sounds interesting. And, with regard to wrapping around from 'Z' to 'A', don't forget about `%`. 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` on `cloud.cs50.net`, you may execute the below.

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

☐ Fnnc ina.

[4] `man strlen`

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:[5]

`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`.

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 jth character of *k* to the ith 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*.

[5] 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.

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@cloud (~/pset2): ./vigenere FOOBAR
HELLO, WORLD
MSZMO, NTFZE
```

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 `cloud.cs50.net`, you may execute the below.

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

How to Submit.

In order to submit this problem set, you must first execute a command on `cloud.cs50.net` and then submit a (brief) form online; the latter will be posted after lecture on Wed 9/22.

☐ SSH to `cloud.cs50.net`, if not already there, and then execute:

```
cd ~/pset2
```

If informed that there is "no such file or directory," odds are you skipped some step(s) earlier! Not a problem, we'll help you fix in a moment. If the command indeed works, proceed to execute this command next:

```
ls
```

At a minimum, you should see `oldman.c`, `caesar.c`, and `vigenere.c`. If not, odds are you skipped some more steps earlier! If you do see those files, you're ready to submit your source code to us:[6]

```
~cs50/pub/bin/submit pset2
```

That command will essentially copy your entire `~/pset2` directory into CS50's own account, where your TF will be able to access it. You'll know that the command worked if you are informed that your "work HAS been submitted." If you instead encounter an error that doesn't appear to be a mistake on your part, do try running the command one or more additional times. (The `submit` program's a bit old and annoyingly flakey.) But if informed that your "work HAS been submitted," it indeed has.

Now, if you instead see some message informing you that your code is not where it should be, you'll need to fix that before you submit. Review the last two pages of Problem Set 1's spec for some tips!

☐ Anytime after lecture on Wed 9/22 but before this problem set's deadline, head to the URL below where a short form awaits:

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

If not already logged in, you'll be prompted to log into the course's website.

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

This was Problem Set 2.

[6] Note that there is no slash between this tilde (`~`) and `cs50`.