CS103 Spring 2016 — Caesar Decipher

Caesar Decipher

In this assignment, you will implement a simple form of message encryption that dates back to the times of Julius Caesar. This technique, called the *Caesar Cipher*, does not offer much security in modern practice (and in fact the second half of the assignment is to crack it). Nonetheless, it is of historical importance and led to the development of ever-gradually more complex techniques like the Vigenère cipher and the Enigma from WWII.

This assignment will exercise your understanding of the following concepts:

- functions and libraries
- file stream objects
- character arrays and ASCII representation
- command-line arguments

Background

Julius Caesar (100-44BC) was involved in many complex affairs and would on occasion want to send messages secretly. Paraphrasing the 1st-century historian Suetonius,

> If Caesar had anything confidential to say, he wrote it in cipher, that is, by so changing the order of the letters of the alphabet, that not a word could be made out. If anyone wishes to decipher these, and get at their meaning, he must substitute the first letter of the alphabet, namely A, for D, and so with the others.

The following Khan Academy video gives a nice illustration. **However,** you'll be implementing a **different decryption method** (a more sophisticated one) than the one it mentions.

Part 1: Encryption and Decryption

As illustrated in the video, the way Caesar would obscure/encrypt a message is the following:

- write down the "plaintext" message you desire to send secretly.
- write down a new "ciphertext" message: for each letter in the plaintext, write down the letter that appears 3 steps later in the alphabet.
- send the ciphertext instead of the plaintext.

For example, if the plaintext is

`Statue is safe?`

the letter S is replaced by V (since the English alphabet goes ...PQR**S**TU**V**W... and the letter 3 steps after S is V), the letter T is replaced by W, etc, resulting in the ciphertext

`Vwdwxh lv vdih?`

This ciphertext message is what we would actually send. If someone were to intercept it, the meaning of the message would not be immediately obvious — they might think it's in a foreign language — so it provides at least a minimal degree of secrecy.

Your first program, `shift`, will automate this process. It should take two *command-line arguments*: the name of a text file which will contain the plaintext, and an integer shift indicating how many places the message should be shifted forward. It should print the ciphertext to *standard output* (`cout`).

E.g. if you type in the following two commands, you should see this output:

```
$ echo "Statue is safe?" > msg.txt # create a text file
$ ./shift msg.txt 3
Vwdwxh lv vdih?
```

One issue has not been addressed: *what do you do if you fall off the end of the alphabet?* For example, how do we shift Z forwards by 3 steps? We will think of the alphabet as cyclic:

This means that the letter "after" Z is A. Here is an example illustrating this cyclicity:

```
# remember: msg.txt contains "Statue is safe?"
$ ./shift msg.txt 10
Cdkdeo sc ckpo?
```

In detail, the first letter of the plaintext is S, and the letter 10 steps "after" S is C, so the first letter of the ciphertext is C.

Decryption

This *Caesar shift* scheme has the nice property that decryption is basically the same as encryption. For instance, to "reverse" the shift of 3 steps forward, we want to basically move 3 steps backwards. But because we've arranged the letters of the alphabet in a circle, moving 3 steps backwards is the same as moving 23 steps forwards. Thus we can decrypt a message like so:

```
$ ./shift msg.txt 3 > secret.txt
$ more secret.txt
Vwdwxh lv vdih?
$ ./shift secret.txt 23
Statue is safe?
```

Part 1 Requirements and Structure

There is one requirement implicit in the above examples: you should shift lowercase letters to lowercase letters, uppercase letters to uppercase letters, and anything that is not a letter should be left alone.

Your `shift` program should work for any number of steps between 0 and 25. (The number of steps is the

second command-line argument.) You could make it work for other values (like -3 or 30) if you like, but it is not required.

Because Part 2 will re-use many of the ideas from Part 1, you are required to create a library with three specific functions. We will give you a file `caesarlib.h` with the following contents:

```
// caesarlib.h:

// is this char an English letter?
bool is_letter(char ch);

// return shifted image of ch (if ch not letter, don't shift)
// assumes 0 <= steps < 26
char image(char ch, int steps);

// shift all characters in this file and print it to cout
// return 1 if error (file couldn't be opened), 0 if no error
int print_file_image(const char filename[], int steps);
```

You should implement these three functions in `caesarlib.cpp`.

As you develop `caesarlib.cpp` it is a good idea to test it. To help, we will include a program `libtest.cpp` that contains some basic tests:

```
cout << is_letter('g') << " ";
cout << is_letter('G') << " ";
cout << is_letter('?') << endl;
cout << image('S', 3) << " ";
cout << image('w', 23) << " ";
cout << image('?', 10) << endl;
```

When you compile and run it, you should see the following:

```
$ make libtest
$ ./libtest
true true false
V t ?
```

You may add extra functions to `caesarlib` if you like, but you may not delete or change the prototype of any of the three given functions `is_letter`, `image`, or `print_file_image`. We will test them independently and expect them to exhibit the functionality described in the comments above their

declaration.

There happens to be a built-in function for this (it is part of the C++ library `cctype` but can also be accessed through other means). You must **not** use it! Part of the point of this assignment is for you to be able to create a function like this from scratch. All of the necessary header files will be already given to you in the skeleton code.

After creating `caesarlib`, complete the skeleton of `shift.cpp` given to you. Make sure you understand the parts of the skeleton we give you pre-written; do not delete them. Compile your code with `make shift` and then test `shift` on the examples given above.

You can upload your part 1 files for testing before proceeding to part 2.

Part 2: Crack the Code

It turns out that Caesar's method was pretty good for its time. Not only was it innovative (the oldest known cryptographic method), but no record of it being cracked is known until nearly a millennium later, due to the Arab polymath Al-Kindi.

For Part 2 of the assignment, you will implement a specific method that cracks the Caesar cipher. It doesn't crack it perfectly (we'll give examples below) but it works on most ciphertexts that are long enough and not too weird. For example:

```
$ ./shift msg.txt 3 > secret.txt
$ ./crack secret.txt
Statue is safe?
```

What is significant is that we didn't have to tell `crack` the shift number, it figured it out "automatically."

Frequency Analysis

The approach that you must use for `crack.cpp` is based on frequency analysis. In particular, we use the fact that in every language, some letters are more common than others. (It is the same reason that `Q` is worth a lot more than `R` in Scrabble.)

We will assume that in the English language, letter frequencies "on average" occur with a specific frequency distribution:

```
const double enfreq[26] = {0.0793, 0.0191, 0.0392, 0.0351, 0.1093,
 0.0131, 0.0279, 0.0238, 0.0824, 0.0024, 0.0103, 0.0506, 0.0277,
 0.0703, 0.0602, 0.0274, 0.0019, 0.0705, 0.1109, 0.0652, 0.0321,
 0.0098, 0.0093, 0.0026, 0.0156, 0.0040};
```

What this means is that "on average," we expect 7.93% of letters in English texts are the letter A, 1.91% are the letter B, 3.92% are the letter C, et cetera. We will give you a skeleton for `crack.cpp` that includes this frequency table (array) `enfreq` pre-defined for you.

The Khan Academy video suggested just looking at the maximum frequencies, but we will use a slightly more robust method instead. It is based on giving a *score* to each possible shift.

- The ***score*** of a single letter is just its value in the frequency table.
- Then the ***score*** of a decrypted message is the sum of the scores of its individual letters. (Anything that is not an English letter, like a digit or punctuation sign, has a score of 0.)

For example, on the example `./crack secret.txt` above, your program should enumerate the possibilities:

- Add shift of 0: message is "Vwdwxh lv vdih?" which has score 0.0098 + 0.0093 + 0.0351 + 0.0093 + 0.0026 + 0.0238 + 0.0506 + 0.0098 + 0.0098 + 0.0351 + 0.0824 + 0.0238 = 0.3014
- Add shift of 1: message is "Wxexyi mw weji?" which has score 0.4622
- ...
- Add shift of 23: message is "Statue is safe?" which has score 0.9679
- ...
- Add shift of 25: message is "Uvcvwg ku uchg?" which has score 0.2935

Your program should determine the shift with the maximum score (which is the shift of 23 in this case). Then it should print out the image of the message under this shift.

Part 2 Requirements and Structure

Several of the methods in `caesarlib` will be useful again for this program. You must also implement the following two functions inside of `crack.cpp`; they will be useful in structuring your approach:

```
// return score when ch is shifted (if ch not letter, return 0)
double char_score(char ch, int shift)

// return score when contents of entire file are shifted
double file_score(const char filename[], int shift)
```

Again, you can define more functions if you like. However, these two must be present and completed in `crack.cpp`, with these prototypes.

You may test these functions with this test main:

```
// crack.cpp test main
```

```
int main() {
   cout << char_score('A', 3) << " ";
   cout << char_score('x', 10) << " ";
   cout << char_score('?', 3) << endl;
   cout << file_score("secret.txt", 0) << " ";
   cout << file_score("secret.txt", 1) << endl;
}
```

If you compile and run

```
$ make crack
$ ./crack
```

it should output

```
0.0351 0.0238 0
0.3014 0.4622
```

Perform any additional tests you like. You will need to delete, comment or rename that test main in order to add the real `main` function of `crack.cpp`.

Reiterating the example given earlier, the `main` function of `crack.cpp` should take one command-line argument, which is the name of a ciphertext. It should compute the score of the file's contents under all possible shifts from 0 to 25 and determine the shift with the best score. Then, it should print to standard output (`cout`) the image of the file under that shift. Compile it with `make crack`.

Examples and Caveat

You can create small sample files and crack them as follows:

```
$ echo "This file's a test" > test.txt
$ ./shift test.txt 11 > test-enc.txt
$ more test-enc.txt
Estd qtwp'd l epde
$ ./crack test-enc.txt
This file's a test
```

No matter what number of steps you pass as a command-line argument to `shift`, the `crack` program should give the right answer on the last line.

This score-based cracking method is not perfect: it will fail sometimes. For instance, let's repeat the above

example with a different plaintext, `Rhythm & Blues`.

```
$ echo "Rhythm & Blues" > rb.txt
$ ./shift rb.txt 10 > rb-enc.txt
$ ./crack rb-enc.txt
Xnezns & Hraky
```

That's not exactly English! However, the score of this shift "Xnezns & Hraky" is 0.5669, which is bigger than the score 0.5486 of "Rhythm & Blues". Better methods could avoid this by looking at consecutive pairs of letters. However, don't do this. We want you to implement the exact method described above, so for our purposes, this is the correct output.

Starter Code

Obtain the skeleton code for this assignment as follows:

```
mkdir caesar # or wherever you want
cd caesar
wget ftp://bits.usc.edu/cs103/caesar.tar
tar -xvf caesar.tar
```

This contains all of the skeleton code and tests mentioned above, the `readme.txt` file, an additional test file and output, as well as two corpora for the optional extra credit (see below).

Submit

- Answer the questions that are listed in the `readme.txt` file that was contained in your `.tar` file.
- Continue to follow the code style guidelines posted here.
- Use the link on the coursework page to submit your code and readme file.

Extra Credit

The extra credit portion of the assignment is meant to be an extra challenge to stretch your skills. It is **not** meant to be a way to bump up your grades, because it is not worth very much. (For instance, leaving no comments or indentation in the assignment would cost you more style points on the real assignment than you would gain from doing the extra credit.)

There are two extra credit parts. Both of them should be submitted as part of a separate program `extra.cpp`. You can do part 1 without doing part 2.

Strictly enforced collaboration policy: You can't ask course staff for help doing extra credit. We want to focus our energy on getting everyone through the normal portion of the assignment, and encourage

independence for extra credit.

Extra credit part 1: Internationalization

For this part of the assignment, you will make your cracking program work with any language, not just English. Consequently, you will no longer use the array `enfreq`. Instead, your program will take two command-line arguments: the first is the ciphertext, and the second is a *corpus* which is just the filename of a long text document in the desired language (e.g., a dictionary, a bible, Wikipedia, etc).

For example,

```
$ ./extra secret.txt corpus-en.txt
Statue is safe?

$ echo "Ae'ocd aeo m'ocd?" > mystere.txt
$ ./extra mystere.txt corpus-fr.txt
Qu'est que c'est?
```

Your program should:

- scan all letters in the second-named file and create a table of frequencies
- replicate the behavior of `crack.cpp` using that table of frequencies instead of `enfreq`

Note that you may have a lot of duplicated or nearly-duplicated code from `crack.cpp`; that is okay, the extra credit is not about perfecting the style.

Your code should only scan the corpus once. The secure "sandbox" that runs your code may cause your program to fail if you use a lot of I/O.

Extra credit part 2: Language detection

Modify your program so that it takes in multiple corpora, representing different languages. It should try cracking the message in each language (like extra credit part 1) and determine which one is the best match for the given text (which one can achieve the best score). It should print out the filename of the corpus that achieves the best match.

```
$ ./extra mystere.txt corpus-fr.txt corpus-en.txt
corpus-fr.txt
```

You can assume there will be between 2 and 20 languages.

Your code should only scan each corpus once. The secure "sandbox" that runs your code will detect if you try to scan a corpus 26 times and will cause your program to fail.

The single program `extra.cpp` will be responsible for performing both of the tasks from extra credit parts 1 and 2.

Q&A

- Where can we get a table of ASCII values?
- One is in the appendix of your textbook. However, all you really need to know is that the lower-case letters are consecutively numbered, and the upper-case ones are too.
- How can I get the ASCII value of a character, or convert an ASCII number to its corresponding character?
- Look at "casting" in the lecture notes from the second lecture. However, keep in mind that you can also directly compare characters (`'x' < 'y'`) and also that any arithmetic on them (`'e' - 'g'`) will automatically cast them to `int`.
- How can we read through all the characters of a text file?
- See the lecture notes on streams. You must use the `get` method.
- How can we store an entire text file's contents as a string?
- You shouldn't try to do this. It will use much more memory than necessary. Just stream through its characters one a time.
- Can I use C++ `string` objects in this assignment?
- No. (They will be permitted in all future assignments.)
- Should I be using dynamic memory (`new` and `delete`)?
- Probably not. Very few arrays are needed at all and all have constant size. If you choose to allocate arrays with dynamic memory, make sure to clean it up afterwards.
- What should I do if two shifts both have the same maximum score?
- We won't test this case.
- What is `echo`?
- The `echo` command takes a command-line argument and prints it to standard output. We've used it as a shortcut for creating one-line text files.
- Is there a similar approach for cracking general substitution ciphers?
- If you're allowed to mix up the alphabet arbitrarily, there are 26! possible ciphers, which is too much to check exhaustively. Also, maximizing the score doesn't turn out to extract much more useful information than Al-Kindi's top-frequency approach. However, there are clever methods such as this one based on the Markov Chain Monte Carlo approach and frequencies of 2-grams (pairs of letters).
- The extra credit deals with foreign languages. How should my program treat letters with accents, like *é*?
- You only should shift English letters (the 52 ASCII values corresponding to the lower- and upper-case English alphabet). Treat anything else, including accented letters, the same as punctuation (don't shift it, and treat its score as 0).
- Can you explain how letters with accents are represented? I don't see them in my ASCII table.