CITS3007 Project 2024

Contents

Version:	0.1
Date:	3 May 2024

1 Introduction

- This project contributes **30%** towards your final mark this semester, and is to be completed as individual work.
- The project is marked out of 55.
- The deadline for this assignment is **5:00 pm Thu 23 May**.

- You are expected to have read and understood the University Guidelines on Academic Conduct. In accordance with this policy, you may discuss with other students the general principles required to understand this project, but the work you submit must be the result of your own effort.
- You must submit your project before the submission deadline above. There are significant penalties for late submission (click the link for details).

1.1 Project submission

Submission of the project is via the CSSE Moodle server.

- A "testing sandbox" Moodle area will be made available within 1 week of the specification being released, which will provide you with some *minimal* feedback and information you can use while developing and testing your project submission. It will not include space for "coding style and clarity" marks.

  Note that passing these minimal tests is no guarantee of a project achieving a high final mark – students are expected to thoroughly test their code at a variety of levels of optimization, and make use of appropriate static and dynamic analysers.
- A "final submission" Moodle area will be made available no less than 1 week prior to the project being due, where you can submit final code and answers to questions. It will include only minimal tests of code, and will include questions that do not require code or an answer to be submitted, but will be used by markers when assessing coding style and clarity.

1.2 Clarifications and changes to the project specification

You are encouraged to start reading through this project specification and planning your work as soon as it is released. Any queries regarding the project should be posted to the Help3007 forum with the "project" tag.

Any clarifications or amendments that need to be made will be posted by teaching staff on the Help3007 forum.

For an explanation of the process for publication and amendment of the project specification, see the CITS3007 "Frequently Asked Questions" site, under "How are problems with the project specification resolved?".

2 Background

You will need to implement C functions for several tasks, detailed below.

A header file, `crypto.h`, is provided which defines prototypes for these functions.

2.1 The Vigenère cipher

The Vigenère cipher is a simple form of polyalphabetic substitution cipher. The idea behind it was originally described by Giovan Battista Bellaso in 1553 (but was later misattributed to Blaise de Vigenère in the 19th century). The cipher uses a keyword for encryption – the word

is repeated to produce a "key stream" the same length as the plaintext. So if our keyword was "KEY" and our plaintext was "HELLOWORLD", we would obtain the following:

Plaintext	H E L L O W O R L D
Keystream	K E Y K E Y K E Y K

Figure 1: Vigenère plaintext and keystream

Each letter of the message is then encrypted using a Caesar cipher shift determined by the corresponding letter of the keystream. In the example shown in Figure 1, every letter of the plaintext will be encrypted by a Caesar cipher which is shifted by either

- 10 positions (the position of "K" in the alphabet)
- 4 positions (the position of "E" in the alphabet), or
- 24 positions (the position of "Y" in the alphabet)

where positions are counted starting from 0.

Obviously, a longer keyword will result in fewer statistical patterns showing up in our ciphertext. Using a keyword of length 1 is exactly equivalent to applying the Caesar cipher. Using a keyword of length 3 means that every third letter of our plaintext is encrypted using a different Caesar cipher – but if an attacker knew the length of our key, they could simply split the ciphertext into 3 (taking every 1st, 2nd and 3rd character), and analyse each one as we would a Caesar cipher. If the keyword is as long as the plaintext, then very few patterns should show up at all; and if the keyword is also randomly generated, then we have an encryption system equivalent to a one-time pad.

Using the keyword and plaintext from Figure 1:

- the first letter of the plaintext will be encrypted using a Caesar cipher shift of 10 (giving the ciphertext letter "R")
- the second letter similarly, with a shift of 4 (giving the ciphertext letter "I"), and
- the third letter with a shift of 24 (giving the ciphertext letter "J").

So the final ciphertext should be as shown in the following example:

Plaintext	H E L L O W O R L D
Keystream	K E Y K E Y K E Y K
Ciphertext	R I J V S U Y V J N

Figure 2: Vigenère example with ciphertext

Early users of the Vigenère cipher made use of a "Vigenère square" (as shown in Figure 3) to quickly find out what ciphertext letter to use given a particular plaintext and keystream letter, but we will use modular arithmetic. We will first implement the Caesar cipher (functions `caesar_encrypt` and `caesar_decrypt`), and then make use of those functions to implement the Vigenère cipher.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

Figure 3: Vigenère square

2.2 Some useful websites

The following websites implement the Vigenère cipher, and you can use them to check that you understand how the cipher works (or to test your code).

- dCode.fr Vigenère implementation
- Boxentriq Vigenère implementation and cracker
- cryptii Vigenère implementation

3 Tasks

You should complete the following tasks and submit your completed work using Moodle.

3.1 Caesar cipher encryption and decryption functions (20 marks)

You are required to implement functions to encrypt and decrypt C strings using the Caesar cipher. Up to 10 marks are awarded for a successful implementation of these functions. 10 further marks are awarded for coding style and quality of the implementation.

3.1.1 `caesar_encrypt`

The `caesar_encrypt` function encrypts a given plain text using the Caesar cipher algorithm with a specified key within a given range.

```
void caesar_encrypt(char range_low, char range_high, int key,
                    const char *plain_text, char *cipher_text
);
```

A description of this function can be found in the `crypto.h` header file. Briefly, it encrypts `plain_text` and writes the result into `cipher_text`; characters in `plain_text` that fall within the range `range_low` to `range_high` *are* encrypted, but everything outside that range is not – it is simply copied directly into `ciphertext`. (Thus, we can specify for instance that text in the range `'A'` to 'Z' is to be encrypted, but all other characters – such as whitespace, punctuation, and lowercase text – will be left as is.)

The `caesar_encrypt` function cannot fail: it always completes successfully if its preconditions are met.

Example use of `caesar_encrypt`:

```
char plain_text[] = "HELLOWORLD";
char cipher_text[sizeof(plain_text)] = {0};
caesar_encrypt('A', 'Z', 3, plain_text, cipher_text);
// After the function call, cipher_text will contain the encrypted text
char expected_cipher_text = "KHOORZRUOG"
assert(strcmp(cipher_text, expected_cipher_text) == 0);
```

3.1.2 `caesar_decrypt`

The `caesar_decrypt` function decrypts a given plain text using the Caesar cipher algorithm. Calling it with some key n is exactly equivalent to calling `caesar_encrypt` with the key $-n$.

```
void caesar_decrypt(char range_low, char range_high, int key,
                    const char *plain_text, char *cipher_text
);
```

3.2 Vigenère cipher encryption and decryption functions (20 marks)

You are required to implement functions to encrypt and decrypt C strings using the Vigenère cipher. Up to 10 marks are awarded for a successful implementation of these functions. 10 further marks are awarded for coding style and quality of the implementation. The required functionality is described in the `crypto.h` header file; the prototypes for the functions are as follows:

```
void vigenere_encrypt(char range_low, char range_high, const char *key,
                      const char *plain_text, char *cipher_text
);

void vigenere_encrypt(char range_low, char range_high, const char *key,
                      const char *cipher_text, char *plain_text
);
```

3.3 Command-line interface (15 marks)

You are required to implement a function `cli` with the prototype

```
int cli(int argc, char ** argv);
```

The `cli` function is intended to allow your code to easily be called and tested from the `main()` function of a program; the `argc` and `argv` arguments have the same meaning as they do in `main()`. 5 marks are awarded for implementing the function correctly, 5 marks for coding style and quality of the implementation, and 5 marks for writing appropriate documentation for the function using a Doxygen-processable comment (which should start with a forward slash and two asterisks – "`/**`" – like Javadoc-processable comments).

The `cli` function should check whether 3 arguments have been passed (aside from `argv[0]`, which represents the program name), and print an error message and return 1 if some other number has been passed.

The first argument should be one of the following strings, representing an operation to perform:

- `"caesar-encrypt"`
- `"caesar-decrypt"`
- `"vigenere-encrypt"`
- `"vigenere-decrypt"`

If some other string is passed, the function should print an error message and return 1.

The second argument represents a key. If the operation being performed is Caesar cipher encryption or decryption, this will be an integer; if the operation is Vigenère encryption or decryption, the second argument can be any string.

The third argument represents a message – either a plaintext to encrypt, or a ciphertext to decrypt.

The `cli` function should validate that the key is an appropriate `int`, if Caesar encryption or decryption is being performed; if an invalid argument is passed, the function should print an error message and return 1.

Lastly, the `cli` function should invoke the appropriate encryption or decryption function, using `'A'` and `'Z'` as the lower and upper bound, respectively, so as to encrypt or decrypt the message passed in the third argument; it should print the processed message to standard output followed by a newline, and then return 0.

If in your implementation you make use of any functions that can fail, you should ensure you check their result, and if they fail, print an appropriate error message to standard error, and return 1. (You should *not* call `exit()`.)

3.4 Challenge tasks

For students who would like an extra challenge – up to 4 marks can be awarded for completing "challenge" tasks. If you want to complete these, you should contact the unit coordinator at least 4 days before the close of submissions to obtain exact requirements for the tasks.

You will need to implement C functions that analyse and attempt to decrypt a message encrypted with the Caesar cipher and/or Vigenère cipher. Marks for challenge tasks are completely at the discretion of the unit coordinator, depending on the quality and scope of the tasks attempted. Marks awarded for challenge tasks cannot take your project total beyond 100%.

4 Annexes

4.1 Marking rubric

Submissions will be assessed using the standard CITS3007 marking rubrics.

Except where otherwise noted, questions requiring long English answers are marked as per the standard long answer rubric (see https://cits3007.github.io/faq/#marking-rubric).

Questions requiring code will have marks allocated for *correctness*, and for *style and clarity*.

4.2 Code requirements

Note that, as per the standard rubric, your code must compile without errors using gcc in the standard CITS3007 development environment. If it fails to compile, teaching staff will not fix it for you, and you are likely to receive only minimal marks for implementation of functions (though it is still possible to receive marks for style and clarity of code, for those portions of the project you've completed).

If, at the time of submission, some portion of your code is not compiling, you should either comment it out, or surround it with "`#if 0 ... #endif`" preprocessor instructions, so that your code does compile.

Any `.c` files or code submitted should `#include` the `crypto.h` header as follows:

```
#include <crypto.h>
```

Your code must contain the functions required by this specification, and they must exactly match the prototypes in the `crypto.h` file, but you may also write whatever "helper functions" you wish.

Your code must not include a "`main()`" function. If it does, your code will fail to compile when automated tests are run on it, and you are likely to receive minimal marks for implementation.

You may `#include` any header files that you need to, as long as they are available in the standard CITS3007 development environment, and you may use non-standard C functions as long as they are available in that environment.

Although you are strongly encouraged to test your code (testing is discussed in upcoming labs), you should not submit your test code, and it is not assessed.

4.3 Coding style

All code submitted should comply with the coding guidelines listed at https://cits3007.github.io/faq/#marking-rubric.

All C code written should:

a. adhere to secure coding best practices, and
b. be properly documented.

The documentation requirement means that, at minimum, all functions should have a comment just above them describing what the function does.

Functions forming part of the API (and any particularly important internal functions) should have a *documentation block* parseable by Doxygen. (The Caesar and Vigenère encryption and decryption functions in `crypto.h` already have documentation blocks written for them, so you do not need to write them.)

When writing documentation, you may assume

a. that readers have read this project specification and the `crypto.h` header file, so you need not repeat information from them;
b. that any function with a prototype in the `crypto.h` header file forms part of the API; and
c. that if you are unable to complete all the project by the time of submission, there is no need to write documentation for functions you haven't completed.

Except for documentation blocks, all comments should be written as single-line ("`//`") comments – do not use multiline ("`/* .. */`") comments.

As part of keeping your code readable, lines should generally be kept to less than 100 characters long.

Additionally, note that your code should be considered "library" code – it should not be interacting directly with a user or the terminal unless the project specification specifically asks for this. This means your code should:

- never print to standard out or standard error (unless the specification states otherwise); and
- never call `exit()`, but instead return with an error value, unless the specification states otherwise.

If your code prints to standard out or standard error, it is likely to fail the automated tests used to mark portions of your code. If that happens, those portions of your code are likely to receive a mark of 0. Teaching staff will not fix your code to remove statements that print to standard out or standard error.

4.4 Security best practices

You are expected to bear in mind (and apply!) all secure coding best practices that have been discussed in class. However, note that in only one case is an integer value obtained from an untrusted source; in general, you can (and must!) assume that your functions are being called correctly according to their specification.