Lecture 6

Structures

In C

- functions organize sequence of instructions into logical units
- structures groups variables in logical units

A C struct is a named collection of one or more variables, possibly of different types

```
struct slot {
    int x;
    char c;
 }
```

`slot` is the name (tag) of the structure; `x` and `c` are members

Comparison with Java


```java
class Slot {
    int x;
    char c;
}
```

Java

```c
struct slot {
    int x;
    char c;
}
```

C

Difference between the two:

- No inheritance
- No methods
- A Java variable of type Slot is a pointer
- A C variable of type slot denotes the structure with no indirection

Structures


```
struct slot { int x; char c; }
```

Tag names can be used after a struct has been declared

```
struct slot s1, s2;
```

The size of a struct is obtained by calling **`sizeof`**

```
sizeof(s2)
```

Accessing a member is done with the dot operator

```
s1.x
```

Pointers to structures can be defined

```
struct slot* p = &s1;
```

Two equivalent syntactic ways to access members by reference

```
p->x
(*p).x
```

Size

If a structure contains dynamically allocated members, the size of whole struct may not equal sum of its (referenced) parts

```
struct word { char* w; int l; }
```

- `sizeof(struct word)` is 8 bytes.
- Internal padding means that sizeof may be larger than expected

```
struct ex { int a;  char b; int c;  };
```

- Is `sizeof(struct ex) == 2*sizeof(int)+sizeof(char) ?`

Structs in structs...

A structure can contain a member of another structure

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

struct slot {
  struct pos p;
  char c;
} s;
```

Access **x** via: **s.p.x**

The size of slot is exactly the same as if the fields of pos were written inline in slot

In terms of performance there is no cost to nested structures

Recursive structures

What is the meaning of

```
struct rec { int i;  struct rec r; }
```

A structure cannot refer itself directly.

The only way to create a recursive structure is to use pointers

```
struct node {
   char *word;
   int count;
   struct node *left,*right;
}
```

Anonymous Structures


```
struct y;
struct x { struct y *p; /* ... */ };
struct y { struct x *q; /* ... */ };
```

```
struct s* p = NULL; // tag naming an unknown struct declares it
struct s { int a; }; // definition for the struct pointed to by p
void g(void)
{
    struct s; // forward declaration of a new, local struct s
              // this hides global struct s until the end of this block
    struct s *p;  // pointer to local struct s
                   // without the forward declaration above,
                   // this would point at the file-scope s
    struct s { char* p; }; // definitions of the local struct s
}
```

Structures and functions

Structures can be initialized, copied as any other value

They can not be compared directly

- instead one must write code to compare members one by one
- Or compare the addresses of the structures *(usually not the right answer)*

Functions can return structure instances

- What is the cost in terms of memory allocation, copy, and performance?
- What's the difference between arrays and structures in this sense?

```
struct pt { int x, y; };

struct pt mkpt(int x, int y) {
    struct pt t; t.x = x; t.y = y; return t;
}

struct pt p1 = mkpt(0, 0);
```

Typedef

A declaration form that allows us to create new data type names:

```
typedef int len;
len l1, l2;
typedef struct { len x, y;} pos;
pos p1, p2;
```

- Notice the difference. No struct needed when using the type.

```
unsigned int uint5[5];
typedef unsigned int uint5[5]
uint5 arr = {1, 2, 3, 4, 5};
```

Example


```
struct coord {
  float x;
  float y;
  float z;
};
typedef struct coord coord_type;

coord_type add_coord(coord_type a, coord_type b) {
   coord_type sum = { 0.0, 0.0, 0.0 };
   sum.x = a.x + b.x;
   sum.x = a.x + b.x;
   sum.y = a.y + b.y; sum.z = a.z + b.z;
   return sum; }

#include <stdio.h>
void print_coord(coord_type coord) {
  printf(“(%f, %f, %f)”, coord.x,
  printf(“(%f, %f, %f)”, coord.x,  coord.y, coord.z);
  return; }
```

Declaration vs Definition


```
struct hey {
 int foo;
 int bar;
};
```

A structure declaration

```
struct point {
 int x;
 int y;
} pt;
```

A structure definition

Generally, declarations occur outside functions, while definitions typically occur inside. Why?

Initialization


```
struct person {
  char name[40];
  char title[15];
  int ssNum[9];
};
```

```
struct person ae = {"Albert", "Prof", {1,2,3,4,5,6,7,8,9}};
struct person z = {0};
```

What about:

```
char person_name[20] = "Mike";
char person_title[15] = "Guy";
int id[9] = 123;
struct person mike = {person_name, person_title, id}
```

```
struct {int sec, min, hour, day, mon, year;} z
   = {.day=31,12,2014, .sec=30,15,17};
   // initializes z to {30,15,17,31,12,2014}
```

```
struct example {
    struct addr_t { int port; } addr;
    struct {
        int a8[4];
        int a16[2];
    } in_u;
};
struct example ex2 = { // current object is ex2
                       .in_u.a8[0]=127, 0, 0, 1, .addr=80};
struct example ex3 = {80, .in_u={ // changes current object
                                  127,
                                  .a8[2]=1 // this designator refers to the member of in_u
                      } };
```

Initialization


```
#include <stdio.h>
typedef struct { int k; int l; int a[2]; } T;
typedef struct { int i;  T t; } S;
T x = {.l = 43, .k = 42, .a[1] = 19, .a[0] = 18 };
 // x initialized to {42, 43, {18, 19} }
int main(void)
{
    S l = { 1,          // initializes l.i to 1
           .t = x,      // initializes l.t to {42, 43, {18, 19} }
           .t.l = 41,   // changes l.t to {42, 41, {18, 19} }
           .t.a[1] = 17 // changes l.t to {42, 41, {18, 17} }
          };
    printf("l.t.k is %d\n", l.t.k); // .t = x sets l.t.k to 42 explicitly
                                    // .t.l = 41 would zero out l.t.k implicitly
}
```

Structure elements implicitly initialized to zero when defined outside a function; absent any initialization, contain undefined elements inside a function. "Partially" initialized structures have their implicitly initialized elements zeroed.

Aggregate Initialization


```
struct foo { char s[4]; int n; };
struct foo x[ ] = { { { "abc" }, 1 }, // inits x[0] to { {'a','b','c','\0'}, 1 }
                      [0].s[0] = 'q'  // changes x[0] to { {'q','b','c','\0'}, 1 }
                   };
struct foo y[ ] = { { { "abc" }, 1 }, // inits y[0] to { {'a','b','c','\0'}, 1 }
                     [0] = { // current object is now the entire y[0] object
                             .s[0] = 'q'
                            } // replaces y[0] with { {'q','\0','\0','\0'}, 0 }
                    };
```

Readings

K&R - Chapter 6, pp. 127-143