Instructions:

- Print your name in the space provided below.
- This examination is closed book and closed notes, aside from the permitted one-page formula sheet. No calculators or other electronic devices may be used. The use of any such device will be interpreted as an indication that you are finished with the test and your test form will be collected immediately.
- Answer each question in the space provided. If you need to continue an answer onto the back of a page, clearly indicate that and label the continuation with the question number.
- If you want partial credit, justify your answers, even when justification is not explicitly required.
- There are 5 questions, some with multiple parts, priced as marked. The maximum score is 100.
- When you have completed the test, sign the pledge at the bottom of this page and turn in the test.
- If you brought a fact sheet to the test, write your name on it and turn it in with the test.
- Note that either failing to return this test, or discussing its content with a student who has not taken it is a violation of the Honor Code.

Do not start the test until instructed to do so!

Name __________________________________________________________________________

Printed

Pledge: On my honor, I have neither given nor received unauthorized aid on this examination.

__________________________________________

Signed
Okay, Human.
Huh? * Looking at the screen.*
Before you hit `compile`, *looking at you.*
Listen up.

You know when you're falling asleep, and you imagine yourself walking or something.

And suddenly you misstep, stumble, and jolt awake?

Yeah!

Well, that's what a segfault feels like.

Double-check your damn pointers, okay?

xkcd.com
1. [25 points] Recall the Untangle assignment from earlier in the semester. In this question you will write a similar C function where you iterate through a series of records in a memory block.

Each record is made up of 3 bytes, a 1 byte unsigned integer value, and then a 2 byte unsigned integer value. In the example memory dump below the first integer in each record is **bolded** and the second is *italicized*. The first record starts at the first byte of the memory region pointed to by pBuffer.

To get to the next record, you need examine the both integers. The 2 byte value is next the offset, but the 1 byte value indicates the byte ordering of that offset. If any even bit (at bit position 0, 2, 4, …) in the byte sized value is set to 1, then the byte ordering for the 2 byte value is **big endian**, otherwise the byte ordering is **little endian**. Your program should continue processing records until the next record location (or offset) is zero.

Recall when a number is stored using big endian byte ordering, the high order byte is at the low address. So the first two byte values below should be interpreted as 0x000F, rather than 0x0F00. This means you'll have to manually reorder the bytes.

```c
uint16_t miniUntangle(const uint8_t* pBuffer) {
    uint16_t count = 0;
    uint16_t offset = 0;

    // Some sort of loop that runs until the offset is 0,
    do {
        // Get the next record location.
        const uint8_t * next = pBuffer + offset;
        offset = *((uint16_t *) (next + 1));

        // If statement and mask that gets that determines
        // if the right bits are set or unset.
        if (*next & 0x55) {
            // Transform little endian to big endian.
            offset = (offset << 8) | (offset >> 8);
        }

        // Keeps a count of the number of records
        count++;
    } while (offset != 0);

    // Returns the number of records.
    return count;
}
```

```plaintext
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
---------------------------------------------------------
01 00 0F 07 2b 2d f6 28 77 fd 47 b4 0f b2 0e 05 |....+-.(w.G...f.| 00 17 10 8c 77 9e 00 02 00 00 13 5a 20 41 0a 02 |....w....%Z A...|
00 00 00 01 11 11 6b 0e e8 67 e4 0b 2b 89 15 16 19 |...k...j...%QR...|
00 00 00 00 d0 f7 ea f6 51 e4 c7 00 0f ff e8 7a 33 f3 7e 0c |....Q......23.~.|
```
Implement the C function described below. Be sure to note the restrictions and pre and post conditions, failure to meet these conditions will result in a low score.

```c
/**
 * leftpad - Pads the left side of a string with the provided character.
 * 
 * Pre:
 * str - Points to a properly formatted C string.
 * len - Is the number of characters in str, excluding the terminator.
 * c - Is the character we'll use when adding padding to str.
 * num - Is the number of characters to pad.
 * 
 * Post: str has num of character c padded on the left side of the string.
 * The original characters in str are preserved and are shifted out
 * num indices. For example:
 * char hello[100] = {'H', 'e', 'l', 'l', 'o', '!', '\0'};
 * leftpad(hello, 6, '_', 4);
 * printf("'%s'", hello);  // prints '____Hello!' 
 * 
 * Restrictions and Additional Info:
 * - You may NOT use any function declared in string.h
 * - You may NOT use dynamic allocation or a separate array, i.e.
 *   you must move the characters within str.
 * - You may assume that str is long enough to hold:
 *   the original string + num padding characters + 1 null terminator.
 * - Take care to preserve the null terminator or re-add it.
 */
void leftpad(char * str, size_t len, char c, size_t num) {
    // Preserve the NULL terminator with the + 1 below in the loop.

    // Loop to shift out the characters in str, by num slots.
    for (size_t x = 0; x < len + 1; x++)
        str[len - x + num] = str[len - x];

    // Loop to copy num padding characters into str.
    for (size_t x = 0; x < num; x++)
        str[x] = c;
}
```
3. [20 points] Implement the C function described below. Be sure to note the restrictions and pre and post conditions, failure to meet these conditions will result in a low score.

/* Say you could capture the output of your print2505() function in a string, reverse_print2505() would take that output string as input and would produce the corresponding formatting string. For simplicity we won't worry about big or little endian, so we won't include 'b' or 'l' in our specifiers, further, the function should work for 1 or 2 byte values.
 * Pre: input - Points to a properly formatted C string.
 * len - Is the number of characters in input, excluding the terminator.
 * Returns: The function creates and returns the corresponding formatting string. You should choose the smallest number of bytes that still "fit" a number. For example:
 * // Creates and returns "Hello World! % %1 %2".
 * reverse_print2505("Hello World! % 123 256", 22);
 *
 * Restrictions and Additional Info:
 * - You may NOT use any function declared in string.h, ctype.h, or math.h
 * - Dynamic allocation will be required.
 */
char * reverse_print2505(char * input, size_t len) {
    // Pointers are used below but array brackets are fine too.
    char *result, *tmp;

    // Malloc space for the string, the number of bytes should be
    // related to the len parameter. The following size is used since
    // it should be large enough for any string, at most the length
    // should double, plus null terminator. e.g. '1', then I need 2 slots
    // for the '%1' and 1 slot for the null terminator.
    result = tmp = malloc(2 * len + 1);

    while(*input) {
        if (*input < '0' || *input > '9') {
            // Handle '%' characters
            if(*input == '%') *tmp++ = '%';

            // Case for characters other than a numbers
            *tmp++ = *input++;
        } else {
            // Case for a number.
            uint32_t num = 0, multiplier = 1;
            char *leader = input;

            *tmp++ = '%';

            // Count the number of digits in the number.
            while(*leader >= '0' && *leader <= '9') {
                multiplier*=10;
                leader++;
            }

            // Convert to an integer.
            for (; input != leader; input++) {
                multiplier /= 10;
                num += multiplier*(*input - '0');
            }

            // Depending on the number of bytes either
            // write a 1 or 2 to the string
            *tmp++ = (num / 256 == 0) ? '1' : '2';
        }
    }

    // Add the NULL terminator
    *tmp = '\0';

    return result;
}
4. **[15 points]** Perform the indicated conversions. You must show work in order to receive credit!

   a) Convert from 8-bit 2's complement (i.e., signed) form to base-10: 1010 1101

   First, take the negation in 2’s complement: 0101 0011
   Then, expand the positional representation: $2^6 + 2^4 + 2^1 + 2^0 = 64 + 16 + 2 + 1 = 83$
   Finally, take the negation of the result: $-83$

   b) Convert from 8-bit unsigned form to base-10: 1010 1101

   Expand the positional representation: $2^7 + 2^5 + 2^3 + 2^1 + 2^0 = 128 + 32 + 8 + 2 + 1 = 171$

   c) Convert 0x25 hexadecimal to base-10:

   Expand the positional representation: $2*16^1 + 5*16^0 = 37$
5. **[20 points]** Consider the following C struct:

```c
struct _String {
    char *word;
    int len; /* # of characters in the word, excluding the null terminator */
};
typedef struct _String String;
```

All 128 ASCII characters can be represented using only 7-bits, i.e., a char variable always leaves the most significant bit (MSB) unused. Here, we’d like to leverage the MSB for an even parity bit that is defined as below.

For the seven bits of a given ASCII character, the occurrences of bits whose value is 1 are counted. If that count is odd, the parity bit value is set to 1, making the total count of occurrences of 1’s in the whole set (including the parity bit) an even number. If the count of 1’s in the given 7 bits is already even, the parity bit’s value remains 0.

Implement the C function below to update each character of `word` in the `String` struct.

```c
/* Pre: arg - is a pointer to a String instance that is initialized correctly, 
   i.e., word points to some string in memory, and the len is assigned 
   the length of the string excluding the null terminator. 
* Post: 
*      String s; 
*      s.word = malloc (sizeof (char) * 10); 
*      strcpy (s.word, "AC"); // s.word[0] is 65 (A in ASCII), s.word[1] is 67(C) 
*      // Their binaries are 1000001 and 1000011. 
*      s.len = strlen((char*)s.word); // len is 2 
*      parity_update (&s); 
*      // s.word[0] becomes 65 (0100001), i.e., the parity bit is 0 
*      // s.word[1] becomes 195 (1100011), i.e., the parity bit is 1 
*      
* Restrictions and Additional Info: 
*      - You may use only |, &, >> to count 1s of each character 
*      - you must use a loop to count in combination with the bitwise ops 
*      - You may assume the type of word's characters is the same as uint8_t 
*/
void even_parity_update (String* arg) {
    // Loop to read each byte (word) of String, 
    for (int i = 0; i < arg->len; i++) {
        int count = 0;
        int x = arg->word[i];
        while (x) { // Loop to count 1’s, 
            if (x & 1) {
                count++;
            }
            x >>= 1;
        }
        // Set the MSB for even parity, 
        if (count % 2 == 1) { // for odd parity 
            arg->word[i] |= 0x80;
        }
    }
}
```