CS262 Lecture 07
Chapter 6 Structure

Jyh-Ming Lien
Department of Computer Science

The struct Definition

- **struct** is a keyword
- Format:

```
struct name {
    type1 name1;
    type2 name2;
    …
};
```

name1 and name2 are *members* of name

- name represents this structure's tag and is optional
  - we can either provide name
  - or after the } we can list variables
  - We can also use typedef

Examples

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

struct point p1, p2;

p1 and p2 are both points, containing an x and a y value

```
struct {
    int x;
    int y;
} p1, p2;
```

p1 and p2 both have the defined structure, containing an x and a y, but do not have a tag

```
struct  point {
    int x;
    int y;
} p1, p2;
```

same as the other two versions, but united into one set of code, p1 and p2 have the tag point

An Array of Struct

```
struct key {
  char *word;
  int count;
} keytab[100];
```

equivalent

```
struct key {
  char *word;
  int count;
};
struct key keytab[100];
```

Initialize An Array of Struct

```
struct key {
  char *word;
  int count;
} keytab[] = { "auto", 0,
"break", 0, "case", 0, "char", 0,
"const", 0, "continue", 0,
"default", 0, /* ... */ "unsigned",
0, "void", 0, "volatile", 0,
"while", 0};
```

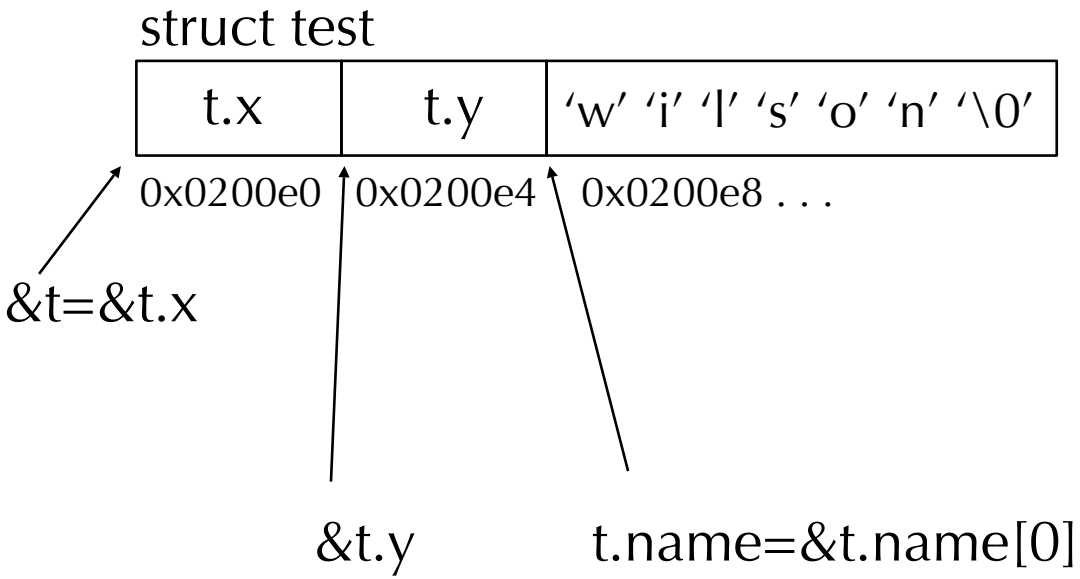
- see ex01.c and ex02.c

Using C Structs

- How do we access a C structure such as:

```
#define MAXLEN 20
struct test {
        int x,
        int y;
        char name[MAXLEN];
} t;
t.x = 2;
t.y = 5;
strncpy(t.name, "wilson", MAXLEN);
trystruct(&t); /* pass to asm via pointer*/
```

Using C Structs

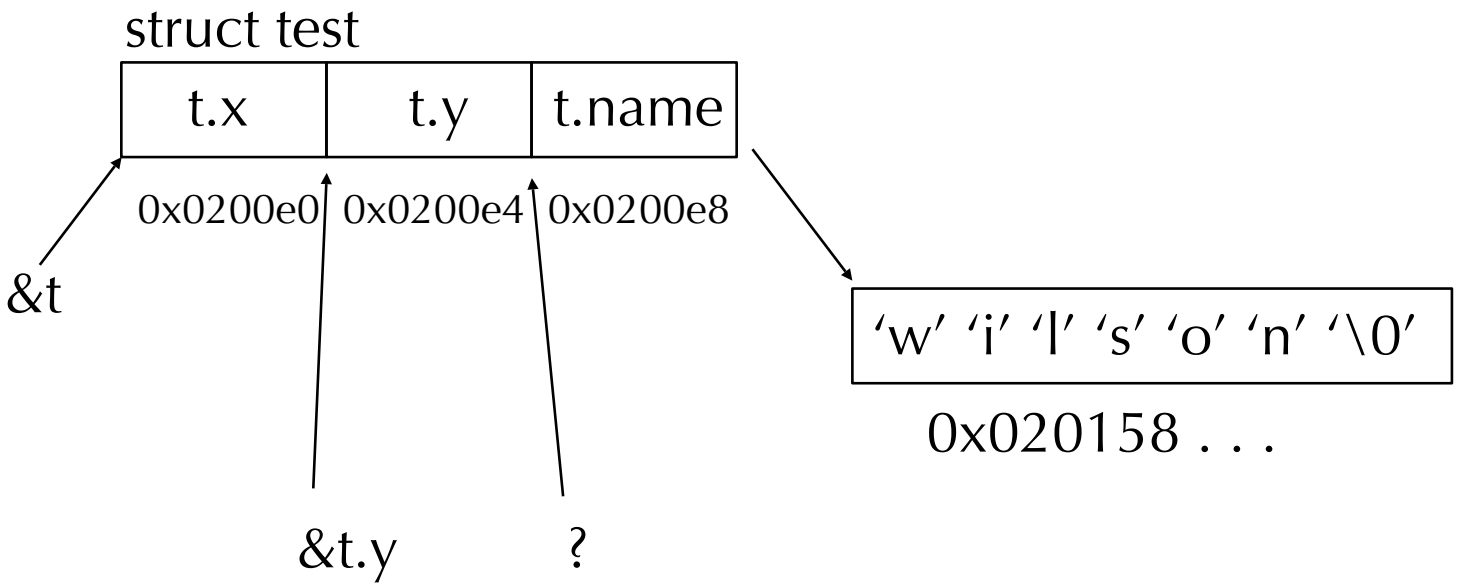


Using C Structs

- However, we would normally have a **pointer to string**:

```
#define MAXLEN 20
char array [MAXLEN];
struct test {
        int x,
        int y;
        char *name;
}t;
t.x = 2;
t.y = 5;
t.name = array;
strncpy(array, “wilson”, MAXLEN);
```

Using C Structs



Accessing structs

- A struct is much **like an array**
  - The structure stores multiple data
    - You can access the individual data
    - you can reference the entire structure
  - To access a particular member, you use the "." operator
    - as in p1.x and p1.y
  - Use "- >" to reference a field **if the struct is pointed to by a pointer**

- Legal operations on the struct
  - **assignment (make a copy)**
  - **taking its address with &**
  - **passing it** as a parameter

structs as Parameters

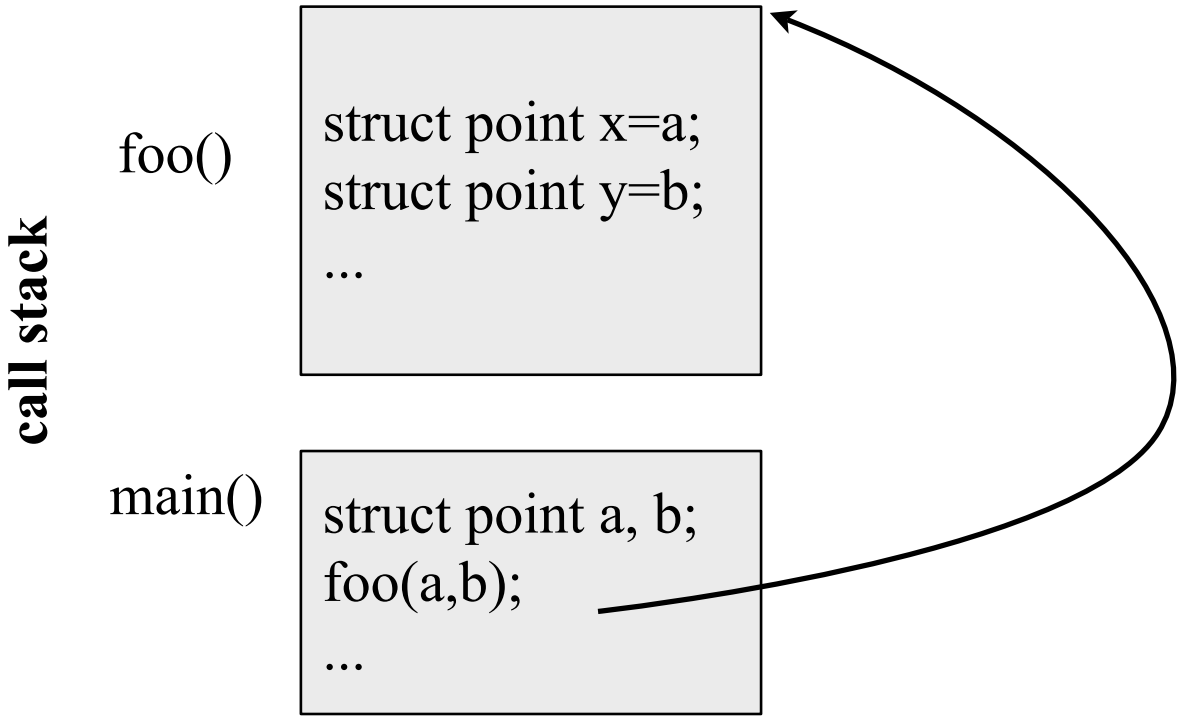
- pass structs as parameters and functions can return structs
  - Passing as a parameter:
    - void foo(**struct point** x, **struct point** y) {…}
      - notice that the parameter type is not just the tag, but preceded by the reserved word struct
  - Returning a struct:

```
struct point createPoint(int a, int b)
{
    struct point temp;
    temp.x = a;
    temp.y = b;
    return temp;
}
```

So, what happened in call stack when both functions are called and returned?

structs as Parameters

- C functions are call-by-value
- void foo(struct point x, struct point y)



Inputting a struct in a Function

- We will need to do multiple inputs for our struct
  - Rather than placing all of the inputs in main, let's write a separate function to input all the values into our struct

- Will this code work?

```
#include <stdio.h>

struct point  { int x; int y;  };

void getStruct(struct point);
void output(struct point);
void main( )
{
    struct point y = {0, 0};
    getStruct(y);
    output(y);
}

void getStruct(struct point p)   {
    scanf("%d", &p.x);
    scanf("%d", &p.y);
    printf("%d, %d", p.x, p.y);
}

void output(struct point p)   {
    printf("%d, %d", p.x, p.y);
}
```

One Solution For Input

- In our previous solution, we passed the struct into the function and manipulated it in the function, but it wasn't returned

- In our input function, we can instead create a temporary struct and return the struct rather than having a void function

```
void main( )
{
        struct point y = {0, 0};
        y = getStruct( );
        output(y);
}
```

```
struct point inputPoint( )
{
    struct point temp;
    scanf("%d", &temp.x);
    scanf("%d", &temp.y);
    return temp;
}
```

We could also pass the address of y and treat the struct like an array, we will see this next, but it requires a change in how we handle the members of the struct

Pointers to Structs

- The previous solution had two major flaws:
  - It requires **twice as much memory**
    - we needed 2 points, one in the input function, one in the function that called input
  - It is slow
    - required copying each member of temp back into the members of the original struct
  - So instead, we might choose to use a pointer to the struct
  - We see an example next, but first…

- If a is a pointer to a struct, then to access the struct's members, we use the **->** operator as in **a->x**

Pointer-based Example

see ex03.c and ex04.c

Nested structs

- In order to provide modularity, it is common to use already-defined structs as members of additional structs
- Recall our point struct, now we want to create a rectangle struct
  - the rectangle is defined by its upper left and lower right points

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

struct rectangle {
    struct point pt1;
    struct point pt2;
}
```

If we have
struct rectangle r;
Then we can reference
r.pt1.x, r.pt1.y,
r.pt2.x and r.pt2.y

Now consider the following
struct rectangle r, *rp;
rp = &r;

Then the following are all equivalent
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
But not rp->pt1->x (since pt1 is not a pointer to a point)

typedef

- typedef is used to define new types
  - The format is
    - typedef description name;
  - Where description is a current type or a structural description such as an array declaration or struct declaration
  - Examples:

```
typedef  int Length;          // Length is now equivalent to the type int

typedef *char[10] Strings;    // Strings is the name of a type for an array of 10 strings

typedef struct node {         // declares a node structure that contains
        int data;             // a data item and a pointer to a struct of type node
        struct node *next;
 };
```

We can simplify our later uses of node by doing the following

```
typedef struct node aNode; // this allows us to refer to aNode instead of struct node
```

Linked Structures

- Our last topic is in building linked structures
  - lists, trees
- These are **dynamic structures**, when you want to add a node, you allocate a new chunk of memory and attach it to the proper place in the structure via the pointers
  - Each node in the structure will have at least one datum and at least one pointer
  - In linked lists, the pointer is a next pointer to the next node in the list, in a tree, there are left and right children pointers
  - We will use malloc to allocated the node
  - We will need to traverse the structure to reach the proper place to insert a new node
    - A linked list example is given separately on my web site

Declarations for Nodes

```
struct node {
   int data;
   struct node *next;
};

node *front;
```

front is a pointer to the first node in a linked list. It may initially be NULL. Traversing our linked list might use code like this:

```
temp = front;
while(temp!=NULL)
{
    // process temp
    temp=temp->next;
}
```

```
struct treeNode {
   int data;
   struct treeNode *left;
   struct treeNode *right;
};
```

Our root node will be declared as treeNode *root;

It is common in trees to have the root node point to the tree via the right pointer only with the left pointer remaining NULL

Declarations for Nodes

Create a linked list with 4 elements

```
struct node {
   int data;
   struct node *next;
};

node *front=malloc(sizeof(node));
front->next=malloc(sizeof(node));
front->next->next=malloc(sizeof(node));
front->next->next->next=NULL;
```

How do you use a for loop to simplify this?

Declarations for Nodes

- Note the difference

```
struct node {
   int data;
   struct node next;
};
```

recursive definition is NOT allowed

```
struct node {
   int data;
   struct node *next;
};
```

allowed since it's NOT recursive definition