ENGG1003 - Programming Assignment 1
English Text Ciphers

Brenton Schulz

Due date: 6pm Monday of Week 8
Submission: Upload your final `.c` file to Blackboard as an assignment submission
Marking: During your Week 8 lab
Weighting: 20% of your final grade

1 Introduction

A *cipher* is an algorithm which encrypts a message into *cipher text* so that it can be safely transmitted without an eavesdropper being able to (easily) read it. For the purposes of this assignment the message and ciphertext will be stored as strings of ASCII characters.

Cipher algorithms always perform two tasks: encryption and decryption. The encryption process takes a "message" and "key" as inputs and produces cipher text. The decryption process performs the reverse: it turns cipher text and the key back into the original message.

The exact details of how the key and message are processed to produce the cipher text vary between algorithms. However, the general principle is that algorithms employ a mathematical function which is "easy" to calculate with the key but "difficult" to invert without it. A rigorous discussion of modern cryptographic techniques can be found in the excellent (but *highly* mathematical) text, An Introduction to Mathematical Cryptography (available in the Auchmuty library).

This assignment will cover three cipher algorithms:

1. The classical "rail-fence" cipher
2. A modified "2-level" rail-fence cipher
3. The substitution cipher

Your task is to write a C program which performs the following broad tasks:

- Encryption of a message using the classical rail-fence cipher algorithm
- Encryption of a message using the 2-level rail-fence cipher
- Decryption of a message using the 2-level rail-fence cipher
- (HD level) Partial decryption of a block of cipher text encrypted with a substitution cipher *without the key*

2 Cipher Algorithms

2.1 2-Level Rail-fence Cipher

This section describes a *transposition* cipher algorithm which has been created for this project. To my knowledge code for this cipher does not commonly exist on the Internet.

A transposition cipher is one which creates ciphertext by re-ordering the characters in the message. This is similar to an anagram except the output cipher text looks like random characters, not a new word.

The algorithm documented here is an extension of the classical rail-fence cipher. You should read about it here and here before continuing. You can view C code for this cipher here. Your implementation should be done from scratch and not simply re-use existing code.

The encryption algorithm involves two broad steps:

1. Writing the message on a 2D grid where each row is called a "rail". The message "zig-zag"s between the top and bottom rails, one message character per column. The height (number of rows) is the "key".
2. The cipher text is created by reading the characters off the grid in a top-to-bottom-left-to-right sequence.

Where the classical rail fence cipher has a "key" which is a single integer, A, your algorithm will use two integers A, and B with $A > B$ and $B > 1$; alternating between them when writing out the message on the fence rails. This algorithm reduces down to the classical rail fence cipher if $A = B$.

2.1.1 General Example

If the message characters are denoted `A, B, C, D, ... , etc` and the cipher key is chosen as $A = 4$ and $B = 2$ then the message characters would be written on the rails as:

```
A-------I-------Q
-B-----H-J-----P-
--C-E-G---K-M-O--
---D-F-----L-N---
```

Observe the general pattern in the algorithm:

1. Down A rails (to `D`)
2. Up to a peak of B rails (up 1 to `E`)
3. Down $B - 1$ rails back to the bottom (down 1 to `F`)
4. Up to a peak of A rails (up 3 to `I`)
5. Repeat until whole message has been written to the grid

The ciphertext can then be read off left-to-right-top-to-bottom:

`A, I, Q, B, H, J, P, C, E, G, K, M, O, D, F, L, N`

Optional For the purposes of this assignment your message may be "padded" with random characters so that it completes a full cycle on the rails. ie: It always finishes on the top rail. Note that ASCII lower case letters take the values 97 to 122 so a random lower case letter can be generated with `rand()%26+97`. Similarly, a random upper case letter is generated with `rand()%26+65`. Note that a N cycles require $N \times (2A + 2B - 4) + 1$ letters.

2.1.2 Worked Example

Encrypting the message "`Move B company 1km West.`" with $A = 5$ and $B = 3$:

NB: "-" indicates an unused spot on the rails; " " indicates a space in the message.

NB 2: The final "q" is a random character padding so that the algorithm finishes on the top rail.

```
M----------n----------q
-o---------a-y---------.-
--v--- ---p--- --- ---t--
---e-b-c-m-----1-m-W-s---
---- ---o-------k---e----
```

Listing 1: The filled in rails.

The cipher text is then:

```
Mnqoay.v p  tebcm1mWs oke
```

2.1.3 Decryption

Designing and implementing the decyption algorithm will be mostly up to you to work out. However, to get you started, one decryption method could perform the following general steps:

1. Re-create the 2D grid from the cipher text and key (eg: Listing 1)
2. Read off the original message by performing the encryption algorithm again, except reading from the rails instead of writing

Given the key numbers A and B and the encrypted cipher text the first decryption step requires you to rebuild the grid show in Listing 1.

There are many methods to do this. One option is to observe that you need to know which grid location each letter goes in and realising that you have the means to work this out - the encryption algorithm does it! The encryption algorithm writes message letters to this grid so you can repeat that algorithm and, say, write a "1" in all locations which have a letter written to them and "-" (or, in C, a zero) otherwise.

Given $A = 4$ and $B = 2$ from the previous example the grid will then look something like this:

```
1----------1----------1
-1---------1-1---------1-
--1---1---1---1---1---1--
---1-1-1-1-----1-1-1-1---
----1---1-------1---1----
```

Then you can loop over the grid left-to-right-top-to-bottom and write cipher text characters to all grids with a "1" to get back to what is seen in Listing 1.

Once that is done the original "zig-zag" algorithm can again be applied to read off the message in the correct order or you can observe that there is only one letter per column so you can read all non-zero values left to right.

2.1.4 C Implementation Hints

In no particular order:

- The whole message can be stored in an array
- The grid (ie: rail fences) can be a 2D array
- When encrypting you can remove the need for a 2D array if you calculate 1D array indices in the message instead of doing it "graphically" with a 2D array

2.2 Substitution Cipher

A substitution cipher encrypts a text message by replacing each of the 26 message letters to an encrypted letter. Each letter is only chosen once. The "key" is therefore knowledge of all 26 different substitutions. The number of possible key combinations is: $26! \approx 4 \times 10^{26}$.

Example with a substitution chosen from the qwerty keyboard layout:

```
Message letter:      ABCDEFGHIJKLMNOPQRSTUVWXYZ
Cipher text letter:  QWERTYUIOPASDFGHJKLZXCVBNM
```

As before, a message is encrypted by substituting each letter in the top row with the one below it:

```
Message:      PLEASE GET MILK AT THE SHOPS
Cipher text : HSTQLT UTZ DOSA QZ ZIT LIGHL
```

There is no neat algebraic method to write the encryption function. It is effectively a "look-up-table" where each letter, x_n, becomes a different letter, y_n based on a fixed substitution rule.

2.3 Substitution Cipher Attacks

With 26! different encryption keys a brute force attack is not possible. Even when testing a million combinations *per second* it would take almost 1000 times longer than the *age of the universe* to test every encryption key. Aside: the German Enigma machine (used in WWII) used a algorithm closely related to the substitution cipher and used a new encryption key every day. Even with modern computers a naive brute force attack would not have been possible. You can read about the Enigma decryption efforts on Wikipedia.

Decryption must therefore rely on extra information which reduces the key search space. It is possible, for example, to perform a statistical analysis of the cipher text to estimate which letters were used for 'e', 't', 'a', 'z', etc.

If the message is assumed to be normal English text then other assumptions can be made. Any single letter word is likely to be 'a' or 'i', the latter being easier to spot if the message is case sensitive. Likewise, the most common three letter word is likely to be 'the'.

By making educated guesses about the most common short words and letters a subset of the encryption key can be deduced. This knowledge, coupled with a dictionary, and some kind of "spell checker" algorithm such as *Levenshtein distance*, can then be used in an attempt to work out further letter substitutions.

The WWII efforts to decrypt Enigma relied heavily on *cribs*. These were known, or assumed, sections of plain text for a given encrypted message. For example, many German messages would include regular weather reports in the same format and would reveal several of the day's letter substitutions.

3 Programming Task

Write a C program which performs the following tasks, selectable by the user:

1. Encryption of a message using the classical rail-fence cipher algorithm
2. Encryption of a message using the 2-level rail-fence cipher
3. Decryption of a message using the 2-level rail-fence cipher
4. (D/HD level) Decryption of a block of text encrypted with a substitution cipher but *without being provided the key.* This cipher text will be released at the same time as the project; you have about a month to analyse it. Your program must perform all analysis calculations (eg: letter or word statistics and dictionary comparisons) from scratch.
5. (HD level) Decryption of a **previously unseen** block of cipher text encrypted with a substitution cipher *without the key.* This block of text will be provided at marking time.

All tasks you attempt needed to be coded in a **single .c file** and some kind of user interface designed which can select between them.

Your code **must** include the following functions:

```
void railFence(char *message, char *cipherText, int length, int A);
```

```
void railFence2(char *message, char *cipherText, int length, int A, int B,
    int dir);
```

The function `railFence2()` accepts an argument `dir` which controls whether the function performs encryption or decryption. When `dir=0` the function should encrypt (ie: read from `message[]` and write to `cipherText[]` and when `dir=1` it should decrypt (ie: read from `cipherText[]` and write to `message[]`). The function should perform any required padding to `message[]` by either declaring a new variable of an appropriately larger size and copying the message over or only creating padding characters when required. Just make sure you don't try to access `message[]` at an index outside its boundary.

The functions above are allowed to call other functions you write. If you do not attempt a feature (eg: decryption of the 2-level rail-fence algorithm) the function prototypes must still be defined as they are above (eg: you must not remove the `dir` variable).

Your own functions are strongly recommended for the decryption of substitution ciphers but they may be of your own design.

3.1 User Interface Specification

3.1.1 Inputs

For all data inputs (message, keys, cipher text, algorithm selection, etc) you may choose from the following methods (**NB:** methods are *not* worth equal marks):

- Hard-coded variable initialisation (eg: `char m = "This is a test message";`)
- Read from `stdin`
- Read from a file using the C standard file I/O library

It is *strongly* recommended that variable initialisation be used while debugging your programs and file I/O only be implemented as a final feature.

If file I/O is implemented the file name may be hard coded. You are permitted to read the file name(s) as command line arguments but, at time of writing, this is beyond the scope of ENGG1003 and will not attract extra marks.

When being graded, you should ensure that multiple inputs can be tested quickly. This is especially true for the decryption tasks which will be tested with multiple blocks of cipher text.

3.1.2 Message Specification

Different rules on the input specification will apply depending on whether your program reads from `stdin` or from a file. The crucial difference being that when reading from a file you are able to determine the file length before reading data but when reading from `stdin` this is not possible.

Therefore, please observe the following specifications:

- When reading from `stdin`:

  For maximum marks your program should be able to encrypt a message up to 8 kB (8192 bytes) long. It should also, however, terminate reading the message when a termination character is read. As such, the following rules apply to the message input when reading from `stdin`:

  1. The ~ character must not appear in the message
  2. A single ~ character indicates the end of the message
  3. A newline character in the message should read into the program and be encrypted just like any other allowed character.

  For debugging, new lines can be entered into initialised strings with `\n`.

  A ~ character is typed by holding `shift` and pressing the key to the left of "1" on a standard US keyboard. Among gamers it is colloquially known as the "console key" as it allows access to the command console in many FPS games since Quake 1.

- When reading from a file:

  If you use file I/O the length of the message stored in the file **must** be measured prior to declaring any arrays which store the message or which are required for encryption. As such, projects using file I/O should support message lengths which are only be limited by the amount of RAM the executing machine has installed.

Remember that arrays in C (since C99) can be declared with a size stored in an integer *variable*. ie: The size of the array does not need to be known until the line on which it is declared.

3.1.3 `stdin` Functions

There are many functions in the C standard library which allow for reading from `stdin`. As such, there are many solutions to how the specification above can be implemented.

Note that `scanf()`, as covered in lectures with the `%s` format specifier, only reads one "word" (it stops at whitespace).

Other potentially useful options which you will need to research (ie: read documentation on) include:

```
fgetc()
fgets()
getchar()
getline()
fscanf()
```

Some of the above are also suitable for reading from files.

You are, under no circumstances, allowed to use `gets()`. This function is a **major** security risk and should never be used in new code. Using `gets()` will incur a marks penalty (see Section 9).

3.1.4 Outputs

All program output should be sent to `stdout`. You are encouraged to also use file I/O for outputs but, due to time constraints while marking, all output should be sent to *both* the file and `stdout`.

3.1.5 Task Selection

It should be easy for a *demonstrator* to quickly test functionality of each task listed in Section 3. It is up to you to decide exactly how this is implemented but the following options are recommended (NB: not all options are worth equal marks):

- Hard code an initialised integer variable which selects between the different tasks.
- Take user input from `stdin` to select each task in a menu system
- (Advanced) Define the task as part of a *header* inside an input file which contains the message and key (or key and cipher text, or just cipher text). The header could be something like:
  - A single integer placed on the first line to indicate the task to be performed, followed by
  - A 2nd, optional, line which contains a key (perhaps start the line with a "weird" character like # to indicate that it is a key), followed by
  - The message or cipher text
- (Advanced-Beyond ENGG1003) Use command line arguments

Each task should be implemented as a different function. Beyond that you are free to choose how many functions are implemented.

All tasks must be accessible from running a **single** `.c` file. The GitHub repository should contain this `.c` file and any other required files (eg: input text).

3.2 Message Text Specification

The substitution cipher is only defined for letters. This leaves a few potential ambiguities:

- What should be done to punctuation and white space?
- What about numerals?
- Should upper and lower case letters be handled differently?

For the purposes of this assignment the following rules will apply when the encrypted cipher text is created:

1. All letters in the substitution cipher's cipher text will be UPPER CASE
2. Any character which is not a letter will be copied to the output unmodified

3.3 Key Format Specification

The rail-fence cipher key is two integers. However they are read in by your program (`stdin`, from a file, etc) the only specification is that A is read first and B second. Your program should enforce the rules that $A > B$ and $B > 1$ and produce an error if either rule is broken.

The substitution cipher key is a string of 26 UPPER CASE letters ordered as-per their alphabetical substitution.

eg: The string `"QAZXSWEDCVFRTGBNHYUJMKILOP"` would cause 'A' to become 'Q', 'B' to become 'A', ..., 'Z' to become 'P'. Substitution of a letter with itself (ie: no change for one or more letters) should be allowed.

3.3.1 ASCII Code

All input data is to be encoded with the ASCII standard. ASCII encoding defines that upper and lower case letters be stored as the following 8-bit integers:

A	65	a	97
B	66	b	98
C	67	c	100
...		...	
Y	89	y	121
Z	90	z	122

If an input byte is outside of the ranges $[65, 90]$ and $[97, 122]$ then it can be copied to the output without modification. If an input byte is in the lower case range, $[97, 122]$, then you should subtract 32 from its value to make it an upper case letter prior to encryption.

4 Tackling the Problem

To complete all parts of this assignment some project management techniques are required.

This is a big task. It is intentionally large because I encourage you to discuss potential algorithms with your peers while writing your program. Implementation in C, however, should all be your own work.

It is also expected that some level of independent research will be needed before you can fully implement all the specifications.

Each programming task listed in Section 3 gets progressively more difficult. I am not expecting anybody to 100% "finish" this task but have designed it so that achieving a pass or credit is a "reasonable" amount of work for an average student. By contrast, achieving a high distinction should be an "extensive" amount of work for the high achievers.

You will need to do some planning before doing too much coding. This will involve understanding the problem at a high level (eg: message and key in, encrypted text out) then breaking it down into smaller sub-tasks (eg: read message in, read key, loop over message doing substitutions, store message somewhere).

Each time you define a sub-task try to describe it as a C function. Can you clearly define the inputs, the processing, and the outputs? If so, it is a good candidate for an *actual* function in your code.

After sub-tasks have been defined the details can be filled in. What variables do you need? Should you use 26 `char`'s to represent 26 letters or should they be in an array?

5 Marking and Submission

Marking will be performed by a demonstrator during your enrolled lab time in week 8. This will involve a Zoom or Discord interview where you demonstrate your code and answer any questions your demonstrator asks. Demonstrators may deduct marks if you do not understand how your code works.

If there are technical difficulties during this demonstration an alternative grading time will be organised. This will not incur any academic penalty.

Your code should be submitted to Blackboard **as a single .c file** before 6pm Monday April 27th 2020.

Code submitted after 6pm Monday 27th April will incur a late penalty.

If you require extra files (ie: any files which demonstrate file I/O features) you do not need to include them when submitting to Blackboard. You do, however, need to be able to describe their format to the demonstrator.

6 Commenting

Your ability to comment will be assessed in this task. The target audience for the comments is a student who has previously failed ENGG1003 and is trying to learn how your program works by reading the source code.

This implies the following guidelines:

- Most lines will need a comment.
  - Tell me *why* your code is doing something. For example, this comment is useless:

```
a = 1; // Set a to 1
```

    but this one could be the difference between a working program and a totally broken one:

```
a = 1; // Re-initialise loop variable to skip first array element
```

- A large block comment should be at the top of your code which describes the high-level operation of the program. It should also include any user-interface notes (eg: how should the demonstrator choose between encryption and decryption?)
- Every function needs to be documented in a block comment above the function definition:
  - What are the inputs?
  - What is the return value?
  - What does the function do?
  - Are there limitations to the function? Must strings be less than a certain length? Are there data type restrictions? etc.
- Program flow control needs to be briefly described.

7 Code Indentation

Your program must follow a code indentation standard. Exactly which one is up to you. A list of common standards can be found on Wikipedia. If in doubt, use "Allman" (as it places matching pairs of braces in vertical lines) or K&R (because I like it).

8 External Resources

For decryption purposes you may find a list of the 10 000 most common English words useful. One can be found on GitHub here: `https://github.com/first20hours/google-10000-english/blob/master/google-10000-english.txt`

A more complete list (466k words) is here: `https://github.com/dwyl/english-words`

Text analysis (counting words, etc) can be done here: `http://textalyser.net/`

There will likely be other online text analysers available, that was just one near the top of the Google results.

9 Mark Allocation

Total marks: 20

Code commenting: **2 marks**

Encryption of at least one line of text with the classical rail-fence cipher algorithm: **5 Marks**

Encryption of at least one line of text with the 2-level rail-fence cipher given plain text and key: **3 marks**

Decryption of at least one line of cipher text with a 2-level rail-fence: **3 marks**

Correct decryption of *three or more letters* from a day-1 provided block of cipher text encrypted with a substitution cipher: **2 marks**

Correct decryption of *four or more letters* from an unseen block of cipher text on marking day: **2 marks**

Decryption of a previously unseen block of cipher text encrypted with a substitution cipher to the point that you can work out what the original text said at time of grading: **1 mark**

The following marks will be awarded for various cipher and plain text input methods (if multiple methods are used demonstrators may, at their digression, award marks based on the "most advanced" method used at any point in the program):

- Hard-coded cipher and plain text: +0 marks
- Cipher and plain text read from `stdin`: +1 mark
- Cipher and plain text read from, and written to, files: +2 marks

The following marks will be awarded for various task selection methods:

- Task selection with a hard-coded variable: +0 marks
- Task selection with a "user friendly" menu system: +1 mark
- Task selection with text read from a file or command line arguments: +2 mark

At the demonstrator's discretion, the following marks may be deducted:

- Comments are woefully inadequate, uninformative, or missing: commenting mark listed above is not allocated
- Code indentation is confusing, not attempted, or is in some way very difficult to read: **-2 Marks**
- Your encryption and decryption algorithms are not in functions: **-3 Marks**
- Your program can only encrypt single words: **-2 Marks**
- Your program can only encrypt a single line: **-1 Mark**
- Your program uses `gets()`: **-3 Marks**

You will note that the total adds up to 22. Grades will be capped at 20.