Heap management

- compiler must address language's memory management plan
- can programmer explicitly request/free memory?
- is garbage collection automated, and scheduled how?
- is there built in protection against memory leaks, dangling pointers, etc?
- when memory is requested, how is the specific memory segment found and allocated?
- when memory is freed, how is it returned to the free “pool”?
- what happens when free memory is low, or fragmented?

Allocation/deallocation

- typically maintain a pool of chunks of free memory
- many different allocation algorithms for handling requests
  - first fit: take from first big enough chunk
  - best fit: take from smallest chunk that's big enough
  - worst fit: take from largest existing free chunk
- on deallocation, need to return to free pool, and ideally consolidate adjacent free chunks into single larger chunks
- we'll look at a gnu-ish approach shortly

Garbage collection

- If just a single pointer/reference to each dynamically allocated item then it's easy to see when to deallocate
- With multiple pointers/references, how do we know when the last reference to something ends?
- When should the item itself be deleted?
- Common approaches are
  - maintain reference counts (delete when count of refs hits 0)
  - sweep algorithms (search out what is/is not still accessible)

Sweep algorithms

- sweep algorithms look to eliminate memory leaks: find things that are no longer accessible and delete them
  - Mark every allocated item in heap as “unreferenced”
  - Go through every pointer/reference currently active, mark what they point at as “referenced”
  - Delete everything that is still “unreferenced”
- can be invoked at fixed intervals, “on demand” by program, or when free memory drops below specified threshold

Reference counts

- often used in “smart pointer” approaches
- when something is allocated, create a smart pointer object that is a pointer into the heap plus a count
- pointers/references to the target item go through smart pointer, count is incremented/decremented as needed
- when count hits 0 delete the memory and set the smart pointer's internal pointer to null (keep smart pointer alive)
- remaining smart pointer also catches dangling references

defragging: copy collectors

- Might have lots of tiny fragments of free memory, but none big enough to satisfy existing request
- copy collector copies all current allocations to a second memory pool, consolidating them into one large block, then freeing original memory pool
- likely to be slow
- breaks programmer code if they're maintaining/using pointers to items (but might play well with smart pointers)