**CIT 593: Intro to Computer Systems**  
**Lecture #21 (11/27/12)**

**Structs**

Unlike Java, C++, and to some extent Python, C is not traditionally considered an object-oriented language. That is, there is no concept of an “object” that has a set of attributes and its own methods, and certainly nothing like inheritance, polymorphism, etc.

However, in C we can group related variables together into something called a **struct**.

For instance, here is how we would define a struct called “flight_info” that contains information about a single flight.

```c
struct flight_info {
    char *origin;
    char *destination;
    char *airline;
    double price;
};
```

Now if we want to create an instance of this struct, we can simply declare one (it can be global, local, whatever) and then start using it:

```c
struct flight_info flight1;
flight1.origin = “PHL”;
flight1.destination = “ORD”;
flight1.airline = “United”;
flight1.price = 500.0;
```

The first line creates a variable called “flight1”; its type is “struct flight_info”.

What does this look like in memory? Recall that the strings “PHL”, “ORD”, and “United” are immutable and are stored in the program text area of memory. But if `flight1` is a local variable, then the three char pointers and the double (for “price”) are all stored on the stack.

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1234</td>
<td>address of first char in “PHL”</td>
<td><code>flight1.origin</code> stored here</td>
</tr>
<tr>
<td>x1235</td>
<td>address of first char in “ORD”</td>
<td><code>flight1.destination</code> stored here</td>
</tr>
</tbody>
</table>
Note that there is no spot on the stack for the variable `flight1` itself. In a way, `flight1` is an array: its value is the address of the first element (specifically, the char pointer `origin`) and that value only exists as an offset in the symbol table. To get to individual elements of the struct, the compiler uses the offset based on the struct's definition. However, you cannot access something like `flight1[2]`; you must use the element's name.

It can get a bit tiresome having to write “struct flight_info” over and over again. C allows you to create your own types using the `typedef` directive. This allows you to give a new name to an existing type. For instance, to create a type called “flight”, which is the same thing as “struct flight_info”, you could do this:

```c
typedef struct flight_info flight;
```

This means “create a new type called `flight` that is the same as what's called `struct flight_info`”.

Now you don't have to keep using “struct flight_info” everywhere:

```c
flight flight2;
flight2.destination = "LHR";
flight2.airline = flight1.airline;
```

**Pointers to Structs**

One other key difference between a struct and an array is that you can have pointers to structs:

```c
flight *p = &flight2;
```

Note that the variable `p` would, in fact, hold the address of the first element of `flight2`.

To use the pointer, you would first have to dereference it:

```c
(*p).price = 200;
```
This can get a bit wieldy, so there is a shortcut:

\[
p->\text{price} = 200;
\]

This is the same as the instruction above, but looks nicer and is much more common. Keep in mind that in order to use this notation, \( p \) would have to be a pointer to a struct, not the name of the struct itself.

**Linked Lists**

Over the course of the next few lectures, we'll start talking about various data structures and their implementations in C.

The field of “data structures” refers to how data is stored, accessed, added, deleted, etc.
  - “Abstract data types” define how the structure is used
  - “Concrete data types” define how the structure is implemented

We've already seen one abstract data type: a stack. You could also argue that an array is a (concrete) data type.

We'll start with the simplest of data structures, which is the linked list.

Linked lists are a simple datatype in which each piece of data is stored in a “node”, and each node contains the data and a pointer to the next node in the chain.

Here is a linked list in which the data in each node is an int. The linked list has a “head”, which is a pointer to the first element; in the last node, the pointer to the next node is null.

```
head
```

```
5 → 7 → 2 → 4 → null
```

Linked lists can be doubly-linked, meaning that each node points to both “next” and “previous”, but we'll only focus on a singly-linked list for now.
Using the example above, here is the definition of a node:

```c
typedef struct Node node;
struct Node {
    int data;
    node *next;
};
```

To refer to the linked list, all we need is a pointer to the first element (the “head” of the list):

```c
// pointer to head of list
node *head;
```

Note that declaring a pointer to a node does not allocate any space for a node! It just allocates space for a variable that holds a node's address.

Now we'll look at some of the functions for operating on a linked list. First, let's see how we would add a new element to the list. We'll say that we're going to add it to the front (head) of the list; this makes things a little bit easier.

Adding a new element to the list consists of three steps:
1. Creating a new node for that piece of data
2. Updating that node's “next” pointer to refer to what is currently the node at the head of the list
3. Updating the head pointer to refer to the newly created node

Here's a possible implementation in which the parameter d is the value to store in the list, and head is a pointer to the first element (head will be null if the list is empty):

```c
void add_to_front(int d) {
    node new_node;
    new_node.data = d;
    new_node.next = head;
    head = &new_node;
}
```

This seems to make sense, right? On line 3, we create a node called `new_node`, and then set its `data` field to d (step 1, above) on line 4. On line 5, we set the value of its `next` pointer to that of `head`, so that it holds the address that `head` holds (step 2, above). Finally, on line 6, we update `head` to point to this new node.
See the problem?

If not, that's okay, but when you run this and try to add a few new values to the list, you'll see something interesting: all the old values will be overwritten by new ones!

Why is this happening? Because new_node is declared on the stack! On line 6, you put the address of a stack variable into head. When you overwrite that data on a subsequent function call, head still points to that same address, and now the old data is lost.

In order to address this situation, we need to dynamically allocate some new memory for the node. We can't use global memory because we don't know in advance (i.e., when we write the program) how many nodes we'll have in our linked list. So we need to use the heap.

The heap is an area of memory in your program that is managed by the operating system. Your program requests space on the heap from the operating system by indicating how much space it will need, and the OS returns a pointer to the space it's freed up for you.

Here's the revised implementation of adding a new element to the list, using the heap instead of the stack:

```c
void add_to_front(int d) {
    node *new_node = (node *)malloc(sizeof(node));
    new_node->data = d;
    new_node->next = head;
    head = new_node;
}
```

On line 3, we use the malloc function to get a pointer to some space on the heap. The argument is the amount of space (number of bytes) we need; in this case, we need enough to hold one node. The return value from malloc is a “void pointer”; it's a pointer, but we don't know to what type. So we need to cast it to a node pointer before we can start using it.

Now new_node holds the address of a piece of memory in the heap, which does not get overwritten by subsequent function calls, so it's safe to put the value of new_node (i.e., the address of the first element in the struct) into head.

One thing we typically want to do with a data structure is determine whether it contains a specific value. For instance, in the example above, we might want to see whether it contains a certain number.
With a linked list, you typically start with the first element (referred to by the head pointer) and then look at all subsequent elements (following the next pointers) until you hit the end.

This function returns 1 if k is found in the linked list, and 0 if not:

```c
int find(int k)
{
    node *n = head;
    while (n != NULL) {
        if (n->data == k) return 1;
        else n = n->next;
    }
    return 0;
}
```