

Spoiler for CS-241 Project 2: Private Key Cipher

Milestones

Break the problem into 5 milestones. Add output to prove to yourself that you have reached the milestone and leave the output in place until you are totally done. Then, just comment out the extra output.

Milestone 1: Check the syntax and print error messages:

- a) Verify that the input contains 4 parameters.
- b) Verify that `argv[1]` is a string of length 2 and is either “-e” or “-d”.
- c) Verify that `argv[2]` is a string of length 12 and that it consists only of the digits ‘0’ through ‘7’ (these characters make all the octal digits).
- d) Verify that `argv[3]` is a string of any length >0 and <1025 , and that it does not contain any characters with codes <32 or greater than 126.

Milestone 2: Separate the input key into the source and sink parts. The key is a string of length 12. These should be thought of as 6 pairs. I call the first digit of each pair the *source* part because it is used to select bits, one at a time, from the input character. The second digit in each pair I call the *sink* part because it is used to determine where each selected bit from each input character will be mapped.

- a) Create two global arrays: `int sourceKeys[7], sinkKeys[7];`
Even though the input key only contains 6 pairs, there are 7 bits that need mapping. There is no need to give 7th input pair because after the first 6 mappings have been created, there is only one source and one sink location left. To avoid handling this case separately, I just append an extra source key and an extra sink key: `sourceKey[6] = 0; sinkKey[6]=0;`
- b) Walk through the 12 digits of the input key, convert each octal character digit to an `int` and place every even indexed key in the `sourceKey` list and every odd indexed key in the `sinkKey`.
- c) Finally, print the two arrays. For cipher `-e 21 21 03 55 30 30`
"Euler"
The output should be:

```
sourceKey = [2 2 0 5 3 3 0]
sinkKey   = [1 1 3 5 0 0 0]
```
- d) Test this for some of the other test cases and make sure you are always getting the keys split correctly.

Milestone 3: Build the encryption and decryption bit mappings. Note: so far we have not even looked at the string to be encrypted except to check it for errors. We will still not look at the string in this milestone. In this milestone, just build the maps from the keys. This, I think, is the hardest step.

- a) Create two global arrays: `int encryptMap[7], decryptMap[7];`
- b) Initialize the values in these arrays to -1. A value of -1 will be used to indicate that the corresponding bit (bit 0 through 6) has not yet been mapped. A value of zero in the array indicates the corresponding bit of the character to be encrypted will be mapped to bit 0 of the encrypted character. For example, if `encryptMap[4]=0`, then bit #4 of each unencrypted character will be mapped to bit #0 of each encrypted character. The -1 values are needed because when you use each key to count through `encryptMap` and `decryptMap`, you need to skip array elements that already contain a mapping.
- c) Use the circular counting to build the values in `encryptMap` and `decryptMap`.

```
For cipher -e 21 21 03 55 30 30 "Euler"
```

The output should be:

```
encryptMap = [4 6 3 2 1 0 5]
```

```
decryptMap = [5 0 6 4 3 2 1]
```

- d) Calculate by hand what maps one of the other test case keys should produce. Then check your code to see if it produces them.

Milestone 4: Only now that you can successfully build the encryption map do you look at the string to be encrypted and do the encryption.

- a) Create a global array: `char outString[1025];`

This same array will be used to store the encrypted string when the input string is an unencrypted string, and to store the unencrypted string when the input string is an encrypted string. This `outString` only needs to be as big as the input string, however, it is easier to just make it +1 the maximum size of the input string (the +1 is for a '0/' at the end).
- b) Loop through the input string one character at a time. Whatever unencrypted character is at `inputString[i]`, will have its corresponding encrypted character build in `outString[i]`.
- c) Set `outString[i]=0;` This will set all 8 bits of the encrypted character to 0.
- d) Start at bit 0 of the character you are going to encrypt.
- e) If bit 0 of the character to be encrypted is a 1, then this 1 needs to be mapped to someplace in `outString[i]`. `encryptMap[0]` tells you where (which bit) that someplace is.

- f) Create a local variable: `unsigned char mapBit`. Set the value of this so that all bits are 0 except for the bit in the correct map location. That is the bit number stored in `encryptMap[0]`.
- g) Set the mapped bit in `outString[i]` to 1.
- h) Go back to step d and repeat on the next unencrypted bit of input character `i`. Stop when you have mapped all 7 bits of `inputString[i]` into `outString[i]`.
- i) If `outString[i]` is less than 32 (i.e. it is a non printing character), then add 192.
- j) The character in `outString[i]` now contains the correctly encrypted character found in `inputString[i]`.

Milestone 5: Reverse the encryption process to do decryption.