/unc/comp211

Systems Fundamentals

Structs !

Structures

- A structure in C is a group of related variables
  - Each variable in the struct is a **member** (also often called a property)
  - You can think of it as like a class with only public properties and no method or constructor (C++/Java-style classes evolved out of C-style structs)

- You declare a structure as such:

```
struct <Name> {
    <type> <member0>;
    ...
    <type> <memberN>;
}
```

- Example:

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

Struct Variable Declaration (1/2)

- The declaration of a struct variable works *almost* as expected:

```
struct <StructName> <variable_name>;
```

- Example:

```
struct Point aPoint;
```

- In a few slides you will learn how to make the struct keyword implicit.
- The same rules about locations of variables in memory apply to structs
  - This is a significant difference from memory-managed languages like Java! In those languages, your objects can *only* live in dynamic, heap memory. You can *only* pass around pointers.

Struct Variable Declaration & Initialization (2/2)

- Zero-initialize all members:

```
struct Point aPoint = { 0 };
```

  - The book does not mention this because it came in the C99 standard:

    "If there are fewer initializers in a brace-enclosed list than there are members of an aggregate, the remainder are initialized implicitly the same as objects with static duration."

- Initialize members, in order, to specific values:

```
struct Point aPoint = { 1.0, 2.0 };
```

  - The order of the values corresponds with the order of the member definitions in the struct (!)

- This ***only*** works when declaring and initializing at the same time.
  - You cannot initialize after declaration or reassign with this syntax.

- As a matter of practice, ***<u>always initialize</u>*** *one way or the other!*
  - A struct's members will be garbage values, otherwise.

Aside: Aliasing Types with typedef (1/3)

- C's **typedef** keyword defines another name for another type

- The syntax is:

```
typedef <existing type> <new-name>;
```

- For example:

```
typedef unsigned int whole_number;
```

- After defining a type, you can use it in place of the original:

```
whole_number x = 0;
whole_number y = 211;
```

Aside: Alias Struct Types with typedef (2/3)

- When declaring struct arrays and variables, most C programmers find it verbose to have to write the struct keyword at every declaration.

- The typedef keyword provides a way out!

- The syntax is the same as before:

```
typedef struct <Name> <new-name>;
```

- Examples:

```
typedef struct Point point_t;
typedef struct Point Point;
```

- After defining two aliases of struct Point, you could use either with the same effect:

```
point_t x = { 0 };
Point y = { 1.0, 2.0 };
```

- Naming conventions around struct typedefs vary project-to-project.
  - Two common conventions illustrated above: suffix with _t or CamelCase
  - In this course, we will opt for a convention of CamelCase struct names

Aside: Alias Struct Types with `typedef` (3/3)

- Consider again the syntax for a typedef:

```
typedef <type> <new-name>;
```

- And the pattern of first defining a struct type and then referencing it later:

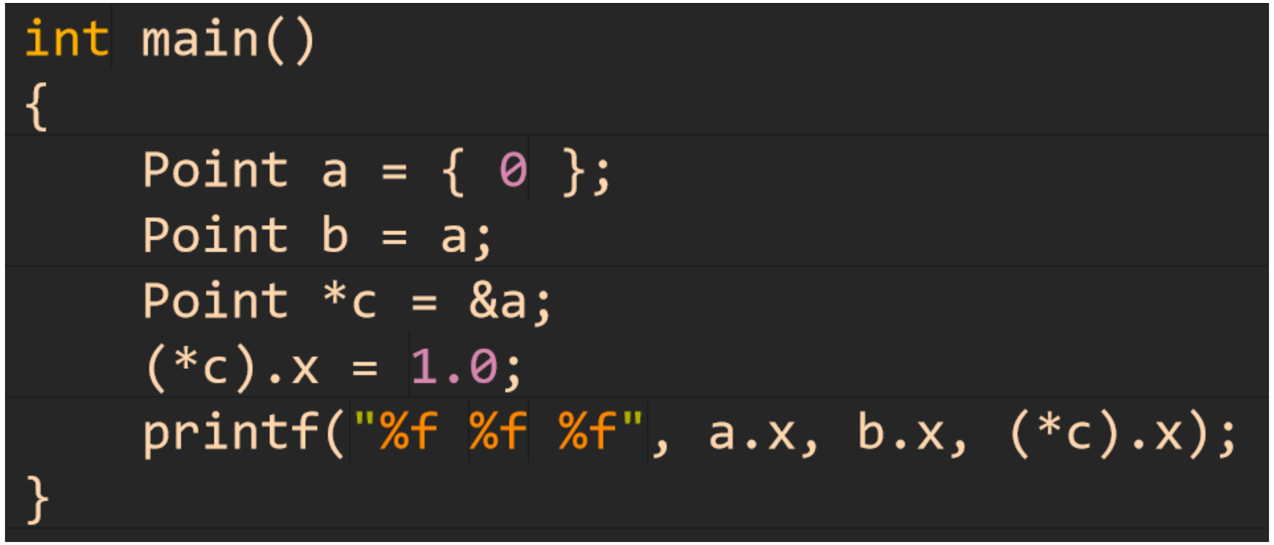
```
struct Point {
    double x;
    double y;
}
typedef struct Point Point;
```

- These two steps are commonly combined into one:

```
typedef struct Point {
    double x;
    double y;
} Point;
```

- Can you get rid of the redundancy of Point being repeated twice?
  - Yes, *but only if you do not need a recursive data type (linked list, tree, etc)*. In this case you could leave off the first Point to specify an *anonymous struct*.
  - Rather than remembering that caveat, we will always be redundant on this front in 211. We'll use recursive data types soon.

Trace the following code.



```
int main()
{
    Point a = { 0 };
    Point b = a;
    Point *c = &a;
    (*c).x = 1.0;
    printf("%f %f %f", a.x, b.x, (*c).x);
}
```

- Diagram the main frame's local variables

- Respond with the printed output.

Using `struct` values

- Access Members

```
aPoint.x
aPoint.y
```

- Assign to Members

```
aPoint.x = 1.0;
aPoint.y = 2.0;
```

- Take the Address Of

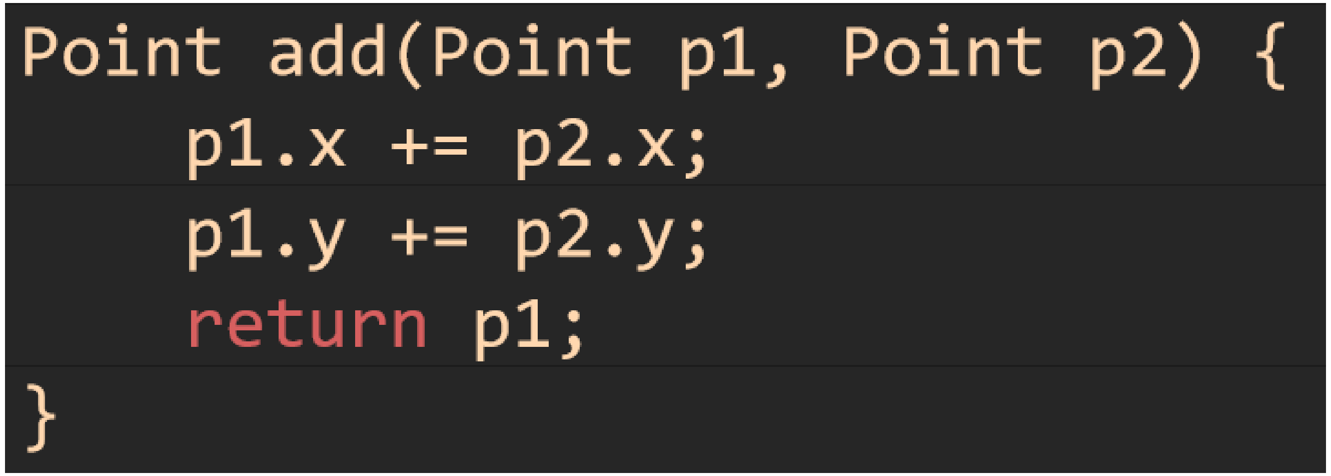
```
Point *aPointPointer = &aPoint;
```

- Copy over all members of a struct

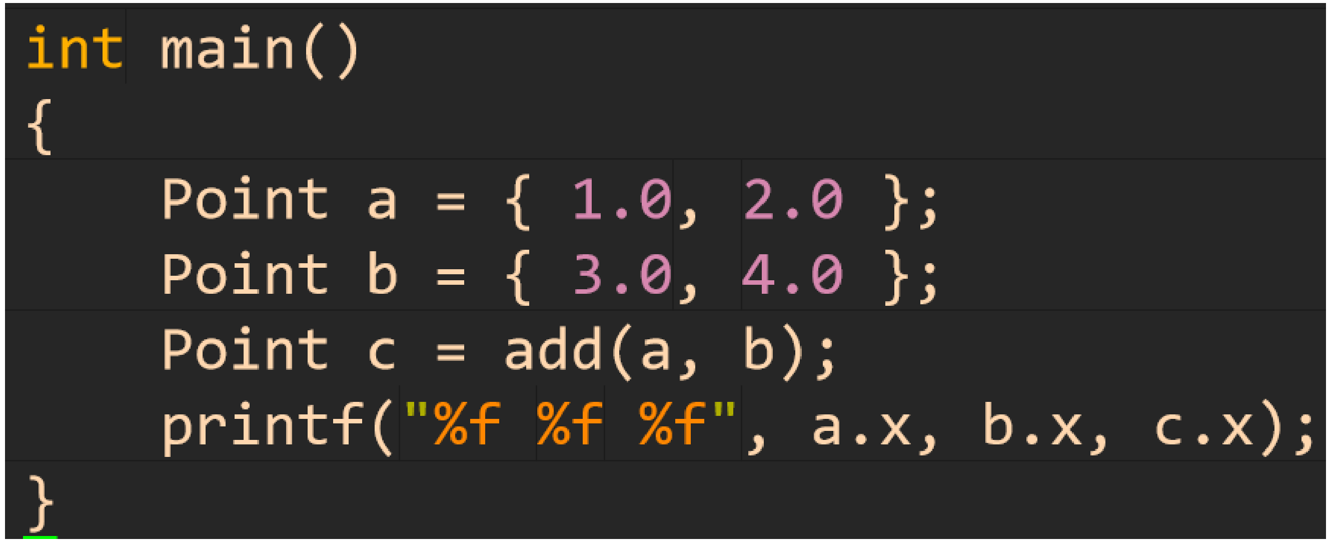
```
Point aCopiedPoint = aPoint;
*aPointPointer = someOtherPoint;
```

Be certain you understand the
big difference
between
a copy of a struct
and
a pointer to a struct
!!!!

Consider the following function...



```
Point add(Point p1, Point p2) {
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```



```
int main()
{
    Point a = { 1.0, 2.0 };
    Point b = { 3.0, 4.0 };
    Point c = add(a, b);
    printf("%f %f %f", a.x, b.x, c.x);
}
```

Accessing Members of struct Pointers with Arrow Syntax

- Consider the following variables:
  - `Point aPoint = { 0 };`
  - `Point *aPointPointer = &aPoint;`

- C provides a convenient arrow syntax for dereferencing a struct pointer and accessing a member:
  - `aPointPointer->x`
  - is syntactic sugar for: `(*aPointPointer).x`

- Also works for lvalues (left-hand side) in assignment statements:
  - `aPointPointer->y = 1.0;`
  - vs. `(*aPointPointer).y = 1.0;`

- When working with pointers to structs, the arrow syntax is strongly preferred.