Dynamic memory issues

- Runtime allocation/release requests for chunks of memory from outside call stack (generally “heap” space)
- Might be programmer specific requests (e.g. new/delete, malloc/calloc) or implicit requests (e.g. on creation of a new object)
- Allocated resources must be accessible within the program, need to be freed after released, and need to prevent misuse
- How are responsibilities divided between programmer and system?

References vs pointers

- As discussed in pointers section, references hide implementation details from programmer, all responsibility for request/access/release dealt with system side
- Pointer-based systems put control (and responsibility) in hands of developer, powerful but risky

  (smart pointers an attempt to bridge the gap)
- For ptrs or refs, system needs to maintain a pool of available memory chunks, find and allocate chunks to fill requests, and return chunks to the pool once released

Pointer-based issues

- Wild pointers: programmer uses an uninitialized pointer as if memory had already been allocated through it
- Invalid pointers: programmer accesses memory through pointer containing an invalid address (out of range, violates alignment boundaries, violates permissions)
- Dangling pointers: programmer accesses memory through pointer after that space has been released/deallocated
- Memory leaks: programmer neglects to deallocate memory once done with it, and/or loses last pointer that references it

Resource allocation/release

- System must keep track of memory available to be allocated (memory pool)
- On request, system must find appropriate chunk of memory and remove from pool, give access to program
- On release, system must return chunk to pool
- Allocation/deallocation needs to handle allocate/release requests quickly, but also to organize memory pool in a way that minimizes fragmentation (carving memory into many many chunks too small to be useful for most requests)

Garbage collection

- Ideally, system recaptures memory and returns to pool once it is no longer in use by the program
- Eager approaches: recapture asap once memory is no longer needed (e.g. reference counts)
- When needed: delay recapture until we reach a point where we can no longer service incoming requests, then go through and recapture everything available
- Periodic: run recapture routines at fixed intervals (or when needed)
- On demand: run recapture routines at programmer request

Garbage collection impact

- Need some memory overhead to track what is/is not in use
- Need some cpu overhead to update what is/is not in use, and to actually perform recapture
- Cpu delay can be significant if recapture needs to explore many chunks of memory, application is paused while this runs (can give application appearance of freezing or stuttering)
- Which garbage collection approach is chosen determines when the recapture (and application pauses) takes place, and how much control developer has over that
- Significant implications for systems that need to run in/close to real time