# Mysterious Parody Bits

## Lab 1 - COMP211 - Fall 2020

"How often have I said to you that when you have eliminated the impossible, whatever remains, however improbable, must be the truth?"

~ Sherlock Holmes, *The Sign of Four (1890)*

## Part 2 of 3. Doughnut Holes and `parity` Bits

Perhaps you've noticed US ASCII is a 7-bit encoding although a `char` can hold 8-bits. The 8th bit is often used for locale-specific encodings, such as fancÿ Ümlaut marks. Since the mystery you are solving contains highly sensitive messages, you'll instead use the 8th bit for *parity*, a technique to help detect corrupted data.

### A Parity Parody

Ancient lore has it if you squint and look closely at your computer's memory you just might find each `char` is made of tiny doughnuts and french toast sticks, often represented as 0s and 1s. If you do find this, you should eat them, that'd be one delicious byte.

The idea of *even parity* is you use a spare bit to always have an even number of french toast sticks in a bitfield. After encoding with even parity, if you later count an odd number of french toast sticks in a byte, then you've detected data *corruption*.

In this lab, you will use *even parity* to ensure an even number of 1s in every byte. One way to do this is to count the number of 1s in the 7-bit ASCII value. If there are an odd number of 1s, then you will set the parity bit to 1 bringing the total number back to even. If there are already an even number of 1s, you will leave the parity bit 0. Here are two examples:

| ASCII | Hex | Decimal | 7-bit Binary | Even Parity Bit |
|---|---|---|---|---|
| 1 | 31 | 49 | 0110001 | 1 |
| 3 | 33 | 51 | 0110011 | 0 |
| C | 43 | 67 | 1000011 | 1 |
| Z | 5A | 90 | 1011010 | 0 |

### Super Shifty Security

Additionally, our parody parity scheme will add *security through obscurity* (that means: no security whatsoever). Rather than encode the parity bit in the highest-order place value, you should encode it in the lowest-order place value, as shown below:

| ASCII | Hex | Decimal | 7-bit Binary | 8-bit w/Parity | Hex w/ Parity |
|---|---|---|---|---|---|
| 1 | 31 | 49 | 0110001 | 01100011 | 63 |
| 3 | 33 | 51 | 0110011 | 01100110 | 65 |
| C | 43 | 67 | 1000011 | 10000111 | 87 |
| Z | 5A | 90 | 1011010 | 10110100 | B4 |

## Code

Complete your work for this part of the lab in `parity.c`. Include the standard header, as usual, and use `gcc` to compile a binary named `parity`.

Since the bytes generated by `parity` will often be outside the visible, 7-bit ASCII range, you have two strategies for testing. First, if you only encode numerical ASCII digits '1'-'9', then their encodings will be visible ASCII characters. Second, if you pass the `parity` output through your `hex` program from part 1, then you will see a hex representation of the output. Both directions are demonstrated below:

```
# Numbers parity encode to visible characters
$ echo "123456789" | ./parity
cefijloqr
# For other tests, use parity first, then hex encode
$ echo "abcdef" | ./parity | ./hex
C3C5C6C9CACC140A
# Compare with the output of ONLY hex encoding
$ echo "abcdef" | ./hex
6162636465660A
```

When `EOF` is encountered in the input of `parity.c`, you should emit a new line character.

## To a great mind, nothing is little

Once your `parity` encoding is working, pipe the output of echoing the following string into your parity encoder and then pipe its output into your hex encoder (be careful to capitalize the first T and include the trailing period!):

*The truth is out there.*

Take the resulting secret value and navigate to http://bit.ly/your-encoding-here to reach the final part of this lab.