Project 2

Due Wednesday, November 13 at 23:59 (11:59 PM)

Introduction to encryption (info)

Messages contain information that we understand. The purpose of encryption is to convert a message (the *plaintext*) into one that does not appear to have information (the *ciphertext*) while still being able to convert that encrypted message back into the original; that is, ideally it should be impossible to differentiate an encrypted message with a randomly generated message.

The Caesar cipher has each letter replaced with the corresponding letter in the alphabet that appears three positions prior to it.

```
HELLOWORLD
EBIILTLOIA
```

To decrypt this ciphertext, it is only necessary to replace each letter with the one three spaces ahead in the alphabet. (Of course, the Romans used the Latin alphabet ABCDEFGHIKLMNOPQRSTVXYZ (no J, U or W—after all, you cannot have a *double*-U if you don't have a U).

At first glance, the ciphertext appears to be a random collection of letters, but if you were to retrieve a large enough ciphertext, you would notice that certain letters appear much more frequently (think `E` which is encrypted as `B`) and certain pairs of letters are commonplace while their reverse is infrequent if not nonexistent (think `TH` which is encrypted as `QE` versus `HT` which is encrypted as `EQ`).

One-time pads (info)

Now, pick a number between 1 and 6, and then roll a die, and add that number that appears onto the number you picked modulo six. For example, you picked the number 4, and the die displayed a 5. In this case, your number is 4 + 5 = 9 = 3. With no additional information, if I told you that the result of the random number plus the original chosen number was 3, do you have any information about the original number? The answer is clearly "no".

Based on this idea, the only secure form of encryption is the use of a one-time-pad. In binary, this works as follows: To encrypt *n* bits,

1. create a sequence of *n* random bits, and
2. determine the ciphertext by taking the exclusive-OR the plaintext and the *n* random bits.

To decrypt the ciphertext, exclusive-OR the ciphertext with the *n* random bits again. For example, to encrypt the string `"HELLO"`, we note that five characters in ASCII occupies 5 x 8 = 40 bits. Thus, we go ahead as follows to create the ciphertext:

```
  0100100001000101010011000100110001001111 ("HELLO" in ASCII)
^ 1010011100010010101101011111110000110111 (40 random bits)
  1110111101010111111110011011000001111000 (the ciphertext)
```

To convert the ciphertext back into the plaintext, we simply need to

```
  11101111010101111111100110110000001111000 (the ciphertext)
^ 10100111000100101011010111111110000110111 (the same 40 random bits)
  01001000010001010100110001001100010011111 ("HELLO" in ASCII)
```

Such an encryption algorithm is 100% safe as long as the random bits are used only once. Another problem is, to decrypt, both the sender and the receiver must have the same 40 random bits. Exchanging such information requires both parties to securely communicate that information. The Soviets used one-time pads for its spies, but one-time pads only work if you use the random bits (or letters) once and only once. Of course, when the Soviets, under pressure to produce a sufficient number of one-time pads, began issuing the same random sequences more than once, the United States was able to exploit this through the Venona project.

Incidentally, producing random characters with no patterns is actually rather difficult. You could find a fair coin and flip it forty times and interpret HEADs as 0 and TAILs as 1; however, the speed of this process is only 1 Hz. Other ways include using radioactive decay (a process currently understood to be truly random) and atmospheric noise (a process so complex that it is for all intents and purposes random).

Your project: stream ciphers

You will author a system that generates a sequence of *pseudo-random* bits, meaning that they appear to be random, but are generated using a mathematical algorithm. Such bits are not truly random, and if the algorithm is poorly chosen, the resulting bits may have many patterns that can be detected and used for decryption.

> "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin."
>
> John von Neumann

The generated sequence of numbers is based on a key, so now if both parties have the same algorithm and the same key, the sender can generate the ciphertext by exclusive-ORing the plaintext with the generated sequence of bits, and the receiver can then recover the plaintext by, once again, exclusive-ORing the plaintext with the identical generated sequence of bits.

You will implement one such common stream cipher.

This cipher requires an 8-byte unsigned integral *key*. This key is to be shared by both the sender and the receiver.

Each message should have a unique key; however, checking to ensure previous keys are not reused is not part of this project. You should, however, try to think about how you could prevent the same key being used twice.

To encrypt, we will start with a state array S of `unsigned char` of size 256, and the values in these entries will be 0, 1, 2, ..., 255, in that order (i.e, $S_k = k$).

We will have to randomize the entries in the array using the given key. To do this, we use the following algorithm:

1. Declare $i \leftarrow 0$ and $j \leftarrow 0$.
2. Looping 256 times, do the following:
    a. Let k be i modulo 64.[1]
    b. Let j be the sum of j plus S_i plus the k^{th} bit of the key (0 being the least-significant bit and 63 being the most-significant bit) modulo 256.
    c. Swap S_i and S_j.
    d. Increment i modulo 256.

This scrambles the entries S_0 through S_{255}. We are now ready to begin encrypting the plaintext. Continue to use the same i and j for the balance of the algorithm (that is, do not reset them).[2]

For every byte in the plaintext, we will exclusive-OR it with the value R found by the following algorithm:

1. Increment i modulo 256.
2. Let j be the sum of j and S_i modulo 256.
3. Swap S_i and S_j.
4. Let r be the sum of S_i and S_j modulo 256.
5. Let R be S_r.

You must repeat this sequence of instructions for every byte in the plaintext.[3] This lets each byte be exclusive-ORed with a pseudo-randomly chosen value between `0b00000000` and `0b11111111`.

When the receiver gets the ciphertext, the receiver will exclusive-OR each byte again with the same sequence of R values (initialized the same way, too), and this will recover the plaintext.

ASCII armour

There is a small problem, however. When you encrypt an ASCII character, the resulting byte could be any value between 0 and 255; however, it is not possible to print to the screen 33 of the 128 ASCII characters, and the remaining 128 values from 128 to 255 may not also render correctly, either. Consequently, it is often necessary to convert arbitrary bytes to printable characters. This is called *binary-to-text encoding* and also called *ASCII armour*. We will convert a set of four bytes into five printable characters as follows.

First, recall how to convert a number into base 16:

1. While the number is not zero:
    a. Find the remainder of n modulo 16, and this value between 0 to 15 is represented as a digit in 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f.

---

[1] A calculation of an expression modulo n is the same as the calculation of the expression % n, so calculating something modulo 256 is the same as performing the calculation then evaluating that expression % 256.
[2] First of four changes.
[3] Some students mistakenly believed that all characters would be encoded with the same R; however, this would result in an encrypted text that is essentially equivalent to the Caesar cipher.

b. Divide the number by 16.

For example, 105598 converted to hexadecimal may be done as follows:

n	Remainder modulo 16	Representation	n/16
105598	14	e	6599
6599	7	7	412
412	12	c	25
25	9	9	1
1	1	1	0

Thus, 105598 in hexadecimal is 19c7e.

Your 32-bit unsigned integer has a value between 0 and $2^{32} - 1 = 4294967295$. You will notice that $85^5 - 1 = 4437053125 > 2^{32} - 1$. Therefore, every 32-bit number can be interpreted as a five-digit base 85 number. You will perform a similar action but your interpretation of the base-85 number will be `'!'` plus the remainder.

> Question: Why are we using `'!'`? Look up the location and value of the exclamation mark in an ASCII table.

For example, if the four bytes are

```
array[4*k]     == 0b10110101
array[4*k + 1] == 0b01110100
array[4*k + 2] == 0b01010101
array[4*k + 3] == 0b10011110
```

so this represents the binary number:

```
10110101011101000101010110011110
```

this has the integral value of 3044300190. Using the above algorithm, this would produce the five values 30, 36, 11, 27, 58 which produces the string of five printable characters `"[<,E?"` (as 30 is the least significant value, so it comes last and `'!' + 30` is the question mark).

Because this must be a printable string, your array size will be a multiple of five plus 1, where the last character will be the null character `'\0'`.

To go back from hexadecimal, you must take the representation and determine the remainder and multiply that by the appropriate power of 16.

For example, 19c7e converted back to decimal is:

Position (k)	Representation	As a value from 0 to 15	16^k
0	e	14	$16^0 =$ 1
1	7	7	$16^1 =$ 16
2	c	12	$16^2 =$ 256
3	9	9	$16^3 =$ 4096
4	1	1	$16^4 =$ 65536

Thus, 19c7e is 14 + 7 × 16 + 12×256 + 9×4096 + 1×65536 = 105598. You will have to do this but with base 85 instead of base 16.

Problem: This algorithm requires the array of bytes to be a multiple of four; however, the message we are encoding may be arbitrarily large. Thus, if the number of bytes being encoded is not a multiple of four, the last bytes should be appended with null characters `'\0'` to get a multiple of four bytes. Of course, you cannot modify the string that is passed, so your code will have to deal with this as a special case.

For example, if you were encoding the string `"Code"`, this occupies five bytes with the last being `'\0'`. Thus, the four characters `'C'` through `'e'` will be encoded and converted into five bytes of ASCII armor. The returned string will be of capacity six, with the last character being a `'\0'` that you placed there (because the ASCII armor characters are between '!' and 'u', none of the other characters can be the null character). If, however, you encode the string `"Coding"`, the seventh character is `'\0'`, but the first six characters are not a multiple of four, so you would actually encode the eight characters `"Coding\0\0"` and then convert this to ten characters of ASCII armor plus an eleventh null character.[4]

Deliverables

You will implement two functions, one that encodes the plaintext message and converts the ciphertext to ASCII armour. The next decodes the ASCII armour back into the ciphertext and then decodes the ciphertext back into the plaintext. The output of each function should be printable.

The plaintext string `plaintext` is a character array of n printable characters of size $n + 1$ where `plaintext[n] == '\0'`. Your encode algorithm will encrypt the n printable characters but not encrypt the trailing null character. You will return a character array of size $5\left\lceil \frac{n}{4} \right\rceil + 1$ where the last character, again, will be the null character.

The ciphertext string `ciphertext` will be an array that has a size of the form $5m + 1$ where the last character is the null character. You will return an array of size $4m + 1$ where the first $4m$ characters are the decrypted characters and the last character is, again, the null character.

Your function declarations are:

```
char *encode( char  *plaintext, unsigned long key );
char *decode( char *cihpertext, unsigned long key );
```

Critical information

The type `char` is signed, and consequently, if some of your encrypted values end up with a leading 1, the compiler will interpret it as a negative number. If you now using that in an arithmetic calculation, it will immediately converted it to a negative integer. Thus, the negative char `0b10000101` will first be converted to `0b11111111111111111111111110000101`, and if you use this in your calculations, you are likely to get the wrong answer. When you are calculating in your ASCII armour sections of code and need to read the entries as parts of a base 85 or base 256 number, you should use `static_cast<unsigned char>( array_name[k] )` instead of just

[4] This clarifies the statement in the previous paragraph; however, the succinct summary in the previous paragraph may be easily misinterpreted.

`array_name[k]`. This tells the compiler to interpret the signed character as if it is an unsigned character. Thus, if the character array has the entry `0b11111011`, normally, if you added this to an `unsigned long`, because the character is negative, it would subtract five from the `unsigned long`, but you will want it to add 251. The static casting ensures that this is what happens.

To see this in action, consider executing the following code:

```
#include <iostream>
// We need this library for std::isprint(...)
#include <cctype>

int main() {
        unsigned long value{1532153553};

        // Equal to 241 as an unsigned char
        //  - this is not a printable character
        char ch{-15};

        // Confirm that 'ch' is not assigned a printable character
        //  - this should print 0
        std::cout << std::isprint( ch ) << std::endl;
        std::cout << (value + ch) << std::endl;
        std::cout << (value + static_cast<unsigned char>( ch )) << std::endl;

        // Print 'ch' cast as an unsigned long
        //  - this prints 18446744073709551601
        std::cout << static_cast<unsigned long>( ch ) << std::endl;

        // Print 'ch' first cast as an unsigned char, and then cast as an
        // unsigned long.
        //  - this prints 241
        std::cout << static_cast<unsigned long>(
            static_cast<unsigned char>( ch )
        ) << std::endl;

        return 0;
}
```

The function `std::isprint(…)` returns `0` if the character is not printable, and a non-zero value if it is printable.

Marmoset testing

Be sure to wrap your main function with these pre-processor directives for testing:

```
#ifndef MARMOSET_TESTING
int main() {
    // Your main function
    return 0;
}
#endif
```

Helper functions

It may be useful if you authored a function:

```
bool is_valid_ciphertext( char *str );
```

which checks that

1. the array is of size $5m + 1$ with the last character being the null character and
2. the first $5m$ characters are all between `'!'` and `'u'`, inclusive.

It may also useful if you authored a function:

```
bool is_valid_plaintext( char *str );
```

which checks for the decrypted message that all the characters before the first `'\0'` are either printable characters or printable whitespace (use `std::isprint( ch ) || std::isspace( ch )` from the standard library `cctype`).

It may be useful to have

```
assert( is_valid_ciphertext( return_array ) );
return return_array;
```

and

```
assert( is_valid_plaintext( return_array ) );
return return_array;
```

as the last two lines of the functions you are authoring.

Sample project and output

The following is the sample code the author used to test the author's implementation.

```
#ifndef MARMOSET_TESTING
int main() {
        char str0[]{ "Hello world!" };
        char str1[]{ "A Elbereth Gilthoniel\nsilivren penna miriel\n"
                     "o menel aglar elenath!\nNa-chaered palan-diriel\n"
                     "o galadhremmin ennorath,\nFanuilos, le linnathon\n"
                     "nef aear, si nef aearon!" };  // [1]

        std::cout << "\"" << str0 << "\"" << std::endl;

        char *ciphertext{ encode( str0, 51323 ) };

        std::cout << "\"" << ciphertext << "\"" << std::endl;

        char *plaintext{ decode( ciphertext, 51323 ) };

        std::cout << "\"" << plaintext << "\"" << std::endl;

        delete[] plaintext;
        plaintext = nullptr;
        delete[] ciphertext;
        ciphertext = nullptr;

        std::cout << "\"" << str1 << "\"" << std::endl;

        ciphertext = encode( str1, 51323 );

        std::cout << "\"" << ciphertext << "\"" << std::endl;

        plaintext = decode( ciphertext, 51323 );

        std::cout << "\"" << plaintext << "\"" << std::endl;

        delete[] plaintext;
        plaintext = nullptr;
        delete[] ciphertext;
        ciphertext = nullptr;

        return 0;
}
#endif
```

The output, where the double quotes are explicitly added to the strings, is here. You will observe that first is the original string, then the encrypted ASCII armour and the decrypted output.

```
"Hello world!"
"l=]V]&9!>trOu*9"[5]
"Hello world!"
"A Elbereth Gilthoniel
silivren penna miriel
o menel aglar elenath!
Na-chaered palan-diriel
o galadhremmin ennorath,
Fanuilos, le linnathon
nef aear, si nef aearon!"
"j!*1O*@cZAqmmIG`M2\X]I7]QP,<i2UN&gA'/bW4(-
5h$_t&Wj(h\d2Lung$ZrN4QGsFQ:08DNrc"hm3g2);em[O.p6uG6"\F1h?%Xq]LV.V/FU_`Wmf@5s
^O$L%!@[[4m[Pm&M+7^lnpASFK!Y4ep5I]Reg5N09^e7.dc_AJIW>Ud/CpRj(juZ'"1BX>1b2R3-
SVr0uUCt_("[6]
"A Elbereth Gilthoniel
silivren penna miriel
o menel aglar elenath!
Na-chaered palan-diriel
o galadhremmin ennorath,
Fanuilos, le linnathon
nef aear, si nef aearon!"
```

For the key used in this example, the entries of S_k for $k = 0, \ldots, 255$ are

```
 41   2   0 143 165  17  52  28 238  15  24  44  72  69 100 116
 99 137 155 174 194 141 219  48 162  35 245  64 151 124  92 185
217 223  63  53 145 134 104 235 195  93 220  13  56 138 184 218
 54  96 102 166 206 216 242  58 114 171  62 240  70 110  66 121
186   8 128 212  29 179  84  87 204 197   7  57 247 153  31 113
193 167  38   1  76  32  59 202  19 187  73 156  49  40 135 164
  9 146 108 132  20  71 233 173  46 125 139 119  51 133 252 211
222 227  94 142  45 126 161 246  43 251 111 109 236 250 229  39
208  60 147 149  47 150 163 129 160  18  12 203   5 201 192   6
 78  26  86  90 169   3 158  34 180 168  79 191 210 182  97 243
213   4  55  89 209  11 170  16 249 228  80  91  23  85  37  83
123 224 207  27 221  81 248 255 254  22 253 190 136  68 118 205
 25 103 152 215 200 241 188  65 237 148  74 154 226 131 225 231
117 140 177 101 130 189 159 127  82  30 176  67 172 106 181 105
230  21 232 214 144 115 112 234  14 122 244 175  77  36 107  10
178 157  42  33  98  61 196 183 239 199  75  95 120  88 198  50[7]
```

[5] Second of four changes.
[6] Third of four changes.
[7] Fourth of four changes.

Please note, it is pure coincidence that the first few entries contain a '2' and a '0'. In order to print these out, it is necessary to use `static_cast<unsigned int>`; otherwise `std::cout` will attempt to print them out as characters, as many of these values are unprintable when interpreted as characters.

The author's mistake

To be fair, the reason a mistake occurred in my original code, and I will explain that here. In order to determine if the k^{th} bit of the key (a long integer) is 1, I did a bitwise AND of `(1 << k)` and the key. The problem is, the compiler interpreted the "1" as an integer, not a long, and so if $k > 31$, then `(1 << k)` ended up equaling zero. Instead, you must tell the compiler "treat this literal integer as a long and not an int." To do this, you can use `(1L << k)`. The "`L`" appended to the literal says to the compiler, "this literal is 64 bits." The following code demonstrates this:

```
#include <iostream>

int main();

int main() {
        std::cout << (1 << 30) << std::endl; // int
        std::cout << (1 << 31) << std::endl;
        std::cout << (1 << 32) << std::endl;

        std::cout << (1U << 30) << std::endl; // unsigned int
        std::cout << (1U << 31) << std::endl;
        std::cout << (1U << 32) << std::endl;

        std::cout << ( 1L << 30) << std::endl; // long
        std::cout << ( 1L << 31) << std::endl;
        std::cout << ( 1L << 32) << std::endl;
        std::cout << ( 1L << 63) << std::endl;
        std::cout << (1UL << 63) << std::endl; // unsigned long

        return 0;
}
```

When you run this code, you should ask yourself why the various outputs are what they are. For this project, does it matter if you use `1L` or `1UL`?[8]

References

[1] J. R. R. Tolkien, "The Fellowship of the Ring", George Allen an Unwin, 1954.

[8] This section is an explanation of the error.