C - POINTERS (CONT.)

RECAP LAST CLASS

- pointers store memory address
- pointers associated with a certain type
  - `int *` vs `char *`
- `&` -> "address of" (dereference) operator
- `*` -> "value of" operator
- print using `printf` command with `%p` formatter

STACK VS HEAP

- Two different areas of memory
- Typically implemented with stack and heap data structures, respectively
- How a variable is declared -> where it goes
- Everything we've looked at so far is on stack

STACK

- New block whenever a function called
- Block = scope
  - local variables stored in that block
  - block freed when function returns (aka variables disappear)

HEAP

- Memory allocated dynamically
- Memory can be allocated and freed at any time
- Entire program has access to the heap (no real scope)
- Memory must be explicitly freed
- Memory leaks
  - Lose access to memory
  - Computer doesn't know memory is no longer accessible

POINTERS AND ARRAYS

- Array name = pointer to element 0 that can't be changed
- Pointer arithmetic becomes useful
  - walking down array

ALLOCATING MEMORY

- `malloc(size)` -> request a certain number of bytes of memory
  - on the heap
  - contiguous
  - returns address of start of block
  - needs to be cast to appropriate pointer

```
int *p;
p = (int *) malloc(sizeof(int)*100);
```

ALLOCATING MEMORY (CONT.)

- `calloc(count, size)`
  - `count` -> number of items
  - `size` -> size of item
  - allocated contiguously
  - initialized to 0

```
int *p;
p = (int *) calloc(100, sizeof(int));
```

ALLOCATING MEMORY (CONT.)

- `realloc(ptr, size)`
  - `ptr` -> current pointer to memory trying to resize
  - `size` -> overall size wanted
- may not be enough room to enlarge
  - creates new, copies to new, frees the old
  - additional not guaranteed to be 0 filled

```
int *p = (int *) malloc(sizeof(int)*10);
p = (int *) realloc(ptr, sizeof(int)*20);
```

FREEING MEMORY

- `free(ptr)`

```
int *p = (int *) malloc(sizeof(int)*10);
free(p);
```