CMPT 300: Operating Systems
School of Computing Science
Fall 2015, Section D1

Assignment #1: C Programming under Unix
Due Date: Thursday, September 24, 2015

Assignments in this Course:

All four assignments in CMPT 300 this term will be related to each other. Although it may not be obvious now, you will be using the knowledge (and some code) from this assignment in your future assignments. Therefore, please take the time to understand the concepts and to write clean, readable code.

Also, as a student in this course, it is essential to have a solid understanding of how memory needs to be managed when aids such as Java's garbage collection are not present; operating systems and most systems software do not use garbage collection.

Overview:

Students in CMPT 300 must have knowledge of C programming and the UNIX environment, as per the course prerequisites. This assignment is meant as a "refresher" of what you (should have) learned in CMPT 225 (or in an equivalent class). This assignment should not involve any topics which have not been covered in your previous courses and is intended to be a short exercise to review concepts that you will need for CMPT 300.

For this assignment, you will write a program in C using standard C file I/O functions: `fopen()`, `fscanf()`, `fgets()`, `fprintf()`, `fputs()`, `fclose()`. You **cannot** use streams, the Standard Template Library (STL) or any other C++ constructs. The program you will write is called `lyrebird` and it will perform its task on a text file. This assignment will test your file I/O basics, string handling, and dynamic memory management. The task will involve the decryption of a text file.

Standard Comment About Design Decisions

Although many details about this assignment are given in this description, there are many other design decisions that are left for you to make. In those cases, you should make reasonable design decisions (e.g., that do not contradict what we have said and do not significantly change the purpose of the assignment), document them in your source code, and discuss them in your report or README file. Of course, you may ask questions about this assignment (for example, on the mailing list) and we may choose to provide more information or provide some clarification. However, the basic requirements of this assignment will not change.

Project: `lyrebird`

Imagine you have been hired by the security agency of your choice as a software developer. News has come to your office that some not-so-nice people are passing encrypted messages over Twitter. It is important for you to find out what those encrypted tweets say and whether they contain any information that is dangerous to those you work for.

Fortunately for you, a mathematician within your office has come up with an algorithm that decrypts the tweets. The algorithm consists of four steps:

1. To confuse onlookers, the encoders have added extraneous symbols to the encoded text at regular intervals. These appear every 8th character (starting at position 8), so, for step 1, these characters should be removed. For example, `txdgaclr?xkbq'pxvztl` becomes `txdgacl?xkbq'pvztl`.

2. After performing step 1, the number of remaining characters (including spaces and punctuation) should be a multiple of 6. This is because they were encrypted in groups of 6. In step 2, transform each group of 6 into an integer using base 41. For example, examine the first group of 6 in `txdgacl?xkbq'pvztl`, i.e., `txdgac`. Using the table at the end of this document, `t` has a numerical value of 20, `x` ↔ 24, `d` ↔ 4, `g` ↔ 7, `a` ↔ 1, and `c` ↔ 3. So, in base 41, `txdgac` looks like

20	24	4	7	1	3

   which, expanded positionally, is equal to $20 \cdot 41^5 + 24 \cdot 41^4 + 4 \cdot 41^3 + 7 \cdot 41^2 + 1 \cdot 41^1 + 3 \cdot 41^0$ = 2385229779. When translated, the other two groups of 6 become 1482371458 and 3522459006.

3. Each cipher number (C) obtained from step 2 can now be mapped onto a similar plain-text number (M), according to the formula $M = C^d \bmod n$ where d is the constant 1921821779, and n is the constant 4294434817. In step 3, perform this transformation. Using the function described above on our example, (2385229779, 1482371458, 3522459006) → (941826252, 2708339983, 3128117427).

4. Each of the plain-text numbers translates into a group of 6 characters. So, in step 4, use the inverse of the function used in step 2 to get the final decrypted text. For this example, the final string is "`hello world!#  `".

While the mathematician has provided you with this working algorithm, he or she is not a good programmer. It is up to you to create a piece of software that implements this algorithm in order to decrypt the large number of encrypted tweets that your security agency is seeing. We will refer to this project as *project lyrebird*, named after a type of bird that is also known to "encrypt" its tweets using everyday sounds (https://www.youtube.com/watch?v=XjAcyTXRunY).

Input/Output and Behavior Specification:

Your program should have the name `lyrebird` and should take exactly two command-line arguments. The first command-line argument is the name of a file containing the encrypted tweets that need to be decrypted. The second command line argument is the name of a file to save the decrypted

tweets to. An example of how the program is started from the command line is:

```
$ ./lyrebird encrypted_tweets.txt decrypted_output.txt
```

The contents of the input file are encrypted tweets, with one encrypted tweet per line. Note that tweets have a maximum length of 140 characters, which means that there is a maximum length to each encrypted tweet (not 140 characters, but some finite number nonetheless). The output file should contain each decrypted tweet on a separate line in the same order as in the encrypted file. For example, if the contents of `encrypted_tweets.txt` is

```
 aa )pyepyt.fd':ymkcnp# dzcs\fkwnze-vy&gfz?h-)!s'\bfia
e-ituuzf$kqnljz?-hcme#nwmjo(mf?qgifgv p#x
):u#rplf,vr&xgnqpbzx'\bffia
```

Then the output file `decrypted_output.txt` should contain

```
there was an old woman who swallowed a fly.
i dunno why she swallowed that fly,
perhaps she'll die.
```

No output should appear in the terminal unless an error occurs. The format of any error messages are up to you as long as they clearly describe the error.

Required Design:

As appropriate, you must use C memory allocation (e.g., `malloc()`, `free()`) and C file I/O functions (e.g., `fopen(), fscanf(), fclose()`). Because of the use of MEMWATCH (see below), you cannot use C++ streams, the Standard Template Library (STL), or the C++ `stdlib` (e.g., cannot use type/class `string`). Also, your TA may not have any expertise in C++ and therefore we cannot guarantee support for languages other than C.

You must write a Makefile for your program. When someone types `make`, your Makefile should build the executable program `lyrebird`. When someone types `make clean`, your Makefile should remove the executable `lyrebird` (if any), all `.o` files (if any), and all `core` files (if any).

It is IMPERATIVE that your program properly deallocates ALL dynamic memory in a correct fashion (i.e., using `free()`) before your program terminates, or else your assignment will LOSE marks. To check that your program properly allocates and deallocates ALL dynamic memory it uses, you must use the MEMWATCH package, which is simple to do.

Using MEMWATCH

This term, we will use Version 2.71 (stable) of MEMWATCH. You can download the package at http://www.linkdata.se/sourcecode/memwatch/. The TA will expect that your files have been compiled with the header file `memwatch.h` and with file `memwatch.c` in your working directory.

In **all of your source files** (either directly or indirectly), you must add the directive

```
#include "memwatch.h"
```

and when you compile, you must compile `memwatch.c` along with your source file with the variables `MEMWATCH` and `MW_STDIO` defined. As an example:

```
gcc -DMEMWATCH -DMW_STDIO main.c memwatch.c
```

When you run your program, if you get a message in your output that reads something like:

```
MEMWATCH detected 5 anomalies
```

it means you have not deallocated dynamic memory properly. In particular, this message indicates that 5 allocated structures have not been deallocated. You should also check the MEMWATCH log file for any reports. If your assignment is not properly compiled with MEMWATCH enabled or if MEMWATCH reports that your memory allocation/deallocation was incorrect, then you will lose marks.

What to Hand In:

You will submit your assignment through http://courses.cs.sfu.ca. Your submission should be a `zip` archive file with the name `submit.zip`. The zip file should contain the following:

1. A **README** file (ASCII text is fine) for your assignment with: (1) your name, (2) student number, (3) SFU user name, (4) lecture section, (5) instructor's name, and (6) TA's name clearly labeled. All these items of information should also be part of **each file** that you submit (e.g., as a comment in your code files). The **README** file must also include a short description of your program, as well as a description of the relevant commands to build (e.g. `make all`) and how to execute your programs including command line parameters. As per the academic honesty guidelines, you should list your sources and the people you have consulted within this **README** file.
2. Your source code file(s) for `lyrebird`, including all header files. Do NOT submit any MEMWATCH files, as the TA will use his own fresh copy of that code, but the use of MEMWATCH should be enabled in your code and `Makefile`.
3. Your `Makefile`.

NOTE: Do **not** submit files or test data **not** described above. Only submit what is requested and what is required to compile your program (except, of course, the MEMWATCH files).

Marking:

This assignment is worth 5% of your final mark in this course. **This is an individual assignment. Do not work in groups.** Review the course outline on this matter.

The assignment itself will be marked as follows: 70% for the correctness of the program when we test

it using the CSIL Linux machines, using `gcc`, and 30% for the quality of the implementation (design, modularity, good software engineering, coding style, useful and appropriate comments, etc.).

Note that the correctness mark will be computed solely on how your program runs and not on what the code looks like. If your source code, **as submitted**, does not compile and run (using the submitted Makefile) on the CSIL Linux workstations using `gcc`, you will receive a mark of zero for correctness. Review the Course Outline on this matter.

When it comes to your quality of implementation mark, all that you have learned about good programming style and comments in your code will apply. Having correct code is important, but good style, design, and documentation are also important. We cannot provide an exhaustive list of what we will look for, but an incomplete list includes: a comment for each source code file, a comment for each procedure/function, a comment for each significant (global or local) variable, good choice of names/identifiers, proper modularity (e.g., do NOT put all/most of the code in `main()`), checking function return values for errors, etc.

NOTE: There are a number of programs that you can download off the Internet that provide similar functionality to what you are asked to implement for this assignment. We are familiar with them. Therefore, do **not** download these programs; write your own solution to this problem. Modifying someone else's program (including programs that you can download) is against the requirements of this assignment and is an Academic Offense. If you have any doubts about whether your actions are permissible or not, you should ask the instructor **before** proceeding.

Hints:

- To get you started, we've created an assignment #1 care package that can be downloaded at https://courses.cs.sfu.ca/2015fa-cmpt-300-d1/pages/A1CarePackage. The care package is a zip file containing a basic Makeflie, C source files for Hello World, and an executable called `encrypt` that will allow you to create encrypted test files. Feel free to use this care package as a starting point.
- For step 3 of the decryption algorithm, you may want to google “modular exponentiation” or “exponentiation by squaring”.
- You may want to refresh your memory about what C libraries exist and what functions/variables they contain. Here is a decent reference:
  https://www-s.acm.illinois.edu/webmonkeys/book/c_guide/
- You may want to get comfortable working with Makefiles. Here is a decent tutorial to get you started: http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/
- Before you submit, make sure your `submit.zip` file works from within a fresh directory. That way, you make sure that `submit.zip` contains all the needed files for testing (with the exception of MEMWATCH, of course).
- Remember, make sure that your program does **not** produce any debugging or extraneous output during **normal** execution. Only the requested output should be generated. Marks will be deducted for incorrect and other unrequested output. That said, it is acceptable to have output to report an actual error.

Further hints may be given later on the course mailing list, if warranted. Be sure to read the emails on the mailing list on a regular basis.

Table of Plain Text and Cypher Text Characters

0	<space>	11	k	21	u	31	!
1	a	12	l	22	v	32	?
2	b	13	m	23	w	33	(
3	c	14	n	24	x	34	)
4	d	15	o	25	y	35	-
5	e	16	p	26	z	36	:
6	f	17	q	27	#	37	$
7	g	18	r	28	.	38	/
8	h	19	s	29	,	39	&
9	i	20	t	30	'	40	\
10	j						

Note: The symbols 37-40 ($/&\) can never be used in plain text, but may be used within cypher text.