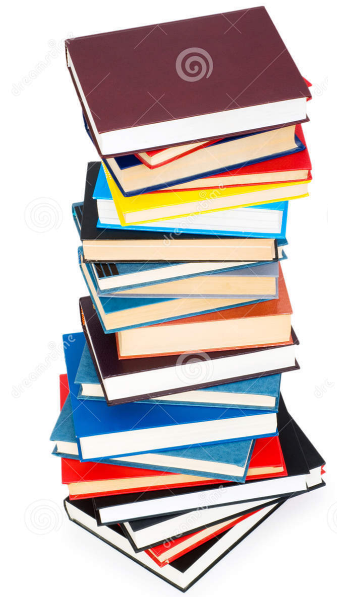
Today’s lecture

- **Arrays/pointers as parameters**
  - Pointers and pass by reference
- **Stack allocation**
  - Stack declaration, scope, lifetime
- **Heap allocation**
  - malloc, realloc, free
  - How to use as client
  - Contractual guarantees
- **Stack versus heap allocation**
  - Features/limitations
  - How to choose which to use



C parameters are pass-by-value

Parameter is callee's variable, distinct from caller's, value is copied

```
void change(char ch)
{
   ch = toupper(ch);
}


int main(void)
{
   char letter = 'a';
   change(letter);
```

letter is <u>unchanged</u>

To achieve pass-by-reference, do manually

```
void change(char *p_ch)
{
   *p_ch = toupper(*p_ch);
}


int main(void)
{
   char letter = 'a';
   change(&letter);
```

letter <u>is changed</u>

Same applies to pointer parameters!

Draw a picture!!

```
void change(char *str)
{
   str = str + 1
}


int main(void)
{
   char *name = "drain";
   change(name);
```

name is <u>unchanged</u>

```
void change(char **p_str)
{
   *p_str = *p_str + 1;
}


int main(void)
{
   char *name = "drain";
   change(&name);
```

name <u>is changed</u>

Let’s code and draw!

```
/afs/ir/class/cs107/samples/lect7
```

Heap allocator analogy

- Request memory by size (malloc)
  - Receive room key to first of connecting rooms
- Need more room? (realloc)
  - Extend into connecting room if available
  - If not, trade for new digs, bellman moves your stuff for you
- Checkout when done (free)
  - You remember your room number though
- Errors! What happens if you…
  - Forget to check out?
  - Bust through connecting door to neighbor?
    - what if neighboring room in use? yikes!
  - Return to room after checkout?

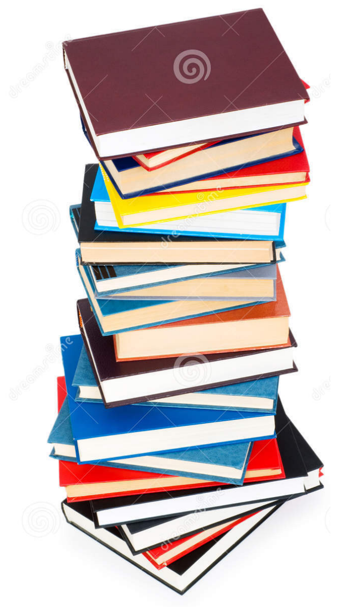
Heap allocator functions

```
void *malloc(size_t nbytes);
void free(void *ptr);
void *realloc(void *ptr, size_t nbytes);
```

- **Contractual guarantees**
  - NULL on allocation failure
  - Address of memory is contiguous block of at least nbytes
  - Not recycled unless you call free
  - Realloc preserves existing data
- **Undefined behaviors**
  - What are initial contents? How many bytes actually reserved?
  - What happens if write outside bounds, use after free, free twice, realloc non-heap address?

Stack allocation (i.e. "local variables")

- **Very efficient**
  - Fast to allocate/deallocate, ok to oversize
- **Not especially plentiful**
  - Total stack size fixed, default 8MB
- **Convenient**
  - Automatic allocation/deallocation on function entry/exit
  - Can declare and initialize in one step
- **Reasonable type safety**
- **Somewhat inflexible**
  - Declarations are fixed at compile-time, cannot add/resize at runtime
  - Scope/lifetime dictated by control flow in/out of functions



Compare/contrast to: heap allocation

- **Moderately efficient**
  - Will search for available space, update record-keeping
- **Very plentiful**
  - Heap enlarges on demand to limits of address space
- **Allocation/deallocation under programmer control**
  - Can precisely determine lifetime
- **Very flexible**
  - Runtime decisions about how much to allocate and when, can resize
- **Lots of opportunity for error**
  - Low type safety
  - Forget to allocate, allocate wrong size, free before done, etc.
  - Leaks
    - Much less critical

How do you choose which to use?

- **Use stack if possible, go to heap only when you must**
  - Stack is safer, more efficient, more convenient
- **What requires heap?**
  - Very large allocation that could blow out stack
  - Dynamic construction, not known at compile-time what declarations will be needed
  - Need to control lifetime — memory must persist outside of function call
  - Need to resize memory after initial allocation
- **With heap, comes responsibility**
  - Your responsibility for correct allocation at right time and right size
  - Your responsibility to manage the pointee type and size
  - Your responsibility for correct deallocation at right time, once and only once
  - Valgrind is your friend!