Memory Allocation III

CSE 351 Summer 2021

Instructor:

Mara Kirdani-Ryan

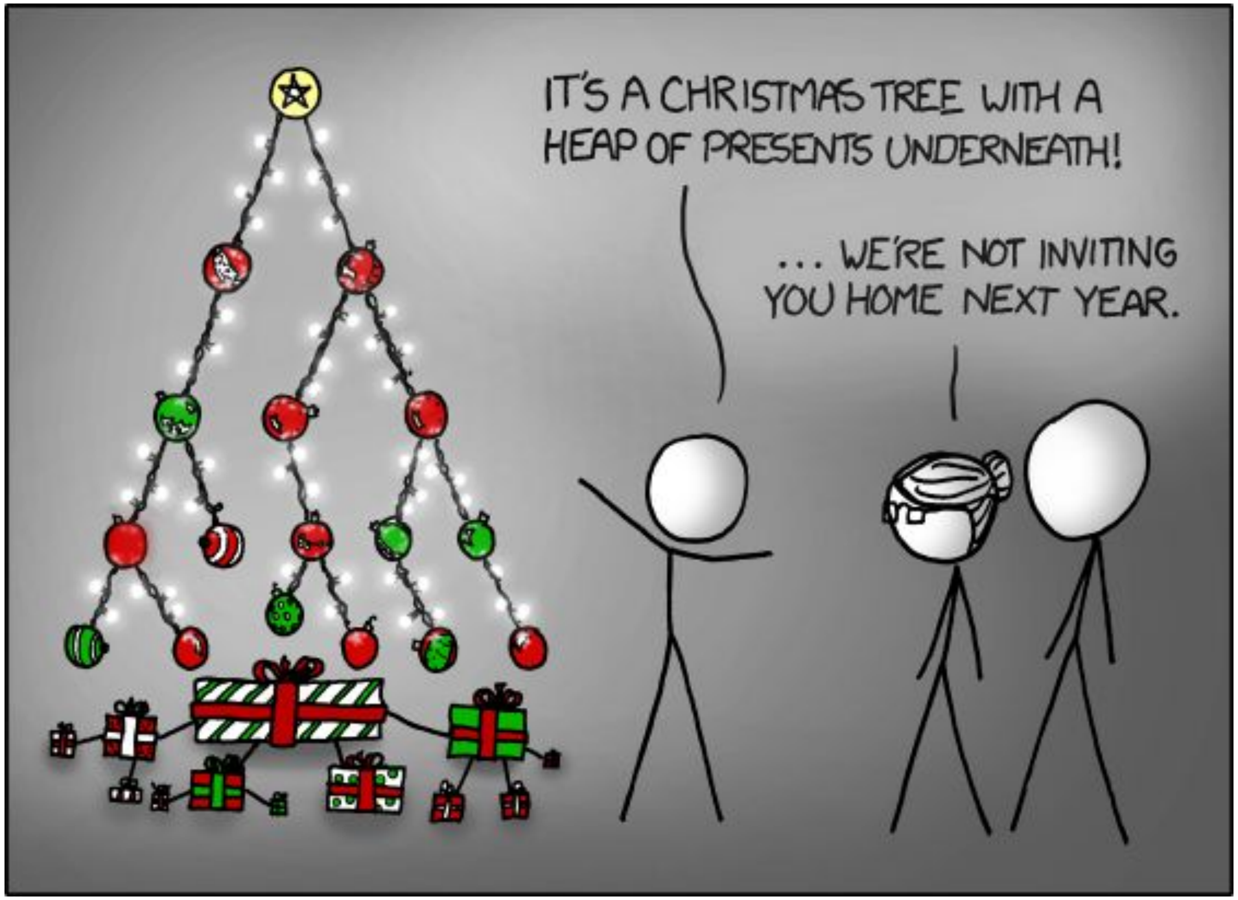
Teaching Assistants:

Kashish Aggarwal

Nick Durand

Colton Jobs

Tim Mandzyuk


IT'S A CHRISTMAS TREE WITH A HEAP OF PRESENTS UNDERNEATH!
... WE'RE NOT INVITING YOU HOME NEXT YEAR.

https://xkcd.com/835/

Gentle, Loving Reminders

- Hw21 due tonight!
- Lab 5 due Wednesday!
  - Email if you're using late days.
  - Focus on understanding concepts, before diving into coding! C can be tricky!
- Unit Summary 3 due Friday!
  - Covering Caches, Processes, VM, Malloc
  - Only up to last friday, we've talked about everything!

And then we're done!

Course Evals are out!

Please, please, please fill them out!

It’s my first time teaching, I’d love to know how y’all felt about it!

Allocation Policy Tradeoffs

- Data structure of blocks on lists
  - Implicit (free/allocated), explicit (free), segregated (many free lists) – others possible!
- Placement policy: first-fit, next-fit, best-fit
  - Throughput vs. amount of fragmentation
- When do we split free blocks?
  - How much internal fragmentation can we tolerate?
- When do we coalesce free blocks?
  - **Immediate coalescing:** Every time `free` is called
  - **Deferred coalescing:** Defer coalescing until needed
    - e.g. when scanning free list for `malloc` or when external fragmentation reaches some threshold

More Info on Allocators

- D. Knuth, "*The Art of Computer Programming*", 2nd edition, Addison Wesley, 1973
  - The classic reference on dynamic storage allocation

- Wilson et al, "*Dynamic Storage Allocation: A Survey and Critical Review*", Proc. 1995 Int'l Workshop on Memory Management, Kinross, Scotland, Sept, 1995.
  - Comprehensive survey
  - Available from CS:APP student site (csapp.cs.cmu.edu)

Memory-Related Perils in C

A)	Dereferencing a non-pointer
B)	Freed block – access again
C)	Freed block – free again
D)	Memory leak – failing to free memory
E)	No bounds checking
F)	Reading uninitialized memory
G)	Referencing nonexistent variable
H)	Wrong allocation size

Find That Bug!

```
char s[8];
int i;

gets(s);   /* reads "123456789" from stdin */
```

Error Type: | Prog stop Possible? | Fix:

Find That Bug!

```
char s[8];
int i;

gets(s);   /* reads "123456789" from stdin */
```

No bounds checking! Buffers could overflow!

Fix: use fgets()

Polling Question [Alloc III]

- Which error is this?

```
int* foo() {
    int val = 0;
       . . .
    return &val;
}
```

Dereferencing a non-pointer

Reading uninitialized Memory

Returning/referencing a non-existent variable

Returning the wrong type

Find That Bug!

```
int **p;

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

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

- N and M defined elsewhere (#define)

Error Type: ☐ **Prog stop Possible?** ☐ **Fix:**

Find That Bug!

```
int **p;

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

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

- N and M defined elsewhere (#define)

Wrong allocation size! We needed to allocate an array of pointers!

Fix: Make sure that types passed to malloc() match

Find That Bug!

```
/* return y = Ax */
int *matvec(int **A, int *x) {
   int *y = (int *)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;
}
```

- `A` is `NxN` matrix, `x` is `N`-sized vector (so product is vector of size `N`)
- `N` defined elsewhere (`#define`)

Find That Bug!

```
/* return y = Ax */
int *matvec(int **A, int *x) {
   int *y = (int *)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;
}
```

- `A` is `NxN` matrix, `x` is `N`-sized vector (so product is vector of size `N`)
- `N` defined elsewhere (`#define`)

We're reading uninitialized memory! What's in y?
Fix: Zero y[i], or use calloc()

Find That Bug!

- The classic `scanf` bug
  - `int scanf(const char *format, ...)`

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

See: http://www.cplusplus.com/reference/cstdio/scanf/?kw=scanf

Find That Bug!

- The classic `scanf` bug
  - `int scanf(const char *format, ...)`

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

See: http://www.cplusplus.com/reference/cstdio/scanf/?kw=scanf

We're dereferencing a non-pointer!
Fix: Use &val

Find That Bug!

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

   ...

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

Find That Bug!

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

   ...

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

We free'd x twice -- double free!
Fix: change to free(y), fix typos

Find That Bug!

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

   ...

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

Find That Bug!

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


   ...


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

We're accessing x after we free!
Fix: Move free, or do another allocation

Find That Bug!

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

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

Find That Bug!

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

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

We lost our pointer to the list! We've just *leaked* all that memory!
Fix: need to free entire list, not just head

Dealing With Memory Bugs

Non-testable Material

- Conventional debugger (`gdb`)
  - Good for finding bad pointer dereferences
  - Hard to detect the other memory bugs
- Debugging `malloc` (UToronto CSRI `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's reallocated in the interim
    - Referencing freed blocks

Dealing With Memory Bugs

Non-testable Material

- Some `malloc` implementations contain checks
  - Linux glibc malloc: **`setenv MALLOC_CHECK_ 2`**
  - FreeBSD: **`setenv MALLOC_OPTIONS AJR`**
- Binary translator: valgrind (Linux), 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

What about Java or Python or …?

Non-testable Material

- In *memory-safe languages*, most of these bugs are impossible
  - Cannot perform arbitrary pointer manipulation
  - Cannot get around the type system
  - Array bounds checking, null pointer checking
  - Automatic memory management
- But one of the bugs we saw earlier is possible. Which one?

Debugging

You're going to write bugs!

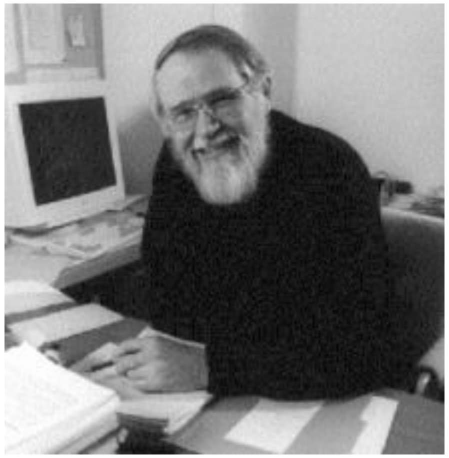
- Seemingly an inevitability

"As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs." ~ Maurice Wilkes (EDSAC)

But, you can prevent some of them!

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

Brian Kernighan



"Defensive Coding"

- Says "I'm going to spend most of my time debugging anyways, may as well spend more time coding up front"
  - Using #defines instead of numbers
  - Coding with comments that record though process
  - Add sensible checks throughout program
    - Are these pointers null?
    - Are these data structures still good?

There's lots of debugging strategies!

You've got to find what works for you!

Mindful, Embodied Debugging

A brief grounding…

Mindfulness

- "Present, attentive mind"
  - There's so many definitions, it's hard to keep track!

- A few questions, instead:
  - Are you focused on what is, or should be?
  - Where's your focus? If it's wandering, can it come back?
  - How are you feeling? What are you feeling?
  - Are those feelings attached to anything in the present moment? If not, can you gently let them go?

Mindfulness, Computing, and me

- What happens when your code doesn't work?
  - For me, I've felt defeated, grumpy, anxious
  - I've tried to fix things, and it still doesn't work
  - *I'll get more grumpy and anxious*
  - I'll keep cycling in this for a while, getting nowhere
  - *Even more anxious and defeated*, *usually bolstered by my lifetime of self-esteem issues*

- Sound familiar?
  - There's lots of research; folks don't think well with high emotional affect
  - What about alternatives?

Mindfulness and debugging

- What happens when your code doesn't work?
  - I try to acknowledge that it's part of the process
  - I'll take a minute, I'll take a few breaths
  - I'll think; *what might've gone wrong?*
  - I'll go looking and won't find anything wrong
  - *I'll feel tired, and maybe anxious*
    - Is there any way that I can make space for what I'm feeling?
    - Can I try and calm down, before trying again?
  - Is there a different approach that I can take, if this one isn't working?

AKA Being Systematic

- What if someone asked you, every 5 minutes, to justify and explain what you were doing?
  - *What are you doing?*
  - *Why are you doing it?*
  - *What else have you tried?*
- Sometimes folks get locked into an approach without evaluating others first!
  - Rearranging lines, without understanding why?
  - Changing constants, without understanding why?
  - Re-running and getting the same results?
  - *What else could you be doing?*
- **Metacognition**: Recognizing your thinking, which is basically just mindfulness

Something to try:

Every 10 minutes (set a timer), evaluate your approach!

What are you trying to achieve?
What have you tried?
What else could you try?
Could you defend your current approach, against alternatives?

Embodiment

- Recognizing your physical form, how you feel in your body, at this moment
  - Closely tied with mindfulness, equally hard to define

- A few questions, again:
  - How do you feel, in your body?
  - Are your feet grounded against the floor?
  - Are you sitting tall? Could you sit taller?
  - Are there any sensations that you notice?
  - How does it feel to be breathing?
  - Could you breathe a little deeper, and a little slower?

Embodiment, Computing, and Me

- Computing's kinda weird, honestly
- Some programmers describe being "in the zone"
  - Incredibly focused, incredibly productive
  - *For me, this looks like hyperfocus*

Embodiment, Computing, and Me

- Computing's kinda weird, honestly
- Some programmers describe being "in the zone"
  - Incredibly focused, incredibly productive
  - *For me, this looks like hyperfocus*

- **Hyperfocus**: trouble regulating attention between different tasks
  - Getting sucked in
  - Oblivious to everything else
  - Immobile for hours at a time
  - "Productive", but not always a good thing

Embodiment, Computing, and Me

- For me, hyperfocus is sometimes fun, sometimes inconvenient
  - When I was little, I'd forget to go to the bathroom
  - Today, I'll be holding my breath unconsciously
  - *Really, I'll forget to be embodied*

Embodiment, Computing, and Me

- For me, hyperfocus is sometimes fun, sometimes inconvenient
  - When I was little, I'd forget to go to the bathroom
  - Today, I'll be holding my breath unconsciously
  - *Really, I'll forget to be embodied*

- CS actively discourages embodiment!
  - Sitting in front of a screen for hours on end
  - Hackathons are a great way to ignore bodily needs
  - Lots of trans* folks in programming, it's a great way to escape dysphoria

Embodiment and debugging

- What happens when your code doesn’t work?
  - As my mood drops, my posture collapses
  - Feelings will manifest in my body; I won’t notice them
  - I’ll be feeling “tense”, but I won’t be able to decompose that into feelings
  - I’ll forget to attend to my needs, or I’ll tell myself that I’ll do that after I’m done

Embodiment and debugging

- What happens when your code doesn’t work?
  - I try to acknowledge that it’s part of the process
  - I’ll take a minute, I’ll take a few breaths
  - I’ll think; *what might’ve gone wrong?*
  - I’ll go looking and won’t find anything wrong
  - *I’ll feel tired, and maybe anxious*
  - I’ll check in with my body, and realize that a cup of tea and a bathroom trip might help me feel more comfortable
  - Sometimes, as I’m making tea, I’ll realize what the issue was
    - Really, I just needed a break

AKA Supporting yourself!

- There are few guarantees for support, besides the support that you can give yourself
  - Supporting yourself is a learning process!
  - I’ve been terrible at it for most of my life,
    I’m just now getting better

AKA Supporting yourself!

- There are few guarantees for support, besides the support that you can give yourself
  - Supporting yourself is a learning process!
  - I’ve been terrible at it for most of my life, I’m just now getting better
- Supporting myself helps my debugging!
  - I’ve figured out so many things on a break
  - I’ve gotten better at recovering from setbacks
  - I’m more comfortable being myself; I’ll always show up for me

An aside; breaks

- People are really different!
  - *What works for me might not work for you*

- We can be mindful of how we're resting!
- What does a break mean?
  - Attending to bodily needs?
  - Checking phones and getting sucked in?
  - Doing 18 other things on the internet?
  - *Do you feel rested after these breaks?*
  - *Is there something that you could do, that might be more restful?*
    - I haven't found anything better than lying on the floor, arms out, eyes closed

Something to try:

Every 10 minutes (set a timer), evaluate yourself!

How do I feel?
Do I have any needs that I should address?
Am I feeling anxious, stressed, or frustrated?
Is there anything I can do to make myself more comfortable?

Good luck finishing lab 5!
Good luck on US#3!

Reach out if you want to meet!
calendly.com/marakr

Freeing with LIFO Policy (Explicit Free List)

	Predecessor Block	Successor Block	Change in Nodes in Free List	Number of Pointers Updated
Case 1	Allocated	Allocated		
Case 2	Allocated	Free		
Case 3	Free	Allocated		
Case 4	Free	Free		

Find That Bug! (Slide 50)

```
int* foo() {
   int val = 0;

   return &val;
}
```

Error Type:

Prog stop Possible?

Fix:

Memory Allocation

- Dynamic memory allocation
  - Introduction and goals
  - Allocation and deallocation (free)
  - Fragmentation
- Explicit allocation implementation
  - Implicit free lists
  - Explicit free lists (Lab 5)
  - Segregated free lists
- **Implicit deallocation: garbage collection**
- **Common memory-related bugs in C**

Wouldn’t it be nice…

- If we never had to free memory?
- Do you free objects in Java?
    - Reminder: *implicit* allocator

Garbage Collection (GC)
(Automatic Memory Management)

- *Garbage collection:* automatic reclamation of heap-allocated storage – application never explicitly frees r

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

- Common in implementations of functional languages, scripting languages, and modern object oriented languages:
  - Lisp, Racket, Erlang, ML, Haskell, Scala, Java, C#, Perl, Ruby, Python, Lua, JavaScript, Dart, Mathematica, MATLAB, many more…

- Variants (“conservative” garbage collectors) exist for C

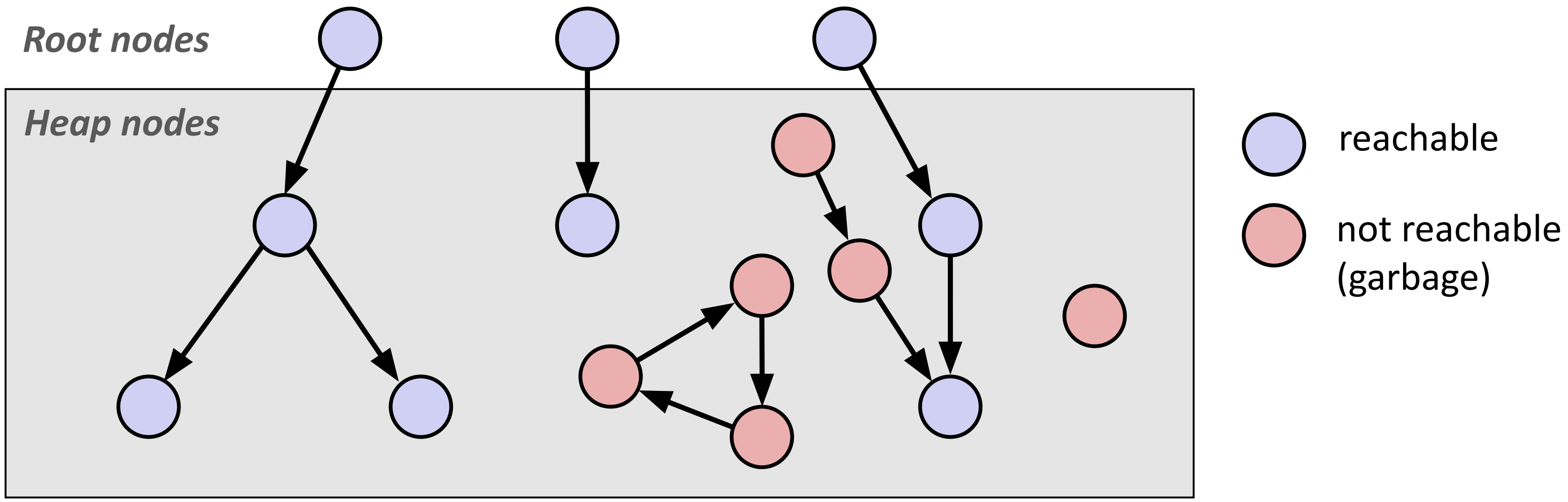
Garbage Collection

- How does the memory allocator know when memory can be freed?
  - In general, we cannot know what is going to be used in the future since it depends on conditionals
  - But, we can tell that certain blocks cannot be used if they are *unreachable* (via pointers in registers/stack/globals)

- Memory allocator needs to know what is a pointer and what is not – how can it do this?
  - Sometimes with help from the compiler

Memory as a Graph

- We view memory as a directed graph
  - Each allocated heap block is a node in the graph
  - Each pointer is an edge in the graph
  - Locations not in the heap that contain pointers into the heap are called ***root*** nodes (e.g. registers, stack locations, global variables)



A node (block) is *reachable* if there is a path from any root to that node
Non-reachable nodes are *garbage* (cannot be needed by the application)

Garbage Collection

- Dynamic memory allocator can free blocks if there are no pointers to them

- How can it know what is a pointer and what is not?

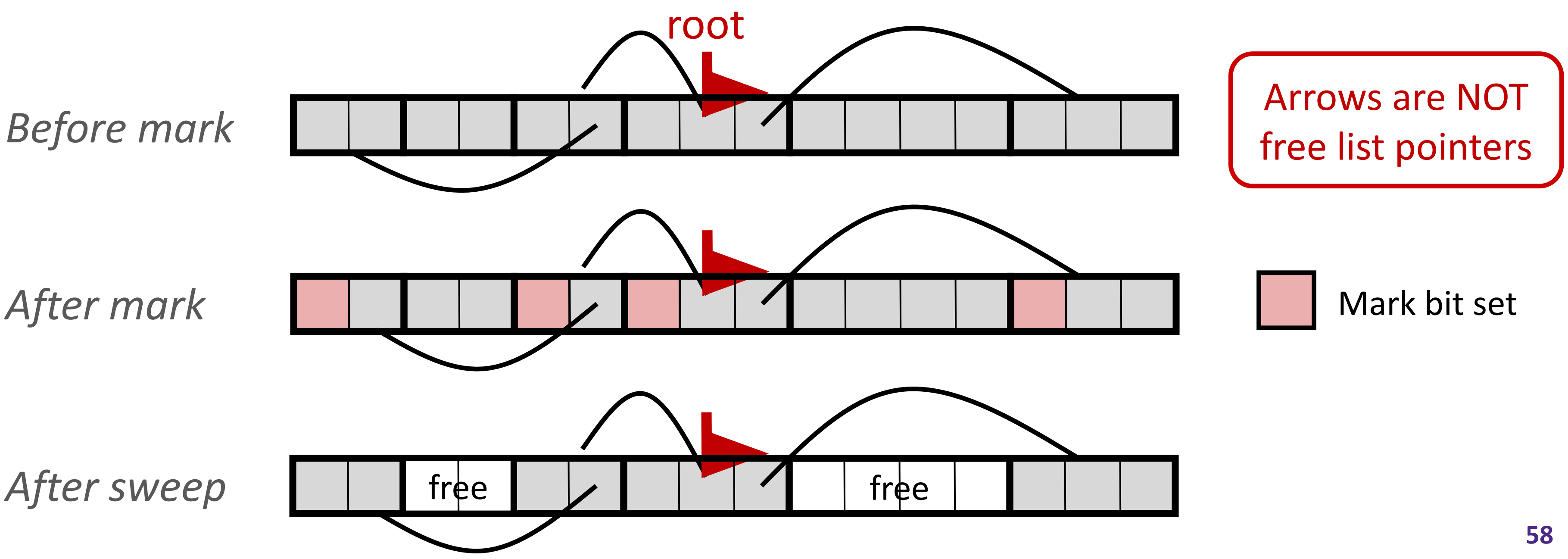
- We'll make some *assumptions* about pointers:
  - Memory allocator can distinguish pointers from non-pointers
  - All pointers point to the start of a block in the heap
  - Application cannot hide pointers (*e.g.* by coercing them to a `long`, and then back again)

Classical GC Algorithms

- Mark-and-sweep collection (McCarthy, 1960)
  - Does not move blocks (unless you also "compact")
- Reference counting (Collins, 1960)
  - Does not move blocks (not discussed)
- Copying collection (Minsky, 1963)
  - Moves blocks (not discussed)
- Generational Collectors (Lieberman and Hewitt, 1983)
  - Most allocations become garbage very soon, so focus reclamation work on zones of memory recently allocated.
- For more information:
  - Jones, Hosking, and Moss, *The Garbage Collection Handbook: The Art of Automatic Memory Management*, CRC Press, 2012.
  - Jones and Lin, *Garbage Collection: Algorithms for Automatic Dynamic Memory*, John Wiley & Sons, 1996.

Mark and Sweep Collecting

- Can build on top of `malloc`/`free` package
  - Allocate using `malloc` until you "run out of space"
- When out of space:
  - Use extra ***mark bit*** in the header of each block
  - ***Mark:*** Start at roots and set mark bit on each reachable block
  - ***Sweep:*** Scan all blocks and free blocks that are not marked



Assumptions For a Simple Implementation

Non-testable Material

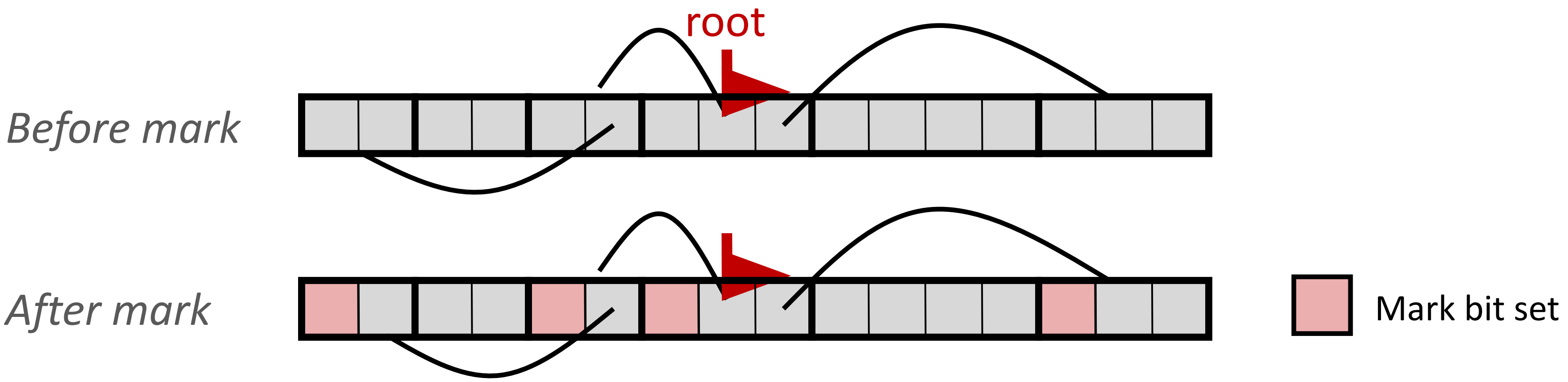
- Application can use functions to allocate memory:
  - `b=new(n)` returns pointer, `b`, to new block with all locations cleared
  - `b[i]` read location `i` of block `b` into register
  - `b[i]=v` write `v` into location `i` of block `b`
- Each block will have a header word (accessed at `b[-1]`)
- Functions used by the garbage collector:
  - `is_ptr(p)` determines whether `p` is a pointer to a block
  - `length(p)` returns length of block pointed to by `p`, not including header
  - `get_roots()` returns all the roots

Mark

Non-testable Material

- Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {                         // p: some word in a heap block
   if (!is_ptr(p))     return;            // do nothing if not pointer
   if (markBitSet(p)) return;             // check if already marked
   setMarkBit(p);                         // set the mark bit
   for (i=0; i<length(p); i++)            // recursively call mark on
      mark(p[i]);                         //    all words in the block
   return;
}
```

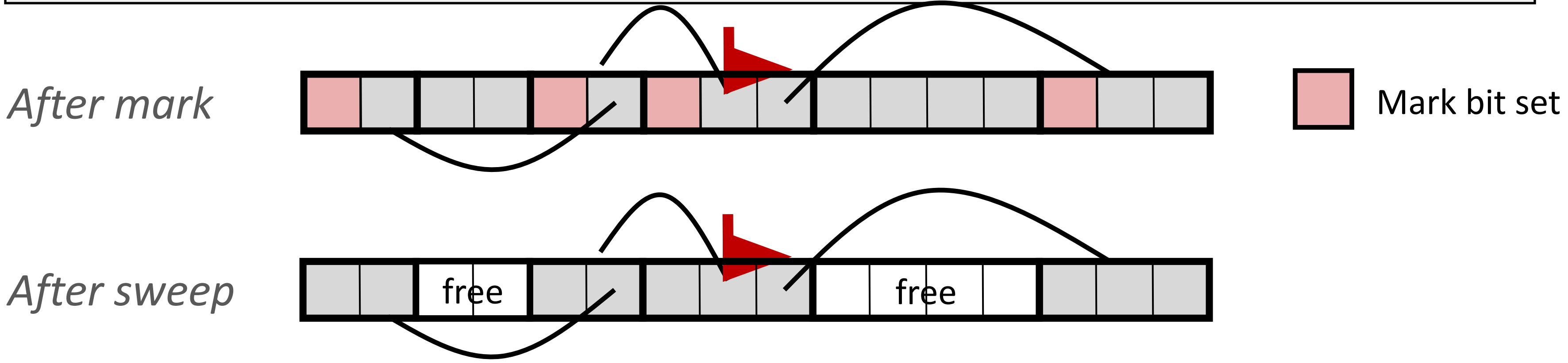


Sweep

Non-testable Material

- Sweep using sizes in headers

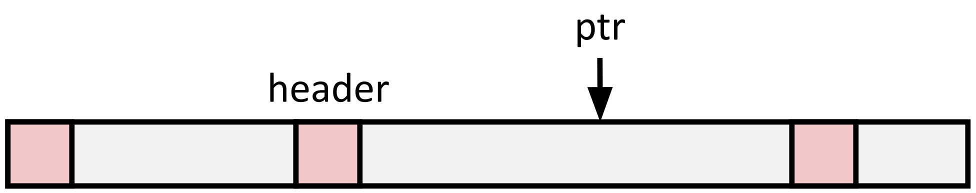
```
ptr sweep(ptr p, ptr end) {              // ptrs to start & end of heap
   while (p < end) {                     // while not at end of heap
      if (markBitSet(p))                 // check if block is marked
         clearMarkBit(p);                // if so, reset mark bit
      else if (allocateBitSet(p))        // if not marked, but allocated
         free(p);                        // free the block
      p += length(p);                    // adjust pointer to next block
   }
}
```



Conservative Mark & Sweep in C

Non-testable Material

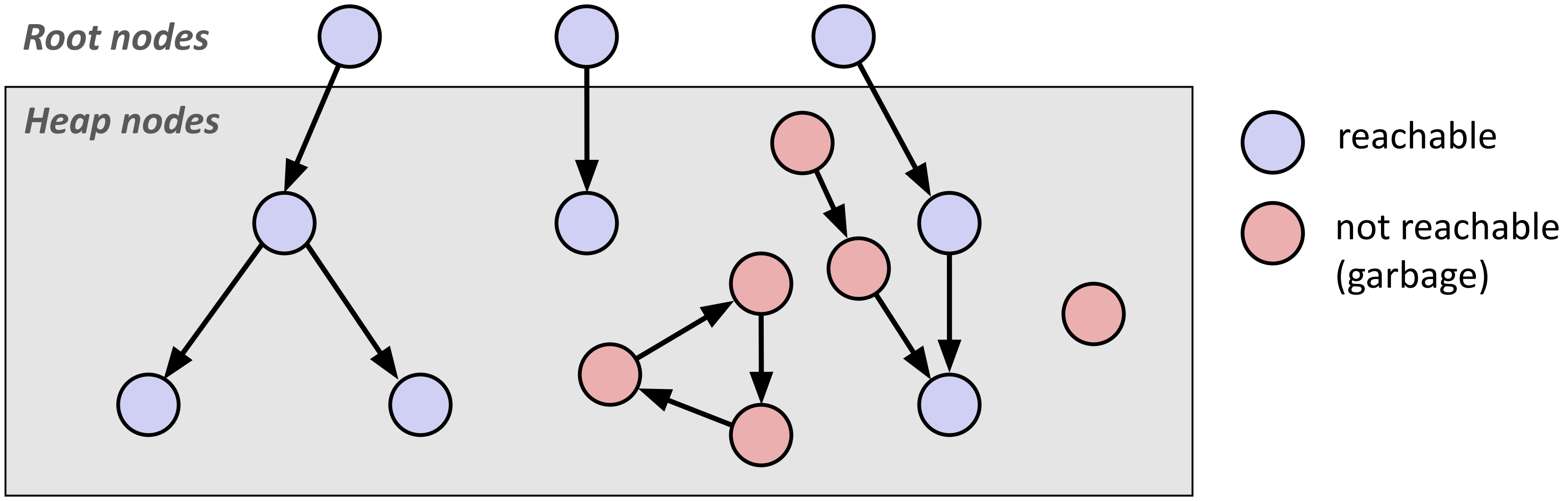
- Would mark & sweep work in C?
  - `is_ptr` determines if a word is a pointer by checking if it points to an allocated block of memory
  - But in C, pointers can point into the middle of allocated blocks (not so in Java)
    - Makes it tricky to find all allocated blocks in mark phase



  - There are ways to solve/avoid this problem in C, but the resulting garbage collector is conservative:
    - Every reachable node correctly identified as reachable, but some unreachable nodes might be incorrectly marked as reachable
  - In Java, all pointers (*i.e.* references) point to the starting address of an object structure – the start of an allocated block

Memory Leaks with GC

- Not because of forgotten `free` — we have GC!
- Unneeded "leftover" roots keep objects reachable
- *Sometimes* nullifying a variable is not needed for correctness but is for performance
- Example: Don't leave big data structures you're done with in a static field


Root nodes
Heap nodes
reachable
not reachable (garbage)