# CS 61C

## Discussion 1: C Basics

John Yang
Summer 2019

## Announcements

- EPA: https://tinyurl.com/john61c
- Sign ups for small group tutoring now open!
- Guerilla Session Tomorrow (5-7 p.m., Cory 540AB)
- Homework 1 Due Tonight!
- Project 1 Released Friday morning

## Today’s Goal

- Cover the basics of C syntax that differ from pre-existing languages like Java, specifically Strings, Arrays, Pointers
- Preview how a program is mapped into memory during execution

## Introduction to C

## Introduction

- "Function", not "Object" oriented: No classes or OOP, we only have structs!
- Memory Management: *You* determine when to allocate a new object using free memory from free store.
    - Be careful! Improper management => Security Issues (CS 161) or Broken Functionality / Unexpected Behavior.
- Pointers: Remember environment diagrams? C has explicit syntax for <u>pointers</u> (int p vs. int *p)
    - Pointer ~ Memory address of where a value is stored!
    - Syntax: <data type> *<variable>
    - Dereferencing: Given a pointer "p", <u>*p</u> tells me the value at p. (Confusing definition! Thinking about a pointer as an *address* helps).
    - Address Of: Given a variable (pointer OR value) "p", <u>&p</u> tells me the address where p is stored AKA where does "p" live?

## Introduction

- Why C?
  - “Irritatingly obedient” - Nick Parlante, Essential C
  - Trade Offs!
    - More Control => 👍Take advantage of hardware (System Specific Code!) / 👎Manual management + responsibilities
    - Simple Set of Functionality => 👍Smaller code base to learn / 👎Fewer readily available functions and abstractions
- Miscellaneous
  - C is “pass-by-value”, pointers help us simulate “pass-by-reference”
  - Fundamental data types: int, char, float, double
  - Type modifiers (toggle storage space for variable) include: short, long, unsigned, signed

## Pointers

## Pointers Overview

- Basics:
    - Syntax: <data type> *<variable>
    - Colloquially, “*variable* is a pointer to a value of *data type*”
    - To *declare* a pointer: int *p;
    - To *dereference* a pointer: *p (gives 4)
- Miscellaneous
    - int *p vs. int* p - No difference
    - int *p, q - *Both* p and q are pointers to integer values
    - Question: Difference between **int *p = 4** and **int p = 4**?

## Pointers Overview

```
// Passing by Reference
void add_one_ref(int *i) {
        *i = *i + 1;
}

// Passing by Value
void add_one_val(int i) {
        i = i + 1;
}

int x = 1;
add_one_ref(&x);
add_one_val(x);
```

Remember!

- * means value at address or dereference
- & means get the address of a variable
- C functions are pass by value
  - a COPY of the function parameters get passed in
  - To bypass this, we use pointers to change values outside the function

## Strings + Arrays

## Arrays Overview

- Arrays: Contiguous Chunks of Memory

```c
/* defines an array of 10 integers */
int numbers[5] = {1, 2, 3, 4, 5};

char vowels[1][5] = {
    {'a', 'e', 'i', 'o', 'u'}
};
```

- Zero Indexing
- Cannot change the size of the array! (Array doesn’t know its own size!)
- Array variable name points at first value of the array (i.e. “numbers” => 1)
- Accessing an array value:
    - Both Arr[0] and *(Arr) => Element at Index 0
    - Both Arr[2] and *(Arr + 2) => Element at Index 2
- *Unlike Java*, we don’t have “add”, “remove”, or other fancy methods (no OOP!)

# Strings Overview

- There is no explicit “string” type in C
- Instead, Strings = Array of Characters!

```c
/* use pointers to a character array to define simple strings */
char * name = "John Smith";
/* define a string which can be manipulated */
char name[] = "John Smith";

char * name = "John Smith";
int age = 27;
/* prints out 'John Smith is 27 years old.' */
printf("%s is %d years old.\n", name, age);

/* prints out the length of value at ‘name’ variable */
printf("%d\n",strlen(name));
```

- Every string has special “null terminator” character, tells us when string ends
  - Why? Because we have to tell C compiler *explicitly* when string terminates
  - Therefore, `char name[11] = "John Smith";` must have length of *11*, not *10*. Extra character = null term!

## Structs

## Structs Introduction

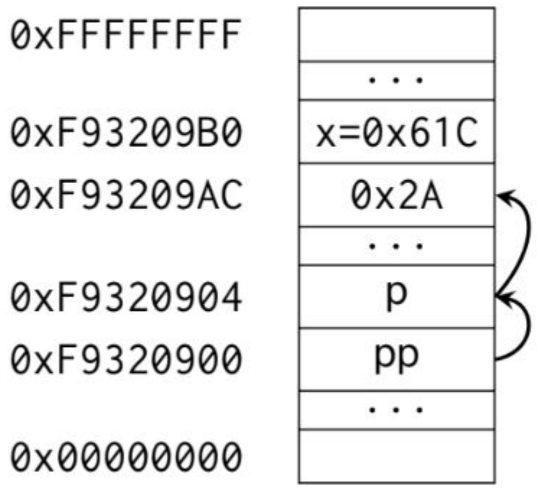
- Remember, C doesn’t have any classes (no OOP design!)

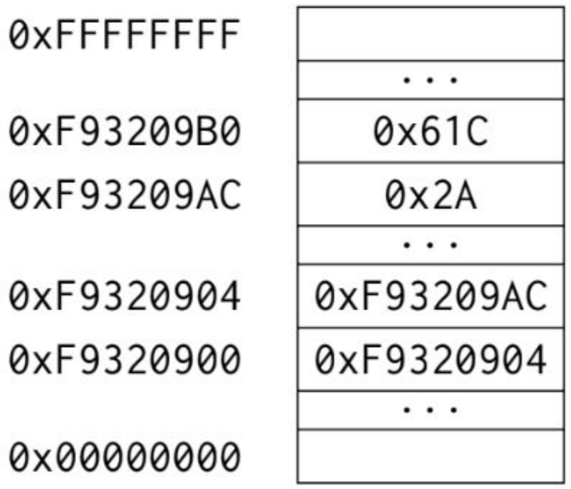
```c
typedef struct {
    char * name;
    int age;
} person;

person * myperson = malloc(sizeof(person));
myperson->name = "John";
myperson->age = 27;

(*myperson).name = "John";
(*myperson).age = 27;
```

- Two main ways to access fields of struct
  - “.” - get the field out of a struct
  - “->” is used to get the field out of a struct **pointer**
    - Also equal to (*ptr).

## Memory Layout

# Memory Layout

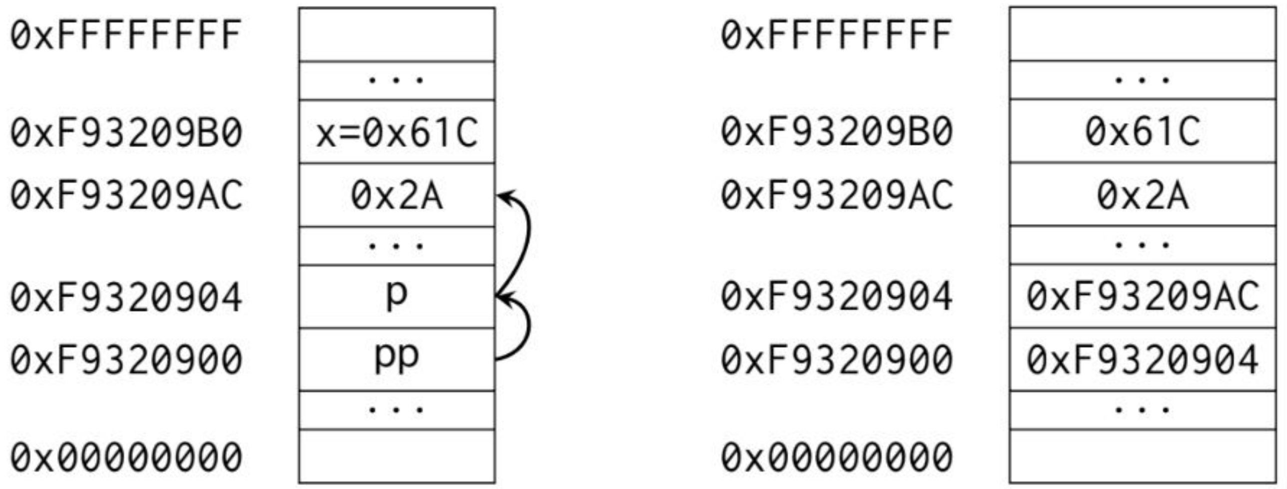




On the left, we have a “box-and-pointer” diagram, but what it looks like underneath that abstraction is what we have on the right.

If int p is located at 0xF9320904 and int x is located at 0xF93209B0, what are the following: *p, p, x, &x

## Memory Layout



- `*p = 0x2A`
- `p = 0xF93209AC`
- `x = 0x61C`
- `&x = 0xF93209B0`

## Discussion 1

## Discussion 1

### 2. Uncommented Code? Yuck!

1. Returns the sum of the first N elements in ARR.
2. Returns -1 times the number of zeroes in the first N elements of ARR.
3. Does nothing. (Pointers vs. Values)

### 4. Problem?

a. Whenever you iterate through an array, pass in the size! What does sizeof(summands) actually give you? The size of the type!

b. When iterating through a string, unlike an array, we terminate when we encounter the *null terminator* character, not by the size.

c. No errors... (We're "Gucci Gang" - Lil Pump)

d. Variable declaration is incorrect! What does `char *srcptr, *replaceptr;` actually give you?