Lecture 7

Dr. Jack Lange

Structs

- C structs are like classes in Java
  - Declares a new type
  - `struct foo;` means there is a type 'struct foo'
- Java classes encapsulates code + data
- C structs include only data
  - No fancy OOP techniques (inheritance, subclasses, etc)
  - But in C : code == data (more on this later)
- Typedef
  - `typedef <original type> <new type>;`
  - `typedef unsigned int uint32_t;`
  - `typedef struct foo foo_t;`

Structs (2)

- structs are just another type
  - Treated exactly the same as primitive types
  - **No special memory state**
- Structs are purely a high level semantic construct
  - Just a collection of other types of data
  - No hardware level representation of *structure*
- How does it look in memory?
  - No metadata indicating it's a struct

Struct example

```
struct foo {
    int x;
    int y;
    char * name
};
```

Compiler sees:
foo.x, foo.y, foo.name

Hardware sees:
12 bytes of memory (int, int, char *)

- Structs are used like any other data type
  - Local declarations are stored on the stack
  - Dynamically allocated instances stored on the heap

Assigning values to structs

- structs are like any over data type
  - Automatically or dynamically allocated

Stack Allocation:

- Fields accessed with '.'

```
int main() {
    struct foo my_foo;

    memset(&my_foo, 0, sizeof(struct foo));

    my_foo.x    = 4;
    my_foo.y    = 10;
    my_foo.name = "George";
}
```

Assigning values to structs

- structs are like any over data type
  - Automatically or dynamically allocated

Heap Allocation:

- Fields accessed with '->'

```
int main() {
    struct foo * my_foo = calloc(1, sizeof(struct foo));

    my_foo->x      = 4;
    my_foo->name   = "George";
    (*my_foo).y    = 10;

    free(foo);
}
```

Structs in memory

- What do structs look like in memory?
  - Raw values of member variables

```
struct foo {
    int x;
    int y;
    char * name
};
```

```
int main() {
    struct foo   my_foo_1;
    struct foo   my_foo_2;

    my_foo_1 = my_foo_2;
}
```

- Copying Structs
  - C provides shallow copies of struct values
    - `memcpy(&my_foo_1, &my_foo_2, sizeof(struct foo));`
  - Copies raw bytes of struct data
    - We know the how many bytes via `sizeof();`

More complex data structures

- **KEY POINT:**
  - structs can contain pointers to the same struct type

```
struct list_node {
    int x;
    struct list_node * next;
}
```

- This is not self referential
  - Rather it points to another variable structure elsewhere in memory
- This allows us to construct collections
  - lists, trees, graphs, etc

Linked Lists

- Linked List properties
  - Linear Search
  - Constant time insertion/deletion
- Consist of at least 2 components
  - List node – contains data
  - List head (pointer to first node)
    - Anchors the list, allows you to find it
    - Allows arbitrary size, no need to track every element

Managing Lists

- Have to manage memory
  - Each node is an independent allocation
    - Almost 100% of the time on the heap
- Freeing Lists
  - Cannot just free the head
  - Must walk the entire list, and free each node

List behaviors

- Easiest construct LIFO (stack)
  - Push and pop from the head of the list
  - Constant time, O(1)
- What needs to change to do FIFO in 0(1)?
- How do you handle random insertion/deletion?