Lecture 10 - Structs (last updated: 2023-10-09)

Structs

Structs allow us to group multiple data components into a single object:

```
struct Point {
    int x;      // struct Point has two components (aka members or fields)
    int y;      // named x and y
};
```

The above struct definition creates a new type "struct Point". We can declare a variable of type "struct Point", and access its members like this:

```
struct Point pt;  // "struct Point" is the type, not just "Point"

pt.x = 2;  // "." is the member selection operator
pt.y = 3;
```

We can initialize a struct in two ways:

```
struct Point pt = { 2, 3 };  // x & y are initialized to 2 & 3, respectively

struct Point pt = { .x = 2, .y = 3 };  // or we can make it explicit
```

This is what pt would look like in memory in a system where int is 4-bytes:

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

In such a system, the following assertion will pass:

```
assert( &pt.y == ((int *)&pt) + 1 );
```

On the left side, &pt.y is evaluated as &(pt.y), which is the address of the y member of the struct variable pt. What we have on the right is a bit more complicated. First, &pt is the address of the whole struct. That's the same address as &pt.x (because x is the first thing in the struct), but the types are different: &pt.x is of type 'int *' and &pt is of type 'struct Point *'. We reinterpret &pt as an 'int *' by type-casting -- (int *)&pt -- and then add 1 to get the address of the second integer, which is finally the same thing as &pt.y.

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 Dream {
        char *kick;
        int current_level;
        struct Dream next;  // compiler error: a struct cannot contain itself
    };
```

Struct size

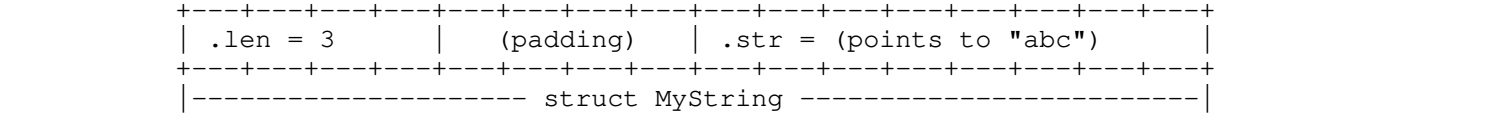
The size and layout of each struct is determined at compile time from how its members are defined, but they may not be exactly what you would expect.

For example, consider the following struct, which stores a string together with its length:

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

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

The layout in a typical 64-bit system looks like this:



There is a 4-byte padding between the members to ensure that the str member is properly aligned. The str member is an 8-byte pointer, and 64-bit systems require that 8-byte data items are placed 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. The sizeof operator returns a byte size of a struct including all paddings.

Pointers to structs

As everything else in C, structs are passed to and returned from functions by value. Because structs may get very large in size, 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) { ... }
```

C provides another member selection operator that we can use with pointers:

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

struct Point pt;

struct Point *p = &pt;

(*p).x = 2;  // '.' has higher precedence, so we need the parentheses

p->y = 3;    // equivalent to (*p).y = 3, and looks much nicer!
```

Here are other situations where we use '->' rather than '.' operator:

```
struct Point pt[1];  // still declaring a single struct on the stack,
pt->x = 2;           // but we now have a pointer (via the array name)

// And here is how we can allocate a struct on the heap:

struct Point *p = malloc(sizeof(struct Point));
p->x = 2;
```

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 linked lists:

```
struct Node {
    struct Node *next;
    int val;
};
```

Here is a simple program that creates a linked list of two nodes:

```
struct Node {
    struct Node *next;
    int val;
};

struct Node *create2Nodes(int x, int y)
{
    struct Node *n2 = malloc(sizeof(struct Node));
    if (!n2) {
        return NULL;
    }
    n2->val = y;
    n2->next = NULL;

    struct Node *n1 = malloc(sizeof(struct Node));
    if (!n1) {
        free(n2);    // don't forget to free the node we already allocated
        return NULL;
    }
    n1->val = x;
    n1->next = n2;  // n1 --> n2 --> NULL
    return n1;
}
```

```
int main()
{
    struct Node *head = create2Nodes(10, 20);
    if (head == NULL) {
        exit(1);
    }

    //         +----+   +----+----+   +----+----+
    //   head: |  --|-->| 10 |  --|-->| 20 |NULL|
    //         +----+   +----+----+   +----+----+

    printf("%d --> %d\n", head->val, head->next->val);

    free(head->next);  // must free the last node first
    free(head);
}
```

The pointer variable 'head' is of type 'struct Node *', and it points to the first node of the linked list. That's basically what we regard as a linked list. That is, our notion of a linked list in this program is simply a pointer to the first node. A NULL pointer represents an empty linked list.

Treating a pointer to a node as a linked list has a nice recursive property. The head->next pointer -- i.e., the 'next' pointer inside the first struct Node -- is also a linked list itself; it's a linked list of one node! This recursive structure enables some elegant algorithms. For example, here is a recursive function that frees all the nodes in a linked list:

```
void free_all_nodes(struct Node *head) {
    if (head != NULL) {
        free_all_nodes(head->next);
        free(head);
    }
}
```

In practice, however, a linked list is typically represented as a separate struct that contains 'head' and a few other useful things like so:

```
struct List {
    struct Node *head;
    struct Node *tail;  // a pointer to the last node for quick access
    unsigned int size;  // how many nodes are currently in the linked list
};
```

Our definition of struct Node above has an integer as the data inside the node. How can we write a generic linked list node that can hold any data type? In C, we can use void *:

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

In this case, however, the data is no longer embedded in the node. The node merely has a pointer to the data. The lifetime of the data needs to be managed independently by the program code that uses the linked list.

Unions (optional topic)

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.

Typedef (optional topic)

Using typedef keyword, we can create an alias for a type name:

```
typedef int MyInt;

MyInt x;  // same as int x;
```

After the typedef keyword, we declare the new type MyInt as if we were declaring a variable named MyInt. Here is another example:

```
typedef int *MyIntPtr;

MyIntPtr p;  // same as int *p;
```

We can also use typedef to give a struct type a new name:

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

In the typedef declaration above, we are defining a new type name 'Point' for an unnamed struct, 'struct { int x; int y; }'. We can then use the Point type without having to say "struct":

```
Point p;
```

Typedefs are also a handy way to make function pointer syntax more tolerable:

```
typedef int (*compar_fn_t)(const void *, const void *);
```

'compar_fn_t' is a new type name for 'int (*)(const void *, const void *)'. (Recall that we declare a new type as if we were declaring a variable.) Now we can write qsort()'s signature using the new type name:

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

Here is the prototype for signal(), a library function that takes a function pointer and returns a function pointer:

```
typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

Without typedef, the prototype looks like this:

```
void (*signal(int signum, void (*handler)(int)))(int);
```

Typedefs are just type aliases and are purely stylistic. There is no agreed upon standard for when and how you should use a typedef. For example, the Linux kernel coding style discourages the use of typedef for structs and pointers.