Lab 1: UNIX `stdio` I/O, simple encryption, and `stat()`

Please read this entire assignment. Take some notes. Think about it some. Take some more notes. Look for answers to your questions in this document.

Due: As shown in Canvas. Submit a single `tar.gz` file.

Do not place ANY directories in your submitted tar file. I will not change into any sub-directories to hunt down your source files. When you create your `tar.gz` file to submit, do it within the directory where you created the source files, **NOT from a higher level directory. If I cannot find your source files in the same directory where I extract your submitted tar file, I will simply give you a zero on the assignment**. Submit a single `tar.gz` file to Canvas for this assignment.

In this assignment, you will be working with UNIX file I/O system calls and library functions. **I urge you to not delay beginning it**.

Part 1 – The `Makefile`

Write a `Makefile` to build the C files.

You must have a single `Makefile` for this lab. Your `Makefile` must contain (at least) the following targets:

Target	Action
`all`	Builds all dependent C code modules for your applications in the directory. This should be the default target in your `Makefile`.
`cae-xor`	Builds all dependent C code modules for the `cae-xor` application in the directory. The dependency of `cae-xor` should be `cae-xor.o`.
`cae-xor.o`	Compiles the `cae-xor.c` module for the `cae-xor` application in the directory, based off of any changed dependent modules. The dependency for `cae-xor.o` should be `cae-xor.c`.
`mystat`	Builds all dependent C code modules for the `mystat` application in the directory. The dependency of `mystat` should be `mystat.o`.
`mystat.o`	Compiles the `mystat.c` module for the `mystat` application in the directory, based off of any changed dependent modules. The dependency for `mystat.o` should be `mystat.c`.
`clean`	Deletes all executable programs, object files (files ending in `.o` produced by `gcc`), and any editor chaff (# files from vi and ~ files from **emacs**). Make sure you use this before you bundle all your files together for submission.

When I build your assignment, I should be able to just type

```
make clean
make clean all
```

to have it completely clean the directory and build `cae-xor`, and `mystat`.

I also strongly recommend that you **use some form of revision control on your source files**. Not only does this reduce the possibility of catastrophic file loss, but it is a LOT better than making `.BAK1`, `.BAK17`, `.BAK4c` copies of your code. Putting this into your `Makefile`, as described in the notes about `make` would make your life better.

You must compile your program using all of the following flags for `gcc`

```
-Wall -Wextra -Wshadow -Wunreachable-code -Wredundant-decls
-Wmissing-declarations -Wold-style-definition
-Wmissing-prototypes -Wdeclaration-after-statement
-Wno-return-local-addr -Wunsafe-loop-optimizations
-Wuninitialized -Werror
```

Putting these flags into the `Makefile` with the `CFLAGS` variable will make your life better. When compiled with the flags, the `gcc` compiler should emit no errors or warnings. **Any warnings from the compiler results in an automatic 20% deduction**.

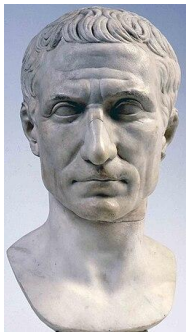
Part 2 – `cae-xor` Cipher

This portion of the assignment is to write a C code program that combines a Caesar-like cipher and an `xor` cipher into a single program. One program that will encipher input with **both** the Caesar-like cipher and an `xor` cipher. A working solution is in my `Lab1` directory (`~rchaney/Classes/cs333/Labs/Lab1`), called `cae-xor`.

A different key can be given for each of the Caesar-like cipher and the `xor` cipher.

- **When encrypting, input always goes through the Caesar-like cipher first; the `xor` cipher follows.**
- **When decrypting, input always goes through the `xor` cipher first; the Caesar-like cipher follows**.
- If no key for either the Caesar-like or `xor` encryption is given on the command line, all data are passed through unchanged.

Let's get to why I call it a *Caesar-like* cipher.



First, the original Caesar cipher only worked with an early Latin alphabet of 23 letters. Your Caesar-like implementation will work on **all the printable ASCII characters** (space to ~ (tilde), in ASCII order). All non-printing characters are just passed through. For example, when encountering a newline character, just pass it through, don't encode it. You can get a good idea for what characters are included in ASCII printing characters by looking

over '`man ascii`'. There is also a program in the `Lab1` directory called `isprint` which will output all the ASCII printing characters. It accomplishes this in 2 different ways (you can look over the `isprint.c` file).

Second, and most importantly, the standard Caesar cipher uses a single value to shift all the input. The Caesar-like cipher for this assignment is more like a Vigenère cipher, the shift for the input varies based on the key. The key for the Caesar-like cipher will be a string. Your code will use the characters in the key string to shift the input. The characters that can make up the string for the key are all the ASCII printing characters, space through tilde. The shift used for the Caesar-like cipher is shown in the table below.

Character in key									
space	**!**	**"**	**#**	**$**	**…**	**{**	**|**	**}**	**~**
0	**1**	**2**	**3**	**4**	**…**	**91**	**92**	**93**	**94**
Shift used for input									

Table 1: Mapping if the ASCII `isprint()` characters to the shift amount for the Caesar-like encryption.

If the input key is a space, the input is shifted zero places (unchanged). If the input key is a single `!` (exclamation point), the shift is a single character. In this case (using a shift string composed of only a single `!` character), an input of an **a** would be shifted to a **b**.

Example 1:

Key used for Caesar-like cipher: **!"#$**

Input text:

a	b	c	d	e	f	g	h	i	j

Shift applied to input (see Table 1 above):

1	2	3	4	1	2	3	4	1	2

Encrypted text:

b	d	f	h	f	h	j	l	j	l

Notice how the key is repeated if the input string is longer than the key. This also applies if the input is more than a single line.

The command line to see the above is:

```
./cae-xor -c '!"#$' -D > example1.cae
```

The type the letters `abcdefghi` and the enter key (the newline character). Then type a Control-d to terminate input.

You can decrypt the output file (`example1.cae`) with the following command:

```
./cae-xor -c '!"#$' -d -D < example1.cae
```

Example 2:

Key used for Caesar-like cipher: **{ | }~**

Input text: **Note that there are 2 lines of input**

a	b	c	d	e	f	g	h	i	j
k	l	m	n						

Shift applied to input: **Note that there are 2 lines of shifts**

91	92	93	94	91	92	93	94	91	92
93	94	91	92						

Encrypted text: **Note that there are 2 lines of output**

]	_	a	c	a	c	e	g	e	g
i	k	i	k						

The command line to see the above is:

```
./cae-xor -c '{|}~' -D > example2.cae
```

The type the letters `abcdefghi` and the enter key (the newline character), type the letters `jkl` and the enter key (the newline character). Then type a Control-d to terminate input.

You can decrypt the output file (`example2.cae`) with the following command:

```
./cae-xor -c '{|}~' -d -D < example2.cae
```

It is important to recognize that when in the input is composed of multiple lines, the location in the encryption string carries over to the next line, it is not (necessarily) restarted. The non-printing character newline (at the end of each of the input lines) is passed through unchanged (in the Caesar-line portion of the assignment).

On to the `xor` cypher. It is a little more straight forward.

A major difference between the Caesar-like cipher and the `xor` cipher is that Caesar-like cipher always manages plain readable text. Plain text comes in and plain text comes out. The `xor` cipher is different. It will write binary data (outside the ASCII printable characters). It will read binary data (characters outside the ASCII printable characters).

With the `xor` cipher, all characters are encrypted, including non-printing characters (such as newlines).

As the name suggests, the `xor` cipher uses the `xor` operator (the ^ circumflex, hat, or caret) to encrypt or decrypt data. Like the Caesar-like encryption, the `xor` cipher uses a string as the key. The string key is chosen from the ASCII printing characters, as is key for the Caesar-like cipher.

Example 3:

Key used for `xor` cipher: **`!"#$`**

Input text: As ASCII characters

a	b	c	d	e	f	g	h	i	\n

`xor` applied to input as ASCII hex values from the encryption string:

0x21	0x22	0x23	0x24	0x21	0x22	0x23	0x24	0x21	0x22

Encrypted text as hex values:

0x40	0x40	0x40	0x40	0x44	0x44	0x44	0x4c	0x48	0x28

Notice how the key is repeated if the input string is longer than the key. This also applies if the input is more than a single line.

The command line to see the above is:

```
./cae-xor -x '!"#$' -D > example3.xor
```

The type the letters `abcdefghi` and the enter key (the newline character). Then type a Control-d to terminate input.

You can decrypt the output file (`example3.xor`) with the following command:

```
./cae-xor -x '!"#$' -d -D < example3.xor
```

You can display the output file using `hexdump` with the following command:

```
hexdump -C example3.xor
```

Example 4:

Key used for `xor` cipher: **{ | }~**

Input text: **Note that there are 2 lines of input**

a	b	c	d	e	f	g	h	i	\n
j	k	l	\n						

`xor` applied to input as ASCII hex values from the encryption string: **Note that there are 2 lines of shifts**

0x7b	0x7c	0x7d	0x7e	0x7b	0x7c	0x7d	0x7e	0x7b	0x7c
0x7d	0x7e	0x7b	0x7c						

Encrypted text as hex values: **Note that there are 2 lines of output**

0x1a	0x1e	0x1e	0x7a	0x1e	0x1a	0x1a	0x16	0x12	0x76
0x17	0x15	0x17	0x76						

The command line to see the above is:

```
./cae-xor -x '{|}~' -D > example4.xor
```

The type the letters `abcdefghi` and the enter key (the newline character), type the letters `jkl` and the enter key (the newline character). Then type a Control-d to terminate input.

You can decrypt the output file (`example4.xor`) with the following command:

```
./cae-xor -x '{|}~' -d -D < example4.xor
```

You can display the output file using `hexdump` with the following command:

```
hexdump -C example4.xor
```

Your program must read input from **`stdin`** and then write back to **`stdout`**. **Do not use** the `fread()`, `fwrite()`, or `dprintf()` functions for i/o with `stdin` and `stdout`. Rather, use the `read()` and `write()` functions. I will update the testing script to give a grade of zero if the `fread()`, `fwrite()`, or `dprintf()` functions are used.

Command line options:

Option	Action
`-e`	Encrypt plaintext (this is the default action).
`-d`	Decrypt the cipher text.
`-c str`	The encryption string to use for the Caesar-like encryption.
`-x str`	The encryption string to use for the xor encryption.
`-h`	Print some helpful information about command line options and exit.
`-D`	**OPTIONAL**: I use this to display some diagnostics about what is going on inside the process. It is very useful. Remember, all diagnostic information must go to `stderr`, not `stdout`. In fact, I recommend you spend a little time using this option with my solution to see how things are going.

- If neither the `-e` (encrypt) nor the `-d` (decrypt) are on the command line, the default is to encrypt.
- If no key is provided for the Caesar-like cipher, data are passed through the Caesar-like cipher unchanged.
- If no key is provided for the `xor` cipher, data are passed through the `xor` cipher unchanged.
- If no key for either the Caesar-like or `xor` encryption is given on the command line, all data are passed through unchanged.

This makes it easier to test the Caesar-like and `xor` encryption individually.

Use `getopt()` to handle the command line. This is a valid command line:

```
./cae-xor -d -e -d -e -d -c 'abc' -c 'xyz' -e -d -e
```

Your C source file must be named **`cae-xor.c`**. My solution has about 250 lines.

Part 3: `mystat`

For this part of this assignment, you will write a C program that will **display the inode meta data for each file given on the command line**. You must call you source file `mystat.c` and your program will be called `mystat`. **The output of your code must exactly match that of my code.**

An example of how my program displays the inode data can be seen by looking at the output form my program. You might also want to look at the output from the `stat` command (the command not a system function, `man 1 stat`). Though the `stat` command not as pretty (or in some cases as complete as the replacement you will write), it is the standard for showing inode information.

Requirements for your program are:

1. Display the file name
2. Display the file type (regular, directory, symbolic link, …) as a human readable string. **If the file is a symbolic link, look 1 step further to find the name of the**

file to which the symbolic link points. **See** Figure 1**. If the file which the symbolic points does not exist, print "Symbolic link - with dangling destination" next to the file type.**

3. Display the device id, both as its hex value and its decimal value (see the stat command).
4. Display the `inode` value.
5. Display the mode as both its **octal** values and its symbolic representation. The symbolic representation will be the `rwx` string for user, group, and other. **See** Figure 1 or '`ls -l`' for how this should look.
6. Show the hard link count.
7. Show both the `uid` and `gid` for the file, as both the symbolic values (names) and numeric values. This will be pretty darn easy if you read through the list of suggested function calls.
8. Preferred I/O block size, in bytes.
9. File size, in bytes.
10. Blocks allocated.
11. Show the 3 time values (`mtime`, `atime`, and `ctime`) in seconds since the epoch and then both local and GMT time and date. This will be pretty darn easy if you read through the list of suggested function calls.

```
lrwxrwxrwx 1 rchaney them 49 Apr 10 08:47 FUNNYbroken -> /u/rchaney/Classes/cs333/Labs/Lab1/DOES_NOT_EXIST
 drwxr--r-- 2 rchaney them  2 Apr 10 08:30 FUNNYdir
 -rw-r-xr-x 2 rchaney them 11 Apr 10 08:31 FUNNYhardlink
 ---------- 1 rchaney them 10 Apr 10 08:33 FUNNYnoaccess
 prw-r----- 1 rchaney them  0 Apr 10 08:32 FUNNYpipe
 -rw-r-xr-x 2 rchaney them 11 Apr 10 08:31 FUNNYregfile
 srwxrw-rw- 1 rchaney them  0 Apr 10 08:36 FUNNYsocket
 lrwxrwxrwx 1 rchaney them 12 Apr 10 08:31 FUNNYsymlink -> FUNNYregfile
```

Figure 1: The FUNNY* files.

System and function calls that I believe you will find interesting include: `stat()` and `lstat()` (you really want to do "`man 2 stat`" and read that `man` entry closely, all of it [**yes really, all of it**]), `readlink()`, `memset()`, `getpwuid()`, `getgrgid()`, `strcat()`, `localtime()`, `gmtime()`, and `strftime()`. Notice that `ctime()` is NOT in that list and you don't want to use it. Since you must be able to show the file type if a file is a symbolic link, I encourage you use `lstat()` over `stat()` (at least initially).

My implementation is nearly 400 lines long, but I have a **lot of comments and dead code** in my file. I have code commented out to support features not required for your assignment. There is no complex logic for this application, just a lot of long code showing values from the struct stat structure from `sys/stat.h`. Honestly, the longest portion of your code will likely be devoted to displaying the symbolic representation of the mode. Formatting these strings is a little *awkweird*. I suggest you create a function. My function is in desperate need of a dramatic refactor.

You need to be able to show the following file types: regular file, directory, character device, block device, FIFO/pipe, socket, and symbolic link.

When formatting the human readable time for the local time, I'd suggest you consider this "`%Y-%m-%d %H:%M:%S %z (%Z) %a`", but read through the format options on `strftime()`.

```
rchaney # d /dev/vda /dev/mem
crw-r----- 1 root kmem   1, 1 Mar 30 19:21 /dev/mem
brw-rw---- 1 root disk 252, 0 Mar 30 19:21 /dev/vda
```

I have some examples in my `Lab1` directory for you to use in testing (the `FUNNY*` files, Figure 1). You can find a block device as `/dev/mem` and a character device as `/dev/vda`. See Figure 2.

I use spaces for formatting the output to look nice (always a priority for me).

Here is an example of running mystat on the `FUNNYhardlink` file:

```
File: /u/rchaney/Classes/cs333/Labs/Lab1/FUNNYhardlink
  File type:                regular file
  Device ID number:         66
  I-node number:            5203335
  Mode:                     -rw-r-xr-x        (655 in octal)
  Link count:               2
  Owner Id:                 rchaney           (UID = 18781)
  Group Id:                 them              (GID = 300)
  Preferred I/O block size: 1048576 bytes
  File size:                11 bytes
  Blocks allocated:         1
  Last file access:         -107617704 (seconds since the epoch)
  Last file modification:   -107617704 (seconds since the epoch)
  Last status change:       1744300868 (seconds since the epoch)
  Last file access:         1966-08-04 03:11:36 -0700 (PDT) Thu (local)
  Last file modification:   1966-08-04 03:11:36 -0700 (PDT) Thu (local)
  Last status change:       2025-04-10 09:01:08 -0700 (PDT) Thu (local)
  Last file access:         1966-08-04 10:11:36 +0000 (GMT) Thu (GMT)
  Last file modification:   1966-08-04 10:11:36 +0000 (GMT) Thu (GMT)
  Last status change:       2025-04-10 16:01:08 +0000 (GMT) Thu (GMT)
```

Notice that the time is given in 3 formats:

1. Seconds since the epoch (the "seconds since the epoch" at the end is something I put into the `printf()` string)
2. The local time (the "local" at the end is something I put into the `printf()` string)
3. The time in UTD (the "GMT" at the end is something I put into the `printf()` string)

If you want to see how my solution displays all the data for the FUNNY* files, run the following command:

```
~rchaney/Classes/cs333/Labs/Lab1/mystat ~rchaney/Classes/cs333/Labs/Lab1/FUNNY*
```

Testing Your Code (STILL UNDER CONSTRUCTION)

There **WILL BE** 3 scripts you can use for some form of testing for this assignment.

1. `Makefile-test.bash` **NOT READY YET**
2. `cae-xor-test.bash` **NOT READY YET**
3. `mystat-test.bash` ready to try

I recommend that you create symbolic links to each of them, not copy them. The command to create symbolic links to them is:

```
ln -sf ~rchaney/Classes/cs333/Labs/Lab1/*.bash .
```

The `Makefile-test.bash` only tests your `Makefile` (or `makefile`). It checked that all programs are built, without warnings from the compiler. It also checks to make sure all the right flags are used with `gcc`.

The scripts `cae-xor-test.bash` and `mystat-test.bash` (when completed) will be used to test your `cae-xor`, and `mystat` programs.

Submitting your code

When you are ready to submit your code, create a single `tar.gz` file for the C source files and the `Makefile`. Upload the file into Canvas. Your submitted file `tar.gz` should contain exactly 3 files: **`cae-xor.c`**, **`mystat.c`**, and **`Makefile`**. Additional files or any directories can result in a grade of zero. Creating a target in your Makefile to automate creation of the `tar.gz` file is an excellent decision. If you don't remember how to do this, check the slides about `make`, or ask me or the TA.

Final note

The labs in this course are intended to give you basic skills. **In later labs, we *assume* that you have mastered the skills introduced in earlier labs.** If you don't understand, ask questions.