HARVARD
School of Engineering and Applied Sciences

Dynamic Memory Allocation

CS61, Lecture 11

Prof. Stephen Chong

October 6, 2011

Announcements 1/2

- Reminder: No section on Monday
  - Monday sections have been rescheduled
  - See website for details
  - Please attend whichever section you can make
- Homework 3 (Buffer bomb) due today
- Homework 4 (malloc) released
  - Design checkpoint due Thursday 13 October, 10pm
  - Final submission due Thursday 20 October, 11:59pm

Announcements 2/2

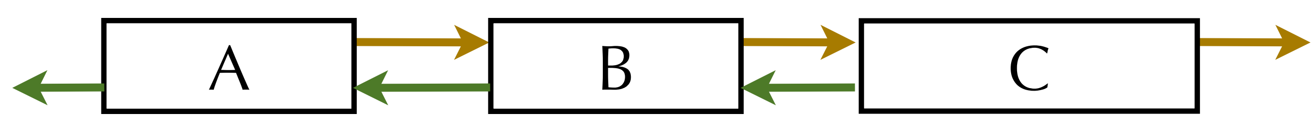
- Interested in concentrating in CS?
  - Don't wait until last minute!
    - Sophomores must declare concentration by Nov 16
  - Prof. Harry Lewis is the Computer Science DUS
    - Office hours posted on his website
    - He is happy to talk about the concentration, or sign you up!
  - Fill in departmental study plan
    - http://studyplan.seas.harvard.edu/ug/
    - Going through exercise will help you understand requirements
    - Can always revise it later
  - To actually enter concentration, need to fill out a form on Registrar's website, and get Prof. Lewis' signature

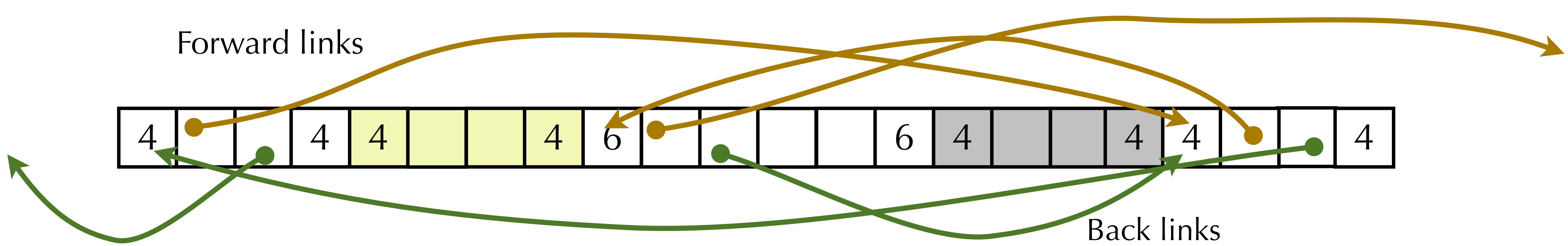
Today

- Free block list management
  - Implicit free list
  - Explicit free list
  - Segregated lists
  - Tradeoffs
- Alignment issues
- Example allocator implementation: `dlmalloc`
- Common memory problems
- Garbage collection
  - Mark and sweep
  - Generational GC
  - Reference counting

Explicit Free Lists

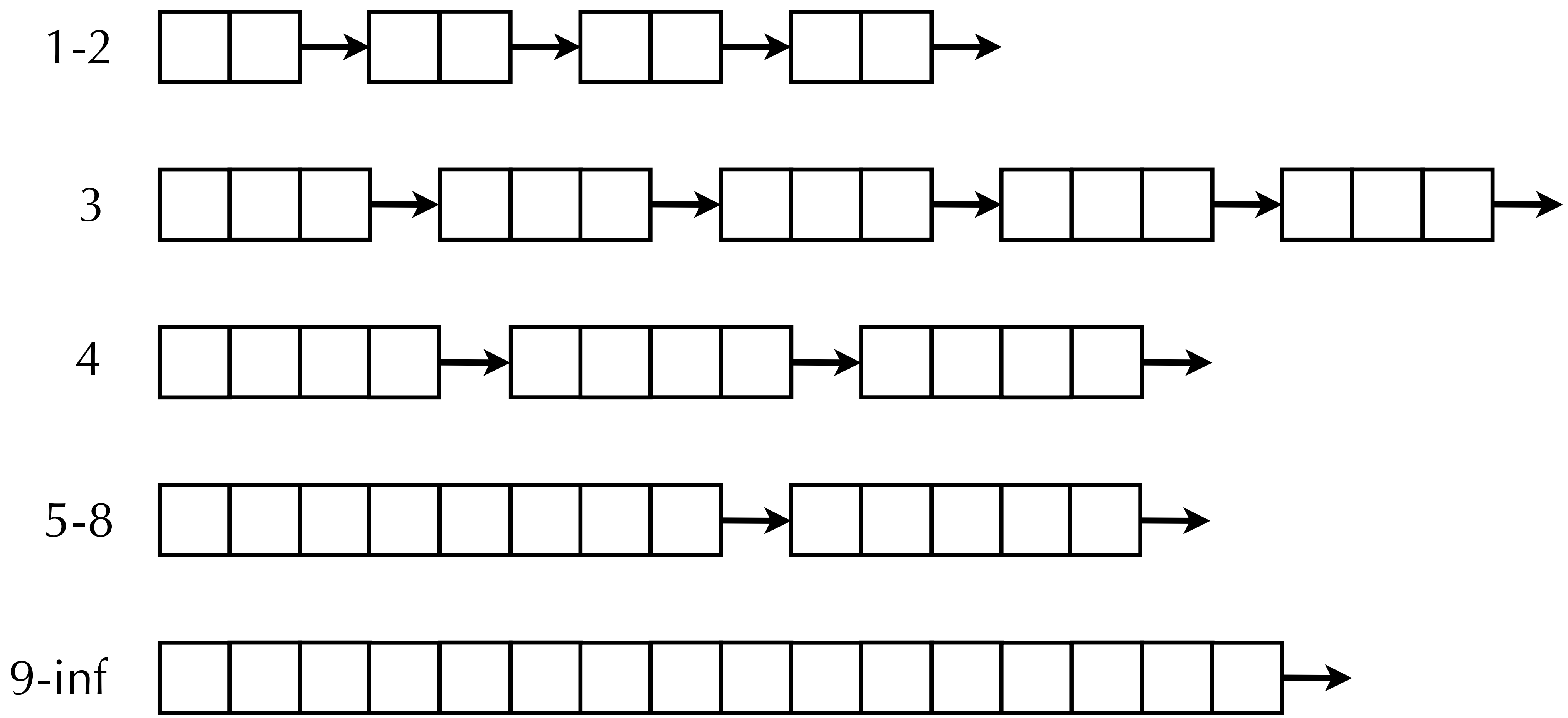
- Use an explicit data structure to track the free blocks
  - In this case, a doubly-linked free list
  - Pointers for list just live in the payload
    - Payload is unused for free blocks!





Segregated List (seglist) Allocators

- Use a different free list for blocks of different sizes!



  - Often have separate size class for every small size (4,5,6,…)
  - For larger sizes typically have a size class for each power of 2

Seglist Allocator

- To allocate a block of size *n*:
  - Determine correct free list to use
  - Search that free list for block of size $m \geq n$
  - If an appropriate block is found:
    - Split block and place fragment on appropriate list
  - If no block is found, try next larger class
  - Repeat until block is found
- If no free block is found:
  - Request additional heap memory from OS (using `sbrk()` system call)
  - Allocate block of *n* bytes from this new memory
  - Place remainder as a single free block in largest size class.

Freeing with Seglist

- To free a block:
  - Mark block as free
  - Coalesce (if needed)
  - Place free block on appropriate sized list

Seglist advantages

- Advantages of seglist allocators
  - Higher throughput
    - Faster to find appropriate sized block: Look in the right list.
  - Better memory utilization
    - First-fit search of segregated free list approximates a best-fit search of entire heap.
    - Extreme case: Giving each size its own segregated list is equivalent to best-fit.

Allocation Policy Tradeoffs

- Data structure for free lists
  - Implicit lists, explicit lists, segregated lists
  - Other structures possible, e.g.explicit free blocks in binary tree, sorted by size
- Placement policy: First fit, next fit, or best fit
  - Best fit has higher overhead, but less fragmentation.
- Splitting policy: When do we split free blocks?
  - Splitting leads to more internal fragmentation, since each block needs its own header.
- Coalescing policy: When do we coalesce free blocks?
  - **Immediate coalescing**: Coalesce each time free is called
  - **Deferred coalescing**: Improve free performance by deferring coalescing until needed.
    - E.g., While scanning the free list for `malloc()`, or when external fragmentation reaches some threshold.

Today

- Free block list management
  - Implicit free list
  - Explicit free list
  - Segregated lists
  - Tradeoffs
- Alignment issues
- Example allocator implementation: `dlmalloc`
- Common memory problems
- Garbage collection
  - Mark and sweep
  - Generational GC
  - Reference counting

Alignment issues

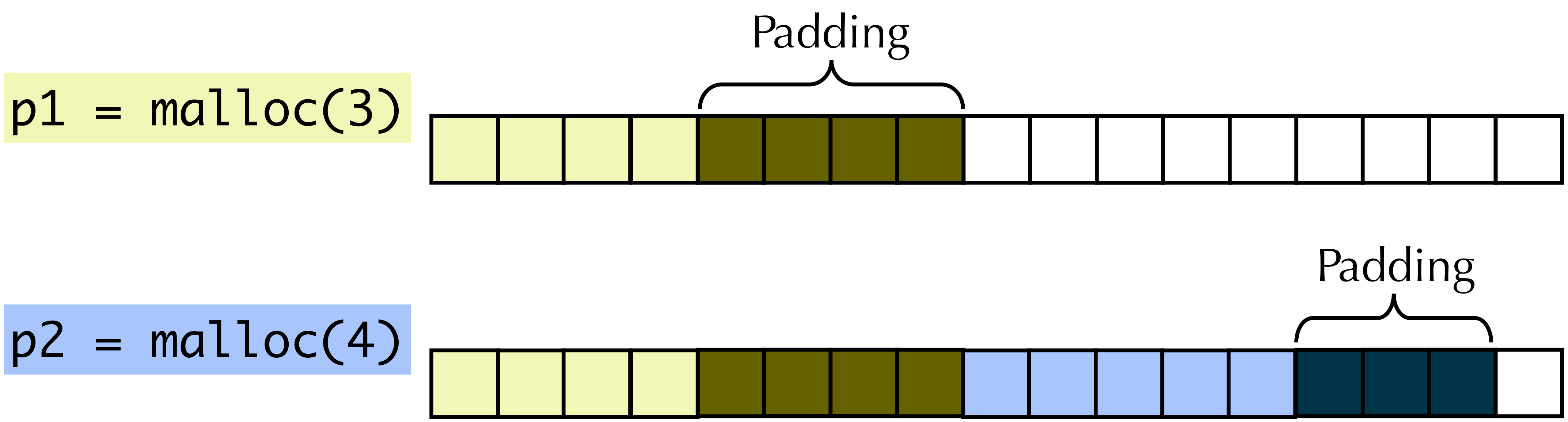
- Most `malloc()` implementations ensure that the returned pointer is **aligned** to an 8-byte boundary.
  - This is to ensure that if the pointer is used to store a struct, it will be properly aligned.

```
struct mystruct *foo;
void *p;

p = malloc(sizeof(struct mystruct));
foo = (struct mystruct *)p;
```

Alignment issues

- Implication: malloc() may have to pad the block that it allocates



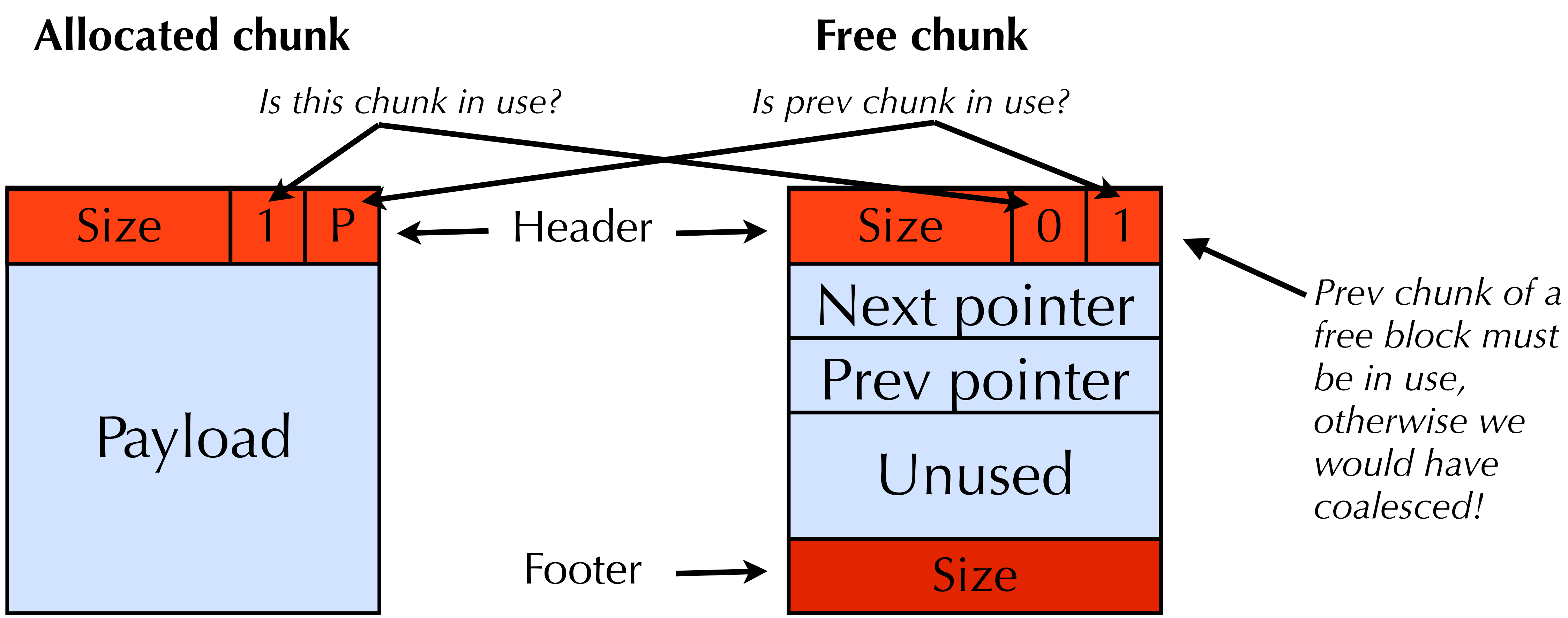
- Padding can often be "recycled" as header, boundary tags, etc.
  - (Not shown in the above example.)
  - Key is that the **payload of the buffer is appropriately aligned**.

Doug Lea allocator

- `dlmalloc`
  - Fast and efficient general purpose allocator
    - Basis of `glibc` allocator
  - Since 1992
- Essentially best-fit
  - Ties are broken in least-recently-used order
    - Reduces fragmentation
  - Deviates for requests less than 256 bytes
    - Prefer space adjacent to previous small request
    - Break ties in most-recently-used order
      - Increase locality of series of small allocations
- All operations bounded by constant factor (in # bits of `size_t`)

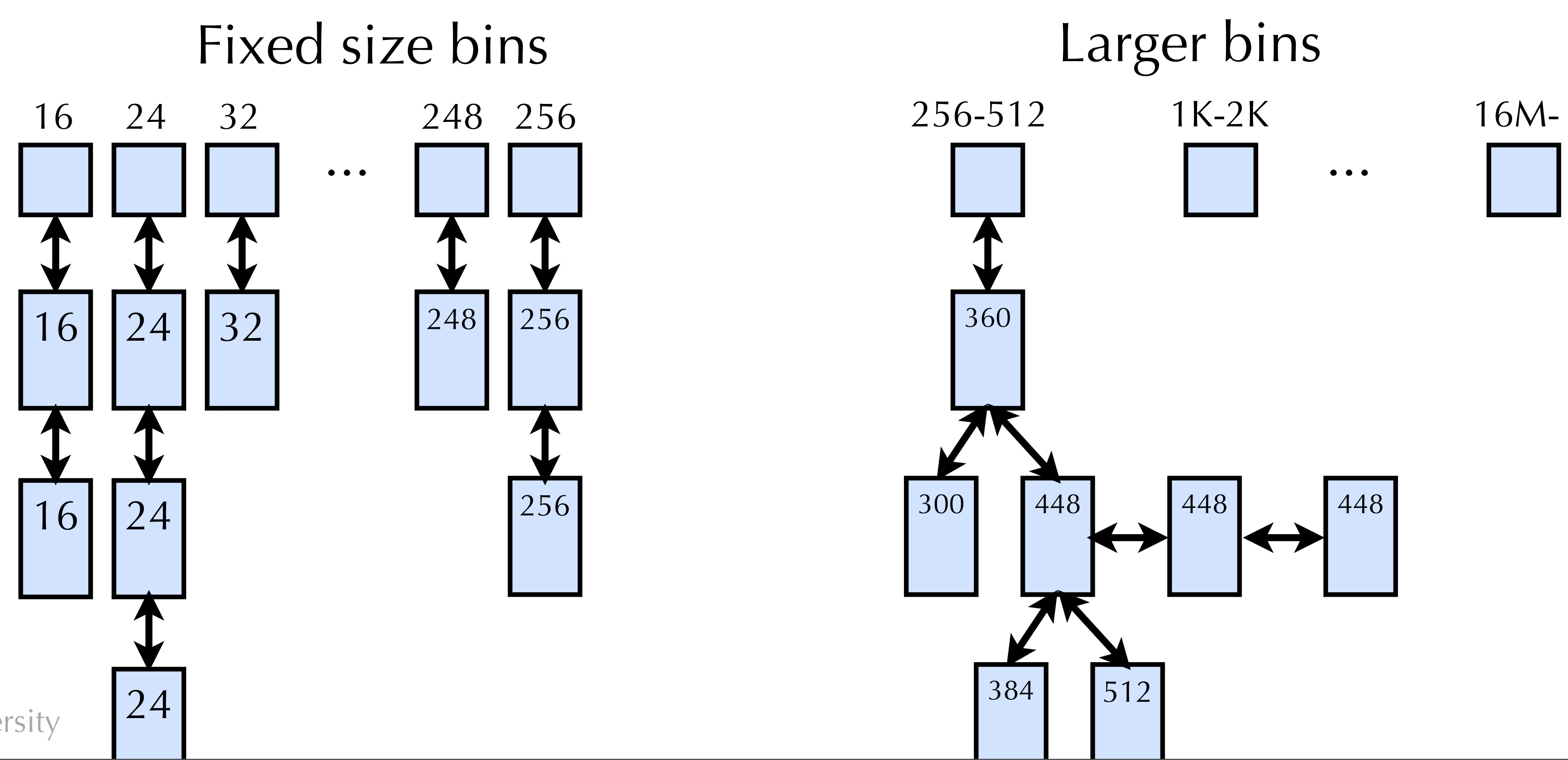
Doug Lea allocator

- Uses blocks (aka "chunks") with boundary tags
  - 8 byte alignment
  - Free chunks have header and footer
  - Allocated chunks have only header
  - A chunk is at least 16 bytes (4 for header + 4 for footer + 8 byte alignment requirement)



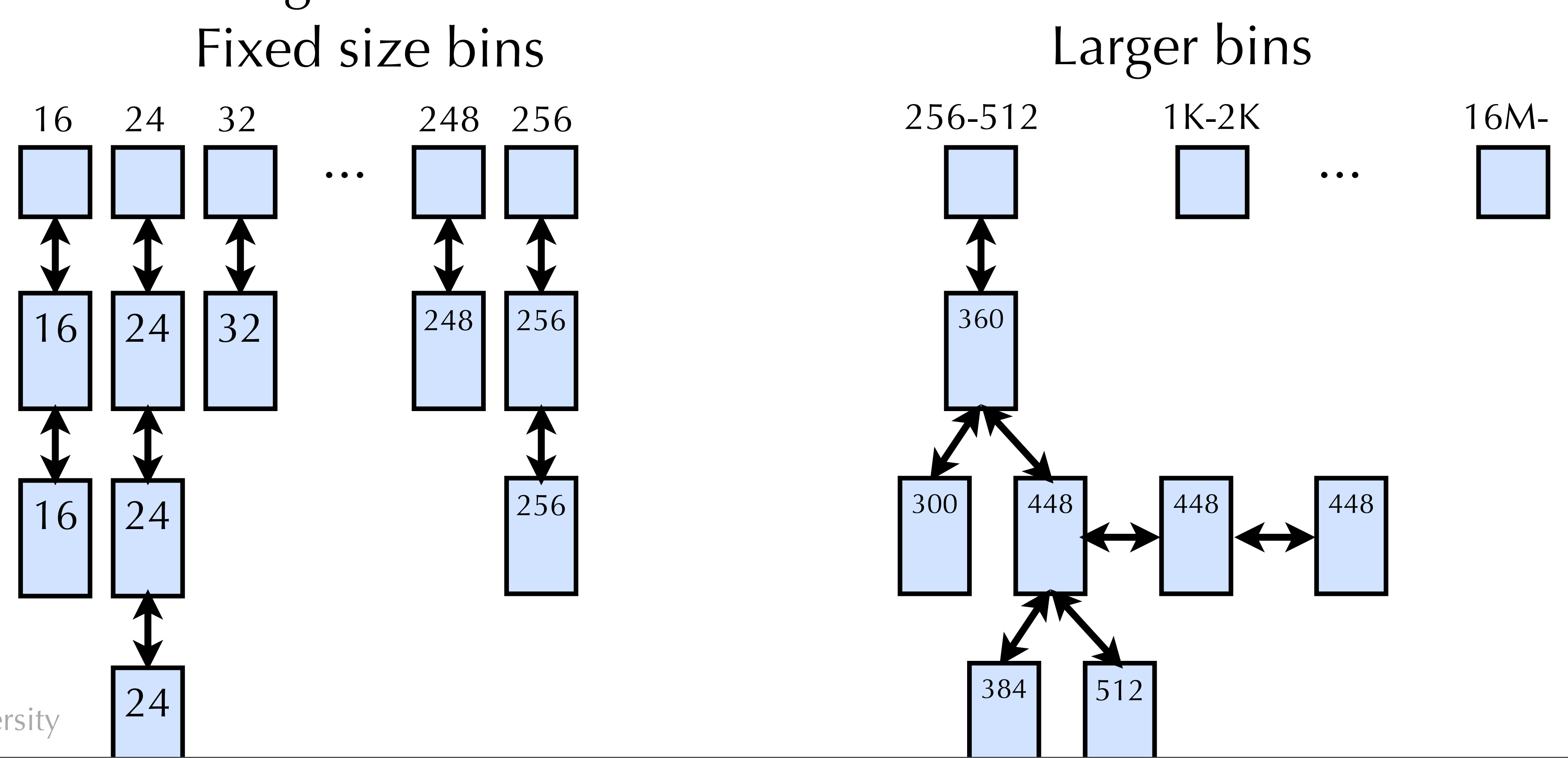
Bins

- Segregated lists
  - Bins for small (fixed size) chunks
    - Chunks stored in circular doubly linked list
  - Bins for large chunks (> 256 bytes)
    - Chunks stored as special kind of binary tree
    - Chunks of same size stored as doubly linked list off node



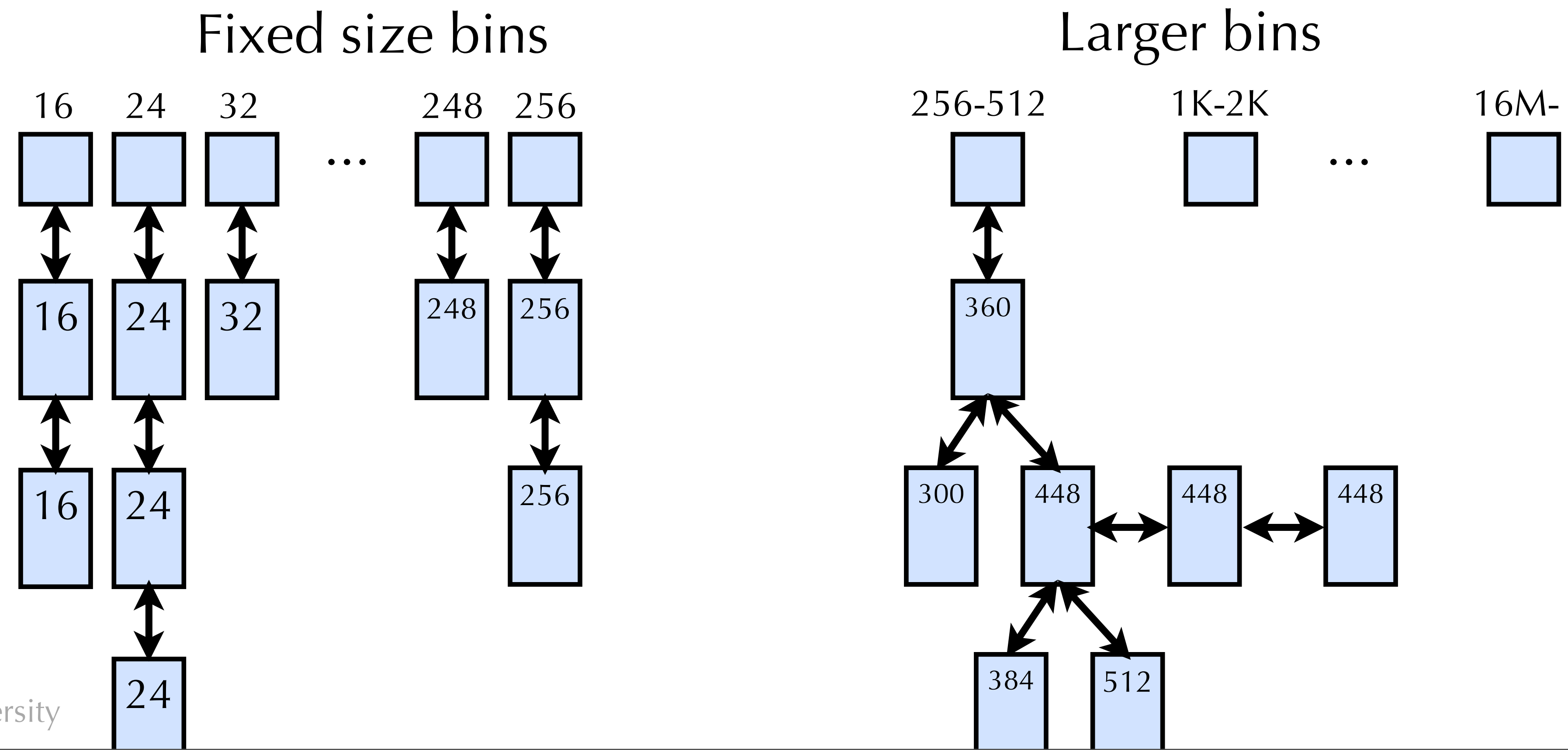
Allocation in dlmalloc

- Find an appropriate size bin
- Look through the structure for the best fit.
  - Lists are arranged so that
    - bins for small chunks are in most-recently-used order
      - To increase locality
    - lists for large chunks of same size are in least-recently-used order
      - To decrease fragmentation



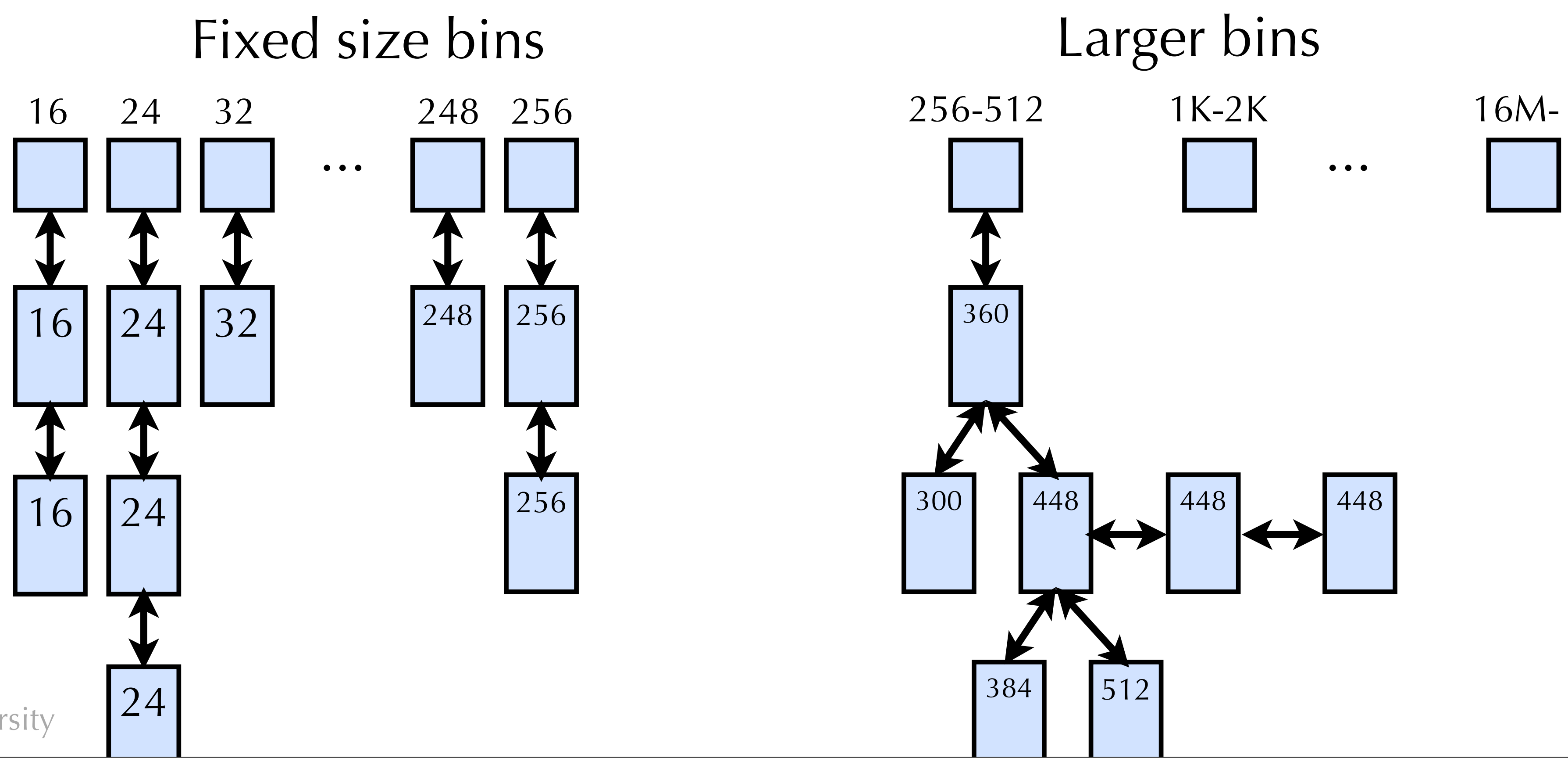
Allocation in dlmalloc

- Algorithm a bit more complicated
- Some other heuristics used for allocation
  - e.g., maintain a “designated victim”, the remainder from the last split, and use it for small requests if possible
- Last free block of heap treated specially



Free in dlmalloc

- Always coalesce on free



Today

- Free block list management
  - Implicit free list
  - Explicit free list
  - Segregated lists
  - Tradeoffs
- Alignment issues
- Example allocator implementation: `dlmalloc`
- Common memory problems
- Garbage collection
  - Mark and sweep
  - Generational GC
  - Reference counting

Memory-Related Perils and Pitfalls

- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks
- Failing to free blocks

Dereferencing Bad Pointers

- The classic `scanf` bug

```
int val;
scanf("%d", val);
```

Not a pointer!

Reading Uninitialized Memory

- Assuming that heap data is initialized to zero

```
/* return y = Ax */
int *matvec(int **A, int *x) {
   int *y = malloc(N*sizeof(int));
   int i, j;

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         y[i] += A[i][j]*x[j];
   return y;
}
```

y[i] not necessarily initialized to zero

Overwriting Memory

- Off-by-one error

```
int **p;

p = malloc(N*sizeof(int *));

for (i=0; i<= N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

Goes through loop N+1 times!

Overwriting Memory

- Not checking the max string size

```
char s[8];

gets(s);
```

Buffer overflow vulnerability!

Overwriting Memory

- Misunderstanding pointer arithmetic

```
int *search(int *p, int val) {

   while (*p && *p != val)
      p += sizeof(int);

   return p;
}
```

Should be p++!
Incorrectly reads every fourth element in array

Overwriting Memory

- Referencing a pointer instead of the object pointed to

```
void trim(char *s, int *length) {
  if (*length > 0) {
    if (s[(*length)-1] == '\n') {
      *length--;
      s[*length] = 0;
    }
  }
}
```

Should be (*length)--!
***length-- is *(length--), which modifies pointer.**

Referencing Nonexistent Variables

- Forgetting that local variables disappear when a function returns

```
int *foo () {
    int val;

    return &val;
}
```

Pointer into stack frame. Valid memory address, but may point to another function's stack frame.

Freeing Blocks Multiple Times

- Nasty!

```
x = malloc(N*sizeof(int));
...manipulate x...
free(x);

y = malloc(M*sizeof(int));
...manipulate y...
free(x);
```

Free same pointer twice!

- What would this do to an explicit free list?
  - Can lead to security vulnerabilities where attacker controls some region of the heap!

Referencing Freed Blocks

- Evil!

```
x = malloc(N*sizeof(int));
...manipulate x...
free(x);
...
y = malloc(M*sizeof(int));
for (i=0; i<M; i++)
   y[i] = x[i]++;
```

Accessing a pointer that has already been freed!

Failing to Free Blocks (Memory Leaks)

- Slow, long-term killer!

```
foo() {
   int *x = malloc(N*sizeof(int));
   ...
   return;
}
```

Object can't be accessed ever again, but is not freed.

Failing to Free Blocks (Memory Leaks)

- Freeing only part of a data structure

```
struct list {
   int val;
   struct list *next;
};

foo() {
   struct list *head = malloc(sizeof(struct list));
   head->val = 0;
   head->next = NULL;
    ...create and manipulate the rest of the list...
   free(head);
   return;
}
```

Only head free, not rest of structure.

Dealing With Memory Bugs

- Conventional debugger (gdb)
  - Good for finding bad pointer dereferences
  - Hard to detect the other memory bugs
- Debugging malloc (CSRI UToronto malloc)
  - Wrapper around conventional malloc
  - Detects memory bugs at malloc and free boundaries
    - Memory overwrites that corrupt heap structures
    - Some instances of freeing blocks multiple times
    - Memory leaks
  - Cannot detect all memory bugs
    - Overwrites into the middle of allocated blocks
    - Freeing block twice that has been reallocated in the interim
    - Referencing freed blocks

Dealing With Memory Bugs (cont.)

- Binary translator: valgrind, Purify
  - Powerful debugging and analysis technique
  - Rewrites text section of executable object file
  - Can detect all errors as debugging malloc
  - Can also check each individual reference at runtime
    - Bad pointers
    - Overwriting
    - Referencing outside of allocated block
- Use a different language
  - Some languages present a different model of memory to programmer
  - Avoid some-to-all of the common memory problems described here

Today

- Free block list management
  - Implicit free list
  - Explicit free list
  - Segregated lists
  - Tradeoffs
- Alignment issues
- Example allocator implementation: `dlmalloc`
- Common memory problems
- Garbage collection
  - Mark and sweep
  - Generational GC
  - Reference counting

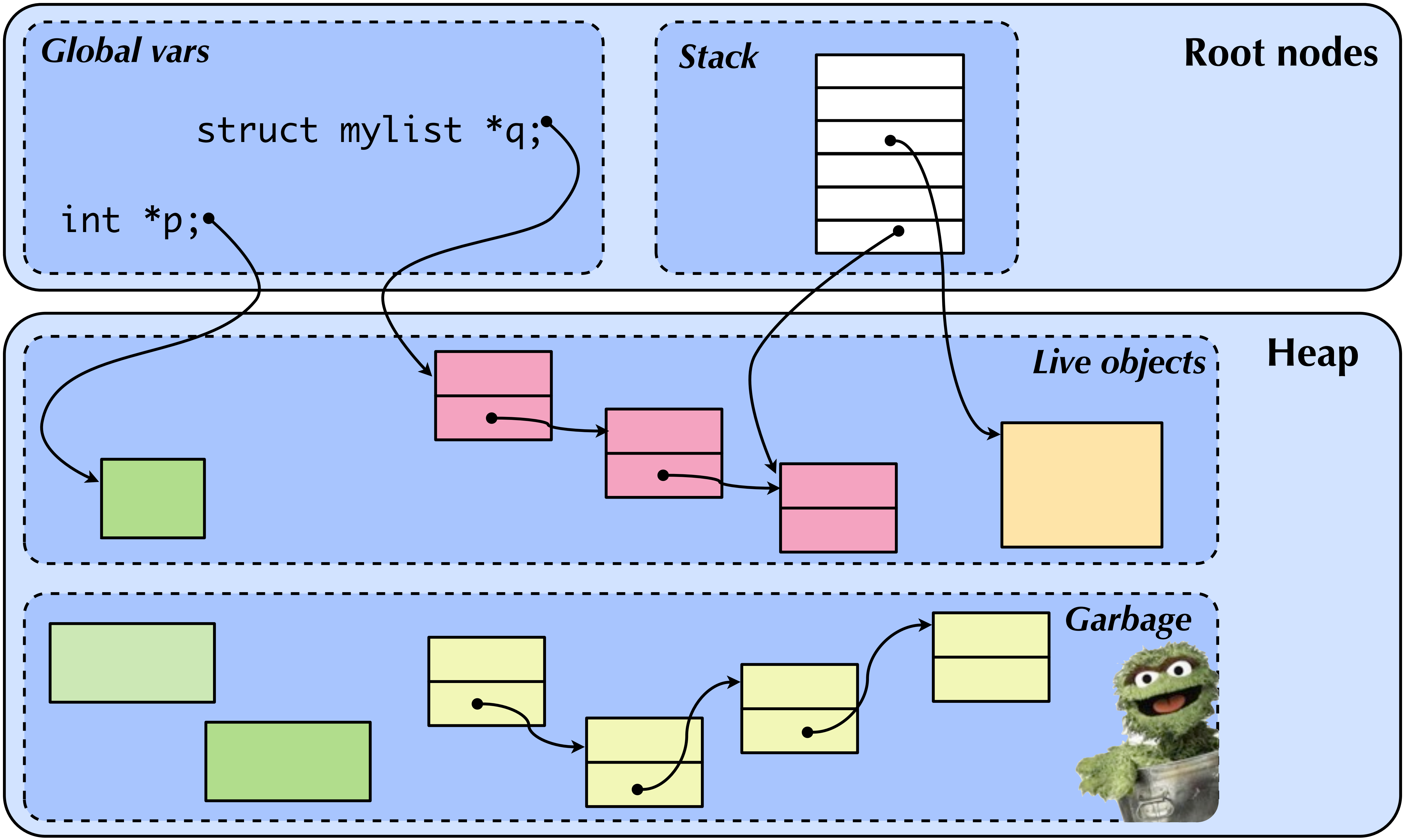
Implicit Memory Management: Garbage Collection

- Garbage collection: automatic reclamation of heap-allocated storage
  - Application never has to explicitly free memory!

```
void foo() {
   int *p = malloc(128);
   return; /* p block is now garbage */
}
```

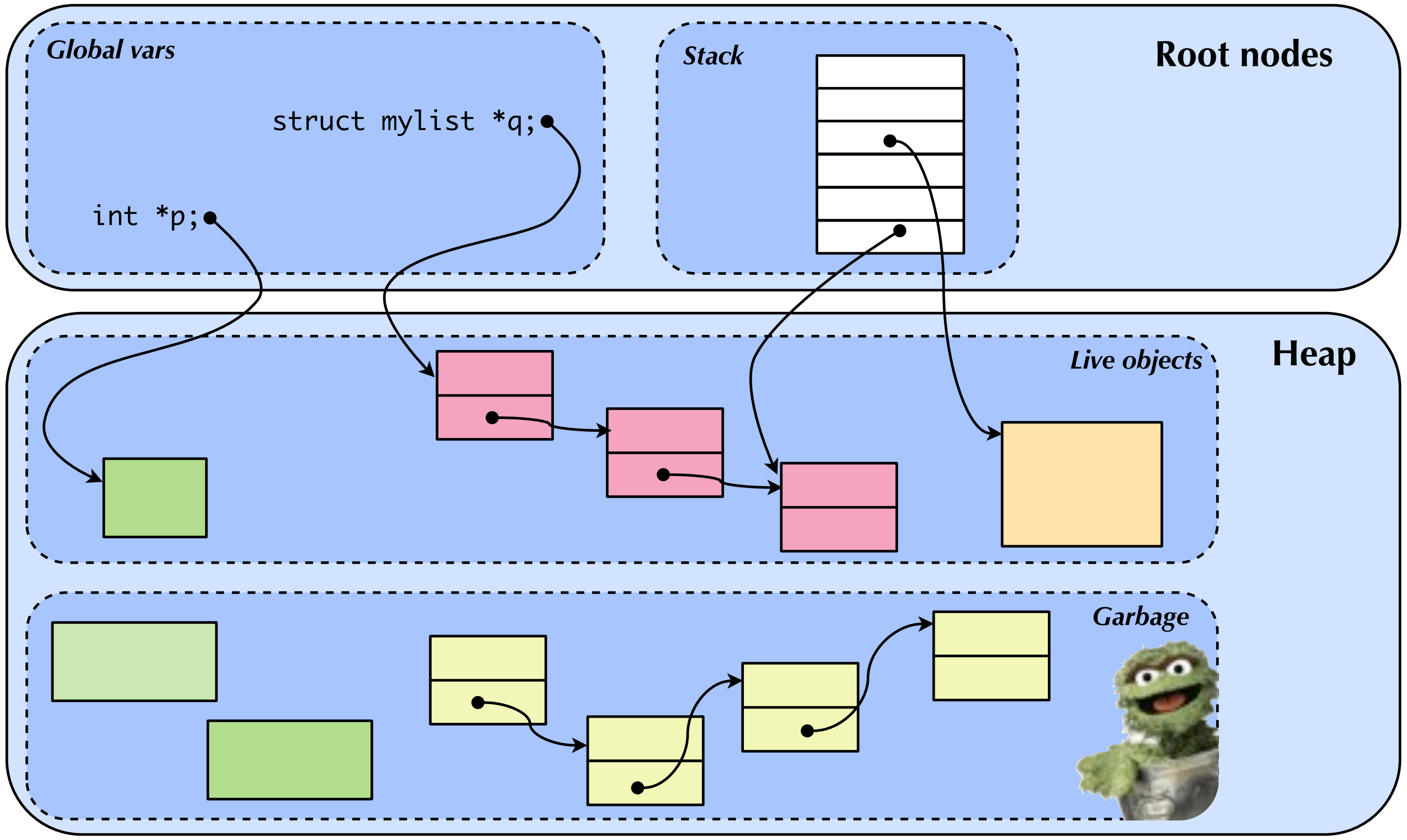
  - Common in functional languages, scripting languages, and modern object oriented languages:
    - Lisp, ML, Java, Perl, Python, etc.
  - Variants (conservative garbage collectors) also exist for C and C++
    - These do not generally manage to collect all garbage though.

Garbage collection concepts



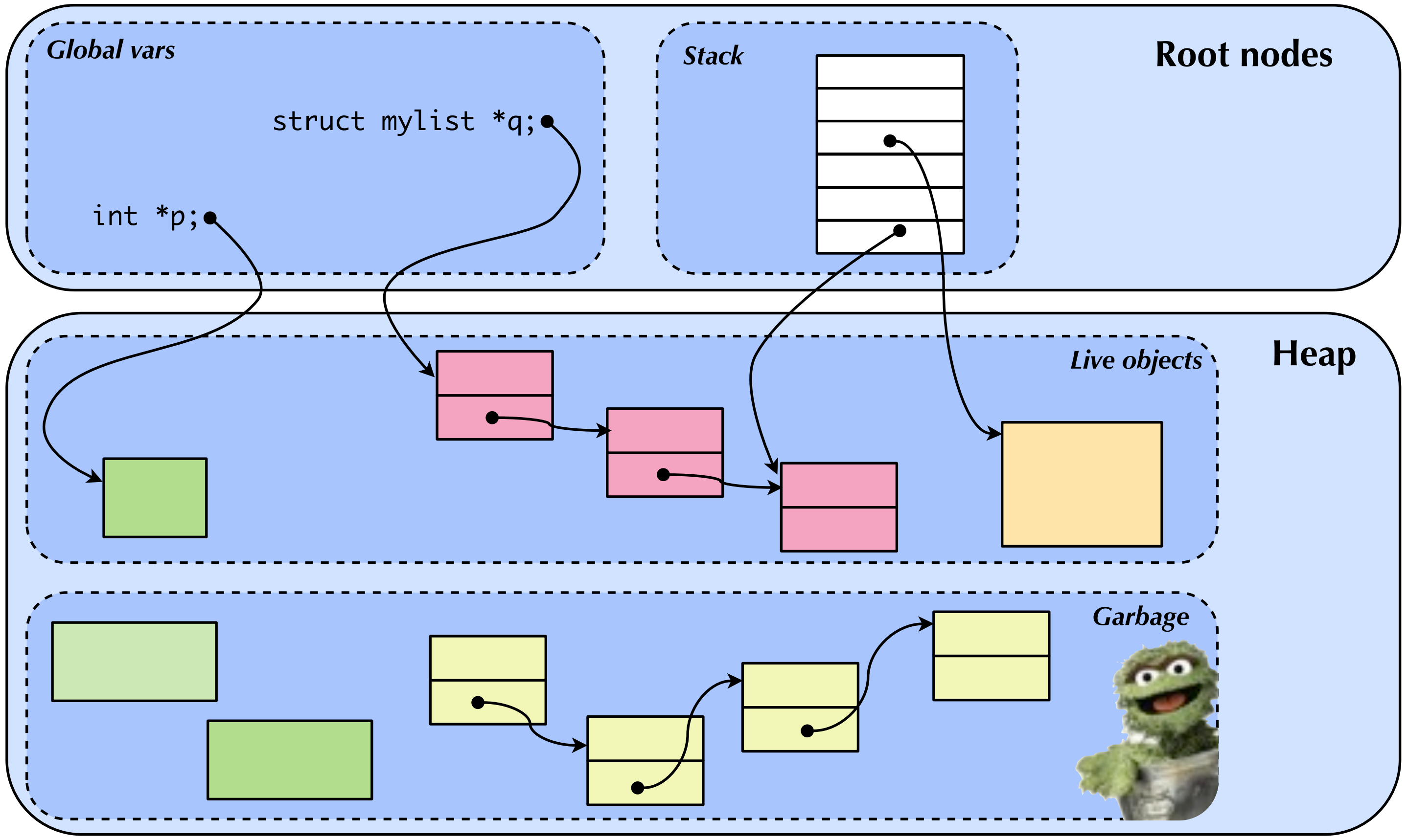
Garbage collection concepts

- A block of memory is **reachable** if there is some path from a root node to the block.
  - If a block is reachable, it's **live** and should not be reclaimed.
  - If a block is not reachable, it's **garbage**.



Garbage collection to the rescue

- Solves some of common memory problems
  - Double free, access after free, some memory leaks, ...
- Say we have a block that is reachable, but never used by the program. What happens?
  - **Memory leak!**
  - Garbage collectors assume all reachable objects can be used in the future.



Challenges

- Challenge: How do we know where the pointers are?
- Global vars are easy: We know the type at compile time.
- What about the stack?
  - GC needs to know which slots on the stack are pointers, and which are not.
  - Depends on call chain of functions at runtime, and how each function stores local vars on the stack.
- What about pointers **between** blocks of memory?
  - Linked lists, trees, etc.
  - GC needs to know where the pointers are internal to each block!

Fake pointers

- In C, you can create a pointer directly from an integer…

```
int i = 0x40b0cdf;
*(int *)i = 42;
```

- But type of `i` is `int`! How do we know it is used as a pointer?
- Conservative garbage collectors treat all bit patterns in memory that could be pointers as pointers.
  - If contents of location could be memory address in heap, assume it is
- Problem: Leads to “false” memory leaks
  - If `int i = 0x40b0cdf` really is just an integer value, not a pointer, memory at location 0x40b0cdf will not be reclaimed!

Fake pointers

- In C, you can create a pointer directly from an integer…

```
int i = 0x40b0cdf;
*(int *)i = 42;
```

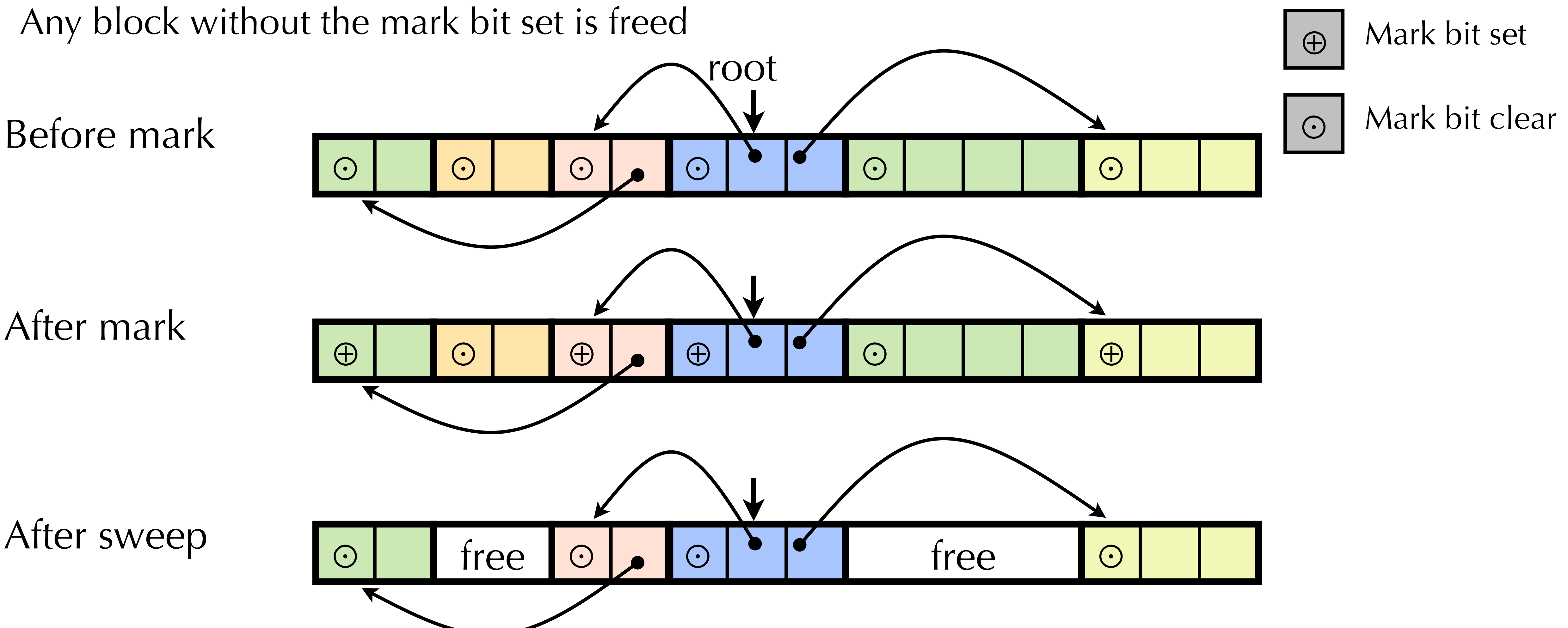
- Solution: Don't let program create pointers from ints.
  - Java (and other languages) never let you “forge” pointers like this.
  - Allows for precise garbage collection: GC always knows where pointers are, since pointers are “special”.

Garbage collection techniques

- We'll consider three techniques
  - Mark-and-sweep
  - Generational garbage collection
  - Reference counting

Mark and Sweep Collecting

- Idea: Use a mark bit in the header of each block
  - GC scans all memory objects (starting at roots), and sets mark bit on each reachable memory block.
- Sweeping:
  - Scan over all memory blocks.
  - Any block without the mark bit set is freed



Mark and Sweep

- Advantages
  - Easily handles cycles
  - Low overhead
- Disadvantages
  - Need to scan all objects
    - Slow if large object set
- Observation: Most objects have short lifespans
  - Most objects created since the last garbage collection
  - If an object survives a few garbage collection cycles, it is likely to survive longer.

Generational Garbage Collection

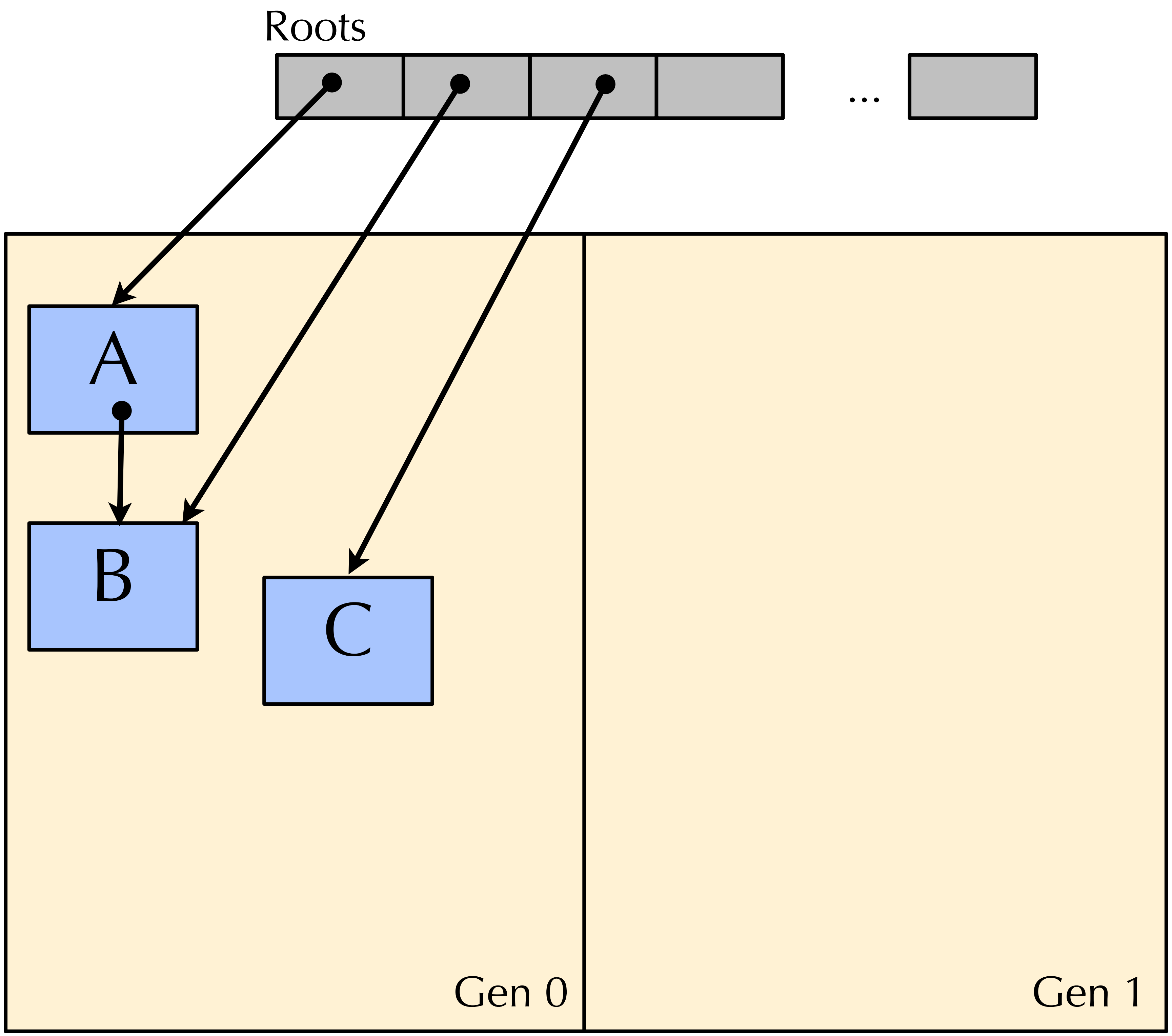
- Key idea: divide objects into **generations**, and garbage collect newer generation more frequently than older generations
- Takes advantage that most objects do not live long
- Used in Java and .NET

Generational GC Algorithm

- Divide objects into generations (Gen0, Gen1...)
  - New objects go in Gen0
- Garbage collect younger generations aggressively
  - Perform GC on Gen(n)
    - mark and sweep, or something else
  - Promote surviving objects of Gen(n) to Gen(n+1)
- May occasionally need a “full” GC

- What does “promotion” mean?
  - Some implementations assign regions of memory to generations
  - Promotion to older generation means copy the object to a different region of memory
    - This is difficult in C. Why?

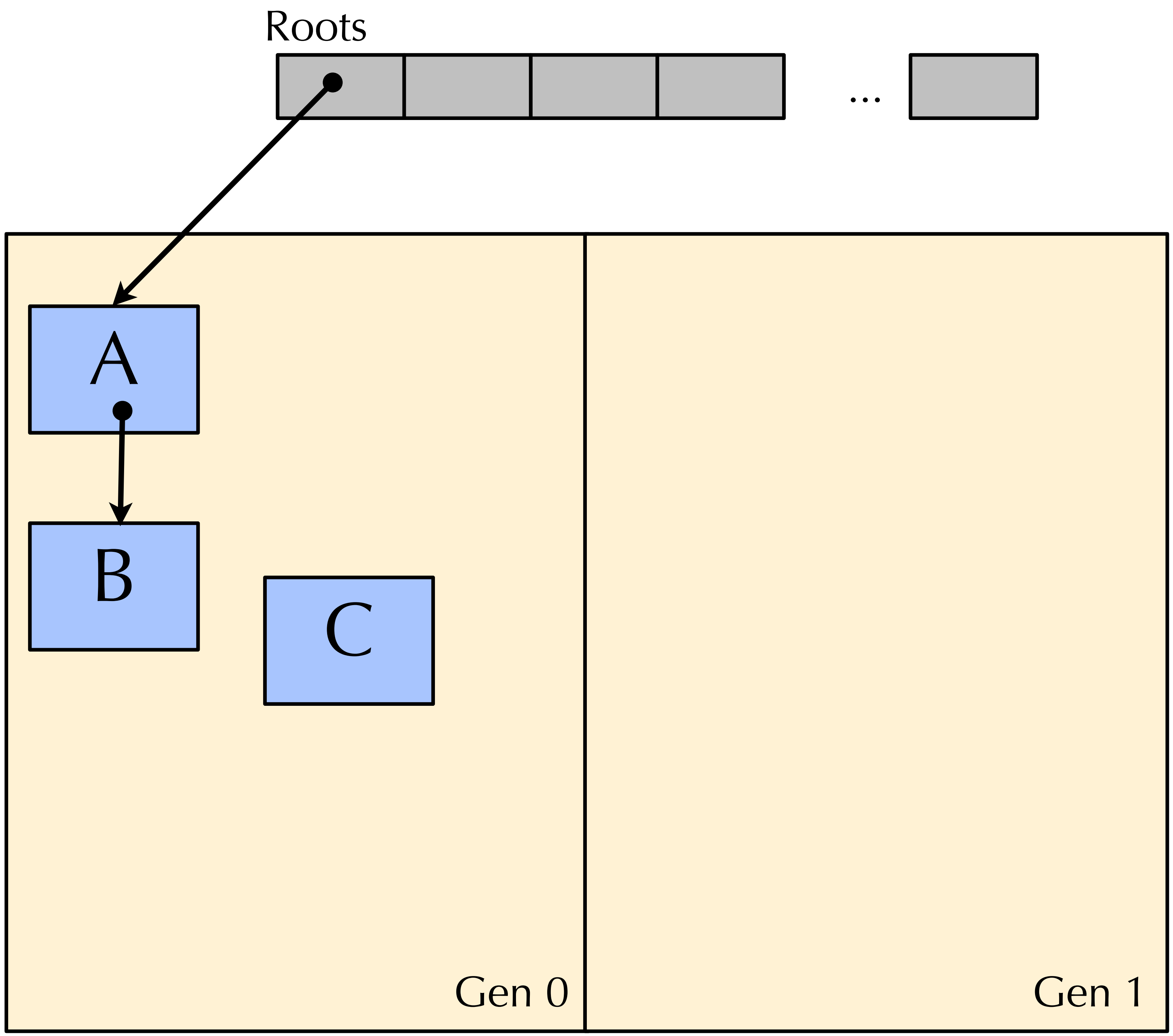
Generational Garbage Collection



We start executing…

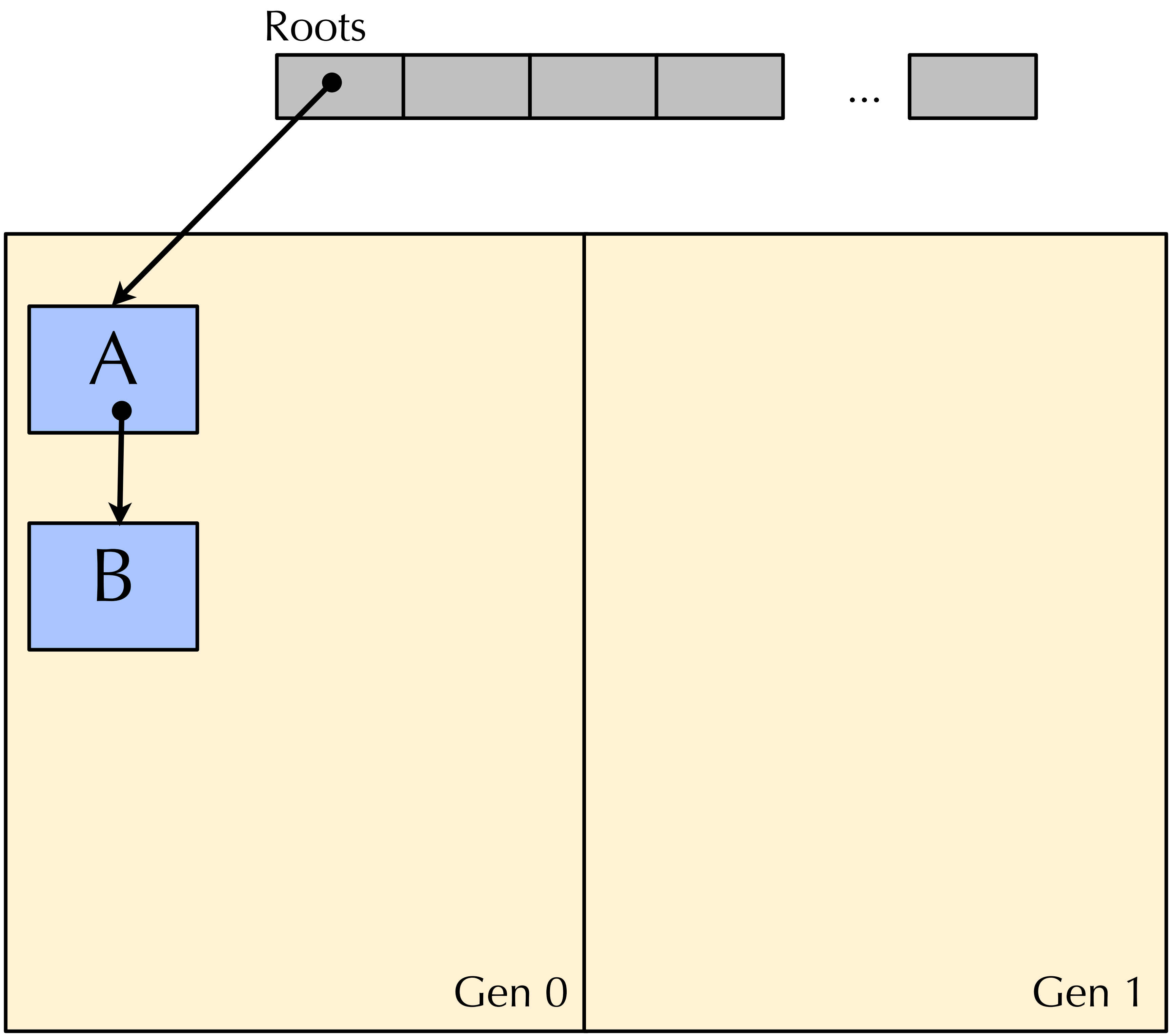
New object go in Gen0

Generational Garbage Collection



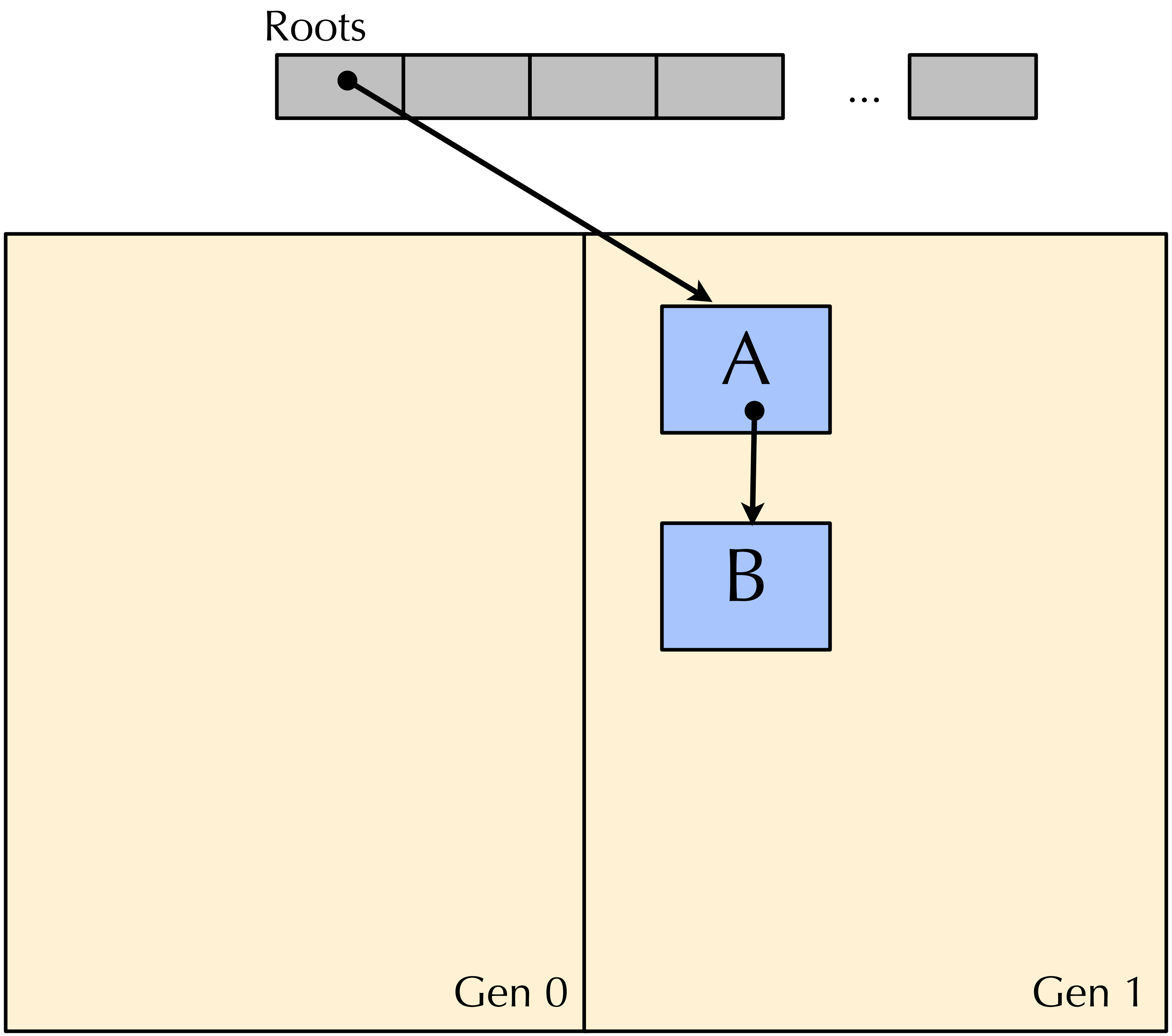
Some objects become garbage

Generational Garbage Collection



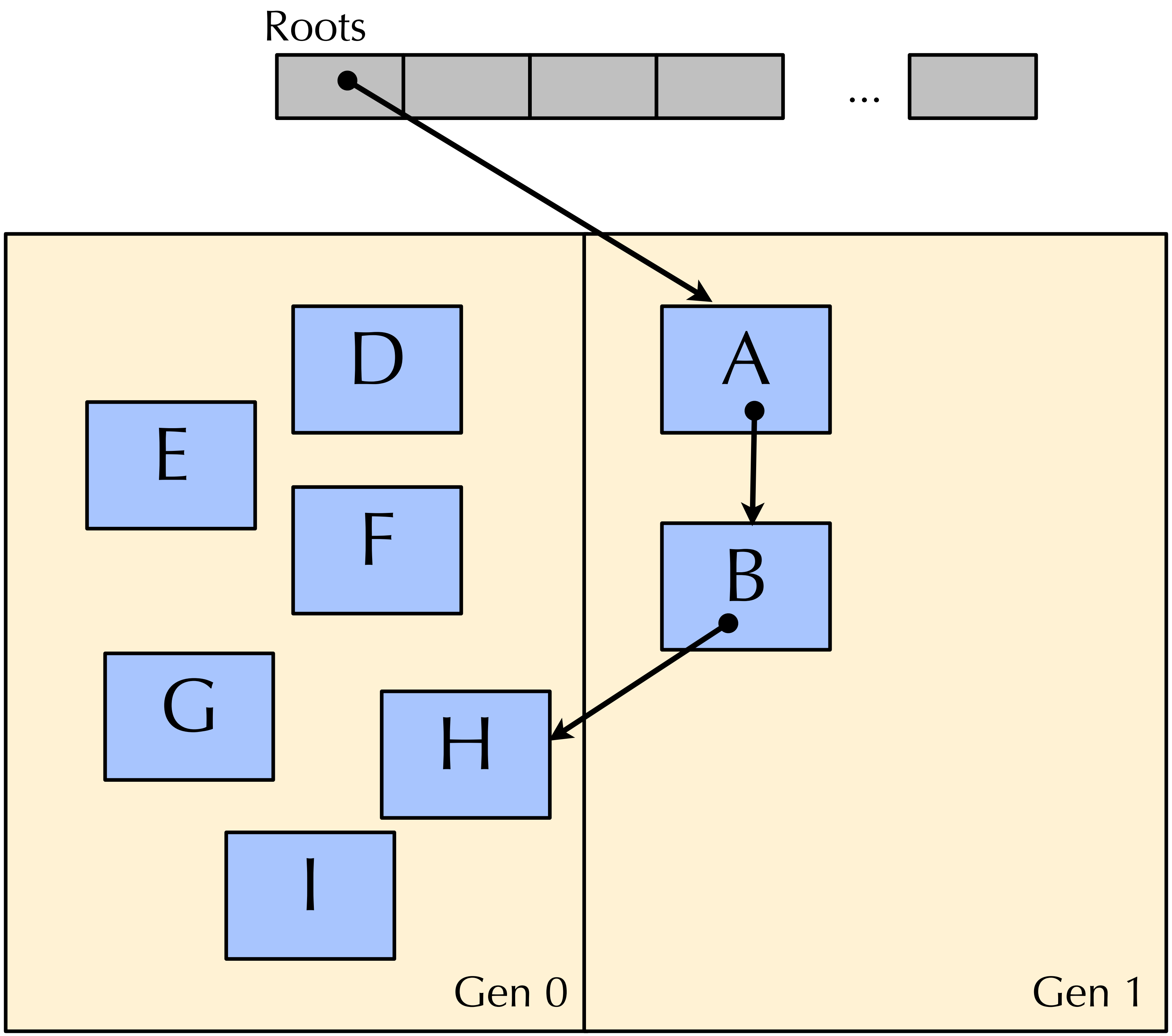
We garbage collect Gen0.

Generational Garbage Collection



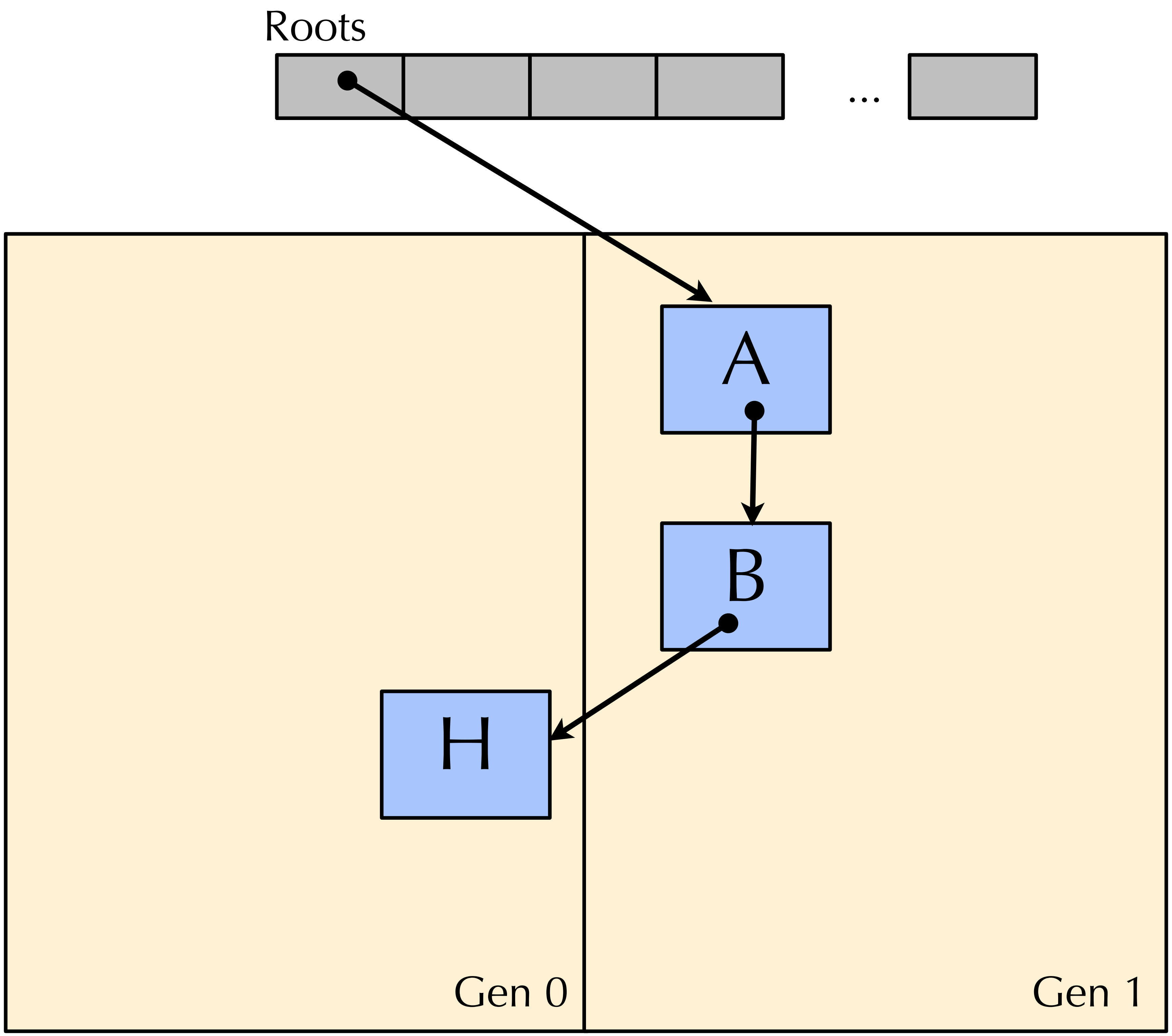
Promote surviving objects from Gen0 to Gen1

Generational Garbage Collection



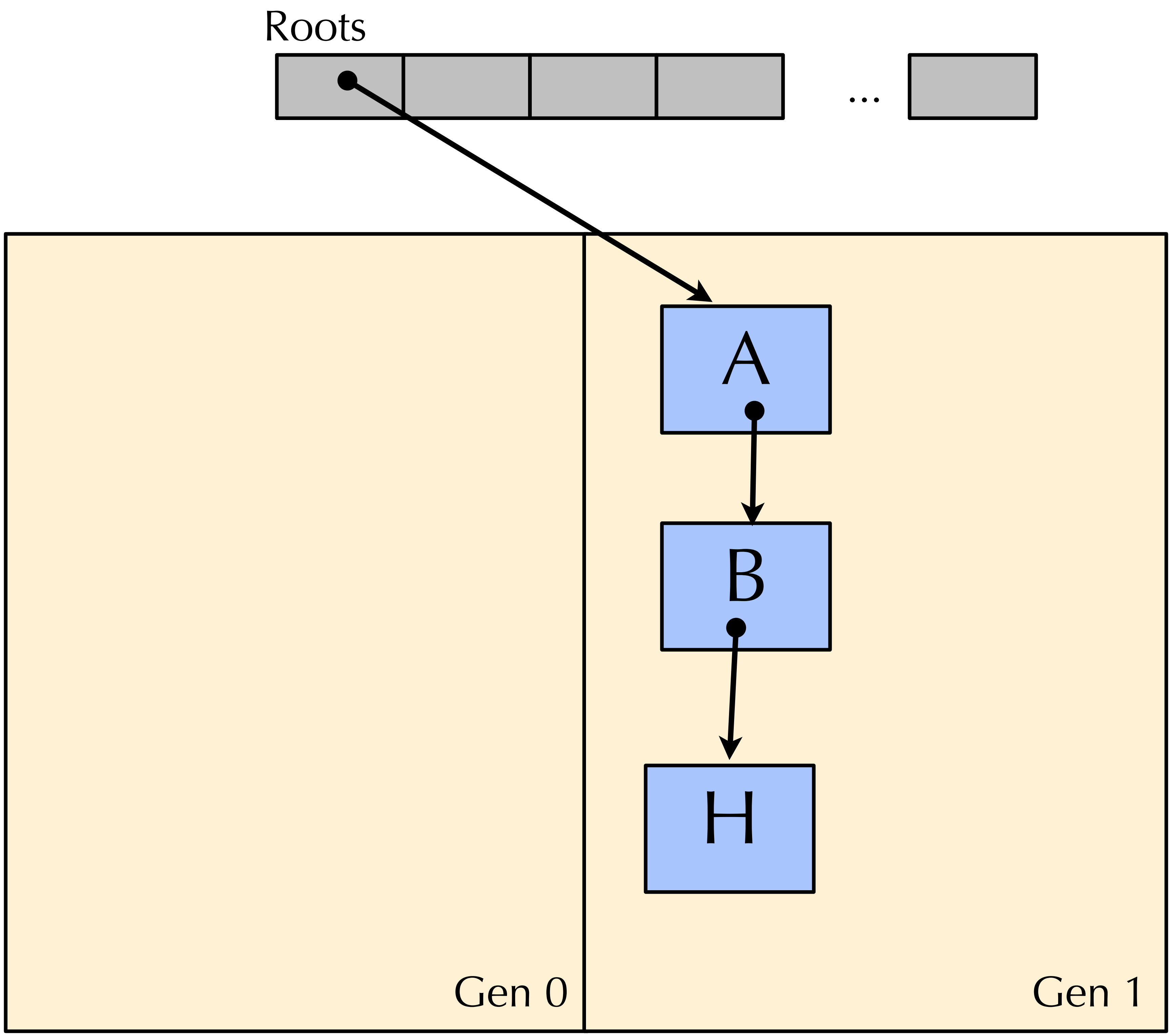
Program continues to execute. Many short lived objects.

Generational Garbage Collection



We garbage collect Gen0.

Generational Garbage Collection



Promote surviving objects from Gen0 to Gen1

Today

- Free block list management
  - Implicit free list
  - Explicit free list
  - Segregated lists
  - Tradeoffs
- Alignment issues
- Example allocator implementation: `dlmalloc`
- Common memory problems
- Garbage collection
  - Mark and sweep
  - Generational GC
  - Reference counting

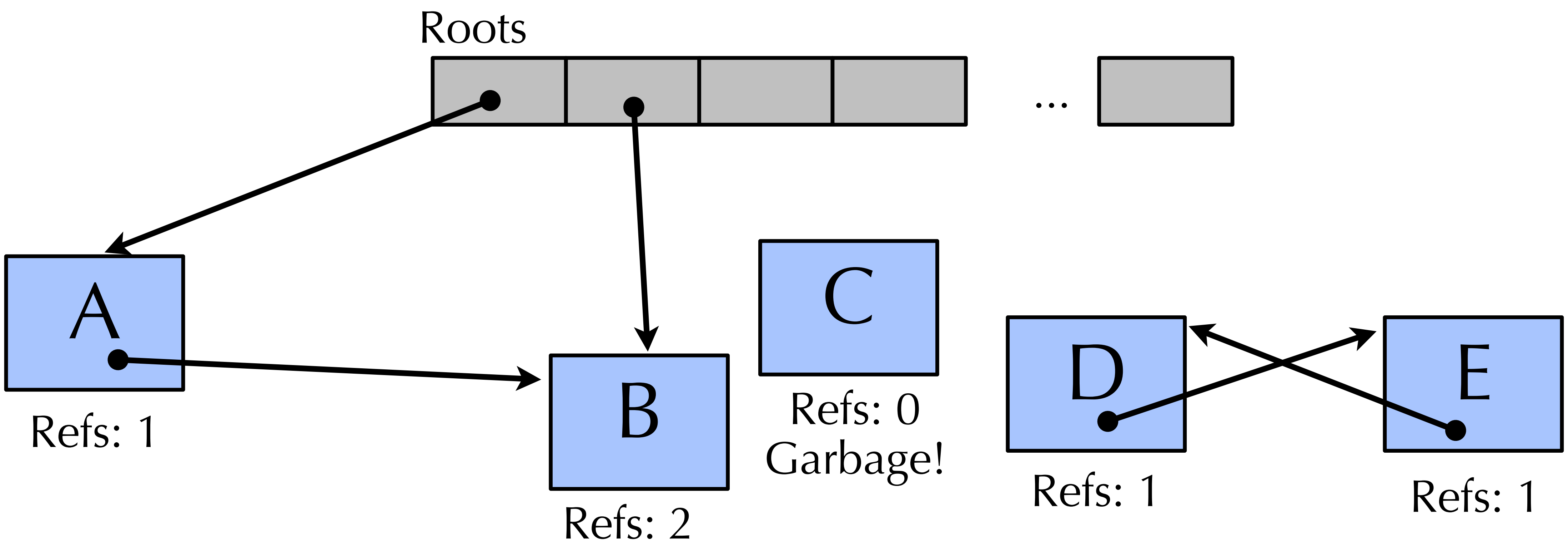
Reference Counting

- An object is garbage if it is not reachable from a root.
- If nothing points to an object (i.e., object is not **referenced**), then it is garbage.

- Problem: How do we decide if an object is no longer referenced?
- Solution: Keep track of number of active references to every object
  - As soon as reference hits 0, can immediately GC object

- Used in: Python, Microsoft COM, Apple Cocoa, etc.
  - Many C hash table implementations use ref. counting to keep track of active entries

Reference counting example



- Problems
  - Changing pointers (including a variable going out of scope) requires updating reference counts. Can be expensive
  - What about cycles? (e.g., D and E)

Next lecture

- Memory and storage technologies