Name: _____ CSCI 237

Practice Problems: C

User-defined data types: the `struct`

Consider the following C structure that contains multiple fields:

```
struct example {
   char a;
   int b;
   char c[2];
   char *d;
};
```

Logically, the struct contains 4 fields, ordered and sized as shown below (each `|----|` represents one byte).

```
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
 a    b    b    b    b    c[0] c[1] d    d    d    d    d    d    d    d    d
```

Given the struct alignment rules, what is the actual size of this structure, in memory?

To programmatically determine the size of any variable or type, we can use the `sizeof` operator (`sizeof` is not a function; it is evaluated at compile-time, so you don't need to worry about dereferencing a null pointer). Below are some `sizeof` examples. For each example, what value is reported by the `sizeof` operator?

`sizeof(int)` _____

`sizeof(struct example)` _____

`sizeof(uint64_t)` _____

`sizeof(char)` _____

`sizeof(char *)` _____

When accessing the fields of a struct, the syntax depends on whether the variable refers to a struct value or a struct pointer. When accessing the fields of a struct value, dot notation is used; when accessing the fields of a struct pointer, arrow notation is used. Note that it is not the type of the particular struct field being accessed that matters!

```
struct example e; // declare a struct variable e
struct example *e_ptr = &e; // pointer with e's addr

e_ptr->a = '!';
e.c[0] = '*';
e_ptr->c[1] = '-';
e.d = "this is a static string";
```

Working with Memory

A `void *` is a "generic pointer". In other words, a `void *` is an address, but it makes no claim about how to interpret any value stored at that address. If we do want a program to interpret the memory pointed to by a `void *`, we must first cast the pointer to a specific type so that we can tell the program to view the memory through that lens.

The `malloc()` function allocates a contiguous chunk of memory and returns a generic pointer (a `void *`) that specifies the start of that newly allocated memory region. Note: the malloc function does *not* initialize the memory in any way, and it may fail and return `0` (a null pointer).

Why might it make sense for `malloc()` to return a `void *` rather than a specific pointer type, e.g., a `char *`?

We can use `malloc()` to allocate memory for a structure (or anything else we want), and the lifetime of that memory will extend beyond the duration of the current function. How would we declare a variable `e_ptr` that is a pointer to a `struct example`, and then assign to `e_ptr` the address of a memory region that is exactly the right size to store a single `struct example`?

If we continue to use malloc over and over, our system will eventually run out of memory. In order to return memory back to the system when we are no longer using it, we must call `free()`. The `free()` function accepts a single argument: a pointer to the start of some memory region that was allocated using `malloc()`. Once a memory region is freed by a call to `free()`, it is no longer accessible. Thus, `free()` can only be called one time per allocated pointer.

Consider the following code that allocates memory for a `struct example` and initializes some of its fields. Write code that frees all of the memory that was allocated by this code. *Hint: the order that the memory is freed in matters!*

```
// code that allocates and partially inits a struct
struct example *e_ptr = malloc(sizeof(*e_ptr));
e_ptr->d = (char *) malloc(14);
memset(e_ptr->d, 0, 14);

...

// code that frees all allocated memory
```

Memory allocated by the `malloc()` function is not initialized. This can be a problem because the memory may have been previously populated with values that would have a meaningful (and incorrect) interpretation if read by the program. We often account for this issue by immediately initializing the memory after a successful call to `malloc()`.
If the allocated memory is a single value or a struct, we may want to assign initial values to the fields themselves. If the allocated memory is a large struct or array, this may be too tedious to do individually, and we may wish to use the `memset()` function (or similar). The `memset()` function takes three arguments: an address, a value, and a length, and it replicates the value a total of len times starting at the specified address. Often we use `memset()` to initialize a memory region to be all 0's.

Write C code below that allocates a `struct example` and initializes each field individually with some starting value.

Write C code below that allocates a struct example and initializes all memory to 0 using `memset()`.

Bitmap Building Blocks

In Lab 4, our bitmap data structure will store and manipulate a set of `n` bits. However, we address memory at byte granularity, so our bitmap's bits will reside in an array of `char` values. The following questions will help you to think about your bitmap's design.

At minimum, how many char values does your bitmap need in order to store 63 bits?

Suppose I want to access the 45th bit in my bitmap. Which byte does this bit fall within? (We will call this value a bit's *bucket index*.)

Within that byte, which bit corresponds to the 45th bit? (We will call this value the *bit index*.) Note that since there are 8 bits in a byte, a bit index falls between 0 and 7.

Use your answers above to write a C expression that calculates the bucket index of the ith bit. (Can you use bit shifts to accomplish this task?)

Use your answers above to write a C expression that calculates the bit index of the ith bit. (Can you use `&` to accomplish this task?)