CSE 341:
Programming Languages

Spring 2006

Lecture 29 — 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?

Why Memory Management?

ML constructors, Scheme's `cons`, Smalltalk/Java's `new`, defining nested functions/blocks create new objects.

Most objects are short-lived.

May *run out of space* if we do not *reuse* parts of memory.

Even if not, programs using *compact* space run faster.

The manual way (e.g., C/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)`.

- Problems

  - free/delete is tedious

  - and error-prone

  - and sometimes cuts across natural module boundaries (Joe creates it, Harry & Sally both use it, who frees it?)

What Could Go Wrong?

Memory management is difficult because we want both:

- *No accesses to 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.

  Examples: dereference after free, or returning a reference to a local variable.

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

Both typically very hard to debug.

Memory Leaks

With manual memory management, a "memory leak" means "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:

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

GC won't reclaim a reachable object (with some exceptions, if your compiler does some fancy analysis/optimization).

Options for the programmer:

- Ignore the problem; it "usually" doesn't come up.

- Set fields to `null` when you're done with them. (Back to manual management, but at least you get a `NullPointerException` in place of silent dangling dereferences)

- Take care not to let "permanent" data grow too big. (Potentially bad example: memoization tables.)

Reachability - The Key to Garbage

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

- All *Roots* are reachable:

  - Globals (top-level bindings / classes / static fields) are roots

  - Local variables from function/method calls that haven't returned (i.e., stack contents) are roots

- 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:

- Examine all roots by crawling stack & globals (next slide)

- Graph Reachability: Recursively follow all fields of reachable objects, without revisiting objects already seen (cycles & cross-links). E.g., BFS or DFS. (next^2 slide)

Crawling amidst the garbage is a dirty business

In practice, crawling the stack and finding fields requires *intimate* knowledge of a language implementation, and possibly deep integration with the compiler.

- Where's the top of the stack?

- Which procedure is that?

- What are its locals and where are they?

- Who called it? What's the layout of *its* stack frame? ...

- Given a heap object, what's its type? How big is it? Which of its fields are other heap pointers? ... (E.g., use "header words," such as class pointers, to locate the fields pointing to other objects.)

Graph Reachability: BFS/DFS
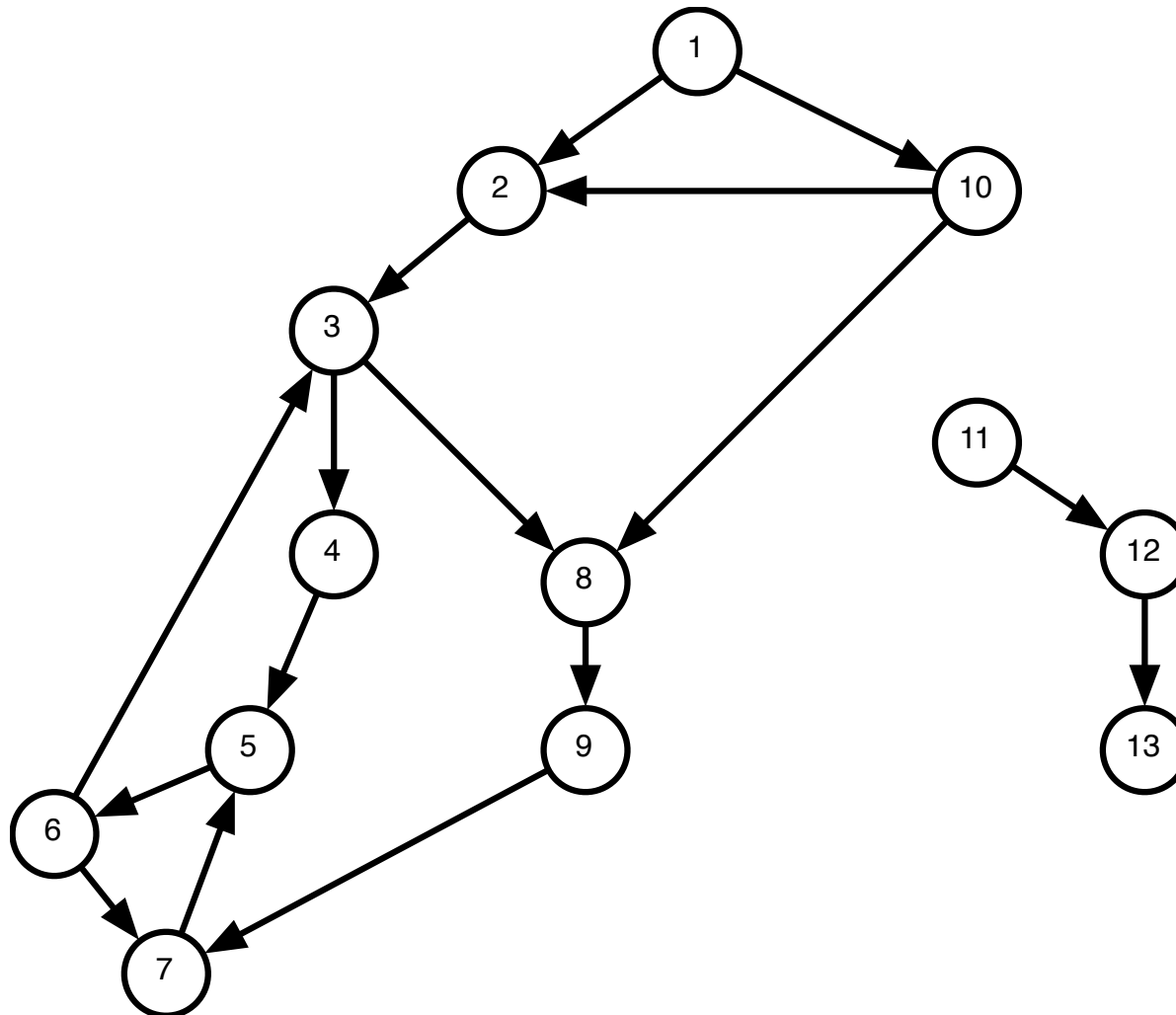
How's the magic work?

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

- run fast

- make allocation fast (e.g., make contiguous areas of memory available)

- minimize fragmentation

- maximize locality

- reduce "pause times." Why?

  - Soft deadlines: Humans don't like "temporary freezes"

  - Hard deadlines: "Your red-button-push is important to us. Please be patient while we collect some garbage, then I'll begin the emergency reactor core shut-down you requested."

Today: sketch the simplest versions of four basic approaches.

Approach 0: Reference Counts

Every object holds a *reference count* = total number of pointers to that object

Ref count incremented/decremented on every change to any pointer

Garbage when ref count hits 0

Pro:

- Fast & simple

Con:

- Requires discipline and/or compiler support to maintain counts

- Fails for circular structures

Approach 1: (Semispace) Copying Collector

- 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.

- Continue until to-space is full.

- The old from-space is empty—it will be the new to-space.

Important Details

We skimmed over two very important details!

- We *moved* objects; that means we must *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. (BFS)

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

Approach 2: Mark-and-Sweep

- 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 revisit 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).

Another Set Of Important Details

- Recall the Dirty Business slide—during sweep, need to be able to find objects in memory, i.e., not by following pointers.

- Our recursive procedure for marking reachable objects better not use space we don't have! (And a Cheney queue won't work.)

  - Use auxiliary space to remember stack or queue of "unprocessed objects" and pull clever tricks if this space fills.

  - Or use really clever "Deutsch-Schorr-Waite" algorithm to "reverse" pointers temporarily while recurring.

- 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," perhaps merging adjacent free elements.

Some Pros and Cons

- No objects move, no fields get changed.

- We don't need 2x more space

On the other hand:

- In practice, if more than about 2/3 of memory ends up marked, you'll GC too often (slow program).

- doubling size of *virtual* memory is usually cheap

- *Fragmentation* can lead to memory exhaustion before a copying collector would.

- Locality may also suffer

Approach 3: Generational collectors

Distribution of object lifetimes is far from uniform. By various estimates, 80+% of objects become garbage very quickly, while the long-lived objects constitute the bulk of the non-garbage.

E.g., copying collectors copy the same objects again and again.

An idea: Have two or more separate areas for young, old, antique ... objects. Use separate semi-space collectors in each area. Collect young space most frequently.

Key problem: need to find/update all old $\longrightarrow$ young pointers when young is collected/compacted, without crawling old space every time.

Solutions: use an indirection table and/or "write barriers" (trap all assignments to pointers in old objects to check whether they point to new ones).

To Learn More

An excellent survey paper:

Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In International Workshop on Memory Management, St. Malo, France, September 1992

Available at:

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