Today's lecture

- **Heap allocation**
  - malloc, realloc, free
  - How to use as client
  - Contractual guarantees
- **Stack versus heap allocation**
  - Features/limitations
  - How to choose which to use
- **Void *, generics**
  - Raw memory
  - Function pointers

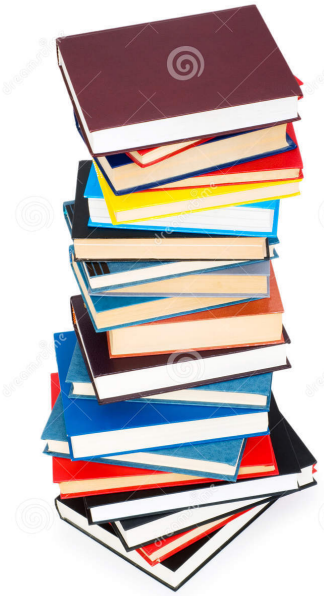
Heap allocator functions

```
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!

Generic pointers

- `void*` pointer
  - Variable of type address with unspecified/unknown pointee type
- What you can do with a `void *`
  - Pass to/from function, pointer assignment
- What you cannot
  - Cannot dereference
  - Cannot do pointer arithmetic
  - Cannot use array indexing (depends on both arithmetic & dereference!)
- Why do you want one?
  - What gain is there in "forgetting" the pointer type?
- Generic functions!
  - All data has an address, by referring to it by address we can smooth over differences in data type

Let's code & draw!

`/afs/ir/class/cs107/samples/lect8`

`generic.c`

Operations that rely on data type

- **Process raw bytes without knowing what they are**
  - memcpy, memmove, memchr, memcmp, …
- **What if need more meaningful behavior per-type?**
  - Do "something" to each element or filter/sort elements
- **Coordinate with client via callback function**
  - As needed. generic operation "calls back" to client who knows the specifics of data
- **Generic implementation**
  - Works in terms of void*, manual pointer arithmetic, raw memory operations,
  - No knowledge of what the data is, only its size
  - Client passes data as void *, "forgets" type
- **Generic client**
  - Supplies the callback function that operates in specific on the data
  - Must cast void* back to specific type