Week 02-1

CSC209 Fall 2023

Dr. Demetres (dee) Kostas

September 19, 2023

Announcements

- A1 due next Wednesday
- Week 2 prepare is over
  - hopefully you submitted it
  - it should be enough to
  - may need to review
    - for loops, functions and conditionals
    - to make sure you can complete assignment

Putting the systems aside

- the next several weeks
  - we will focus on unique aspects of C
  - unique as compared to other languages
- The system tools you've learned
  - should be enough for the time being
    - while we get up to speed in C

Arrays

- sequences of variables
  - all of the same type
  - importantly (in C)
    - they are laid out sequentially
    - in the memory model!

Declaration

```
// generic formula
type name[size];
// an integer array with 3 elements
int arr[3];
// initialize the array with values, 1, 2, 3
int arr[3] = {1, 2, 3};
```

Indexing

- initialization does assignment
- but otherwise can assign with index
  - `arr[0] = 0;`
    - sets the first element to be 0
- can also access values through indexes
  - `arr[2] * 2 // should be 6`

Pointers

- All variables in C store values
- but memory addresses themselves
  - are a kind of value
  - they are just big numbers
    - (8 bytes in size)
- Pointers allow you to have variables
  - that are **understood** as memory addresses

pointing

- A `*` trailing the type
    - indicates a pointer e.g. `int *a;`
- we say that when a variable
    - holds the value of the address
        - of another variable
        - the first one *points to* the second
- e.g. p *points to* a

```
int a = 10;
int *p = &a;
```

Using `*`

- each use of `*` **derefernces**
    - meaning, it uses the value of the variable
    - as a *reference*
        - then retrieves the value
            - indicated by the reference
- this is separate from indicating type!
    - `int *p` is not dereferencing anything
        - just declaring the type as pointer
            - to `int`

Using `&`

- each use of `&`
    - gets the address of
        - whatever statement it prefixes
- Every value is stored *somewhere*
    - meaning it always has an address
    - `&` accesses this address

Connecting it all together

- we need to notice that each variable
- is added to the stack(frame)
  - local to the particular function/context
    - scope of the variable
- Each new value/variable has an address
  - and now we can access address and values
    - using combinations of `*`'s and `&`'s

`array_and_pointer_basics.pdf`

`calls_and_pointers.pdf`