LAB 2

Data Structures and Dynamic Singly Linked Lists in C

WHAT IS A DATA STRUCTURE?

- A way of organizing memory

WHAT IS A STRUCT? WHEN DO WE USE IT?

- A way to organize multiple variables under a single name
- Hold closely related information (name, address, etc.)

- Use it when we want to hold closely related information
- When we need a linked list of any kind

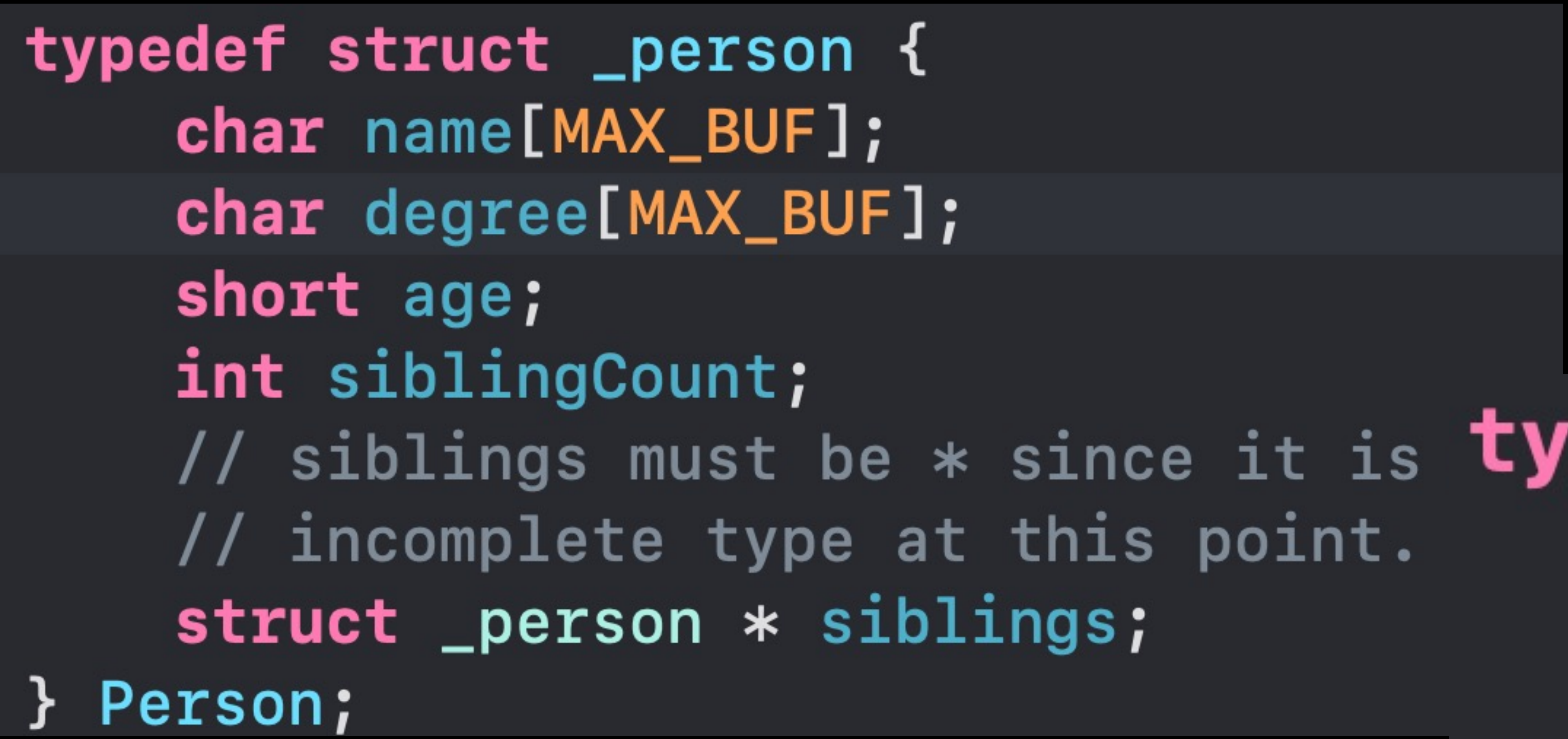
WHAT TYPES CAN BE IN A STRUCT?

- Anything!
- Int
- Double
- A different struct
- The same struct
- Pointer
- Etc.

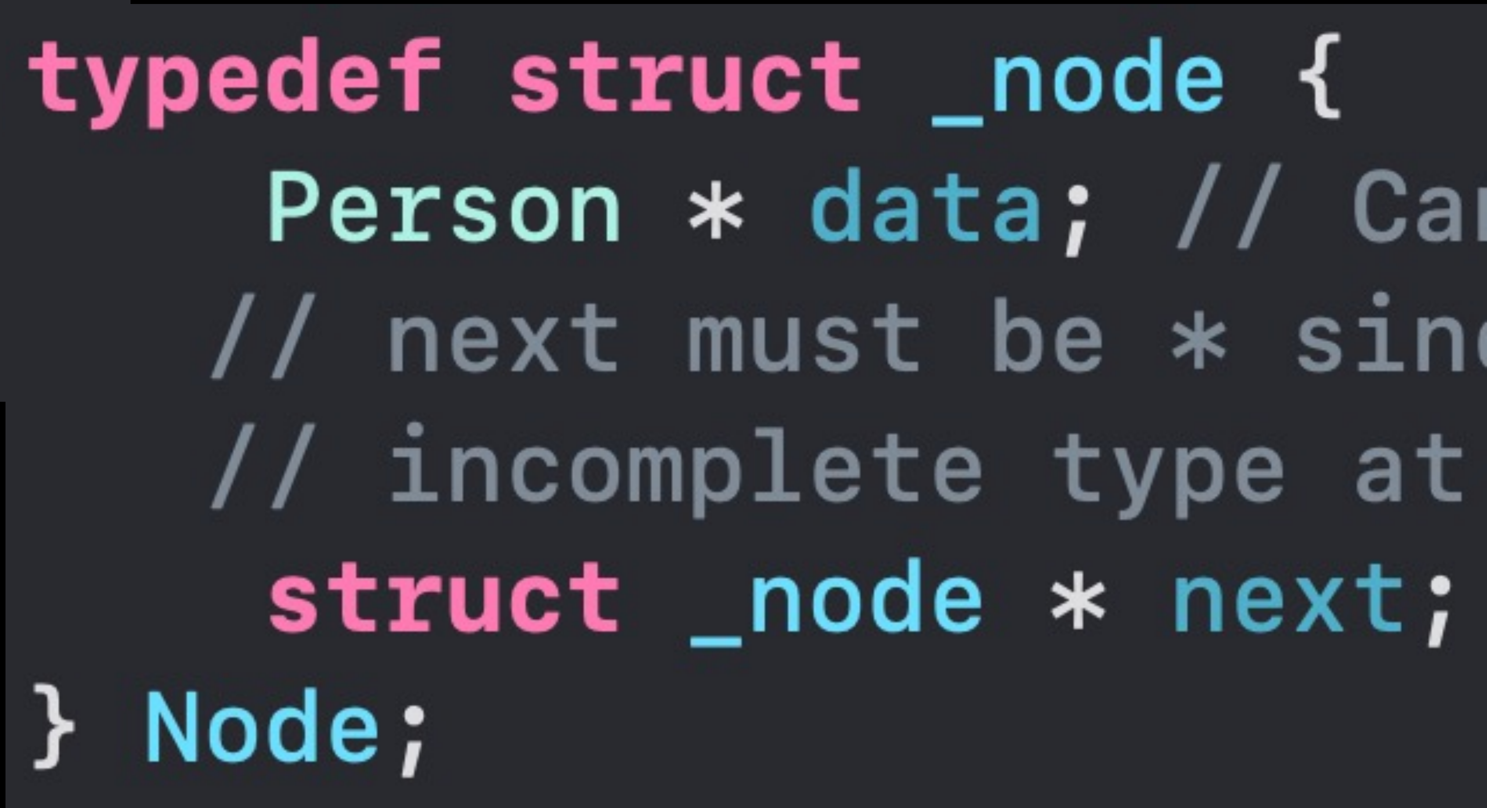
HOW TO REFERENCE A STRUCT WITHIN ITSELF

Where do we put structures?

Header files



```
typedef struct _person {
    char name[MAX_BUF];
    char degree[MAX_BUF];
    short age;
    int siblingCount;
    // siblings must be * since it is
    // incomplete type at this point.
    struct _person * siblings;
} Person;
```



```
typedef struct _node {
    Person * data; // Ca
  // next must be * sin
  // incomplete type at
    struct _node * next;
} Node;
```

IDEAL LAYOUT OF FILES FOR ALL PROJECTS

- Header = .h file; Source = .c file

- 1 header file for your structures
- 1 header/source file for your structure processing
  - Another one for output if you have many structures
  - Or a single header/source file for each structure – depends on specific circumstance
- 1 header/source file for your data processing
  - If there are a lot of different calculations which can be sectioned off, then it should be further broken down
- 1 header/source file for input/output
- 1 source file for main

- 7 files

WHY DO WE WANT TO BREAK FILES DOWN SO MUCH?

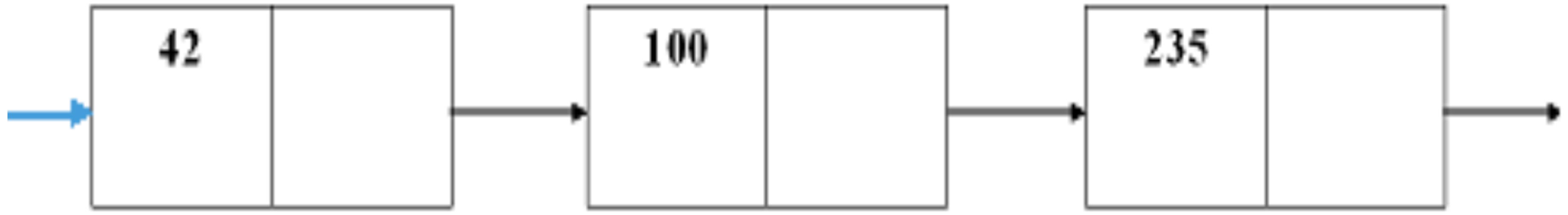
- Code becomes more portable between projects
- You copy files from project to project – nothing more complex
- You can continuously build upon your past work
- It will teach you to make code that is easily modifiable and eventually more general for multiple purposes
  - General purpose code will not always be an option

QUESTIONS?

PROBLEM 1 (MAIN PART)

- There are function prototypes in the instructions
- Copy/paste the structures and prototypes into your header file
- Then work on printing the list
    - If you have finish problem 2, come back and do it in a for loop, while loop, and recursive function

1. Linked lists may be used to implement many real world applications. Recall, linked lists are data structures, which represent collections of nodes that may be accessed sequentially via a pointer to the first node. A node contains data and a pointer to the next node in sequence. When the last node in the list is reached, its next pointer is NULL. A logical view of a singly linked list of integers is illustrated below:



Build an application, in C, for storing contact information (you must have one header file and two source files). For each contact you must store a name, phone number, email, and professional title. Your application must support insertions in order (based on last name), deletions, modifications, and printings of contacts. For this task you will be required to implement a dynamic singly linked list, which grows and shrinks at runtime. Build a menu that allows the user to add, remove, edit, print, store, and load contact information. The *store* feature should write contact information found in the list to a file. The *load* feature should read contact information from the same file into the list. Note: you should store the contact information in a struct called `Contact`. Each `Node` must be a struct, which consists of a `Contact` and a pointer to the next `Contact` in the list. Please see below.

PROBLEM 2 TESTS

- Test functions accept a void param and return a void
- Test functions are self-sustaining
    - If it accepts input or returns something, it is not proper
- Only test a **single** part or function
    - i.e., only test structure creation. Not creation and insertion in the same test

- Specifically,
- A test should test the creation of a node and properly destroy it once it is created. If you have a destroy function, do not use it – only use your creation then manually destroy the node.

NOTE ON TESTS

- X bits = x states
- 1 byte = 8 bits
- 1 int = 4 bytes
- 1 double = 8 bytes

You will never have a program that tests all possible cases because there are 2^{800} states!

A program with 1 int = 256 states!

- Suppose we have a program with 5 ints and 10 doubles in total
  - *Note: This includes parameters and local variables!*
  - 5 * 4 = 20, 10 * 8 = 80, 80 + 20 = 100 bytes ; 100 bytes = 800 bits
- 2^{800} =
  666801443287985427407985179072125779714475832231590816039625781176
  403723781763207152143220087155429074292991059343324044588880165411
  936508036335605233083004609515757951401455846307828591181402472896
  501613588660198169074803747646129116387737 6 (241 decimal digits)

"PERFECT" TESTS

- They cover 100% of code, cover every scenario, and are mythological

- Ideally, we will cover 80% of our code
- We will one test class per class
  - You may want to split tests up when the class is too complicated
    - At this point, you may want to split up your class or redesign
- We want design functions to be testable
  - Can we debug it?
  - Can we call it from outside? (Tricky problem, wait for CS321)
  - It is an algorithm (for some input I, do we ALWAYS get output O?)
    - If you answer no to this question, delete the function and start over

EDGE CASES

- Edge cases (for now, ignore)
  - Cases that should not affect many or any users
  - Look for cases where a value is on the edge and it could cause problems (i.e., num at max & min bounds)
  - Use the same part over and over again (much more than what a typical user would)
    - Keep incrementing a value that may cause unexpected behavior with another variable
    - If you are working with networking, ensure you handle the rst signal correctly
      - Should it crash the program, trigger a reset of something, or simply ignore it?
  - Ex: A finance app that incorrectly rounds

EXCEPTION TESTING

- Exception cases (for now, ignore)
  - You look for cases where an exception is thrown
  - You hope that it is either thrown or not thrown (depending on you needs)

- Overall, we do not like exceptions since they cost a lot of clock cycles!
- When you work with exceptions, you typically want to catch the error, then decide if you want to raise it again or not
  - When we throw it and keep raising it, we may not know where it came from

NORMAL TEST CASES

- This is what we are doing today
- Normal cases
  - You simply make sure the expected result occurs
  - If you have a function that adds two numbers, put in 1 and 2; make sure it equals 3
- Some cases for today:
  1. Ensure nodes are properly created
  2. Ensure nodes are destroyed properly
  3. Ensure nodes are inserted properly
  4. Ensure nodes are removed properly

TESTING DEMO IN SWIFT

- I used Swift so I could show exception cases easier

QUESTIONS?