Wesleyan University, Fall 2023, COMP 211

Homework 3: Working with arrays

Due by 11:59pm on September 26, 2022

---

1. Written problems (10 points)

Problem 1. *Flexibility in being able to interpret bit strings differently allows for some clever algorithmic techniques. One such is* steganography, *which is the practice of hiding one kind of information in another. The goal of steganography is to hide a message from an adversary in such a way that the adversary cannot detect the presence of a message.*

Suppose the following 8 bytes represent a sequence of eight 8-bit ASCII codes (spaces have been inserted spaces for readability, but the spaces have no meaning).

01101000 01101001 00100001 01100001 01101100 01101100 11110100 00100001

Within these 8 bytes, we would like to hide another 8-bit ASCII code. For easy reference, let us label the bit locations in a byte as $b_7b_6b_5b_4b_3b_2b_1b_0$. *Now one approach is to hide information in the least significant bit of each of the eight bytes, that is, the* b_0 *bit of each byte. In the sequence of eight bytes above, we have hidden the character 'q', for which the 8-bit ASCII code is 01110001, by setting the* b_0 *bit of the first byte above to be the* b_7 *bit of 01110001, by setting the* b_0 *bit of the second byte to be the* b_6 *bit of 01110001, and so on, and finally setting the* b_0 *bit of the eighth byte to be the* b_0 *bit of 01110001. An alternative approach we could have taken is to instead hide the 8-bit ASCII code for the character 'q' in the most significant bit (i.e.,* b_7 *bit) of each byte.*

Now, there are multiple different ways of choosing 8 bytes in which to hide the 8-bit ASCII code for 'q'. Keeping this in mind, let us now examine how flexible this steganography technique actually is. First assume that we **do not** *require that the 8 bytes actually spell out something comprehensible. How many different ways are there to choose the values of the 8 bytes such that the most significant bits correspond to the 8-bit ASCII code for 'q'? How many different ways are there to choose the values of the 8 bytes such that the least significant bits correspond to the 8-bit ASCII code for 'q'? Next assume that we* **do** *require that the 8 bytes spell out something comprehensible. Discuss how your answers to the previous questions would change under this new requirement; we are not looking for an exact number now, but a discussion of the possible issues that might arise. Finally, suppose we restrict the 8 bytes to being selected from the set of 8-bit ASCII codes for letters in the alphabet and punctuation: would it be preferable to use the most significant bits or the least significant bits for steganography?*

2. Coding problems (10 points)

Problem 2. *You will write a program to perform a Caesar shift on a string of characters. A "string" in C is really an array of* `char` *values. Have your program first read in the string to be shifted, and then read in the amount to shift by. Your program should then compute and output the shifted string. In the string, the user can include any ASCII character with decimal code in the range [32,126]. This range determines the range of shifts permitted: shift amounts from 0 to 126-32*

= 94 are all permitted. You may assume that the user will enter a string of at most 256 characters, including the newline character (see the hints for information about this). While most of the work for your program will be done in the `main` *function, you must write at least one function that is responsible for performing a Caesar shift. That function might return the result of shifting a single character, or it might shift many characters in a single call.*

A few hints:

- *The function* `scanf` *reads in characters until it hits white space. Since the user may enter a white space character such as a space, you will not be able to use* `scanf`*. Instead you should use the* `fgets` *function. An example usage is*

  ```
  fgets(&plaintext[0], 256, stdin)
  ```

  which will read from standard input (which corresponds to the keyboard, so the user typing) until either a new line character (`'\n'`*) is entered by the user or* 256 − 1 = 255 *characters are entered. That is, the resulting string consists of the characters up to* and including *the first newline character; if there is no newline character in the first* 255 *characters, then the resulting string consists of the first* 255 *characters. The resulting characters are then stored in the array* `plaintext` *(which should have been set to have size 256). When you use* `fgets`*, the third argument should always be exactly* `stdin`*.*
- *It actually isn't quite correct to say that a "string" in C is really just an array of* `char` *values. A "string" in C is actually an array of* `char` *values with the requirement that at least one of the values is ASCII code* 0 *(usually written* `'\0'` *and called the* null character*). The "string" consists of the characters from the beginning of the array up to* but not including *the first null character. Characters from the first null character to the end of the array are not considered part of the string. So if* `s` *is a* `char` *array of length* 256*, and* `s[0] = 'H'`*,* `s[1] = 'i'`*, and* `s[2] = '\0'`*, then s represents the string* `"Hi"` *regardless of the remaining values of* `s`*. You will not need to do any explicit calculations with the null character in this assignment, but it is helpful in understanding some of the built-in functions. For example, the reason that* `fgets` *reads in one fewer characters than its second argument is that it adds a null character at the end of the characters that the user types. You might want to look at pp.30, 38–39 of the Kernighan and Ritchie* The C Programming Language *for more information on strings.*
- *To determine how many characters have been entered by the user, there are several options. One option is to use the* `strlen` *function: if* `s` *is a string (i.e., a* `char` *array with at least one null character), then*

  ```
  strlen(s)
  ```

  returns the number of characters in the array up to but not including the first null character. Note that with respect to the program you must write for this problem, if the user enters a string of fewer than 255 *characters, the last character of the string will be the newline character, and you don't want to shift that character! To use* `strlen` *in your programs, you must have the following directive at the beginning of your program:*

  ```
  #include <string.h>
  ```

You should name your program `hw3a.c`*. Figure 1 shows a few sample runs of the program.*

Problem 3. *You will write a program to* **decrypt** *a Caesar shifted string of characters. Have your program first read in the encrypted string. Your program should then compute and output the possible decrypted strings and the corresponding shift amount. You are free to choose the order* *in which the possible decrypted strings and corresponding shift amounts are printed: e.g., in the sample run in Figure 2, the decrypted string corresponding to using a shift of 0 to encrypt is printed on the first line, and on the second line the decrypted string corresponding to using a shift of 94 is printed, rather than incrementing the shift to 1.*

As with Problem 2, in the string, the user can include any ASCII character with decimal code in the range [32,126], and again this range determines the range of shifts permitted, from 0 to 126-32 = 94. Again, you may assume that the user will enter a string of at most 256 *characters, and you should use* `fgets` *to read the string from the keyboard. While most of the work for your program will be done in the* `main` *function, you must write at least one function that is responsible for performing (or reversing) a Caesar shift. That function might return the result of shifting a single character, or it might shift many characters in a single call. The hints for Problem 2 are also relevant here.*

You should name your program `hw3b.c`*. Figure 2 shows a sample run of the program. Don't forget that if you encrypt a string using* `hw3a` *and run* `hw3b` *on the result, you should see your original string as the result for the corresponding shift.*

3. Going further

This is *not* to be submitted, but is just a bit of food for thought. In Problem 1, you thought about using steganography to embed a single ASCII character into a sequence of ASCII characters. Just as when embedding text into images, the goal is to ensure that the sequence of bytes after embedding is reasonable (as an image, or as text). Can you embed `Hello, Sam` into a sequence of ASCII characters so that the result is meaningful English text? Notice you will need at least 80 characters of text to embed into, since there are 10 characters to be embedded.

4. Submission

Upload your written work as `hw3.pdf`, and your code solutions as `hw3a.c`, and `hw3b.c`, to the Google Drive directory I have created for you named `comp211-f23-USERNAME/hw3/`. You should replace `USERNAME` with your Wesleyan username.

Do not forget that your written work must be submitted as a PDF! And make sure that at the top of each file you have put your name! Do not, however, change the names of the files.

```
$ ./hw3a
Enter a string to encrypt: abcd 43
Enter the shift amount for Caesar cipher: 1
Ciphertext is bcde!54

$ ./hw3a
Enter a string to encrypt: hello world!
Enter the shift amount for Caesar cipher: 23
Ciphertext is  |$$'7/'*${8
```

FIGURE 1. Some sample traces from `hw3a`.

```
$ ./hw3b
Enter string to decrypt: bcde!54
Possible plaintext is bcde!54 using shift of 0
Possible plaintext is cdef"65 using shift of 94
Possible plaintext is defg#76 using shift of 93
Possible plaintext is efgh$87 using shift of 92
Possible plaintext is fghi%98 using shift of 91
Possible plaintext is ghij&:9 using shift of 90
Possible plaintext is hijk';: using shift of 89
Possible plaintext is ijkl(<; using shift of 88
Possible plaintext is jklm)=< using shift of 87
Possible plaintext is klmn*>= using shift of 86
Possible plaintext is lmno+?> using shift of 85
Possible plaintext is mnop,@? using shift of 84
Possible plaintext is nopq-A@ using shift of 83
Possible plaintext is opqr.BA using shift of 82
Possible plaintext is pqrs/CB using shift of 81
Possible plaintext is qrst0DC using shift of 80
Possible plaintext is rstu1ED using shift of 79
Possible plaintext is stuv2FE using shift of 78
Possible plaintext is tuvw3GF using shift of 77
...
Possible plaintext is BCDE`ts using shift of 32
Possible plaintext is CDEFaut using shift of 31
Possible plaintext is DEFGbvu using shift of 30
Possible plaintext is EFGHcwv using shift of 29
Possible plaintext is FGHIdxw using shift of 28
Possible plaintext is GHIJeyx using shift of 27
Possible plaintext is HIJKfzy using shift of 26
Possible plaintext is IJKLg{z using shift of 25
Possible plaintext is JKLMh|{ using shift of 24
Possible plaintext is KLMNi}| using shift of 23
Possible plaintext is LMNOj~} using shift of 22
Possible plaintext is MNOPk ~ using shift of 21
Possible plaintext is NOPQl!  using shift of 20
Possible plaintext is OPQRm"! using shift of 19
Possible plaintext is PQRSn#" using shift of 18
Possible plaintext is QRSTo$# using shift of 17
Possible plaintext is RSTUp%$ using shift of 16
Possible plaintext is STUVq&% using shift of 15
Possible plaintext is TUVWr'& using shift of 14
Possible plaintext is UVWXs(' using shift of 13
Possible plaintext is VWXYt)( using shift of 12
Possible plaintext is WXYZu*) using shift of 11
Possible plaintext is XYZ[v+* using shift of 10
Possible plaintext is YZ[\w,+ using shift of 9
Possible plaintext is Z[\]x-, using shift of 8
Possible plaintext is [\]^y.- using shift of 7
Possible plaintext is \]^_z/. using shift of 6
Possible plaintext is ]^_`{0/ using shift of 5
Possible plaintext is ^_`a|10 using shift of 4
Possible plaintext is _`ab}21 using shift of 3
Possible plaintext is `abc~32 using shift of 2
Possible plaintext is abcd 43 using shift of 1
```

FIGURE 2. Sample trace from `hw3b`.