Elementary Encraption 1
Implementing a Vigenère Cipher in C

Dr. Peter A. H. Peterson <pahp@d.umn.edu>
University of Minnesota Duluth

Overview

In this exercise, you will implement a version of the Vigenère Cipher, using the source code of a basic Caesar Cipher as a starting point. A Vigenère Cipher uses a cycling key that determines the substitution shift value for each symbol in the input.

Requirements

Code and Compilation

1. Your submission will be electronically graded for correctness and uniqueness. Therefore, your code must successfully compile on the MWAH 187 machines running Ubuntu.
2. Your code **must not** use any of the Banned Functions listed in the Build Requirements document or it will be scored a **zero (0)**.
3. You need to create a single C source file capable of generating object files for both an encoder and decoder. Your code will be compiled using the commands
   `gcc vigenere.c -o encoder -D MODE=ENCODE` and
   `gcc vigenere.c -o decoder -D MODE=DECODE`
   See the Build Requirements document for instructions to do this.
4. Your code should not use any libraries explicitly intended for encryption.
5. You should **NOT** use other online examples of the Vigenère Cipher as a coding template; if you do not understand how the Vigenère Cipher works, talk to your instructor or TA.
6. Keys **must** be read in from a file (see Encoder and Decoder, below).
7. Keys will be between 1 and 128 bytes long. **Additional bytes should be ignored.**
8. Keys **may include** non-printable bytes (i.e., non-ASCII symbols). Methods for testing this are described in the sample vectors file (see below).
9. Input will not be limited to text only (i.e., test using binary data like applications or images).
10. Input will not be limited to any particular length (i.e., test using large files).

Encoder and Decoder

Your encoder and decoder must take the following parameters as input:

1. A path to a file containing a key composed of a string of bytes (e.g., a file `keyfile` containing the characters “monkey toast”).
2. A path to an input file, e.g., `/tmp/input`
3. A path to an output file, e.g., `/tmp/output`

Upon running '`encoder /path/to/keyfile /path/to/plaintext /path/to/ciphertext`', `ciphertext` should contain the contents of input encoded using the key. Upon running '`decoder /path/to/keyfile /path/to/ciphertext /path/to/decoded`', `decoded` should contain the contents of `ciphertext` as decoded using the key. If the same key was used in the preceeding example, the contents of `decoded` should **perfectly match** – byte for byte – the contents of `plaintext`.

Sample Vectors

Sample vectors (i.e., sample keys and input along with correctly encrypted and decrypted data) along with the starting code `caesar.c` are provided at

http://www.d.umn.edu/~pahp/labs/vigenere-vectors.tar.gz

for correctness testing. See the included `README.txt` for more information. Grading will use both the sample vectors **and new untested vectors**. Therefore, you should test your code against the sample vectors and various other types of input and keys that you choose.

Submission

Submit a tarball `vigenere.tar.gz` containing a directory `vigenere` with the file `vigenere.c`. To do this properly:

1. Make the directory `vigenere` (`mkdir vigenere`)
2. Move (`mv`) or copy (`cp`) your source file into `vigenere/`.
3. In the parent directory of `vigenere/` (e.g., your home directory), execute `tar cvzf vigenere.tar.gz vignere/`. This will make a gzipped tarball named `vigenere.tar.gz` containing the directory `vigenere/`.

Submit this tarball to your instructor by the specified due date.

Appendix

How do I read a file (e.g., a key) into a character array byte by byte?

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main(int argc, char *argv[]) {

    char *filepath = argv[1];   /* file path is the first argument */
    char data[128];             /* we will only copy up to 128 bytes into the array */
    unsigned int length = 0;    /* length of the data */
    char symbol;                /* we'll read characters into this variable */
    FILE *INPUT;                /* a file pointer */

    if ((INPUT = fopen(filepath, "r")) == NULL) {
        printf("Problem opening key file '%s'; errno: %d\n", filepath, errno);
        exit(1);
    }

    while (length < (128) && (symbol = fgetc(INPUT)) != EOF){
        data[length] = symbol;
        length++;
    }

    fclose(INPUT);

    /* print out the output character by character */
    int i;
    for (i = 0; i < length; i++) {
        printf("%c\n", data[i]);
    }

    exit(0);
}
```

Use of the Key

The encryption key is an array of characters – 8-bit bytes. Arrays in C are 0-offset. That is, `array[0]` is the first character in the array. After the above code executes, the array `data` contains up to the first 128 bytes in the file located at `filepath` and the variable `length` contains the number of bytes in the array. By cycling through the letters in the key (returning to zero after using the last letter), you can repeatedly use the same copy of the array for the key. (If your solution requires you to create a large array with the key repeated many times, rethink your solution.)

Shifting and Characters as Integers

Each byte is a unique sequence of 8 bits. As a result, there are 256 unique bytes, which can be directly treated as integers in C. This means that, to perform a shift, you can directly add a character of input with a character of the key (using the + operator). You should ensure that the addition is performed "modulo 256" – that is, modulo the number of unique 8-bit bytes so that the substitution "wraps around" if the resulting value is greater than 255 or less than 0.

Please note that the C standard does not specify whether type `char` is an unsigned 8-bit integer (0-255) or a signed 8-bit integer (-128-127). On our Ubuntu systems, type `char` is signed, which means that – depending on how you use the key – characters of the key may be interpreted as *negative numbers*. On the other hand, the return type of `fgetc()` (which we use to read characters from the input) is defined to be an `unsigned char` cast to an `int`, which means that values from `fgetc()` will always be between 0-255 (see `man fgetc` for more information). Your code will need to cope with this. One method is to cast the output of `fgetc` to a char before storing it in a character array, like this:

```
array[i] = (char)x
```

The character set in use will determine which unique bytes (or integers) map to which characters and vice versa. Because of this, your code should not depend on any character mapping to any particular integer value. For example, **do not assume** that 'A' == 0 as in the Vigenere Tableau we used in class, but simply treat the characters as integers as described above. Additionally, since keys may contain non-printable bytes, you should expect shifts to be performed on an 8-bit basis (256 values – 0 to 255), not the simple "26 capital letters only" shift we used in class. In other words, instead of a set of 26 shifted alphabets in the tableau, your code will effectively use a set of 256 shifted "alphabets" of all unique 8-bit bytes. Again, you should not need to create a static tableau to complete the assignment – just add the bytes together and make sure the values "wrap around" (i.e., fall between 0-255).

Concrete Examples

The character '!' is decimal value 33 which is hexadecimal 0x21. The character 'A' is decimal 65, hexadecimal 0x41. Shifting "! by A" can be computed by adding '!' (33) to 'A' (65), which equals decimal 98 (hexadecimal 62). In ASCII, byte 98 corresponds to 'b'. In other words, if you had the key '!' and the plaintext 'A', the ciphertext would be 'B'.

Banned Functions and Their Safer Replacements

Certain legacy C string-handling functions are unsafe because they consume or produce input without any bounds checking. As a result, these functions can cause segmentation faults, or worse, buffer overflows leading to segfaults, data corruption or even remote execution. **Your code must not use any of these banned functions or it will be graded as a zero (0)!** Fortunately, all but one of the following functions have safer replacements that include a bounding value.

Banned Function	Safer Replacement
Instead of using:	… you should use:
strcpy	strncpy
strcat	strncat
strtok	N/A
sprintf	snprintf
vsprintf	vsnprintf
gets	fgets
strlen	strnlen

Source: http://stackoverflow.com/questions/6747995/a-complete-list-of-unsafe-string-handling-functions-and-their-safer-replacements (2015/09/14)

How to Pass Build-Time Parameters To GCC

It can be useful for one piece of code to be able to generate binaries with distinct behavior (e.g., an encoder and a decoder). In this way, one can reuse code for file I/O and other tasks, while performing the necessary tasks as required. This can easily be done by setting up your code with the appropriate macro definitions, and passing compile-time defines to GCC. For example, the following code:

```
1: #define ENCODE 0
2: #define DECODE 1
3: #ifndef MODE
4:    #define MODE ENCODE
5: #endif
```

… defines values for ENCODE and DECODE and sets MODE to ENCODE if MODE is not already defined (`#ifndef` means "if not defined"). When compiling from the command line, one can specify compile-time defines using the -D switch and the argument MODE=ENCODE or MODE=DECODE, e.g.: `gcc cipher.c -o cipher -D MODE=DECODE`

Since MODE was defined on the command line, it will not be changed at line 4. Then, in your code, you can test the value of MODE using normal conditionals, and behave accordingly, e.g.:

```
/* start of program */
if (MODE == ENCODE) {
   /* encode here */
} else {
   /* decode here */
}
/* rest of program */
```

The sample code `caesar.c` already includes this code.

How do I create files with nonprintable bytes (e.g., for testing purposes)?

The easiest way I know is to use the Hexedit hexadecimal text editor. If you don't have the command `hexedit` available to you, you can install it by executing `sudo apt-get install hexedit`.

To create a new empty file, execute `touch filename`. Then, edit it with `hexedit` by executing `hexedit filename`.

In Hexedit, the left side of the screen is for hexadecimal data, which is displayed in ASCII on the right side of the screen. Non-printable characters are represented using a period (.). Pressing Tab shifts you from the left and right side of the screen. On the left side, you can enter arbitrary hexadecimal values, many of which will result in nonprintable data. Hexadecimal values are from 0-F, i.e., 0-9 and A-F, representing all 16 four-bit bytes. For example, the string “deadbeef” is valid hexadecimal data and is also non-printable.

When you are finished editing your file, press control-x to exit, whereupon Hexedit will ask you if you want to save your changes. For more information, see `man hexedit`.

How can I see how/where my encrypted or decrypted output differs from the correct versions?

Executing `diff file1 file2` will show lines that differ between `file1` and `file2.` Diff will consider some files to be binary (if they have certain non-printable characters) in which case it will only report whether the files are different. This can be annoying, since encrypted data will often appear to be “binary”. It's more helpful to be able to look at the exact differences in hexadecimal, which can help to indicate why and where a program “went off the rails.”

The easiest way to look at the difference between two files in hexadecimal is to use the utility `hexdiff`. You can install it by executing `sudo apt-get install hexdiff`. Then, you can hexdiff two files by executing `hexdiff file1 file2`. Quit `hexdiff` by pressing q.

This will show precisely where the differences are in the two files. Since the sample vectors and “correct” encryptions are in the sample vectors directory, you can use this to compare your program's output to known correct answers. This can be invaluable for debugging.

How can I perform decimal and hexadecimal arithmetic on Ubuntu?

Open the GNOME calculator. Under the Mode menu, select Programming. Then, choose betweeen Decimal mode (control-d) or Hexadecimal mode (control-h). You can convert answers between decimal and hexadecimal by using the hotkeys.