COMP 2401 A/C – Assignment #2

Due: Thursday, October 19 at 11:59 pm

1. Goal

For this assignment, you will write a program in C, in the Ubuntu Linux environment, to provide the end user with the ability to encrypt and decrypt messages, using a secret key. The algorithm that you implement will use a *chaining* approach. With chaining, each byte (i.e. each character) is encrypted in a slightly different way, because the encryption of each byte is dependent on the encryption result of the previous byte. This prevents the same character value from being encrypted identically every time. This approach is based on the Cipher block chaining (CBC) mode of the Advanced Encryption Standard (AES) encryption algorithm.

If the end user of your program chooses to encrypt a message (called the *plaintext*), your program will output the encrypted sequence (called the *ciphertext*) to the screen. If the user chooses to decrypt an encrypted sequence (ciphertext), your program will print out the decrypted message (plaintext).

2. Learning Outcomes

With this assignment, you will:

- write a program in C that is correctly designed into modular functions, and correctly documented
- organize a C program by reusing existing functions as much as possible
- use bit masks and bitwise operators to manipulate values at the bit level

3. Instructions

Your program will prompt the user to indicate whether they wish to encrypt a readable message (the plaintext), or decrypt an already encrypted sequence of numbers (the ciphertext) back to its readable form. You will implement the encryption algorithm described below, and you will deduce and implement the corresponding decryption algorithm.

Your program must follow the programming conventions that we saw in the course lectures, including but not restricted to the correct separation of code into modular functions, basic error checking, and the documentation of each function. A more comprehensive list of constraints can be found in the *Constraints* section of this document.

3.1. Understand the encryption algorithm

3.1.1. Both the plaintext and the ciphertext must be stored as arrays of `unsigned char`s. We will manipulate the individual characters using bit masks, exactly as we saw in the the course material, section 2.1.2. Remember: plaintext characters are stored in memory using their numeric ASCII value.

3.1.2. The encryption algorithm processes each plaintext byte, one at a time, with a key that is initialized at the beginning of the program. The initial key value must be identical for both the encryption and the decryption processes, and it must be updated identically with every byte that's encrypted or decrypted. If we encrypt a message with a specific initial key, we must use the **same** key for decryption. If we use a different initial key for decryption, we will get garbage instead of plaintext when we decrypt the ciphertext.

3.1.3. Cipher-block chaining means that part of the encryption process for each plaintext byte requires the encrypted value of the *previous* plaintext byte. So our algorithm needs an initial value for encrypting the first plaintext byte, since there is no previous ciphertext byte to use. This *initial value* (IV) is set to `0b11001011`.

3.1.4. The encryption process has two separate parts:
- the initialization of the key, and
- the encryption of each plaintext byte into a ciphertext byte, using that key

3.1.5. **Initialization of the key:**

Definition #1: Two *mirror bit positions* inside a byte are two bit positions that are the same as each other, relative to each end of the byte. For example, bit positions 0 and 7 are mirror bit positions, because they are both in the first position at either end of a byte. Bits 1 and 6 are also mirror bit positions (both are in second position from either end of the byte), as are bits 2 and 5, and so on.

Definition #2: The *mirror bit value* of a bit in a specific position is the value (zero or one) of the bit at the mirror bit position. For example, the mirror bit value of the bit in position 2 is the value (zero or one) of the bit in position 5.

The initialization of the key takes place as follows:

(a) prompt the user to enter a partial key value; this must be a number between 1 and 15, inclusively, so that only the four least significant bits have values; this number is stored as an `unsigned char`

(b) left-shift the partial key by 4 bits, so that the four least significant bits are moved into the four most significant bit positions

(c) loop over the four most significant bits (from bit position 7 to 4, inclusively) of the partial key

(d) for each of the four most significant bits, assuming `i` as the current bit position:
   (i) compute a secondary bit position as `(i-1) mod 4`
   (ii) set the key bit at the secondary position to the *inverse* of the key bit at position `i`

3.1.6. **The encryption algorithm:**

Once the key has been entered and initialized as described above, the program must prompt the user to enter the plaintext. This is entered from the command line as a single string, and it is stored as a null-terminated array of `unsigned char`s. You have been provided with a function in the base code to assist you with this.

The Encryption Algorithm:

```
For every byte of plaintext:
  (1) process the key and update its value, as follows:
        (a) if the current key value is a multiple of 3, perform a circular left-shift of the key by 3 bits
        (b) otherwise, perform a circular left-shift of the key by 2 bits
  (2) encrypt the plaintext byte using the key updated in step (1) above and the previous byte of
      ciphertext (or the IV if encrypting the first byte of plaintext), as follows:
        (a) initialize a temp byte to zero
        (b) loop over every bit position, starting at bit 0; for every bit position i:
              (i) if the key bit at position i is one, perform circular right-shift of the previous ciphertext
                  byte by 1 bit, otherwise keep the current value
             (ii) set the temp byte bit at position i to the result of an xor operation between the plaintext
                  bit at position i and the previous ciphertext bit at the mirror bit position
        (c) once the loop ends, return the temp byte value as the ciphertext byte
```

3.2. Implement the encryption functionality

3.2.1. You will begin with the base code in the `a2-posted.c` file that is posted in *Brightspace*. **Do not** make any changes to the provided code, including the function prototypes.

3.2.2. Implement the `getBit()`, `setBit()`, and `clearBit()` functions, exactly as we saw in class.

Your program **must** reuse these functions to access and make all changes to the bits inside each byte. Do not access or modify any bits without using these functions.

3.2.3. Implement a function that takes a byte and performs a circular left shift on it by one bit, and another function that does a circular right shift. As we discussed in class, in a *circular* shift, the bits shifted in are the same bits that were shifted out.

3.2.4. Implement the `unsigned char initKey()` function that prompts the user to enter a correct partial key, then initializes the full key as described in instruction 3.1.5. The function returns the initialized key as the return value.

3.2.5. Implement the `unsigned char processKey(unsigned char currKey)` function that updates the key from its current value in `currKey`, as described in step (1) of the encryption algorithm above. The function returns the updated key as the return value.

3.2.6. Implement the `unsigned char encryptByte(unsigned char pt, unsigned char key, unsigned char prev)` function that encrypts the given plaintext byte `pt`, using the given key and the previous byte of ciphertext `prev`, as described in step (2) of the encryption algorithm. The function returns the corresponding encrypted ciphertext byte as the return value.

3.2.7. Implement the `void encode(unsigned char *pt, unsigned char *ct, unsigned char key, int numBytes)` function that takes an array of plaintext characters stored in parameter `pt`, which contains `numBytes` bytes. The function encrypts each plaintext character in the array into its corresponding ciphertext byte using the given key, as described in the encryption algorithm in instruction 3.1.6. Each encrypted byte is stored into the ciphertext array `ct`.

3.3. **Implement the decryption functionality**

Write the functions required to implement the decryption algorithm. The decryption processes the key identically to the encryption, but the decryption of each byte using the updated key must perform the xor operation between different values than the encryption does.

NOTE: The xor operator ($\oplus$) has a very useful property. Assume that `a` and `b` are any numeric value, and that $a \oplus b$ results in `c`. Then it is also true that $b \oplus c$ results in `a`, and that $a \oplus c$ results in `b`. So your decryption algorithm does not need to "reverse" the xor operation that was performed during encryption. Instead, it must perform the xor between different values than during encryption.

3.3.1. Implement the `unsigned char decryptByte(unsigned char ct, unsigned char key, unsigned char prev)` function that decrypts the given ciphertext byte `ct`, using the given key and the previous byte of ciphertext `prev`. The function returns the corresponding decrypted plaintext byte as the return value.

3.3.2. Implement the `void decode(unsigned char *ct, unsigned char *pt, unsigned char key, int numBytes)` function that takes an array of ciphertext bytes stored in parameter `ct`, which contains `numBytes` bytes. The function decrypts each ciphertext byte in the array into its corresponding plaintext byte using the given key. Each decrypted byte is stored into the plaintext array `pt`.

3.4. **Implement the main control flow**

3.4.1. Implement the `main()` function to first prompt the user for whether they want to encrypt a string, or decrypt a sequence of numbers.

3.4.2. The user is then prompted for a partial key, and the full key is initialized as described in instruction 3.1.5. Your code must perform all error checking. Remember, a recoverable error like entering a value out of range is never a reason to terminate a program. Your code should keep prompting for a correct value until one is entered.

3.4.3. If the user chooses to encrypt a string, your program does the following:
- (a) read a plaintext string from the command line, using a function provided in the base code; because this string will contain multiple space-delimited words, you cannot use the `scanf()` library function for this; the provided function uses `fgets()` instead
- (b) encrypt the plaintext entered by the user, as described in the encryption algorithm
- (c) print out to the screen the space-separated ciphertext bytes, as decimal numeric values

3.4.4. If the user chooses to decrypt an encrypted sequence, your program does the following:
- (a) read a space-separated sequence of decimal numeric values from the command line, up to a sentinel value of `-1`; the `scanf()` library function is the best choice for this, because one call to `scanf()` reads a single value up to a delimiter, which can be either a space or a newline character
- (b) decrypt the sequence entered by the user
- (c) print out to the screen the resulting plaintext as a string of ASCII characters

NOTE: The user should be able to decrypt any ciphertext that was produced by the algorithm with the same key value, and it should produce the original string as plaintext.

3.5. **Test your program**

You must test your program thoroughly. If your algorithm is correctly implemented, you will be able to decrypt the following sequence to its correct plaintext, using partial key 9 (bragging rights go to the first student who emails me the correct decrypted message!):

```
086 144 168 059 158 250 223 183 142 109 019 161 114 208 203 118 144 132 112 220 157 231
026 172 077 038 035 045 126 131 102 024 180 150 168 059 158 250 141 039 069 103 005 033
114 145 237 031 185 150 253 223 135 106 029 177 193 115 159 230 062 163 068 -1
```

NOTE: You must try different values for the partial key. As long as you use the same value for both encryption and decryption, the algorithm should work.

3.6. **Document your program**

You must document your program, as we covered in the course material, section 1.2.4. Every function except `main()` must have a block of comments before the function to indicate: (1) the function purpose, (2) the correct role for **each** of its parameters (covered in section 1.2.3 of the course material), and (3) the possible return value(s) if applicable. The program as a whole is documented in a `README` file. **Do not** use inline comments as a substitute, as they cannot correctly document a procedural design.

3.7. **Package your files**

3.7.1. Your program must be contained within one (1) source file that is correctly named, for example `a2.c`, that contains your `main()` function at the top of the file, followed by the remaining functions.

3.7.2. You must provide a plain text `README` file that includes:

(a) a preamble (program author, purpose, list of source, header, and data files, if applicable)

(b) compilation and launching instructions, including any command line arguments

3.7.3. Use either the `tar` or the `zip` utility in Linux to package the files above into one `.tar` or `.zip` file.

3.7.4. Do not include any additional files or directories in your submission.

4. Constraints

Your design and implementation must comply with all the rules of correct software development and programming conventions that we have learned during the course lectures, including but not restricted to:

4.1. Your program must be correctly separated into modular, reusable functions, and it must adhere to the correct programming conventions for C programming in Linux.

4.2. The code must be written using the C11 standard that is supported by the course VM, and it must compile and execute in that environment. It must not require the installation of libraries or packages or any software not already provided in the course VM.

4.3. Your program must not use any library functions or techniques not covered in this course, unless otherwise permitted in the instructions.

4.4. Do not use any global variables.

4.5. If base code is provided, do not make any unauthorized changes to it.

4.6. Your program must reuse predefined constants and functions that you implemented, where possible.

4.7. Your program must perform all basic error checking.

4.8. Do not use recursion where iteration is the better choice.

4.9. You may implement helper functions, but only if the design is consistent with the provided instructions. Design and implementation choices that undermine the learning outcomes will not earn any marks.

4.10. All functions must be documented, as described in the course material, section 1.2.

4.11. To earn full program execution marks, your code must compile and execute without warnings or errors, and it must be implemented in accordance with all instructions.

5. Submission Requirements

5.1. You will submit in *Brightspace*, before the due date and time, one `tar` or `zip` file that includes all your program files, as described in the **Package your files** instruction.

5.2. Late submissions will be subject to the late penalty described in the course outline. No exceptions will be made, including for technical and/or connectivity issues. Do not wait until the last day to submit.

5.3. Only files uploaded into *Brightspace* will be graded. Submissions that contain incorrect, corrupt, or missing files will be graded as is. Corrections to submissions will not be accepted after the due date and time, for any reason.

6. Grading Criteria

• 20 marks:	code quality and design
• 50 marks:	coding approach and implementation
• 25 marks:	program execution
• 5 marks:	program packaging