CS 241
Data Organization Structures

Brooke Chenoweth

University of New Mexico

Fall 2017
`void main(void)
{
    char data[] = "Hello World";
data[2] = 'X';
    char *linePt = &data[3];
    *linePt = 'Z';
    printf("[%s], [%s]\n", data, linePt);
}

What is the output?

[HeXZo World], [Zo World]"
K&R Section 6.1: Basics of Structure

A structure is a collection of named data items.

/* x and y are members of the structure point. */
struct Point { int x; int y; };
/* A structure does not reserve storage. */
/* It only defines the type of storage. */

struct Point pt; /* This reserves storage. */

Alternate syntax defining and instantiating a structure.

Like class names in Java, in CS-241, we will use structure names that start with a capital letter.
Accessing the Members of a Structure

Individual members are accessed with the ‘.’ operator.

```c
struct Point {
    int x;
    int y;
} pt;

/* Assignment to the members of a structure. */
pt.x = 5;
pt.y = 8;

/* Initializing a structure when it is instantiated. */
struct Point maxpt = {320, 200};
```
struct Point
{
    int x;
    int y;
};

struct Point pt;

/* Not CS241 standard */
struct Point {
    int x; int y;
} pt;

struct Point
{
    int x;
    int y;
} pt;

struct Point {int x; int y;} pt;
Section 6.2: Structures and Functions

```c
struct Point { int x; int y;};
struct Point makepoint(int x, int y)
{ /* reuse of the variable names x and y is good. */
    struct Point temp;
    temp.x = x;
    temp.y = y;
    return temp; /* Returned by value. */
}

void main(void)
{
    struct Point p1 = makepoint(5,7);
    printf("p1=(%d, %d)\n", p1.x, p1.y);
}
```
Passing a Structures as an Argument

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

void incrementPoint(struct Point p)
{
    p.x++;  
    p.y++;  
}

void main(void)
{
    struct Point p1 = {4, 7};
    incrementPoint(p1);
    printf("p1=(%d, %d)\n", p1.x, p1.y);
}
```

What is the output?  p1=(4, 7)
Section 6.4: Pointers to Structures

```c
struct Point { int x; int y; };

void incrementPoint(struct Point *p) {
    (*p).x ++; /* . has higher precedence than *, */ /* so *p.x++; is a syntax error. */ /* Dereferencing a pointer, then accessing */ /* a member is so common that it is given */ /* a special notation. */
    p->y ++;
}

void main(void) {
    struct Point p1 = {4, 7};
    incrementPoint(&p1);
    printf("p1=(%d, %d)\n", p1.x, p1.y);
}
```

Output: p1=(5, 8)
Warning: Function Returns Address of Local Variable

```c
#include <stdio.h>

struct Point { int x; int y;};

struct Point* badPointer(int x, int y)
{
    struct Point temp; /* local var */
    temp.x = x;
    temp.y = y;
    return &temp; /* Oh, no! Returning address! */
}

void main(void)
{
    struct Point* p1 = badPointer(5,7);
    printf("p1->(%d, %d)\n", (*p1).x, (*p1).y);
}
```
Section 6.3: Arrays of Structures

Parallel Arrays

```c
char *keyword[NUM_KEYWORDS];
int keycount[NUM_KEYWORDS];
```

Array of Structures

```c
struct KeyStructure
{
    char *word; // allocates space for a pointer.
    int count;
}
key[NUM_KEYWORDS];
```

```c
key[0].word = "if";   key[0].count = 0;
key[1].word = "for";  key[1].count = 0;
key[2].word = "char"; key[2].count = 0;
key[3].word = "int";  key[3].count = 0;
```
gcc Compile Error

```c
#include <stdio.h>
#define NUM_KEYWORDS 32;
int main()
{
    struct KeyStructure
    {
        char *word;
        int count;
    } key[NUM_KEYWORDS];
    /* Code continues below... */
```

foo.c: In function main:
foo.c:9: error: expected ] before ;

What did I do wrong here?
Structure Operations

- . and -> allow access to members.
- Also can be assigned, passed as parameters, returned by functions.
  - C is call-by-value, so structure parameters are copied.
  - Often use a pointer to the structure instead.
- Structures cannot be compared via ==.
  - Structures may contain padding to align the fields properly. This may cause different “garbage” bits in the structure that have the same field values.
  - You have to compare structures field-by-field.
**Nested Structures**

C allows structures to be nested.

```c
struct Rect
{
    struct Point bottomLeft;
    struct Point topRight;
};

struct Rect r;
r.bottomLeft.x = 0;
```
Using the type `struct StructName` to define a structure variable is a bit verbose. The C `typedef` operation allows you to provide a synonym for an existing type name.

```c
typedef int Length;
```

defines `Length` to be a synonym for `int`. Thus:

```c
Length foo;
```

defines a variable `foo` of type `int`. 
Typedef...

It is important to remember that typedef does not define a new type, it merely defines a synonym for an existing type. The types are the same, and C won’t complain if you interchange them:

```c
typedef int Length;
Length foo;
int bar = 10;
foo = bar;
```
Self-referential structures

A structure cannot contain an instance of itself, but a structure declaration can contain a reference to its own (incomplete) type.

```c
struct ListNode
{
    int data;
    struct ListNode* next;
};
```
Unions

A union is a data type that stores several variables, perhaps of different types, in the same memory. Because the same memory is used, only one variable at a time can hold a value.

```c
union
{
    char       msg[20];
    int        total;
    short      tax;
} info;
strcpy(info.msg, "hello");
info.tax = 10;
info.total = 1000;
/* Only info.total is valid at this point */
```
Unions...

- Unions have similar syntax to structures, but don’t forget that only one field at a time is valid.
- The union variable info contains three variables, msg, total, and tax. sizeof(info) is the maximum size of these variables, or 20 bytes.
- Unions can be nested, and can contain structures, arrays, etc.
An auxiliary variable is often used to remember what is stored in a union – often this variable and the union are stored in a structure:

```c
struct
{
    int type;
    union
    {
        char msg[20];
        int total;
        short tax;
    } info;
} foo;
```

foo.type = 0;
strcpy(foo.info.msg, "hello");
foo.type = 1;
foo.info.total = 10;