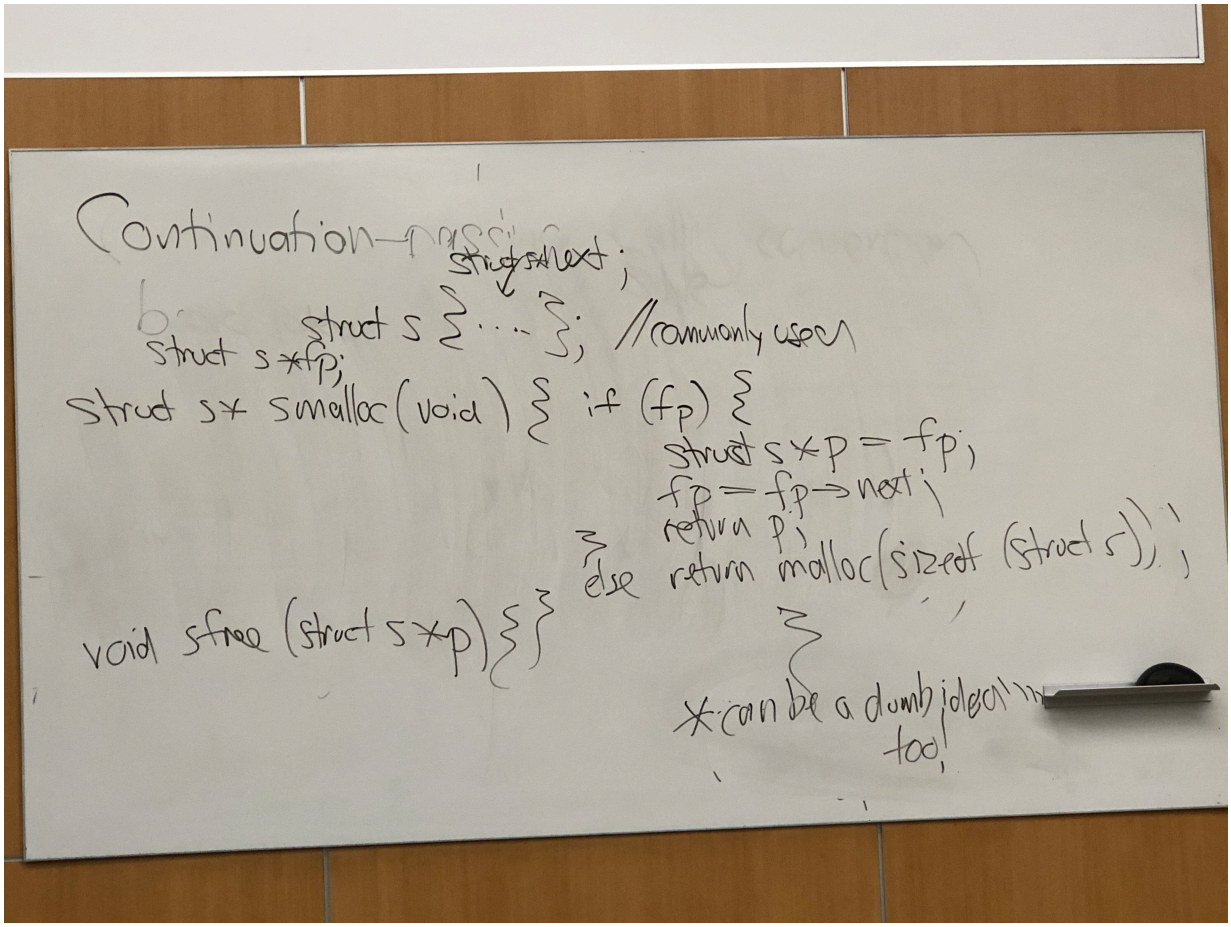
CS131 Week 9

Week 9 Lecture 1

==========

First half of lecture was on memory allocation and storage management in C: CS111 stuff
Some example stuff from first half...



Malloc
malloc is **user-mode**

- means NO syscalls, faster

Linux:

- mmap (.....)
  - changes current process' page tables

When to use storage library functions

- getchar : read :: malloc : mmap : free ===> for small objects
- malloc use mmap ===> for large objects

Garbage Collection

Solves 2 problems

1. dangling pointers
    1. free(p); *p;
    2. * HAVE TO FIX because CORRECTNESS issue
2. memory leaks
    1. *p; …….[no p usage later]
    2. * Don’t HAVE to fix, just PERFORMANCE issue

Mark and Sweep

Classic approach “mark & sweep”

- Start at root (stack, register, global static storage)
- Follow pointer list to check all the other blocks of allocated memory (in heap) to find garbage
- Scans entire heap

mark: find all reachable objects, mark them, tell you which objects are in use
sweep: free all unmarked objects
+: simple, no ref counts
-: program freezes occasionally
***has to work with free list, manipulate it, is annoying but effective. will go over better methods in next lecture

Problems with Garbage Collection

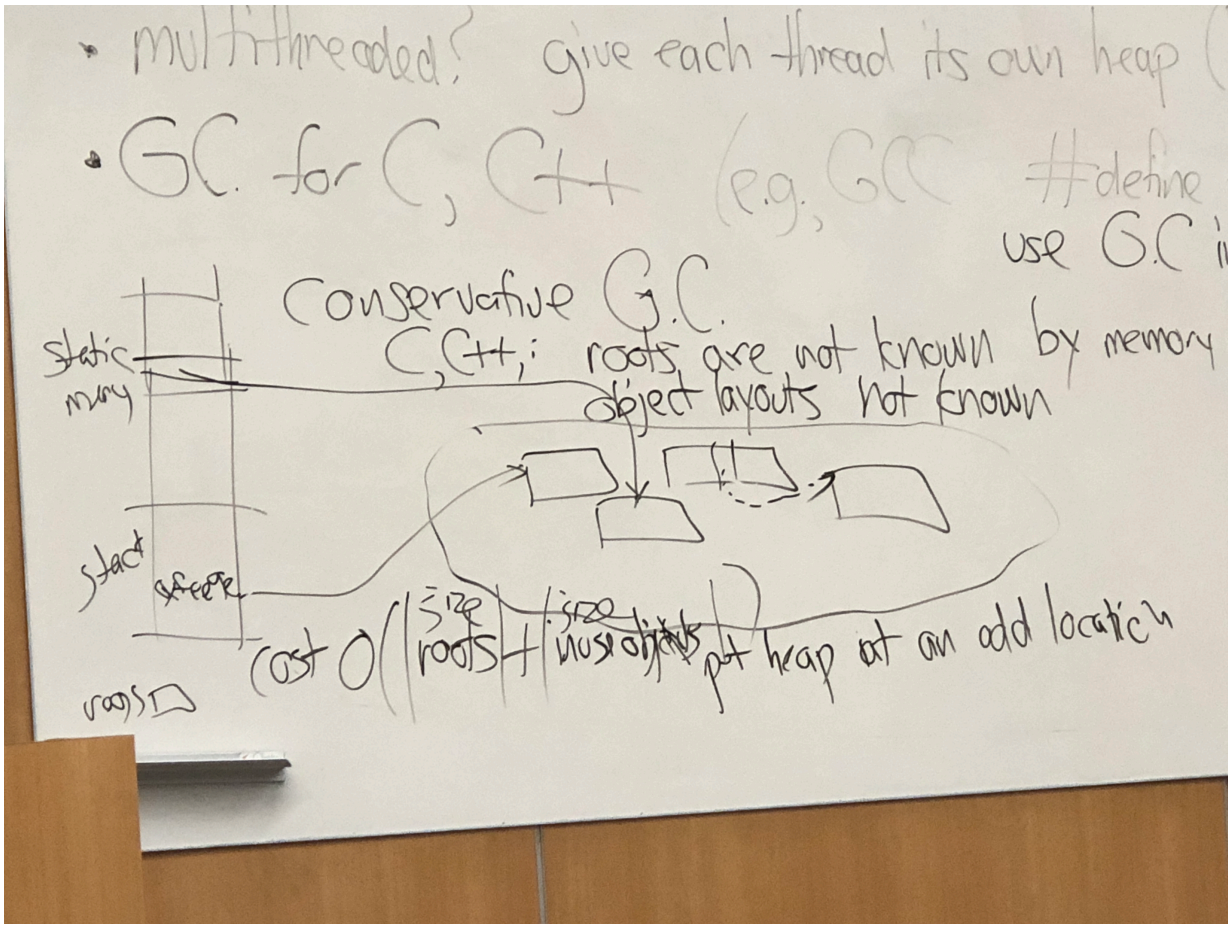
how do we handle…
multithreaded: give each thread its own heap?

- GC for C, C++ (e.g. gcc #define free(p) /**/)
    - use Garbage collector internally
    - **conservative garbage collector**: C, C++, roots are unknown, object layouts unknown
    - will sometimes mistakenly see large integer in stack, think its a pointer, will think a block of memory in heap that large int “points to” is in use, will not free block
    - as a result, might not free blocks that aren’t in use
    - but will NEVER have dangling pointers

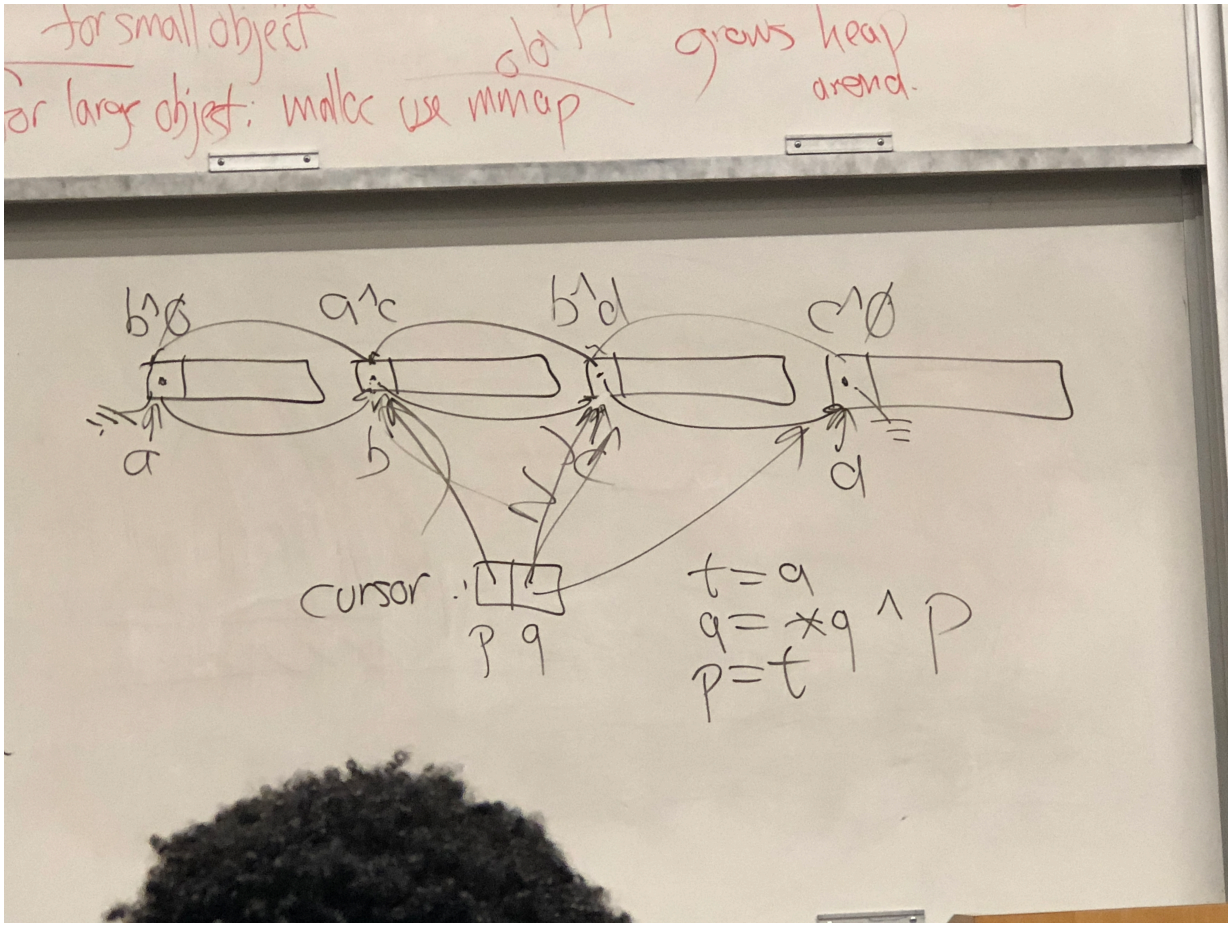
For Garbage Collector to work: need to know which objects are reachable

- thus, need to know where the roots are

- problem is, C and C++ DON'T KNOW WHERE THE ROOTS ARE



Implementing Doubly Linked List using only 1 pointer per node
Solution: use XOR

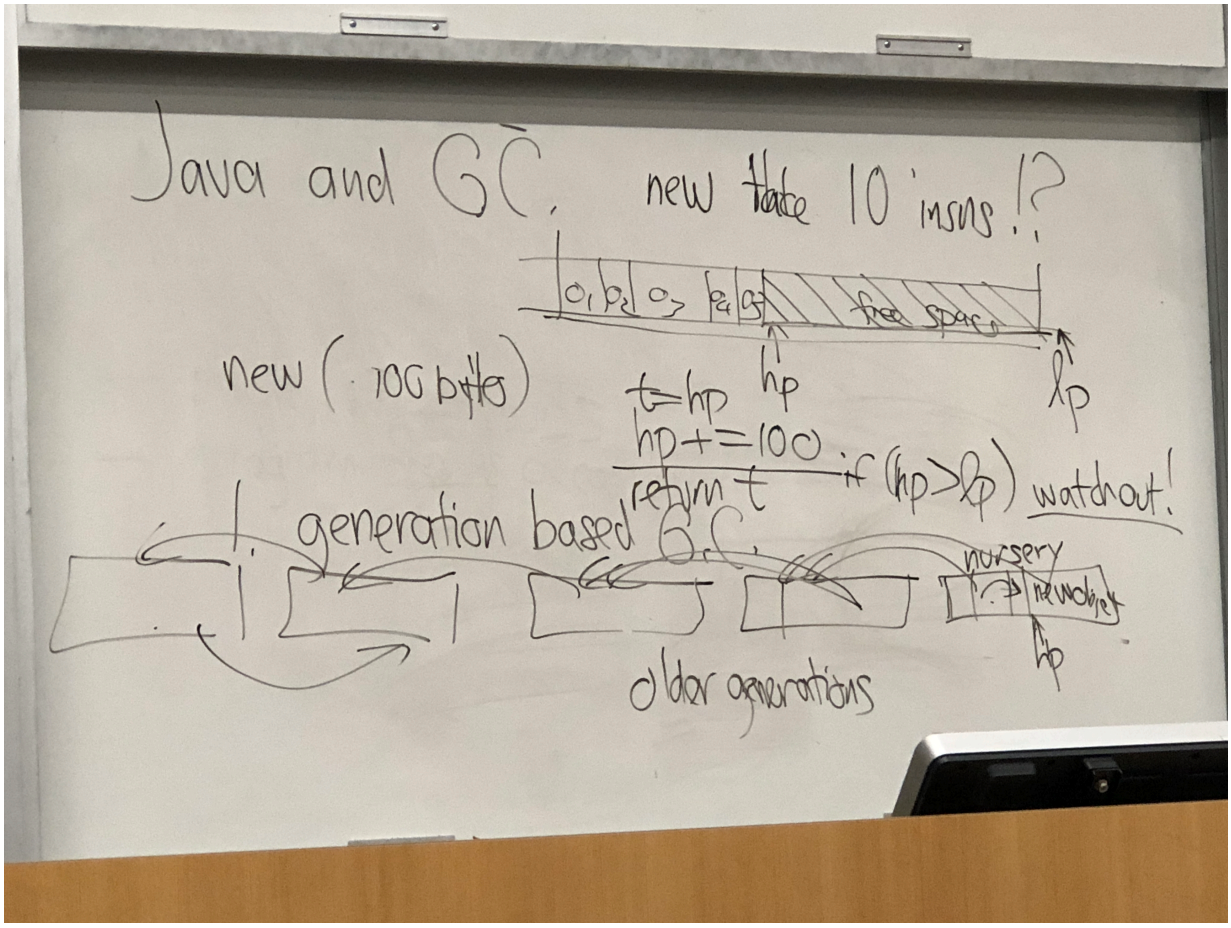


This sort of doubly linked trick with XOR will make garbage collection fail

- when you XOR things together it wont look like memory addresses anymore, just liike normal number

Java and Garbage Collection

new takes 10 insns

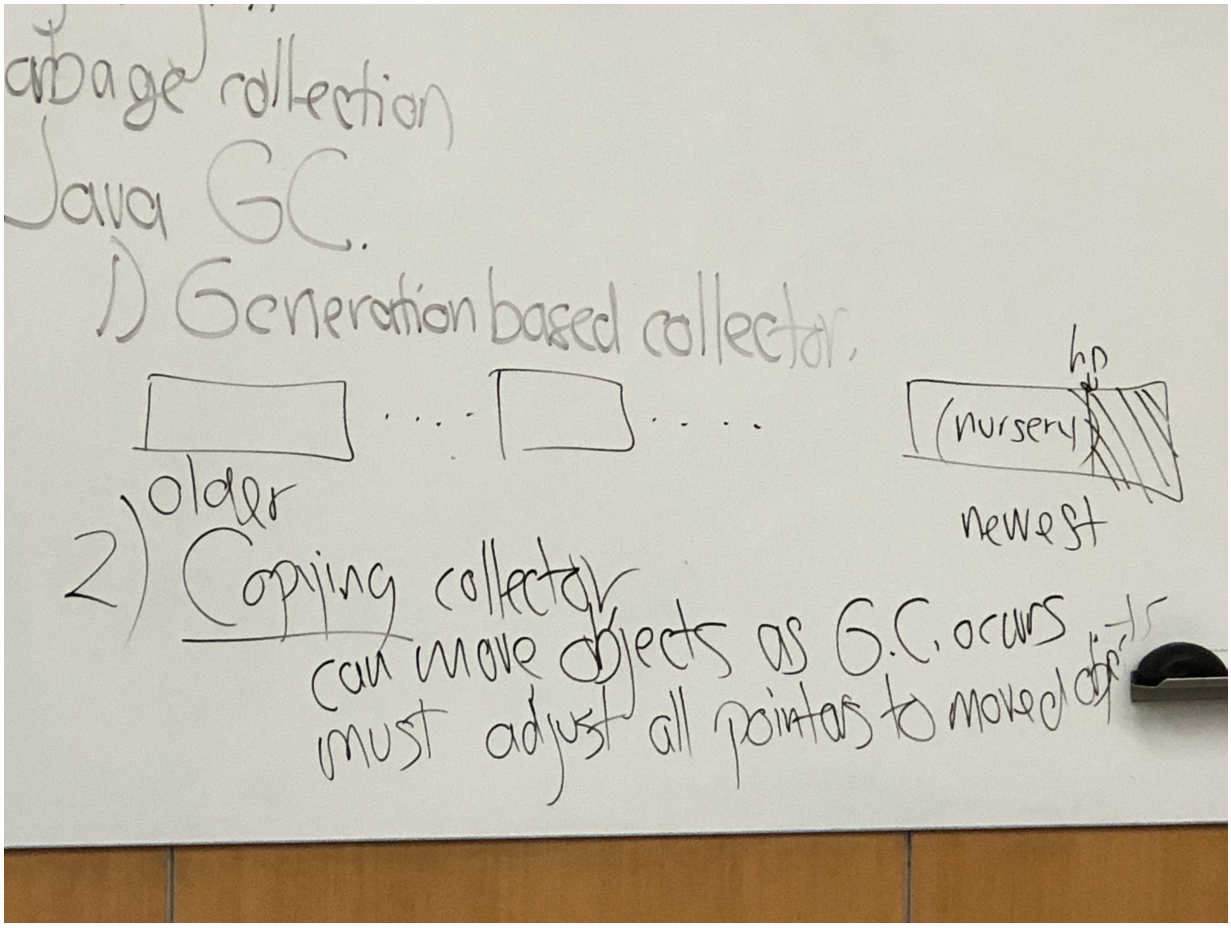


Week 9 Lecture 2

==========

Storage Management

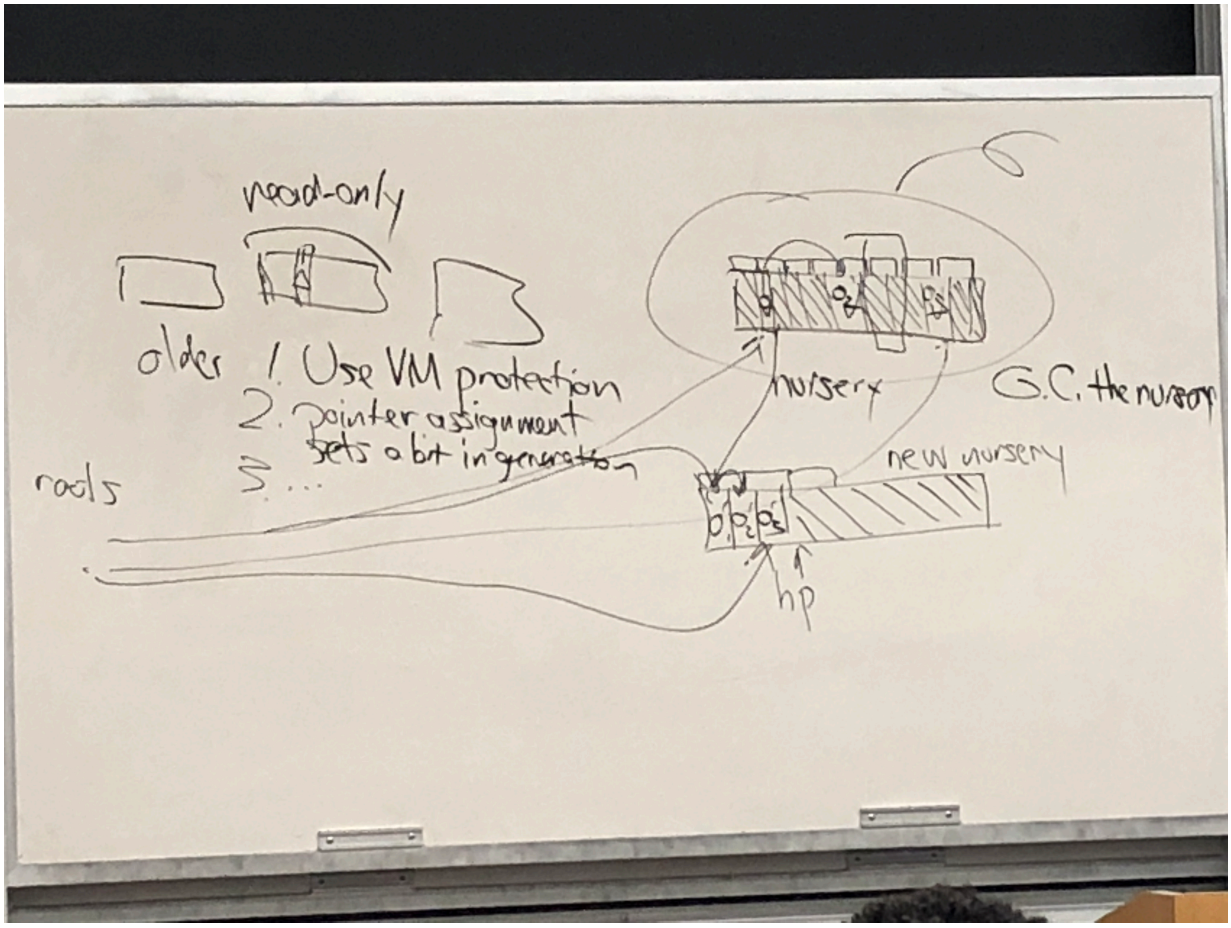
- Garbage Collection
  - Java GC (2 types)
    - 1) Generation-based collector



- 2) Copying collector
  - can move objects as GC occurs, must adjust all pointers to moved objects

Java Garbage Collection

- **Generation-Based Collector**
  - # proportional only to in-use objects, not total objects
- **Copying Collector**
  - cache lines are used more efficiently
  - a) in use object packed same cache
  - b) no need to touch free object - no free list
  - NOTE: Major performance benefits for copying collectors



Other Garbage Collection methods

- **multithreaded GC**
  - each thread gets its own nursery
- **real-time GC**
  - hard upper bound on new C() via incremented GC (it's tricky!)
  - hard since you have to continually make sure you keep track of things to garbage collect, while the program is being mutated (it's running)
- **GC in a different thread**
  - specific other thread performs garbage collection
  - PROBLEM: while finding out objects to garbage collect, the memory is changing

From outside sources

These new *generational collectors* had lots of improvements over the old stop-mark-sweep style:

- **GC throughput:** they could collect a lot more garbage a lot faster.
- **Allocation performance:** allocating new memory no longer required searching through the heap looking for a free slot, so allocation

became effectively free.

- **Program throughput:** allocations became neatly laid out in space next to each other, which improved cache utilisation significantly. Generational collectors do require the program to do some extra work as it runs, but that hit seems empirically to be outweighed by the improved cache effects.
- **Pause times:** most (but not all) pause times became much lower.

They also introduced some downsides:

- **Compatibility:** implementing a generational collector requires the ability to move things around in memory, and do extra work when the program writes to a pointer in some cases. This means the GC must be tightly integrated with the compiler. There are no generational collectors for C++.
- **Heap overhead:** these collectors work by copying allocations back and forth between various 'spaces'. Because there must be space to copy to, these collectors impose some heap overhead. Also, they require various pointer maps to be maintained (the *remembered sets*), further increasing overhead.
- **Pause distribution:** whilst many GC pauses were now very fast, some still required doing a full mark/sweep over the entire heap.
- **Tuning:** generational collectors introduce the notion of a "young generation" or "eden space", and program performance becomes quite sensitive to the sizing of this space.
- **Warmup time:** in response to the tuning issue, some collectors dynamically adapt the young generation size by observing how the program runs in practice, but now pause times depend on how long the program is running for as well. In practice this rarely matters outside of benchmarking

Python and GC

1. Some Python code runs atop Java VM!
2. CPython uses **reference counts**