CS 2123 Bootcamp
Structs
Dr. Mark Robinson
typedef

- Associates an identifier with a data type
- Great for:
  - Shortening long type declarations
  - Can help infer variable’s intended use
  - Ease differences in system data type sizes
typedef int feet, inches;
typedef double weight;

feet myHeightFt = 7;
inches myHeightIn = 4;
weight myWeight = 272.5;
sizeof Operator

- Returns the size of a data type in bytes
- Very handy
- Note: use \%lu as format specifier when printing to prevent warnings
- Same precedence as unary operators
**sizeof and Promotions**

Given:

- `sizeof int (bytes): 4`
- `sizeof short int (bytes): 2`
- `sizeof long int (bytes): 8`
- `sizeof float (bytes): 4`
- `sizeof double (bytes): 8`
- `sizeof long double (bytes): 16`

<table>
<thead>
<tr>
<th>Expression</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sizeof 150 * 4</code></td>
<td>16</td>
</tr>
<tr>
<td><code>sizeof (150 * 4)</code></td>
<td>4</td>
</tr>
<tr>
<td><code>sizeof (150.0 * 4)</code></td>
<td>8</td>
</tr>
<tr>
<td><code>sizeof (150.0 * (float) 4)</code></td>
<td>8</td>
</tr>
<tr>
<td><code>sizeof ((float) 150.0 * 4)</code></td>
<td>4</td>
</tr>
<tr>
<td><code>sizeof ((int) (150.0 * 4))</code></td>
<td>4</td>
</tr>
</tbody>
</table>
Structures

- A derived data type (or a customized data type)

- **Aggregate** other **heterogenous** data types (primitives and derived)
  - Comprised of other data types (its **members**) that all have something in common

- An array is an aggregate derived type of **homogenous** data types (each element is the same type)
Structure Syntax

- tag is optional (but can’t re-declare variables without it)
- Can declare variables of the struct at declaration time, or later
- Data types inside the struct are called member variables

```c
struct [tag] {
    <data type> <member identifier>;
    ...
} [variable identifier1, identifier2...];
```
Example

- Access a member using the dot access operator .

```c
struct person {
    char ssn[12];
    char * firstName;
    int age;
} people[10];

//create a single variable of person
struct person me;

strcpy(me.ssn, "111-22-3333");
printf("%s\n", me.ssn);
```
Typedefs for Simplification

- Use a typedef to shorten usage of struct tags when declaring variables
  - e.g., typedef struct person friend;

- Then can simply use the typedef identifier
  - e.g. friend bob;
Header Files

- Help decompose (modularize) codebase
  - Functionally similar statements are placed in same file
- Increases code reuse
- Helps readability
- Decreases maintenance
Example

- Put person struct declaration in a separate header file
- Any program that wants to use our custom data type includes the header file
Example

```c
struct person {  
    char ssn[12];  
    char * firstName;  
    int age;  
} people[10];

typedef struct person person;

...  
#include "person.h"

// create a single variable of person
person me;

int main(int argn, char * args[]) {
    strcpy(me.ssn, "111-22-3333");
    printf("%s\n", me.ssn);
    return(EXIT_SUCCESS);
}
```
Size of a Struct

- Really simple!
- Add up the size of all member data types
- For size calculations, pointers only take up 4 bytes
- What is the size of our person struct?
Structs as Parameters

- Remember: C is pass-by-value
- Passing a struct into a parameter passes a copy of the argument

Nothing else to see here... for now
//BAD WAY OF PASSING A STRUCT
void initSSN1(person p) {
    strcpy(p.ssn, "222-33-4444");
    printf("%s\n", p.ssn);
}

//GOOD WAY OF PASSING A STRUCT
void initSSN2(person *p) {
    strcpy(p->ssn, "333-44-5555");
}

Nothing else to see here... for now
Assigning one struct variable to another copies its member values.

- Primitive members are value copies.
- Derived types copy references (so pointers, etc. point to the same place in memory).

Example:

e.g., `person you = me;`
More Structure Declaration

- Can typedef a struct declaration immediately
- Does not have a tag, but typedef label can be used

```c
typedef struct {
    char ssn[12];
    char * firstName;
    int age;
} person;

person people[10];
```
Structure Initialization

- Can initialize at declaration using braces
  - e.g., `person bob = { "111-22-3333", "Bob Smith", 42 };`

- Can also initialize all members to zero (null)
  - char array [0] = null char, pointers are null
  - e.g., `person tmp = {0};`
  - does generate a warning though
Structs as Parameters

- Remember: C is pass-by-value
- Passing a struct into a parameter passes a copy of the argument
  - Try not to ever do this
- Instead, declare a pointer to the struct and pass that
  - Example: passing 4 bytes instead of 20
Example

//BAD WAY OF PASSING A STRUCT
void initSSN1(person p) {
    strcpy(p.ssn, "222-33-4444");
    printf("%s\n", p.ssn);
}

//GOOD WAY OF PASSING A STRUCT
void initSSN2(person *p) {
    strcpy(p->ssn, "333-44-5555");
}

- Access a struct pointer’s member using the arrow access operator ->
Exercise 1

- Write a C program that **adds** and **multiplies** pairs of **integers** (can statically allocate memory)

- Implement the pair as a struct and functions in the files pair.c and pair.h (functions should take struct pointers)

- Write the main program in a file called pairs.c

---

Output:

- Printing a...
  - Pair first: 1  second: 10
- Printing b...
  - Pair first: 2  second: 20
- ADDING a and b...
  - Pair first: 3  second: 30
- MULTIPLYING a and b...
  - Pair first: 2  second: 200

\[(a, b) + (c, d) = (a+c, b+d)\]
\[(a, b) \times (c, d) = (a*c, b*d)\]