Memory Management

- activation records (frames)
  - static
  - LIFO
- heap
  - LIAO (last in any out) - freed in any order

currying breaks these boundaries

```
(define f (lambda (x)
    (lambda (y)
    (+ x y))))
```

- activation records don't need to be on stack
  - in Scheme: AR's are on heap (optimized to stack in many cases)

Heap Management

- LIAO (objects can be freed in any order)
- stack, global registers
  - not controlled by heap manager, but it needs to know if it accesses the heap
- remove objects that are not reachable from directly accessed variables
- how to keep track of roots?
  - program tells heap manager where the roots are (explicit calls)
  - compiler records root locations (records in read-only tables visible to heap manager)
  - managed heap (garbage collection)

```
char *buf = malloc(1000);
...
fee(buf); // dangling pointer
return buf != NULL; // undefined behavior
```

- keep LL to know where free areas are -> "free list"
- auxiliary heap to keep track of free areas (bad)
- store this info in the free list itself
- `malloc(N)` - cost = O(1) if N $\leq$ size of first area

- cost = O(free list) if free data is at the end
- ^ first fit

best fit:

- search for best free data block'
- cons:
  - need to check entire list every time
- roving pointer - point to the largest free block
- suppose we allocate something too big:
  - `malloc(10000)`
- external fragmentation - enough space but its scattered around
- internal fragmentation - user asks for n bytes but we give n + 3
  - more space in an area than requested

Mark and Sweep

- assume the system knows where the roots are, if objects are in use, and other stuff
- mark - recursively descend from roots, mark as you go
- sweep - free unmarked objects

1. mark
2. sweep
3. unmark remaining objects - O(1) (realtime garbage collection)

- more efficient to use normal mark and sweep

Problems with C/C++

- (Problem A) dangling pointers
- (Problem B) memory leaks
  - survives until the next mark and sweep

Conservative G.C.

- can be used in C, C++

```
define free(p) ((void) 0) // won't have problem A
```

- when running mark phase, look at entire stack and look at registers for what looks like pointers to the heap

- occasionally mark something that isn't in use -> minor memory leak

Python G.C.

- reference counts - every object knows how many other objects reference it
- pros:
  - delete object when count hits 0 - O(1) memory reclaims
  - simple
  - immediate frees
  - no big overhead
- cons:
  - has to update 2 pointers
  - slower computations
  - cost of assignment goes up
  - need storage for reference counts

faster G.C. in Java:

- generation-based collector
  - oldest, middle, youngest, nursery
- free space is in nursery
  - has two pointers, hp (heap pointer) and lp (limit pointer)
- objects point to older objects but not to newer ones
- all it needs to is is G.C. just the nursery

```
malloc(a)
void *p = hp;
hp += n;
if (lp < hp)
    ouch(); // G.C. or add generation
return p;
```

- pros:
  - do minor G.Cs rather than look at the entire heap

copying collector

- pros:
  - better caching
  - cost proportional to objects in use not total number of objects

- cons:
  - update roots
  - make copies (memcpy)
- can't use finalize with this collector -> needs 2 collectors