C Programming Pointer Casts and Data Accesses

For this assignment, you will implement a C function similar to `printf()`. While implementing the function you will encounter pointers, strings, and bit-wise operations. The function must conform to the following interface specification:

```
// print2505() is our variant of the of the printf() function.
//
// Like printf(), our function has a formatting string and conversion
// specifiers. Unlike printf(), the data or variables we're printing
// reside in the array pointed to by the data parameter. Further, for
// simplicity our function will only print unsigned numbers.
//
// Pre:
//      out    - Points to an already opened file stream.
//      format - Points to a C string containing formatting information
//               and is not NULL.
//      data   - Points to array of uint8_t values or may be NULL when there
//               are no conversion specifiers in the formatting string.
//
// Post:
//      The conversion specifiers and characters in the formatting string are
//      printed using the provided char_out() function. If an invalid
//      conversion specifier is encountered then no more conversion specifiers
//      or characters are processed or printed.
//
// Returns:
//      The number of characters printed. If an invalid conversion specifier is
//      encountered then return -1.
//
// Restrictions:
//      -You may NOT use array brackets for this assignment. Any array
//       accesses must use pointer arithmetic and the dereference operator.
//      -You may NOT use I/O, e.g. printf(), scanf(), fgets(), or variants.
//      -You may NOT use arrays, beyond the given parameters format and data.
//      -You may NOT use other libraries or built-in functions to convert to
//       or from big or little endian format.
//      -You may NOT use math.h or string.h, nor any function declared within.
//      -Further, you MUST use the provided char_out() function to print the
//       characters one at a time.
//
// See the header comments for more examples and information.
//
int print2505(FILE *out, const char *format, const uint8_t *data);
```

As mentioned above, our function takes a formatting string like `printf()`. Unlike `printf()`, any variables or values your function prints will reside in the array pointed to by the `data` parameter. Each time a conversion specifier is encountered in the formatting string, your function will access some of the bytes in the array. Like `printf()`, conversion specifiers start with `%`, followed by a number (`1`, `2`, `4`, or `8` only), which is the number of bytes for the whole value, then a lower case `'b'` or `'l'` for big endian or little endian byte ordering.

The following are the valid conversion specifiers for `print2505()`:

```
%1b, %1l, %2b, %2l, %4b, %4l, %8b, %8l
```

For single byte values, `%1b` and `%1l`, simply fetch a byte from the array. `%2b` will get a 2 byte value from the array pointed to by `data`, interpreted in big endian order, while `%2l` would get a 2 byte value, interpreted in little endian order, etc. Big endian ordering puts the most significant byte in the lowest address (and least significant byte in the highest address), while little endian ordering puts the most significant byte in the highest address (and least significant byte in the lowest address). In this context, the address is the array index (or pointer offset), so the 0 index is the lowest address. Since you can't use array indices for this assignment, using pointer arithmetic `data + 0` is the lowest address and `data + 1` is the next lowest address, etc.

`print2505()` also supports `%%`, a special case, which prints the literal `'%'` and doesn't examine the data parameter.

For example, say we that invoke `print2505()` with the formatting string and array shown below. The `%%` just prints `'%'`, but each of the remaining conversion specifiers accesses and interprets the bytes in the array.

```
uint8_t data[] = {0x01, 0x00, 0x00, 0x88, 0x00, 0x0a, 0x01, 0x00,
                      0x0b, 0x02, 0x20, 0x03, 0x01, 0x01, 0x02, 0x0b,
                      0x02, 0x01, 0x04, 0x01, 0x01, 0x02, 0x0c, 0x01,
                      0x00, 0x07, 0x01, 0x01, 0x05, 0x01};
```

```
print2505(out, "%% %1b, %1l, %2b, %2l, %4b, %4l, %8b, %8l", data);
```

The color-coded conversion specifiers above correspond to the color coded chunks of memory shown below. This table was created with `hexdump`, but this is just another way to visualize the array shown above. Each value in the table is a **byte**. **Note**: that the value for each byte is shown in a human-readable format where high-order bits appear on the left while the low-order bits left on the right, so `0x01` is the value `1`, not `2`.

```
            0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
----------------------------------------------------------------------------
00000000   01 00 00 88 00 0a 01 00  0b 02 20 03 01 01 02 0b  |................|
00000010   02 01 04 01 01 02 0c 01  00 07 01 01 05 01        |................|
----------------------------------------------------------------------------
            0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
```

The function sequentially reads bytes, where the number of bytes for each value is determined by the conversion specifier, until there are no more conversion specifiers and we reach the end of the formatting string. Below is the output corresponding of the function call above:

```
% 1, 0, 136, 2560, 16780034, 16843552, 147213616205070594, 73466072845517068
```

Getting Started

Begin by downloading the posted files:

`main.c`	sample test driver; modify as you like
`print2505.h`	C header file for compiling with test driver --- do not modify!
`print2505.c`	C source file for implementation of the `print2505()` function
`char_out.h`	C header file for compiling with test driver --- do not modify!
`char_out.c`	C source file for the implementation of the `char_out()` function --- do not modify!

The file `print2505.h`, includes header comments and a few more examples. Pay attention to the comments in the header file. All the stated pre- and post-conditions are part of the assignment.

You can compile these files by using the following command in a Linux shell. **A warning:** if you copy and paste the `gcc` command from the `pdf` document, the dashes (-) don't always copy correctly and you'll receive an uninformative error message from `gcc`. Either type out the command by hand or delete and retype the dashes after copying this command.

```
gcc -o driver -std=c99 -Wall main.c print2505.c char_out.c
```

Using this command with the supplied files will yield an executable named `driver`. You can execute the testing code by using the following command in a Linux shell:

```
./driver
```

Although the given code will compile correctly, the resulting program will not satisfy the requirements of the assignment, since the supplied implementation of the required function doesn't do anything. So, you must correctly complete the implementation of the required function.

You may, and should, modify the supplied testing code since it's not exactly thorough. But, if your solution doesn't compile with the supplied testing code, it won't compile with the real testing code either.

It's not likely with the given restrictions, but you may need to add `#include` directives to your `.c` file for any C Standard Library features you use. You may write secondary "helper" functions if you like; if so, those must be defined and declared within the supplied `print2505.c` file. Such functions should be declared as `static`.

What to Submit

You will submit your `print2505.c` file, containing the implementation of the specified C function. Be sure to conform to the specified function interfaces.

Your submission will be compiled with a test driver and graded according to how many cases your solution handles correctly.

This assignment will be graded automatically. You will be allowed up to ten submissions for this assignment. Test your function thoroughly before submitting it. Make sure that your function produces correct results for every test case you can think of.

Any additional requirements specified above will be checked manually by a TA after the assignment is closed. Any shortcomings will be assessed a penalty, which will be applied to your score from the Curator.

The course policy is that the submission that yields the highest score will be checked. If several submissions are tied for the highest score, the latest of those will be checked.

The *Student Guide* and other pertinent information, such as the link to the proper submit page, can be found at:

http://www.cs.vt.edu/curator/

Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the following pledge statement in the submitted file:

```
//     On my honor:
//
//     - I have not discussed the C language code in my program with
//       anyone other than my instructor or the teaching assistants
//       assigned to this course.
//
//     - I have not used C language code obtained from another student,
//       or any other unauthorized source, either modified or unmodified.
//
//     - If any C language code or documentation used in my program
//       was obtained from an allowed source, such as a text book or course
//       notes, that has been clearly noted with a proper citation in
//       the comments of my program.
//
//     <Student Name>
```

Failure to include this pledge in a submission will result in the submission being disallowed during code review.

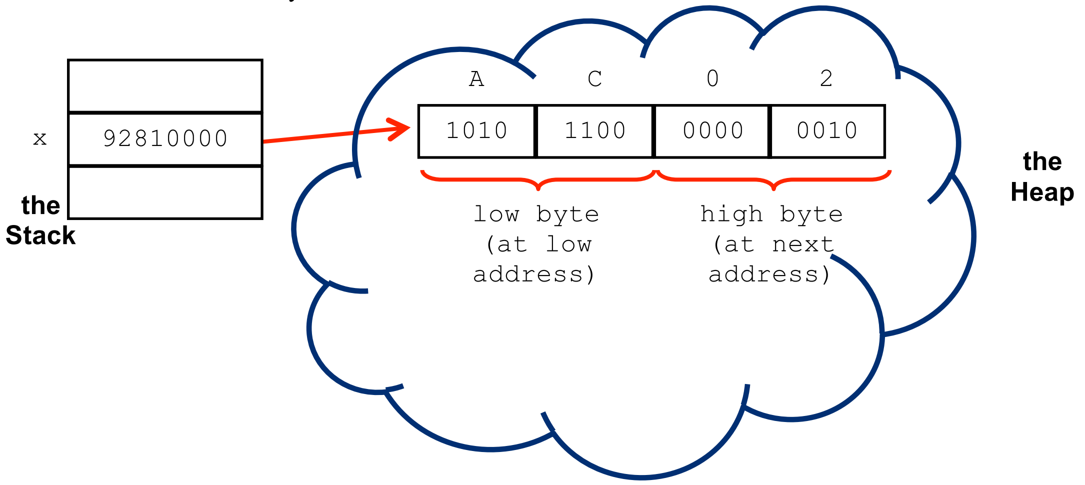
Brief tutorial on using pointer arithmetic and pointer typecasts in C:

This assignment does not require any esoteric portions of the C language, but it does require an understanding of pointer manipulations. First of all, you must understand the effect and uses of a pointer typecast. Consider the following snippet:

```
uint16_t *x = malloc(sizeof(uint16_t));     // 1
*x = 684;                                   // 2
```

Statement 1 causes the allocation of a two-byte region of memory, whose address is stored in the pointer `x`. Statement 2 stores the value 684 (`0x2AC` or `0000 0010 1010 1100` in binary) into that two-byte region. Let's assume that the address returned by the call to `malloc()` was `0x00008192`.

So the current situation in memory would be:



(The bytes are stored in little-endian order, just as they would be on any Intel-compatible system.) If you dereference the pointer `x`, you'll obtain the contents of the two bytes beginning at the address stored in `x`. That's because `x` was declared as a pointer to something of type `uint16_t`, and `sizeof(uint16_t)` is 2 (bytes).

Now consider:

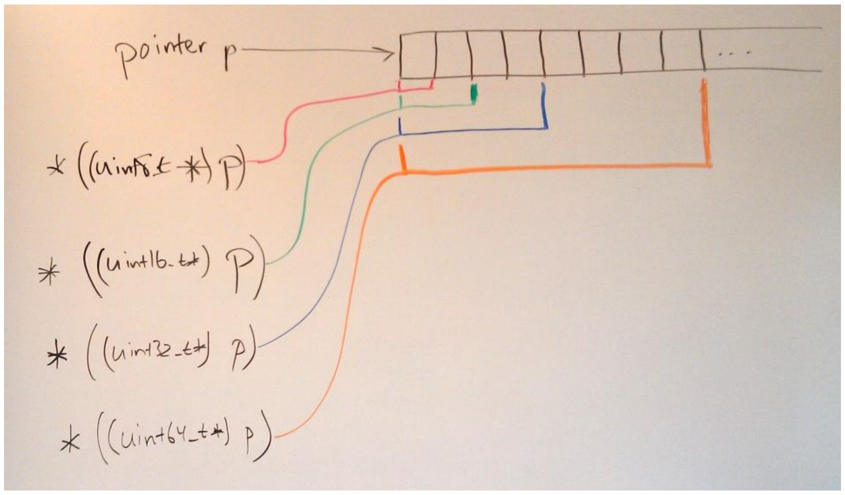
```
uint8_t *y = NULL;                      // 3:   y == 0x00000000
y = (uint8_t*) x;                       // 4:   y == 0x00008192

uint8_t   z = *y;                       // 5:   z == 0xAC     (1 byte)

uint16_t  w = *x;                       // 6:   w == 0x02AC   (2 bytes)
```

The effect of statement 4 is that `y` takes on the same value as `x`; pointer variables are 32 (or 64 bits) bits wide, regardless of the type of target they may take, and so the value of `x` will fit neatly into `y`. So, why the typecast? Simply that C is somewhat picky about assigning a pointer of one type to a pointer of another type, and the typecast formalizes the logic so that the compiler will accept it. If you dereference `y`, you'll obtain the contents of the single byte at the address stored in `y`, since `sizeof(uint8_t)` is 1. Hence, statement 5 will assign the value `0xAC` or `172` to `z`, but statement 6 will assign the two-byte value `0x02AC` or `684` to the variable `w`.

Here's a brief summary:



Note how we can use the type of a pointer to determine how many bytes of memory we obtain when we dereference that pointer, as well as how those bytes are interpreted. This can be really useful.

The second thing to understand is how pointer arithmetic works. Here is a simple summary:

```
T *p, *q;       // Take T to represent a generic type.
. . .           // Assume p gets assigned a target in here.
q = p + K;      // Let K be an expression that evaluates to an integer.
```

Then the value of `q` will be: `p + K * sizeof(T)`. Note well: this is very dangerous unless you understand how to make use of it. In some respects, this is really quite simple; maybe too simple. The essential thing you must always remember is that if you want to move a pointer by a specific number of bytes, it's simplest if the pointer is a `char*` or `uint8_t*`, since the arithmetic will then provide you with byte-level control of the pointer's logical position in memory.

The following loop would walk the pointer `y` through the bytes of the target of `x` (the `uint16_t*` seen earlier):

```
uint8_t *y = (uint8_t*) x;

uint32_t bytesRead = 0;

while ( bytesRead < sizeof(uint16_t) ) {

   printf("%"PRIx8"\n", *y);  // print value of current byte in hex;
                              //    'x' causes output in hex;
                              //    'X' capitalizes the hex

   ++y;                 // step to next byte of target of x
   ++bytesRead;
}
```

You might want to experiment with this a bit…

Now for copying values from memory into your variables (which are also in memory also, of course)... The simplest approach is use appropriate variable types and pointer typecasts. Suppose that the `uint8_t` pointer `p` points to some location in memory and you want to obtain the next four bytes and interpret them as an `int32_t` value; then you could try these:

```
int32_t  N = *p;                // NO.  This takes only 1 byte!

int32_t *q = (int32_t*) p;      // Slap an int32_t* onto the location;
int32_t  N = *q;                // so this takes 4 bytes as desired.

int32_t  N = *((int32_t*) p);  // Or do it all in one fell swoop.
```

The last form is the most idiomatic in the C language; it creates a temporary, nameless `int32_t*` from `p` and then dereferences it to obtain the desired value. Note that this doesn't change the value of `p`, and therefore does not change where `p` points to in memory. So, if you wanted to copy the next few bytes you'd need to apply pointer arithmetic to move `p` past the bytes you just copied:

```
p = p + sizeof(int32_t);  // Since p is a uint8_t*, this will move p forward
                          //    by exactly the size of an int32_t.
```

One final C tip may be of use. The C Standard Library includes a function that will copy a specified number of bytes from one region of memory into another region of memory: `memcpy()`. You can find a description of how to use the function in any decent C language reference, including the C tutorial linked from the Resources page of the course website.

The C Standard, section 6.5.6 says:

> When an expression that has integer type is added to or subtracted from a pointer, the result has the type of the pointer operand. If the pointer operand points to an element of an array object, and the array is large enough, the result points to an element offset from the original element such that the difference of the subscripts of the resulting and original array elements equals the integer expression. In other words, if the expression **`P`** points to the *i*-th element of an array object, the expressions **`(P)+N`** (equivalently, **`N+(P)`**) and **`(P)-N`** (where **`N`** has the value *n*) point to, respectively, the *i*+*n*-th and *i*−*n*-th elements of the array object, provided they exist. Moreover, if the expression **`P`** points to the last element of an array object, the expression **`(P)+1`** points one past the last element of the array object, and if the expression **`Q`** points one past the last element of an array object, the expression **`(Q)-1`** points to the last element of the array object. If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined. If the result points one past the last element of the array object, it shall not be used as the operand of a unary **`*`** operator that is evaluated.