A10: Do you want to know a secret?

Summary:

The music industry has a lot of money invested in maintaining their copyrights from illegal copying and sharing. To protect their rights, they implement various forms of Digital Rights Management (DRM). Central to DRM is that the data in the file is encrypted in a way that (ideally) only the rightful owner of the file can decrypt. Let's build an encryption/decryption program to see firsthand how this works. The code you will create is literally —with just a few changes—a theoretically unbreakable encryption. For a few reasons described below, we will implement the encryption and decryption within functions. While this scheme could be used to encrypt any data, including audio, video, or other files, for simplicity of demonstration, you should show that your program works by encrypting and decrypting a small piece of text, such as a favorite song lyric.

Skills to be Obtained:

- Work with strings and character data as both text and numerical data
- Generate and use sequences of random numbers
- Create and use functions

Details:

For this project, it is recommended that you pair up with a partner. In addition to all the usual benefits of working with a partner, doing so this time will allow you to test your programs by sending your partner an encrypted message and ensuring that your partner can successfully decrypt it. Your partner may need to use the same development environment (Dev-C++ / X-Code / NetBeans) as you for that to work.

We will encrypt the data using a very strong technique that is based on a "substitution cipher" scheme. In its simplest form, a substitution cipher is a very weak encryption. The simplest form(1) is what you may have seen in the back of a comic book or cereal box. This method just shifts each letter by a constant number of spots, "rotating" around from the bottom of the alphabet back to the top as needed. For example, a shift of +1 replaces A with B, B with C, and so on until Z is replaced by A. Instead of this +1 shift, the most common shift used with English language communication uses a value of +13 and is called ROT13 (meaning rotate 13 spots). In ROT13, each letter is replaced by the letter 13 spots down, so A gets replaced by N (letter 14), B gets replaced by O (letter 15), and so on until Z gets replaced by M. Since 13 is half the number of letters in the English alphabet, ROT13 has the distinction of being bidirectional: A becomes N, and N becomes A; B becomes O, and O becomes B; Z becomes M, and M becomes Z.

A shift can be represented mathematically. First, each letter is assigned a numerical value. For example, we can start counting at 0 so that A=0, B=1, and so on until Z=25. To shift letters down, we just add a constant value to each number and then convert the numbers back to letters. To demonstrate, in a shift-2 substitution, the word CAT is changed to the numbers (2 0 19), then we add 2 to each number to get (4 2 21), and then we turn these numbers back to letters to get the encoded text ECV.

How do we accomplish the "wrap-around" for numbers that are greater than 25 after the shift? The answer is to use the modulus operation (the '%' operator in C++). We saw modular math before, with our test for whether a number is odd or even. If we take the value of each shifted number mod 26, then we effectively "wrap around" the number 26 back to 0, the number 27 back to 1, and so on[(2)]. To demonstrate using shift-2: XYZ becomes (23 24 25); we add our shift value 2 to get (25 26 27); we take each number mod 26, so 25 % 26 = 25, 26 % 26 = 0, and 27 % 26 = 1 to get (25 0 1); and finally we convert back to letters to get the encrypted message ZAB. You can verify in your head that this worked: shifting each letter down by 2, XYZ → ZAB.

To decrypt a message, we do basically the same thing as encryption, but we *subtract* the shift value instead of add it. We can use the same mod 26 to stay within 0 to 26—though be careful with negative numbers! Some C++ compilers make negative numbers come out negative after the mod operation. In order to get around this issue, you can test if a number is negative after the mod operation and add 26 if so.

As an example of decryption, the shift-2 encrypted message VQA becomes (21 16 0), we subtract 2 from each to get (19 14 -2), we take each number mod 26 and add 26 as needed to get (19 14 24), and we convert back to our original message: TOY. You can verify in your head that our math worked again: shifting each letter back 2 from the encrypted message VQA → TOY.

One of the reasons this kind of shift substitution is cryptographically weak is because if you figure out how much one letter has shifted, you can quickly shift every other letter in the encoded text by the same amount and recover the original text. To make our encryption stronger, we need a substitution that is less predictable.

We can make this scheme much stronger by having each letter in the message use a *different* shift value. In other words, the first letter in the message might use a shift value of 3, the second letter might use a value of 25, the third letter might use 10, and so on. As long as the sender and the receiver agree in advance on the sequence of shift values to use, this technique is quick to both encrypt and decrypt, yet remains completely uncrackable by any outside party[(3)]. The encrypted message could be sent by any means—even something completely public, like a Twitter feed.

The sequence of shift values should be a series of random integers. We will call this sequence the "encryption pattern." In our work here, we will use the random number generator[(4)] built into your C++ development environment to generate the encryption pattern. The sender encrypting the message and receiver decrypting that message will use the same seed for the random number generator so that they each have the same encryption pattern. We will call this shared seed the "key."

In this project, you will develop codes that perform the encryption and decryption scheme described above. At a minimum, your code only needs to function for capital letters (with words running together as in THISISASECRET). Specific tasks that should be accomplished by your codes include:

- Ask the user for the key and use that key as the seed for the random number generator.
- Ask the user whether this will be an encryption or decryption operation.
- Ask the user for the text to be encrypted or decrypted. The user should be able to input the entire message as a single string.
- Send the string input by the user to either the encryption or decryption function, which returns the encrypted or decrypted message.
- Output to the screen the encrypted or decrypted message.

The encryption and decryption operations **must** be written as separate functions in your code. This will allow us to quickly copy-and-paste these functions into future programs, or to quickly replace these functions with, perhaps, better encryption/decryption functions we write in the future. The functions should each have just one input—either the entire string input by the user or just a single character from the message—and one output—that string or character having been encrypted or decrypted.

<u>Milestones Towards Completion:</u>

As you have seen, for large projects like this, it is usually very beneficial to set intermediate goals. Consider the following as useful milestones. As you work through them, use `cout` commands often, so that you can see what is happening inside your program as it runs! First, create a program that asks the user for the various inputs, stores them as variables of appropriate type, and then outputs them back to the screen. Second, have your program additionally output to the screen the user's input text one character at a time (hint: if you have a string variable named `str`, then the 1st character is `str[0]`, the 2nd is `str[1]`, the 3rd is `str[2]`, and so on). Third, have your program send the string one character at a time to an encryption or decryption function as appropriate before outputting to the screen. For now, this can be a "dummy" function that just returns the input without changing it. Fourth, have your encryption/decryption functions perform a constant shift up/down on each character, like the shift-2 described above Note: the numeric "value" of each character is given in an ASCII table: A = 65, B = 66, etc. You might want to first make 'A' have a value of 0 so that the mod function works as expected. Fifth, have your functions shift each character by a random amount. The sequence of random integers should come from the random number generator, seeded with the key input from the user.

<u>Extended Work:</u>

Though not required, you may perform one of the following extended work opportunities for 1 bump-up point each:

- The encoding becomes both richer and less obvious when you allow for capital and lowercase letters as well as spaces and other punctuation. Extend the encryption scheme to allow for input and output of these additional characters.
- Can you automate the retrieval and/or sending of encrypted text? For example, instead of prompting the user to input the encrypted text, can the code automatically retrieve it from a Twitter feed, email account, or other source?
- Any other findings or extensions to the method that you found particularly useful, curious, and/or interesting. You may wish to run your ideas by the Director before pursuing this option, just to make sure it is of sufficient complexity to count as extended work.

Debriefing Requirements:

Please report on your work by midnight of the due date listed on Blackboard. The report should be a single pdf document and include all of the sections as given in the example memo on Blackboard. Include with your submission all raw code(s) as separate file(s) in a single submission. Your report should show examples of your codes functionality. This is a report to the head of the Union, not a list of answers. Thus, the entire report should have text that neatly flows from beginning to end. The text should provide the context for any and all figures and tables included in the report. Do not assume that the person reading the report remembers anything about what you have been tasked to perform. In other words, the report should be readable on its own, without someone having read this document.

Notes and Further Reading:

1. This simplest type of encoding is known as the Caesar cipher, named after one of its earliest known users, Julius Caesar. From the very beginnings of recorded civilization to the present day, the development of encryption methods (and their inverse—encryption cracking methods) have been motivated mostly by military and political uses.

2. Sometimes modular arithmetic is called “clock math,” because the minute and second hands are counting mod 60 (0 → 1 → 2 → … → 58 → 59 → 0 → 1 → …).

3. Claude Shannon, perhaps the most important mathematician to study information theory, proved that using a different shift value for each character is, if done correctly, impossible to crack. He showed this while working at Bell Labs during World War II, a war fought and won to a large degree by cryptoanalysts. This manner of encryption was used extensively during the Cold War, including as part of the secure communication link between the heads of state in Moscow and Washington, DC.
   For More: This form of encryption is known as the “one-time pad”, and an excellent introduction to the topic can be found on Wikipedia. Wikipedia’s biography of Claude Shannon is also a very good read.

4. Actually, this is a “pseudo-random number generator”. Nothing a computer does on its own is truly random. There is a mathematical function creating the “random” numbers in a chaotic but fully defined way. Because of this, the encryption you are creating is not actually uncrackable, just pretty difficult. Different C++ development environments may use different generator functions; that is why partners on this assignment that are using different development environments may not be able to decrypt each other’s messages. If you really wanted to make this uncrackable, you’d need a source of truly random numbers that both the sender and receiver can access. You can buy very large databases of actual random numbers, or you can make your own by recording some natural process. Some examples that have been used are the sound of a microphone in the wind, the “static” on an unused radio frequency, and the brightness of a small section of a lava lamp.