

# Assignment 3

COMP 206, Fall 2016

Due: Sunday November 13th, (23:59) via My Courses  
100 marks total

## Warm-Up Exercises

Not for marks. The TAs or instructor will solve these for you in office hours, if you wish.

### Tutorial 1 - Practice with CMake

Beginning with the sample files posted on My Courses:

- week8\_multi\_file\_main.c
- week8\_multi\_file\_swap.c
- week8\_multi\_file\_swap.h
- CMakeLists.txt

Build the provided code with the usual sequence of commands:

- mkdir build
- cd build
- cmake ..
- make

Attempt to add a new program (your own main.c) that also builds with the swap library.  
(Hint: if you do this right, it will be *\*extremely\** easy. The point is to show you how nice it is to use cmake)

### Tutorial 2 - Fake the Time

Programs that need to use the current time (e.g., a demo version that should only run for 1 month after download) often check the current time through C's built in functionality described in "time.h": <http://www.cplusplus.com/reference/ctime/>

The provided program "tut2/time\_check" is an example. Luckily, time\_check uses the *\*shared library\** form of the time functionality, and with our 206 knowledge, we can extract the answer from this program before the due date. Create a C file and compile it into a shared object that will falsely provide a date in the future, like this example:

```
$ ./time_check
Now: 2016-11-3 18:25:4
I will not tell you the answer until after the due date:
2016-11-14 0:0:0
$ export LD_PRELOAD=./libtime.so; ./time_check
Now: 2048-7-9 18:3:26
The answer is
$
```

Hidden to keep the secret.

# 1. Break the Cypher (30 marks)

This question involves extracting data that has been encrypted by us using the Reverse Caesar Cypher algorithm. That is:

```
void encrypt( unsigned char *source, int source_len, int key );
```

- Assumes character values are unsigned integers in the range 0-255
- For each character, adds key, wrapping around so that 256 becomes 0
- Reverses the order of the characters within the string
- Examples:
  - Input "a" with key 5 goes to "f"
  - Input "9" with key 10 goes to "C"
  - Input "a9" with key 3 goes to "<d"

**(Part A)** Write a program "q1a\_decrypt.c" that takes 2 command-line arguments: (1) an integer key; and (2) a path to a file containing string data that has been encrypted using the Reverse Caesar Cypher algorithm. Your program must output the decrypted (original) data as standard out, followed by a newline.

Note that the message should only read properly when the correct key is provided (this is the "security" given by the encryption algorithm). When run with the wrong key, the data will be unreadable or even contain non-printing characters, newlines, null characters etc.

All sample files include the "null" character as the last byte in the file - you should make sure not to interpret this as part of the data (i.e., do not reverse and shift it). The maximum message size we will use during marking is 1000 characters.

```
$  
$ ./decrypt 3 usual_greeting_key3.crypt  
hello world  
$ ./decrypt 10 usual_greeting_key3.crypt  
a^eeh[0A]hke]  
$ ./decrypt 1 usual_greeting_key3.crypt  
jgnnq"yqtnf  
$
```

**(Part B)** Using an assumption about the original data (such as that it is English text), it is possible to “crack” simple encryption schemes such as the Reverse Caesar Cypher, **without needing the key**. A simple brute-force approach is to decrypt repeatedly with many keys until the output has the desired properties.

For this question, we make your life even easier with a guarantee: The provided data files without a key in their filename only contain characters in the ranges a-z, A-Z, spaces and null-terminators (`\0`). They all contain at least one (a or A) and at least one (z or Z).

Write a program “q1b\_crack.c” that takes 1 command-line argument: a path to a file containing the encrypted data. Your program must output 2 lines of standard output: (1) the key that was found with brute-force search and (2) the resulting decrypted data.

Running `$ ./crack happy_halloween.crypt` should print:

```
200
```

```
Arent you scared these daze
```

Files used in marking Part B will always follow the stated guarantee. You do not have to handle files with the wrong properties.

## 2. Matt's Chat Daemon (50 marks)

Write a C program “q2\_text\_chat.c” that implements a turn-based chat system. Chat user interactions occur as terminal I/O and data is passed between chat processes as plain-text files. Only plain-text file I/O can be used to communicate the chat messages, you cannot use networking or other system calls. Your program must be run as follows:

```
$ ./q2_text_chat <incoming_file> <outgoing_file> <chat_username>
```

The program must begin by checking for an existing message in <incoming\_file>. If the incoming file does not exist yet, or if it does not hold a valid message, you must print the line “Nothing received yet.” If a valid message exists, your program should print one line of terminal output of the form:

Received: <msg>

The rest of the program continues by alternating 1 send and then 1 receive forever until <ctrl-d> or <ctrl-c> are pressed – this defines *turn-based* behaviour.


To send, the user must be prompted for a message by printing “Send: “ on the terminal, and waiting until one line of text is entered on stdin (until user hits return). Write the message into <outgoing\_file> as “[<chat\_username>] <line>”.

To receive, check for **new** messages in <incoming\_file>. That is, keep track of the previous message that was printed to screen and keep monitoring the file forever until the message in <incoming\_file> differs from the previous one. When a new message does arrive, display it using “Received: <message>”. It is OK not to print the same message twice, even if the other user tries to repeat.

Note that 2 processes of this program are required to achieve meaningful “chatting”. In order to chat properly, the <incoming\_file> of one process must match the <outgoing\_file> of the other process, and vice-versa.

Make sure to apply appropriate checking on arguments, file inputs and terminal inputs to ensure your program always works as expected and does not crash.

Sample chat session:



```
Assign2
$
$
$ ./q2a_text_chat a.txt b.txt Dave
Nothing received yet.
Send: Hello friend?
Received: [Dave2] Yes, I'm here!
Send: Ok great. Let's talk about 206.
Received: [Dave2] I'd love to, what a great course!
Send: That's right. Nothing more to say now.
Received: [Dave2] Agreed. Bye for now then.
Send:
Session terminated due to end of input stream.
$

Assign2 - Window 2
$
$
$ ./q2a_text_chat b.txt a.txt Dave2
Received: [Dave] Hello friend?
Send: Yes, I'm here!
Received: [Dave] Ok great. Let's talk about 206.
Send: I'd love to, what a great course!
Received: [Dave] That's right. Nothing more to say now.
Send: Agreed. Bye for now then.
^C
$

Assign2 - Window 3
$
$cat a.txt
[Dave2] Agreed. Bye for now then.$
$cat b.txt
[Dave] That's right. Nothing more to say now.$
$
```

### 3. Private Chatting (20 marks)

Write a C program that combines the previous two pieces of functionality by implementing an encrypted chat system that uses the Reverse Caesar Cypher. Your program must take 4 arguments:

**\$ ./q3\_encrypted\_chat <incoming\_file> <outgoing\_file> <chat\_username> <key>**

All the same chat interactions from Question 2 must also be used here so that the users of the chat do not experience any change due to the encryption.

When <incoming\_file> and <outgoing\_file> are inspected during the chatting process, all data stored in the files must be encrypted properly using the Reverse Caesar Cypher. This can be tested by running the program from Question 1a.

e.g., **\$ ./decrypt 3 a.txt** -> must print the latest chat message sent, in readable form.

As you are reusing functionality from Q1 and Q2, you may choose to reuse your previously written files as objects or libraries or to recreate the functionality independently for Q3. This is your choice. However, you must provide a "CMakeLists.txt" file that allows us to compile your solution for Q3 using the the following commands (relative to the submission directory):

```
$ mkdir build
$ cd build
$ cmake ..
$ make
$ ./q3_encrypted_chat (the chat starts correctly...)
```

## Submitting your solutions:

You will submit a single zip file that contains all of the C files and the CMakeLists.txt holding your solutions. Create the zip file with the command:

```
$ zip A3_solutions.zip q1a_decrypt.c q1b_crack.c q2_text_chat.c CMakeLists.txt  
<any additional .c or .h files created for Q3>
```

The TAs will mark your code by compiling it, and running several different test cases. It must compile without errors and produce an output program to be awarded any marks, so test carefully at Trottier as always. Remember to include a reasonable level of code comments and run-time error checking.