ECE/CSE 474
Assignment 2
C programming with bit-manipulation

(V2.0 rev. for clarification 20-Apr-21)

(V2.1 typos & func. names 30-Nov-21

In this assignment we will learn how to inspect and modify individual bits in memory. Remember that **everything** in computers is represented by just a series of 0's and 1's. This means that the software has to remember how each part of memory is converted from 0's and 1's to something else. Here's a simple example: Suppose the memory location 0xAFF5C3 (that's a hex number, don't worry about it now) contains

`0xAFF5C3` →

0	0	1	0	1	0	1	0

We will focus on the exact bits in memory and the powerful (but yes, low-level) C-functions which allow you to work with them.

This 8-bit segment of memory could be a `uint8_t` type (an 8-bit integer between 0-255). In that case, `00101010` equates to 2+8+32 = 42. However, if it is a `char` (character), the same pattern of `00101010` equates to '*' (the asterisk character).

This is important because hardware control bits need to be set or cleared in order to control a computer chip from the inside. You could turn on or off on-chip peripheral devices to save battery life for example.

Template Start with the template file, `c_asgn02_empty_Sp21.c`, which you will complete in this assignment. The file contains comments which indicate where to put each part of your code. There is already a `main()` which will run the different pieces of your code and `printf` statements which will separate the output from your different sections. The `main()` function also contains pre-written test code which will verify that your functions are working. Blank spaces for your functions are at the end of the file. **Please read the entire file before starting work so you understand where things go.** Please do not modify the test code. There is also a sample output file which shows you what the completed code should do.

Compiler Differences You may do this assignment on Linux or on Microsoft Visual C or any other C Compilers. There are a few small differences between C compilers. The sample code should compile on both linux (gnu gcc compiler) and Visual C if you set the appropriate `#define` statement near the top of the code file.

Printing In Lab 1 we used pre-defined simplified printing. Now, study the strings and codes used in the simplified printing functions and the several `printf()` statements in the C Assignment 02 test code. A good reference on `printf()` is [Here]

2 C Bitwise Operations

2.1 Bitwise Operations

We are going to take a number (such as your UW student ID) and mangle it beyond recognition. Write the function `int mangle(int sid){}` which performs the following three steps on any `int`:

1. right shift it by 2 bit positions
2. clear bit 7 (after the shift)
3. complement bit 4 (e.g. 00001000). If bit 4 is one, make it 0. If bit 4 is 0, make it 1.

2.1.1 More Bit Manipulation

Write the function `bit_check(int data, int bits_on, int bits_off)`, to check an `int` to see if a specific set of bits (aka a "bit mask", call it `bits_on`) is set AND that another set of bits (`bits_off`) is clear. Returns 1 if the `int` called `data` matches the bit masks, and 0 if not.

However, note there is a special case. For example, suppose I write

```
int data = 0xFF;
bit_check(data, 0x32, 0x20);
```

there's a problem! Because `0x32 & 0x20 = 0x20` , we are asking that bit 0x20 must be **BOTH** off and on! Your function should return -1 for any case of testing that any bit is BOTH off and on.

2.2 Basic Pointer Declaration and use

In this part we will use pointers to designate capitol letters of the alphabet which are put into memory through the pre-defined array `a_array[]`.

Write a function `pmatch()` which will take a character and return a pointer to the element in `a_array[]` which matches it. If there is no such character (e.g. `pmatch('m')`), then return the `NULL` pointer.

2.3 Pointer Arithmetic

`nlet(C)`

You can meaningfully do certain kinds of math with pointers (but e.g. `x = sqrt(ptr)` is meaningless). Here you should write a function to give the next letter in the alphabet in response to a pointer into the alphabet array. If the pointer points to 'Z', there is no next letter so return -1. if the pointer does not point to an element of the array, also return -1.

2.4 Pointer Arithmetic II

`ldiff(C1,C2)`

Find the number of characters between two letters of the alphabet using pointer arithmetic. Include C2 in the count but not C1. For example `ldiff('A','E') == 4` is true and `ldiff('E','A') == -4` is true. Check for errors. If either letter is not a capital letter, return a negative number less than -26.

2.5 Structs and sizeof

You might have used classes in a previous programming course/language. In C we don't have classes but `structs` are basically like a class with no methods.

Here you are going to set up a `struct` to represent some data about people, and a couple of functions which work with pointers to those structs. Specifications for the `struct` are:

```
/*
 * Define a typedef struct  called Person containing:
 *     1) 20 char string: FirstName
 *     2) 30 char string: LastName
 *     3) 80 char string: StreetAddr
 *     4)  5 char string: ZipCode
 *     5)    double: Height  // height in meters
 *     6)    float: Weight  // weight in kg
 *     7) long int: DBirth  // birthday (days since 1-Jan-1900)
 * /
```

Your function `print_person()` must accept a pointer to a `Person` struct, and `neatly` print out all data (see sample output). The street address can store 80 characters (see above) but the printout should be limited to 60 characters.