Winter 2005 Lecture 27— Automatic Memory Management
What Every CS Student Should Know About Garbage Collection
From The Beginning...

• What is memory management and why do we need it?
• What errors does safe memory management prevent?
• What is “drag” and why is it undesirable?
• What safe approximation does GC make?
• What are some basic GC algorithms?
• Why are real GCs so much more complicated?
• Tricks for “programming against” a GC.
Why Memory Management?

Calling an ML constructor, Scheme’s cons, Smalltalk/Java’s new creates a new object. So does defining a nested function/block (see homework 5).

So non-trivial programs may run out of space if we do not reuse parts of memory (a really big array of bits). Even if you don’t run out, programs using compact space run faster.

The manual way (e.g., C):

• Reclaim space for local variables when execution leaves the function/block. (Callers cannot access these stack “objects”.)

• Reclaim other space (heap objects) when the programmer says to, e.g. free(x) or delete(x).
What Could Go Wrong?

Memory management is difficult because we want both:

- **No accessing reclaimed objects** (i.e., no “dangling-pointer dereferences”): If the space has been reused for another object, this will lead to crashes or silent data corruptions. Very expensive to detect at run-time.

- **No memory leaks**: If we do not reclaim enough, we may occupy much more space than we need.

If you could return a reference to the space occupied by a local variable, this could also lead to a dangling-pointer dereference.

The “traditional” definition of a memory-leak uses a key idea in memory management: *reachability*...
Reachability

Whether specified or not, most languages have a notion of reachability:

- Globals (top-level bindings / classes / static fields) are reachable.
- Local variables from function/method calls that haven’t returned are reachable (i.e, the stack is reachable).
- Any object referred to by something reachable is reachable.
- Nothing else is reachable.

Informally, it’s easy to imagine an algorithm to find what’s reachable:

- “Crawl the stack and globals” to get *roots*
- Keep recurring by following all fields of reachable objects
- Don’t recur on objects already seen (cycles)

In practice, crawling the stack and finding fields requires intimate knowledge of a language implementation.
Memory Leaks

In a language with manual memory management, a “memory leak” typically refers to “unreachable heap objects that have not been reclaimed”.

After all, they will never be reclaimed (no way to pass them to free).

But as we’ll see, a garbage-collector reclaims unreachable objects, so many people say “a language with GC cannot have memory leaks”.

While technically true with the right definitions, it’s misleading: For a broader view of “memory leak” (not enough reclaimed) it’s a lie!

Example: Store a huge data structure in a static field of a Java class. Never access that field again.

This is the extreme case of drag: The time between an object’s last access and its reclamation.
Space Leaks in GC’d Languages

Mostly, if an object is reachable, a GC won’t reclaim it.

- In practice, good systems can ignore some “stack roots” but few if any do anything smart for globals.

Options for the programmer:

- Ignore the problem; it usually doesn’t come up.
- Set fields to `null` when you’re done with them. (Problem: Back to manual management, but at least you get a `NullPointerException`)
- Take care not to let “permanent” data grow too big. (Potentially bad example: memoization tables.)
- Use a little-known language feature: “weak pointers”
Weak Pointers

- A weak pointer does not make pointed-to objects reachable.
- But following a weak pointer requires a run-time check.
- This may reclaim too much, but not too little.
- Modest slowdown to garbage collection.
How’s the magic work?

Production-quality GC’s are very sophisticated and use lots of tricks to:

- run fast
- reduce “pause times”
- make allocation fast (e.g., make contiguous areas of memory available)
- minimize fragmentation

Today we’ll just sketch the simplest versions of two basic approaches.

But first: why do “pause times” matter

- Soft deadlines: Humans don’t like “temporary freezes”
- Hard deadlines: Medical/air-traffic/nuclear equipment doesn’t like “I’ll handle that input when I’m done garbage-collecting”
(Semispace) Copying Collection

- Divide memory into two equal-size contiguous pieces.
- Allocate objects in one-space until it’s full (easy and fast).
- We now have a full from-space and an empty to-space.
- Copy the reachable objects into to-space.
- Restart the “real program” (called the mutator), allocating into the partially full to-space.
- The old from-space is empty—it’s the new to-space.

Note: The GC uses “header words” (e.g., class pointers) to figure out where the fields pointing to other objects are.
Wait A Minute

We skimmed over two very important details!

- We *moved* objects; that means we better *change* any references to those objects too!

- Our recursive procedure for copying reachable objects better not use space we don’t have! (GC during GC not an option.)

Solutions:

- A *Cheney* queue: Two pointers into to-space all we need to keep track of what needs to be recursively traversed.

- Forwarding pointers: We can use space in the old objects to record where they moved to. (Use to update fields and not follow cycles.)
Mark-Sweep Collection

- Allocate objects until you (almost) fill the space you have.
- Mark: Starting from the roots, find all reachable objects. Mark them (set a bit in the header word). Don’t recur on already-marked objects.
- Sweep: Scan through memory. If an object is unmarked, reclaim it. Otherwise, unset the bit (or next GC can’t reclaim it).

Note:

- We don’t need 2x more space
- No objects move, no fields get changed.
Wait Another Minute

- In practice, if more than about 2/3 of memory ends up marked, you’ll GC too often (slow program).
- Allocation isn’t nearly as simple:
  - We need to find some space big enough for the object.
  - Can make “free lists”, but want to “segregate them by size”
  - *Fragmentation* can lead to memory exhaustion before a copying collector would.
- Our recursive procedure for copying reachable objects better not use space we don’t have! (And a Cheney queue won’t work.)
  - Can use some auxiliary space to remember “objects to recur on” and pull clever tricks if this space fills up.
  - Can use really clever “Deutsch-Schorr-Waite” algorithm to “reverse” pointers temporarily while recurring.
To Learn More

An excellent survey paper:


Available at:

http://www.cs.utexas.edu/users/oops/papers.html