Lecture 08 - Structs and function pointers (last updated: 09/29/22)

Arrays are but one data structure supported by the C language; they limit us to contiguous elements of the same type, only allow us to access those elements by index (rather than by something more descriptive):

```
    int mypoint[2];     // a pair of integers, e.g., (x, y)

    int x = mypoint[0]; // what we really want to write is mypoint.x
    int y = mypoint[1]; // what we really want to write is mypoint.y
```

In this lecture, we'll learn how to model other in-memory data structures in C.

Structs

Structs allow us to specify a data structure consisting of a fixed number of named elements:

```
    struct Point {
        int x;
        int y;
    };
```

This struct definition must be written at the top-level (e.g., it cannot be inside of a function body); don't forget the trailing ';'!

The above struct definition creates a new type named "struct Point", which we can define and use like so:

```
    struct Point pt;

    pt.x = 2;
    pt.y = 3;
```

Note C syntax requires us to write "struct" before "Point" as the type name.

Structs are so-called because they describe the in-memory structure of a type. This is what pt looks like in memory:

```
    +---+---+---+---+---+---+---+---+
    | .x = 2        | .y = 3        |
    +---+---+---+---+---+---+---+---+
    |----------- struct pt ---------|
```

The struct definition of struct Point tells the compiler to find the .x field at byte-offset 0, and the .y field at byte-offset 4 (because .y comes after .x). So writing:

```
    &pt.y
```

is the same as writing:

```
    (int *) ((char *) &pt + 4)
```

Note that the struct layout (i.e., the byte offsets) may differ depending on what hardware and platform you compile this program for. For example, the layout may look different on a platform where ints are 8 bytes.

Structs can also have fields of different types. For example, the following struct MyString allows us to quickly access the length of a string without scanning it for a null terminator byte:

```
    struct MyString {
        unsigned int len;
        char *s = str;
    };

    struct MyString s;

    s.str = "abc";
    s.len = strlen("abc");
```

The layout might not be exactly what you expect:

```
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | .len = 3      |   (padding)   | .str = (points to "abc")      |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    |--------------------- struct MyString -------------------------|
```

This is because C inserts padding between struct fields to ensure that fields are correctly aligned (e.g., 8-byte pointers should begin at addresses that are a multiple of 8). You do not need to know the precise algorithm, but you should keep a few things in mind:

- The first field of a struct is always at offset 0; padding will never be inserted before the first field.
- Struct padding may also be added at the end of structs, to ensure that arrays of structs are also correctly aligned.
- The size of structs may end up larger than you expect due to padding. When in doubt, use sizeof and field names rather than computing offsets yourself.

The size and layout of each struct is determined and fixed at compile time. Struct definitions may contains fields of any type, including pointers, arrays, and other structs (as long as they don't contain themselves):

```
    struct TwoPoints {
        struct Point p1;
        struct Point p2;
    };

    struct Buffer {
        unsigned int len;
        char buf[1024];
    };

    struct Foo {
        struct TwoPoints pp;
        struct Foo f;       // compiler error: a struct cannot contain itself
    };
```

Pointers to structs

Since structs may get very large in size (e.g., struct Buffer), it can be more efficient to refer to them via pointers rather than copy them around by value:

```
    void inefficient_arg(struct Buffer buf) { ... }

    void efficient_arg(struct Buffer *buf) { ... }
```

Thus, a common pattern when using a pointer to a struct is to first dereference it, and then access the field:

```
    (*b).len
```

C provides syntactic sugar to make this look nicer:

```
    b->len
```

This notation should be read as, "dereference b, and access the len field." Keep in mind that this notation still performs a dereference; if b is a NULL pointer, this expression will segfault.

Self-referential structs

Structs cannot contain themselves, but they can contain pointers to themselves. We can exploit this ability to implement data structures such as singly linked lists:

```
    struct IntListNode {
        int data;
        struct IntListNode *next;
    };
```

Or doubly linked lists:

```
    struct IntDListNode {
        int data;
        struct IntDListNode *next;
        struct IntDListNode *prev;
    };
```

Or trees:

```
    struct IntTreeNode {
        int data;
        struct IntTreeNode *left;
        struct IntTreeNode *right;
    };
```

We can also change up the type of the data payload:

```
    struct FloatListNode {
        float data;
        struct FloatListNode *next;
    };

    struct PointListNode {
        struct Point data;
        struct PointListNode *next;
    };
```

How do we write a generic (singly) linked list that works with any type?
In C, we can use void *:

```
    struct Node {
        void *data;         // points to the data held by this node
        struct Node *next;
    };
```

Conventionally, the "next" element of the last element of a linked list is just a NULL pointer. For instance, this is what an empty list looks like:

```
    struct Node *head = NULL;
```

Unions (optional)

Unions are similar to structs, except all fields occupy the same memory location. For example:

```
    union LongDouble {
        unsigned long   as_long;
        double          as_double;
    };
```

The layout of union LongDouble looks like this:

```
    +---+---+---+---+---+---+---+---+
    | .as_long / .as_double         |
    +---+---+---+---+---+---+---+---+
    |------- union LongDouble ------|
```

The .as_long and .as_double fields both share the same 8 bytes.

Unions aren't used very often, but they are useful for reinterpreting the bit pattern of one type as another, without error-prone pointer casting:

```
    union LongDouble v;

    v.as_double = 3.14;

    printf("%lu\n", v.as_long);
```

Unions are also useful for representing data that may be one of type or another, but never both at the same time. For instance, the following struct carries an extra "tag" flag to indicate whether the data field carries long or a double:

```
    struct LongOrDouble {
        union LongDouble data;
        int tag;        // data.as_long when non-zero; data.as_double otherwise
    };

    struct LongOrDouble l;
    l.tag = 1;
    l.data.as_long = 1234;

    struct LongOrDouble d;
    d.tag = 0;
    d.data.as_double = 3.14;
```

This coding pattern is known as a "tagged union."

Unions can have fields of different sizes; the sizeof a union is the sizeof its largest field.

Function pointers

Machine code---the instructions that your computer hardware executes---is also stored in memory. It is located at the very bottom of the memory address space:

```
     +---------------------+
     |                     |
     |        stack        |     local variables
     |                     |
     +---------------------+
               |
               v

               ^
               |
     +---------------------+
     |                     |
     |        heap         |     malloc()ed variables
     |                     |
     +---------------------+
     +---------------------+
     |                     |
     |        data         |     global variables
     |                     |
     +---------------------+
     +---------------------+
     |                     |
     |        code         |     executable machine code
     |                     |
     +---------------------+
```

For security reasons, the OS typically prevents us from writing to code section, and there's usually nothing interesting for us to read from there either. However, function pointers are useful because they allow us to pass functions to other functions, and help us write generic code in C.

The usual motivating example is qsort(); quicksort is tricky enough that we want to implement it generically for all types of arrays, but we can't implement it without knowing how to compare elements of the array. So, we ask the caller to provide a pointer to the function used to compare two elements:

```
    void qsort(void *base,      // the array we want to sort
               size_t nmemb,    // how many elements in that array
               size_t size,     // the size of each element in that array
               int (*compar)(const void *, const void *));  // how to compare
```

The declaration for compar:

```
    int (*compar)(const void *, const void *)
```

says that compar is a pointer to a function that receives two arguments (whose types are both const void *), and returns an int.

Here are some example comparison functions:

```
    int compare_int(const void *v1, const void *v2) {
        int x = *(int *)v1;
        int y = *(int *)v2;
        if (x < y)
            return -1;
        else if (x > y)
            return 1;
        else
            return 0;
    }

    int compare_double(const void *v1, const void *v2) {
        double x = *(double *)v1;
        double y = *(double *)v2;
        if (x < y)
            return -1;
        else if (x > y)
            return 1;
        else
            return 0;
    }
```

Now we can give them to qsort() to let it sort different types of arrays:

```
    qsort(array_of_100_ints, 100, sizeof(int), &compare_int);
    qsort(array_of_200_doubles, 200, sizeof(double), &compare_double);
```

Note that compare_int() and compare_double() must have the exact type that qsort() asks for; this is why both comparison functions take void * parameters, and cast them to int * and double * internally.

Function pointer syntax can be tricky to read and write! The following two declarations are very different:

```
    int (*f1)(float f);
    int  *f2 (float f);
```

f1 is a pointer to a function that takes a float argument and returns an int;
f2 is a function that takes a float argument and returns a pointer to an int.
The parentheses surrounding f1 make a big difference here.

Typedef (optional)

Having to write the struct or union keyword every time we declare a Point or LongDouble can get cumbersome. You can use a typedef to get around this requirement:

```
    typedef struct {
        int x;
        int y;
    } Point;

    typedef union {
        unsigned long   as_long;
        double          as_double;
    } LongDouble;
```

Now we can declare and define variables using the struct or union keywords:

```
    Point p;
    LongDouble v;
```

However, typedefs do not work with self-referential structs. In these cases, we can specify a struct name and a typedef name:

```
    typedef struct Node {
        void *data;
        struct Node *next;  // we can't use ListNode inside of its typedef
    } ListNode;

    ListNode n;
```

Typedefs are also a handy way to make function pointer syntax more tolerable (though it isn't very obvious what type name we're defining):

```
    typedef int (*compar_fn_t)(const void *, const void *);
               // ^^^^^^^^^^^ the name of our new type
```

Now we can write qsort()'s signature more succinctly:

```
    void qsort(void *base, size_t nmemb, size_t size, compar_fn_t compar);
```

Typedefs are just type aliases and are purely stylistic. There is no agreed upon standard for when and how you should use a typedef. We don't require you to use them in this class, but you may be required to do so in other contexts; when in doubt, consult the appropriate style guide.