What to Expect from Your Garbage Collector

Hans–J. Boehm

SiliconGraphics
Computer Systems

Disclaimers:

Many things are grossly oversimplified, but hopefully not misleading.

Opinions expressed here are personal and occasionally controversial. They may not reflect SGI's position.

Some of the names used here are Trademarks.

Infoworld, Mar 12 1998:

"On your marks, get set, compile; the race to enhance Java heats up"

"[Sun's HotSpot technology] also introduces a new method of garbage collection, called Train, which is designed to remove any unwanted software objects from a Java environment.

While most garbage–collection algorithms are based on the "Bone Collector," Train is more effective, according to Sun officials, because it is interruptible. So when Virtual Machine is busy it won't waste time house cleaning."

Clarifications:

1) This talk is not about collecting bones.

2) Even early JVMs use a variety of garbage collection algorithms.

For more corrections, stay tuned ...

Manual Memory Management:

malloc/free, new/delete, etc.

- programmer must determine when memory is unneeded and return it to system.

- performance is fairly transparent

- error prone, esp. with complex data structures
- symptoms unrelated to bugs
- complicates interfaces
- added overhead to keep objects uniquely referenced

Automatic garbage collection (GC):

Java, Smalltalk, C/C++ libraries, ...

- **garbage collector identifies and reclaims inaccessible memory**
- **programmer does not need to worry about deallocation**
- **performance is different from manual memory management**

Garbage Collector Taxonomy

- Reference counting
- Tracing collectors – traverse live objects
  - Marking collectors
    - Nonmoving Marking Collectors
    - (Compacting Mark and Sweep)
  - Copying collectors
  - (Hybrids)

Reference Counting:

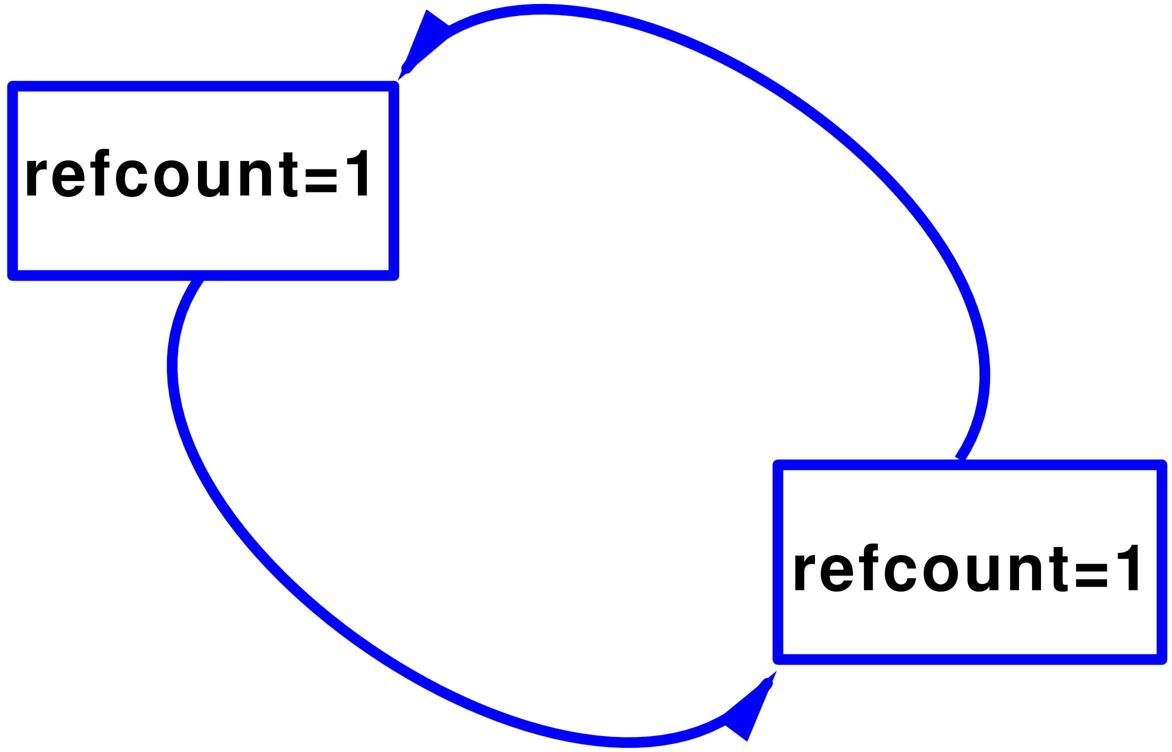
- Conventional nonmoving allocator.
- Each object maintains a count of the number of pointers that refer to it.
- Object can be deallocated when count reaches 0.
- In C++, can almost be hidden in a portable.

– pointer assignment `p = q` becomes

```
if (q != NULL)
   atomically{++(q -> refcount);}
if (p != NULL) {
   size_t new_count;
   atomically{
        new_count = --(p -> refcount);
   }
   if (new_count == 0)
        dealloc_decr_counts(p);
}
p = q;
```

slow!

– Cycles of garbage:



Reference count evaluation:

- Small space overhead.
  Good for some embedded apps.
- Good for large objects. (Stay tuned.)

- Pointer assignments and pointer parameters are expensive.
- Nonallocating code is slowed down.
- Needs atomic increment.
- Leaks cycles.

Reference count use:

- Fairly common in portable C++ apps.
- Uncommon in language runtimes. Notable exception: Lucent's Inferno.
- Xerox Cedar, DEC Modula implementations moved to tracing collector.
- Ignored from here on.

Nonmoving Marking GC

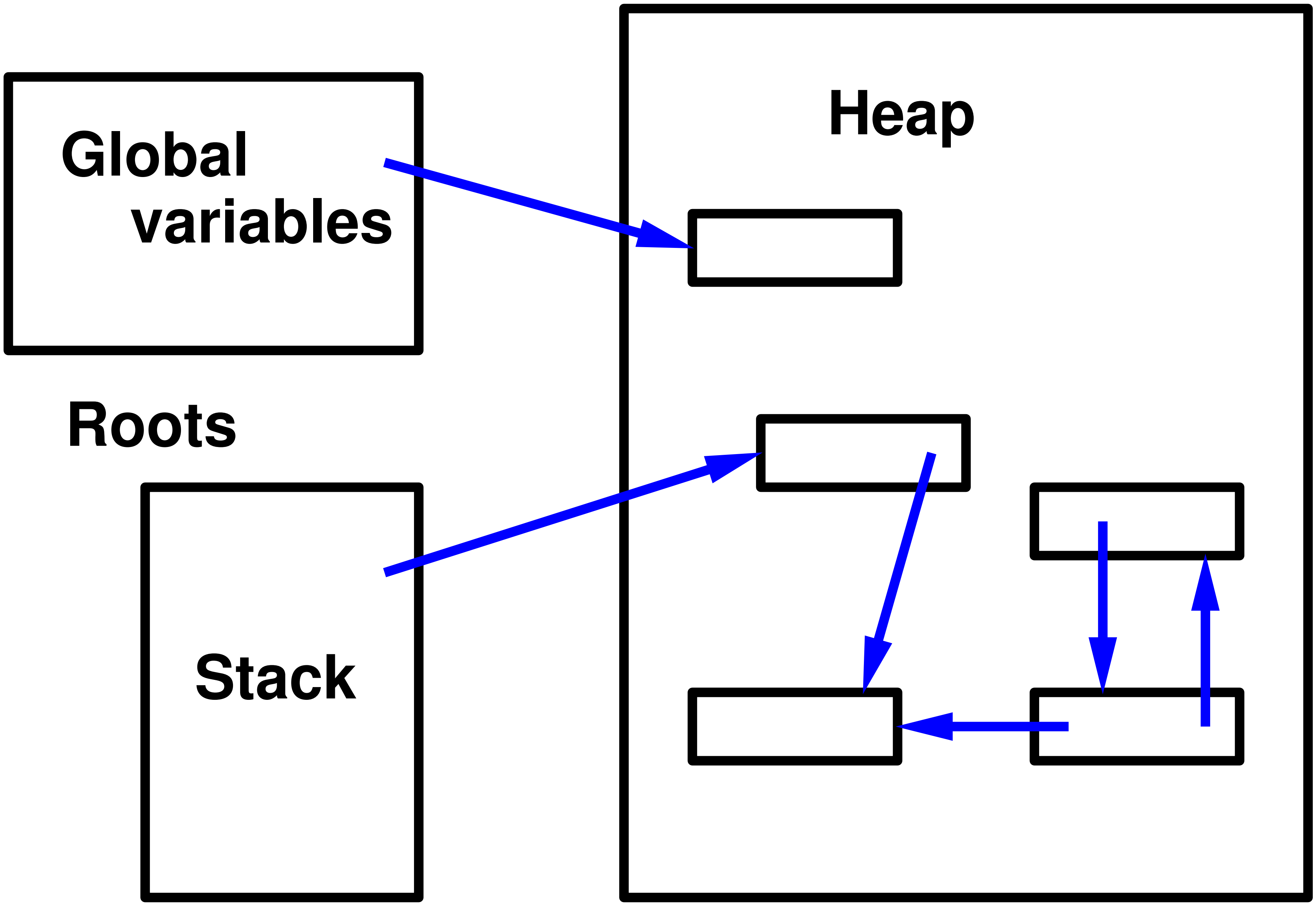
- Can use a mostly conventional allocator. (In practice it's tuned to the collector.)

- Occasionally look for objects that can't possibly be accessed. Reuse those.

- Specific variants:
  - Mark–Sweep (bad implementation)
  - Treadmill

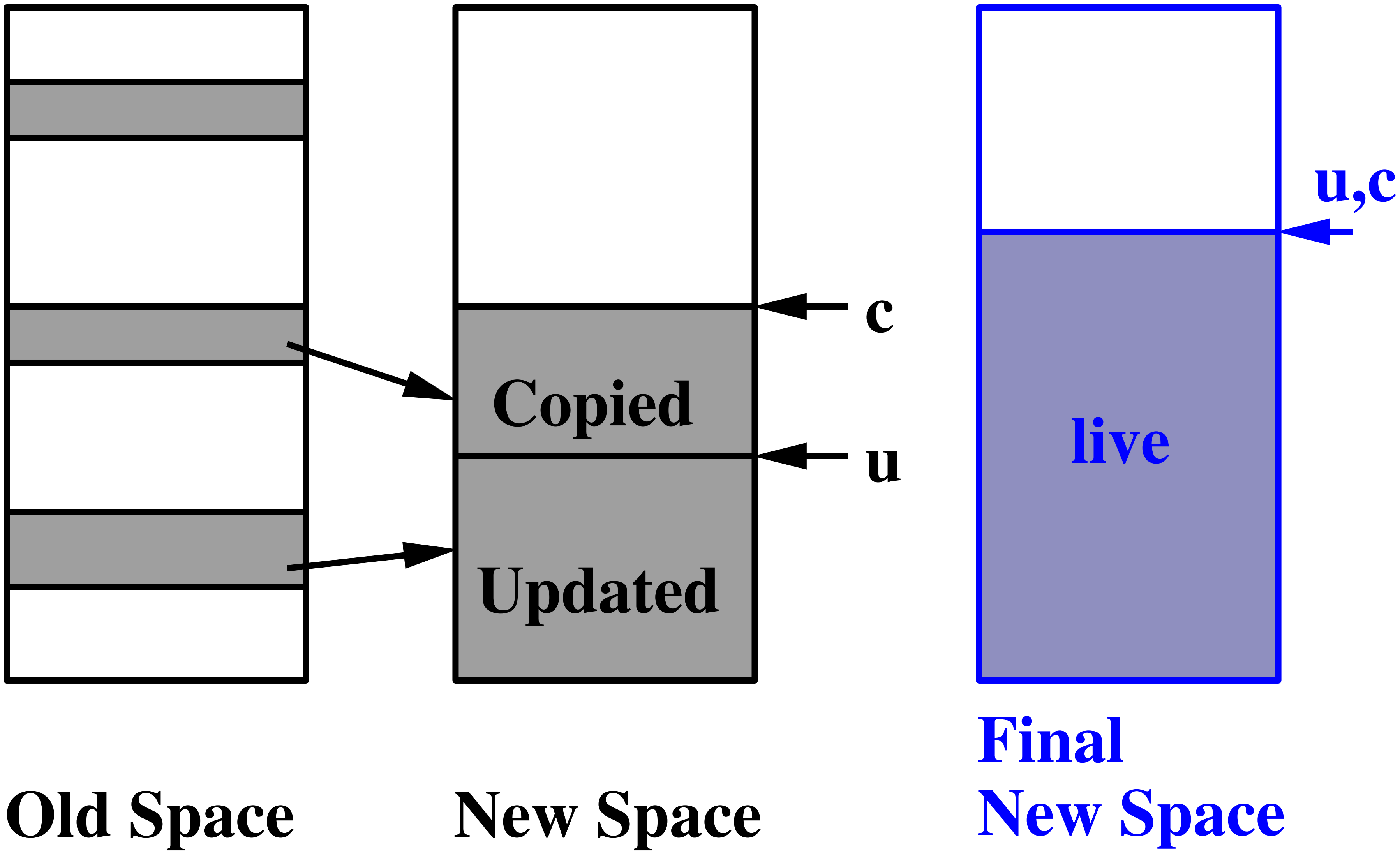
Algorithm:

1. Mark all objects referenced by pointer variables (*the roots*).
2. Mark all objects referenced by newly marked objects.
3. Repeat step 2 until no more objects are marked. (Typically a stack of newly marked objects is maintained.)
4. The allocator reuses unmarked objects.


Heap
Global
variables
Roots
Stack

Copying collection (Cheney)

- Compacts: free space is contiguous
- Can use bump–the–pointer allocator
- Memory is divided into 2 spaces.
  Only one is active at a time.
- Garbage collector copies live memory from one (old space) to another (new space) which then becomes active.



Copied
Updated
c
u
u,c
live
Old Space
New Space
Final
New Space

Performance characteristics:

Total allocation/deallocation time is

GC:

<time in allocator>
\+ <GC time>/<allocations per GC>

malloc/free:

<time in allocator (`malloc`)>
\+ <time in `free`>

Time in allocator:

- Largely independent of object size in all cases.

- Small for compacting (*e.g.* copying) GC.

- Marking noncompacting GC comparable to `malloc`:
  - + Allocate in address order ==> better locality (allocator & client)
  - – mark–bit handling (sweep)

<GC time>/<allocations per GC>:

GC time is proportional to amount of reachable memory.
Larger Heap requirements ==> longer GC times.
Larger Heap requirements ==> more allocations per GC.

Allocatable heap size should be (at least) a constant factor (*e.g.* 1.5) * live data.
Disclaimers: Many JVM implementors initially got this wrong.
Paging matters.

<GC time>/<allocations per GC> contd:

Assume heap = 1.5 * live,
steady state.

For each GC cycle:

- 1/3 heap is allocated.
- 2/3 heap is scanned by collector.

Each byte of allocation costs 2 bytes of scanning/copying.

Allocation cost is proportional to allocated object size.

Malloc vs. GC (again):

`malloc`: Total allocation cost (mostly) independent of size.

GC: Allocation cost proportional to size.

either: Initialization time usually proportional to size.

Small objects, lots of available heap space favor GC.

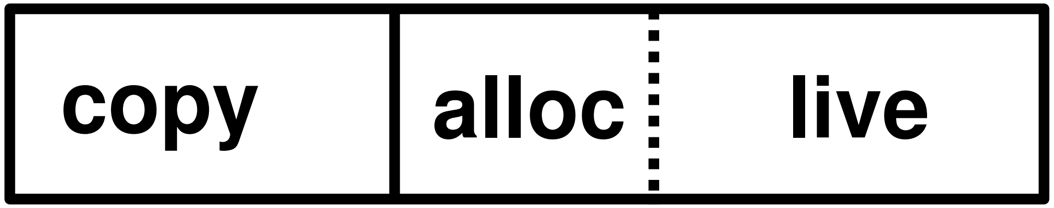
2000 reversals of 1000 element list, (133Mhz R4600 Indy)

malloc, 8 byte nodes	**8.6, 4.4***
gc, 8 byte nodes	**7.3**
gc, 8 byte nodes, 2MB heap	**6.6**
malloc, 100 byte nodes	**14.1, 6.9***
gc, 100 byte nodes	**31.3**
gc, 100 byte nodes, 2MB heap	**16.7**

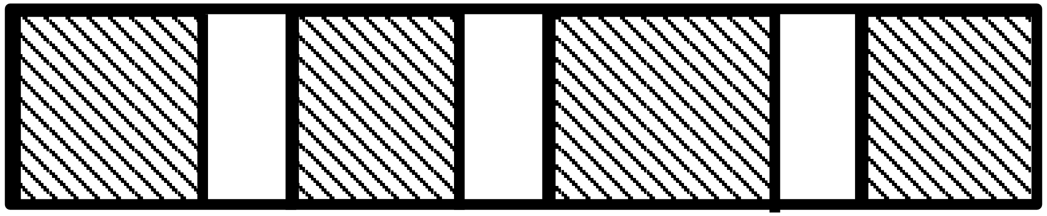
*** inlined malloc**

Copying vs. Marking

– Copying needs extra space to copy to, e.g. 2.5*live vs 1.5*live.



– Marking (w/o compaction) risks fragmentation. 3 chunks of 1MB can't be used for a 2 MB object.



Copying vs. Marking contd.:

- Usually marking requires significantly less space. But copying has better worst–case space behavior.

- Copying allows faster allocation. (Not important here. But stay tuned.)

- Marking and scanning objects is usually faster than copying.

- Marking does not require touching pointerfree objects.

Conservative GC:

- Copying collectors need to know exact pointer locations, since pointer values are changed.

- Treating an integer as a pointer in a marking collector is mostly benign; extra objects are marked and retained.

Marking collectors can be (and are) used with conventional C/C++ code (e.g. Geodesic’s Great Circle, Boehm/Demers/Weiser collector).
"If the bits look like a pointer, treat it like one."

Partially conservative marking collectors allow for better language interoperability.
"Use what type information you have."

Variations

Generational collection:
Young objects treated differently

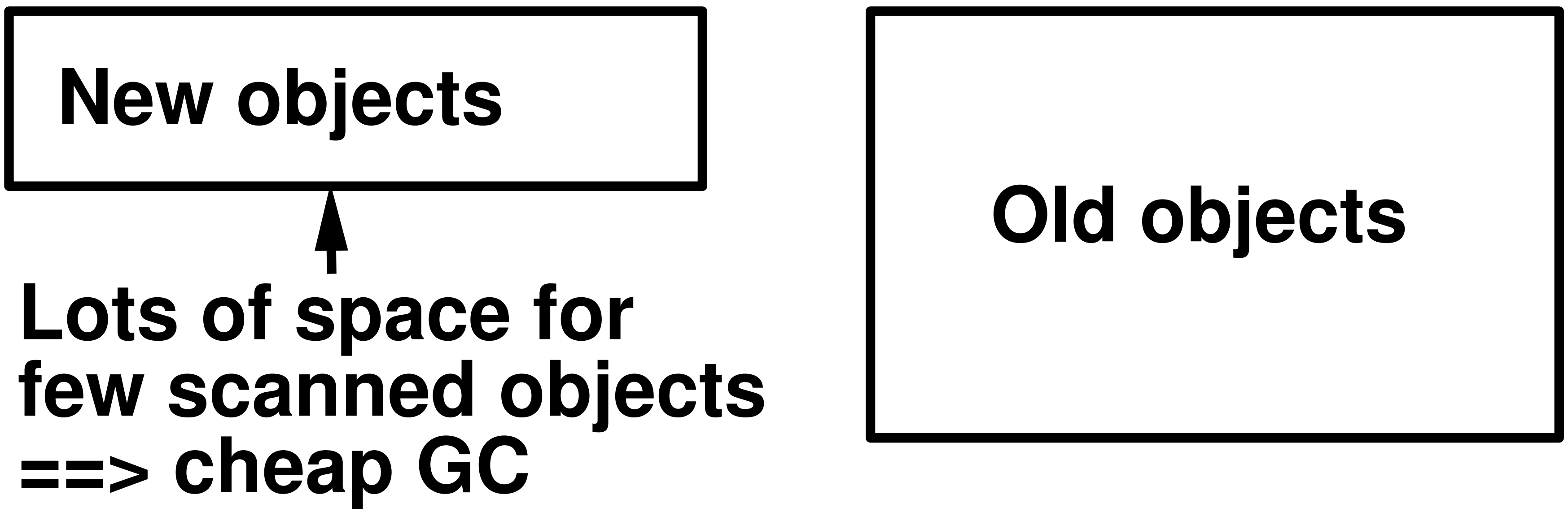
Incremental collection:
Reduce GC pauses

(Opportunistic collection:
Collect during idle time)

Generational Garbage Collection:

Observation:

Often many objects are very short–lived.
Life expectancy of newly allocated objects is often much shorter than those that have survived collections.



Generational collection contd:

Can use copying GC for young objects in modest space ==> cheap short–lived allocation.

Many collections are quick and touch little of the heap.

Need to track pointers from old to new ==> overhead, often for allocation free code.

Doesn’t help during construction of large data structures.

Generational collection, summary:

- Short–lived allocation much more efficient.
- Great for some programs (Java?).
  No help for others (C/C++).
- Appearing in some new Java VMs.
- Still need occasional full collections:
  Not a replacement for incremental GC

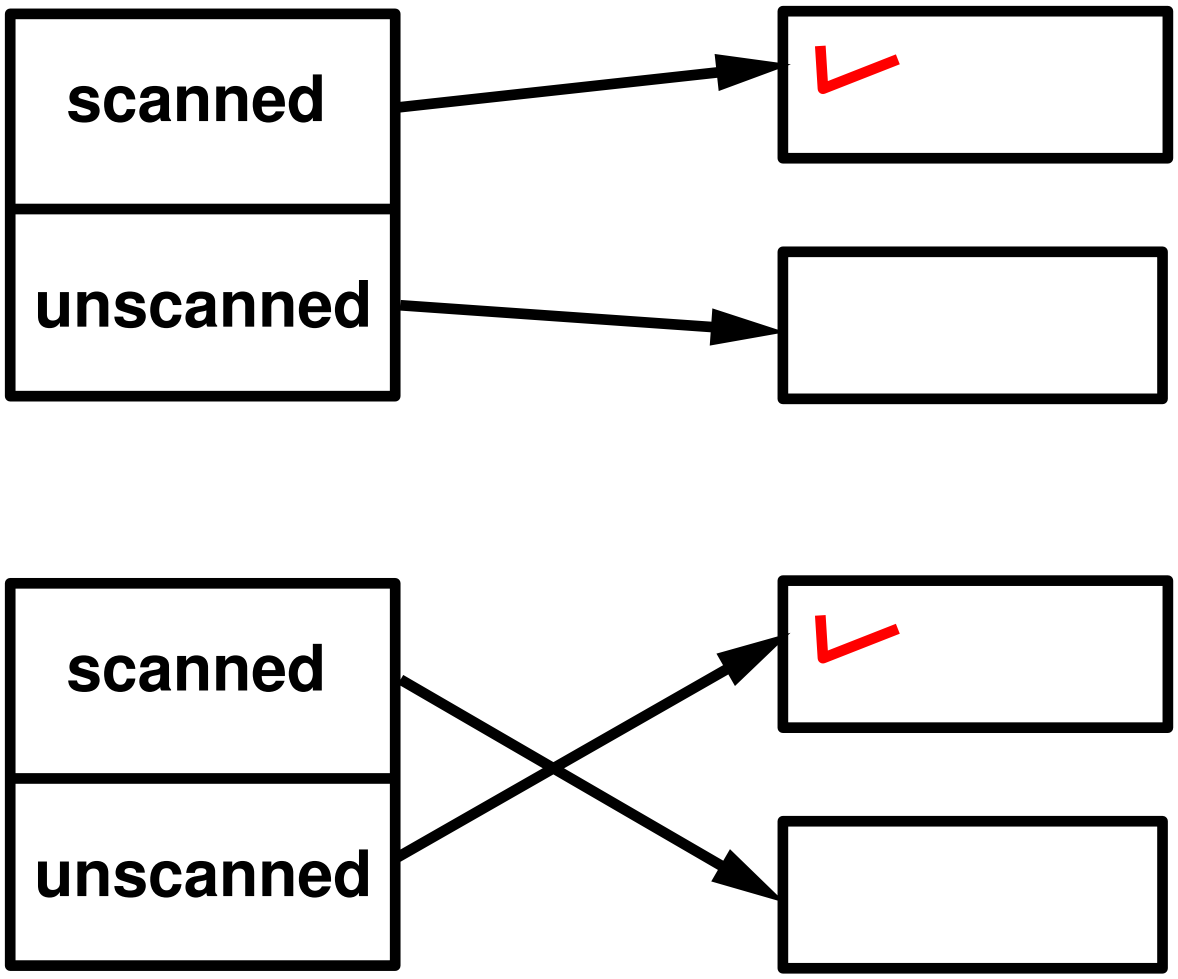
Incremental GC:

Problems:

- For large heaps (10s of MB), GC may take substantial time.

- Simple GCs don't allow user code to run during GC.

    - Objects might move (copying GC)
    - Objects may falsely appear to be unreachable.


scanned
unscanned
scanned
unscanned

Incremental collection contd:

- Want either
  1. A little bit of GC work during each allocation, or
  2. (less robust?) a separate GC thread running concurrently with client.

- This is different from opportunistic abortable collection (*e.g.* Sun JDK 1.1), which may help if there are idle periods.

Incremental collection contd:

- Many known solutions.
- Some are hard real–time.
- Require at least one of:
  - Logical heap snapshot
  - Tracking heap updates for fixup
  - Tracking heap pointers between regions
- Often slows down non–allocating code
- Increases space usage
- Easier for nonmoving collectors
- Possible for copying collectors (Train)

Demo:

"Slow motion"
(486SX20, 8MB, GC compiled –g)
Boehm/Demers/Weiser collector

`http://reality.sgi.com/boehm/gc.html`

Set heap to 3 MB (12);
Allocate 1.6 MB (6.4) in 100 byte nodes;
Keep it accessible;
For each iteration:
- draw frame (code from Jeff Eppler);
- allocate and drop 12K;