**Structs**

- The **struct** is C's version of a class-like data structure
  - **struct** type has a name and a list of members
  - Like the instance variables of a Java class
  - Can allocate variables using the **struct** type, just as we did with primitive types
  - A value of a particular **struct** type takes up enough space to hold all its members
  - More options than Java's new `Class` operation

**Example**

```c
struct S {  // C++ style structs
  int i;
  float f;
  char* s;
};

S s;  // allocates space for an int, float, & ptr
S* ps;  // allocates space for a ptr
```

**C vs. C++ struct types**

- In C++, **struct S { ... }** introduces a new type named S
- In C, the type has to be referred to as "struct S", not "S"
- Ex:
  ```c
  struct S { ...);
  struct S s;
  struct S* ps;
  ```

**Accessing members**

- The main thing to do with a struct value is read and update its members
- Use Java-like dot notation to access members, on either side of assignment
- Ex.
  ```c
  S s;
  s.i = 5;
  s.f = s.i + 3.1415927;
  s.s = NULL;
  ```

**Pointers to structs**

- Can dereference a pointer to a struct and then access its members
  ```c
  S* ps = &s;
  (*ps).i = 5;
  (*ps).f = (*ps).i + 3.1415927;
  ```
- Syntactic sugar: `ps->m = (*ps).m`
  ```c
  S* ps = &s;
  ps->i = 5;
  ps->f = ps->i + 3.1415927;
  ```

**An example**

- Let's define a linked list of integers
- What does it look like, abstractly?
- How does that look physically, in C?
- What operations on linked lists, abstractly?
  ```c
  e.g. addFirst, addLast, findItem
  ```
- How do they look physically, in C?
Data structure declarations

```c
struct Link {
    int data;      // [why not int*?]  
    Link* next;    // [why not Link?] 
};

Link* emptyList = NULL;
```

An operation

```c
Link* addFirst(Link* list, int data) {
    Link* nNewLink = new Link;  // C: .. = (Link*) malloc(sizeof(Link))
    newLink->data = data;  
    newLink->next = list;  
    return newLink;
}
```

Why not this?

```c
Link* addFirst(Link* list, int data) {
    Link nNewLink;    // faster: no heap alloc
    nNewLink.data = data;  
    nNewLink.next = list;  
    return &nNewLink;
}
```

Another operation

```c
Link* addLast(Link* list, int data) {
    Link* lastLink = findLastLink(list);  
    if (lastLink == NULL) {   // empty list
        return addBefore(list, data);  
    } else {   // non-empty list
        addAfterLastLink(lastLink, data);  
        return list;
    }
}
```

A helper

```c
void addAfterLastLink(Link* lastLink, int data) {
    Link* nNewLink = new Link;  
    nNewLink->data = data;  
    nNewLink->next = NULL;  
    assert(lastLink->next == NULL);  
    lastLink->next = nNewLink;
}
```

Another helper

```c
Link* findLastLink(Link* list) {
    if (list == NULL) {   // empty list
        return NULL;
    } else if (list->next == NULL) {  
        // last link
        return list;
    } else {  
        return findLastLink(list->next);  
    }
}
```
A non-recursive version

```c
Link* findLastLink(Link* list) {
    if (list == NULL) { // empty list
        return NULL;
    } else {
        while (list->next != NULL) {
            list = list->next;
        }
        return list;
    }
}
```

Another operation

```c
Link* findItem(Link* list, int data) {
    if (list == NULL) {
        return NULL; // NULL == not found
    } else if (list->data == data) {
        return list; // found it
    } else { // keep searching
        return findItem(list->next, data);
    }
}
```

A non-recursive version

```c
Link* findItem(Link* list, int data) {
    for (;;) {
        if (list == NULL) {
            return NULL; // not found
        } else if (list->data == data) {
            return list; // found it
        } else {
            list = list->next; // keep searching
        }
    }
}
```

An improvement: list header

```c
struct List {
    Link* first;
    Link* last;
};
```

Revised operation

```c
List* addLast(List* list, int data) {
    if (list == NULL) { // empty list
        return addFirst(list, data);
    } else { // non-empty list
        addAfterLastLink(list->last, data);
        list->last = list->last->next; // [why?]
        return list;
    }
}
```

Another revised operation

```c
List* addFirst(List* list, int data) {
    Link* newLink = new Link;
    newLink->data = data;
    if (list == NULL) { // create the list
        list = new List;
        list->first = NULL;
        list->last = newLink;
    }
    newLink->next = list->first;
    list->first = newLink;
    return list;
}
```
Doubly-linked lists

- Extend with a previous link
  ```
  struct DLink {
    int data;
    DLink* prev;
    DLink* next;
  };
  ```
- An exercise for the reader...
- Lots of fun pointer surgery & splicing!

Multiple source files

- Bigger programs need to be broken up into multiple files
  - How does one file get access to things defined in other files?
- In Java:
  - User just writes .java source files
  - Compiler automatically looks in other .class files to see what they publicly export
- In C:
  - User needs to write both .c source files and .h header files

Header files

- Header files (redundantly) declare public functions and types that will be accessed by other .c files
  - Anything not declared is implicitly private to the .c file
- Each .c file #include's the .h files of the things it accesses
  - That way it sees the declarations of those things
- Anything not declared in .h files can't be accessed by other .c files (unless they cheat)

Example

- In link.h:
  ```
  struct Link; // hide its body; allow Link* only
  Link* addFirst(Link* list, int data);
  // no (...) in a prototype
  ```
- In link.c:  
  ```
  include "link.h" // to verify consistency
  ```
  ```
  // full defs of struct Link, addFirst, etc.
  ```
- In client.c:
  ```
  include "link.h" // access public decls
  ```
  ```
  // uses of Link*, calls of addFirst, etc.
  ```

Makefile dependencies

- .c files depend on the .h files they include
- Add to Makefile
  ```
  $ .c: $ .h
  gcc $(CFLAGS) -c $ .c
  ```
  ```
  # additional dependencies:
  link.o: link.h ...
  client.o: link.h ...
  ```
- Have to keep these additional dependencies up-to-date as source files are edited...

makedepend

- makedepend: a tool to construct these extra dependencies automatically from the source files
  ```
  makedepend .c...
  ```
  ```
  # extra dependencies at end of existing Makefile
  ```
  ```
  Add a depend target to Makefile:
  ```
  depend:
  ```
  ```
  makedepend [SRCs]
  ```
  ```
  Also built into gcc as gcc -MM file.c...