LECTURE 19: STRUCTS WITH FUNCTIONS, ARRAYS, POINTERS (K&R §§ 6.1–6.4)

STRUCTS REVIEW

Same:

```
struct point {
  int x;  // member
  int y;  // member
};
struct point pt1, pt2;
```

```
struct point {
  int x;  // member
  int y;  // member
} pt1, pt2;
```

Declares a point structure, then declares two point variables.

Declares a point structure and two point variables in the same statement.

Note that a semicolon is required after the struct definition.

```
struct point maxpt = {320, 200};
```

point variable defined with a list of initializer values.

```
struct rect {
  struct point pt1;
  struct point pt2;
};
struct rect box;

box.pt1.x = 5;
box.pt1.y = 10;
box.pt2.x = 10;
box.pt2.y = 20;

int area =
  (box.pt2.x = box.pt1.x) * (box.pt2.y = box.pt1.y);
```

Note the use of the dot operator, or structure member operator, to access struct members.

WHAT CAN WE DO WITH A STRUCT?

reference members	box.pt2.x = box.pt1.x + width;
assign as a unit	pt2 = pt1;
create a pointer to	struct point *ppt1; ppt1 = &pt1;

NOTE:	We CANNOT compare structures directly. Must compare members.
WRONG!	if(pt1 == pt2) ...
ACCEPTABLE	if (pt1.x == pt2.x && pt1.y == pt2.y) ...

STRUCTS AND FUNCTIONS

A function can return a struct:

```
struct point makepoint(int x, int y) {
  struct point temp;
  temp.x = x;  // note there is no name conflict here
  temp.y = y;
  return temp;
}
```

Three ways to pass a struct to a function:	1) pass components separately
	2) pass entire structure
	3) pass a pointer to structure

/* point in rectangle:
if point p in rect r, return 1, else return 0 */

pass struct:

```
int ptinrect (struct point p, struct rect r) {
  return p.x >= r.pt1.x && p.x <= r.pt2.x &&
         p.y >= r.pt1.y &&  p.y <= r.pt2.y;
}
```

pass struct pointer:

```
int ptinrect (struct point *pp, struct rect *rp){
  return pp->x >= rp->pt1.x &&
         pp->x <= rp->pt2.x &&
         pp->y >= rp->pt1.y &&
         pp->y <= rp->pt2.y;
}
```

STRUCT POINTERS

If ppt1 is a pointer to pt1:
*pp is the structure
(*pp).x, (*pp).y are the members

The parentheses are necessary because . (dot) has higher precedence than *(dereference).

MEMBER OF STRUCTURE OPERATOR ->

Pointers to structures are used often, so we have an alternative notation to access a member via a pointer to a structure.

p -> member	(*pp).x // same as pp->x

With this notation we are dereferencing the pointer and accessing a member of the struct it points to.

NOTE ON PRECEDENCE

Review the precedence chart!

., -> are at the top of the precedence chart

If we have ++pp->x, the implied parenthesization is:
++(pp->x) (member x is pre-incremented)

Example:

```
struct string {
  int len;
  char *cp;
} *p;
struct string xp;
p = &xp;

struct string a = {8 , "yellow"};
struct string b = {9 , "black"};
struct string *ptr_a = &a;
p = &b;
```

expression	same as	value / effect
++p->len	++(p->len)	increments len
*p->cp	*(p->cp)	value is a char
*p->cp++	*((p->cp)++)	value is a char increments cp
ptr_a->len	(*a).len	8
ptr_a->cp	(*a).cp	'y'
ptr_b->len	(*b).len	9
ptr_b->cp	(*b).cp	'b'

ARRAYS OF STRUCTS

What if we want to write a program to count the number of times each C keyword is seen in some code?

Parallel Arrays (what we'd do until now):

```
char *keyword[NKEYS]  // NKEYS = number of keywords
int keycount[NKEYS]   // corresponding counts
```

Array of Structs
(put the word and count in one data structure)

```
struct key {
  char *word; // pointer to a string literal
  int count;  // corresponding count
} keytab[NKEYS];  // array of struct key elements
```

We can also initialize the array as we define it:

```
struct key {
  char *word; // pointer to a string literal
  int count;  // corresponding count
} keytab[] = { "auto", 0,          = { {"auto", 0},
               "break", 0,            {"break", 0},
               // ...                 // ...
               "volatile", 0,         {"volatile", 0},
               "while", 0             {"while", 0}
             };                     };
```

sizeof STRUCTS

sizeof is a compile-time unary operator that produces the number of bytes used to store its operand.

Note that sizeof produces a number of type size_t, which is an unsigned integer type. (The t means type -- it is a typedef created for ints describing size.)

sizeof can be called on a variable or a type:

```
sizeof variable_name
sizeof (type_name)
```

We can use sizeof to derive the number of keywords in our above example:

```
#define NKEYS (sizeof keytab /sizeof(struct key))
#define NKEYS (sizeof keytab / sizeof keytab[0])
```