Memory Allocation III

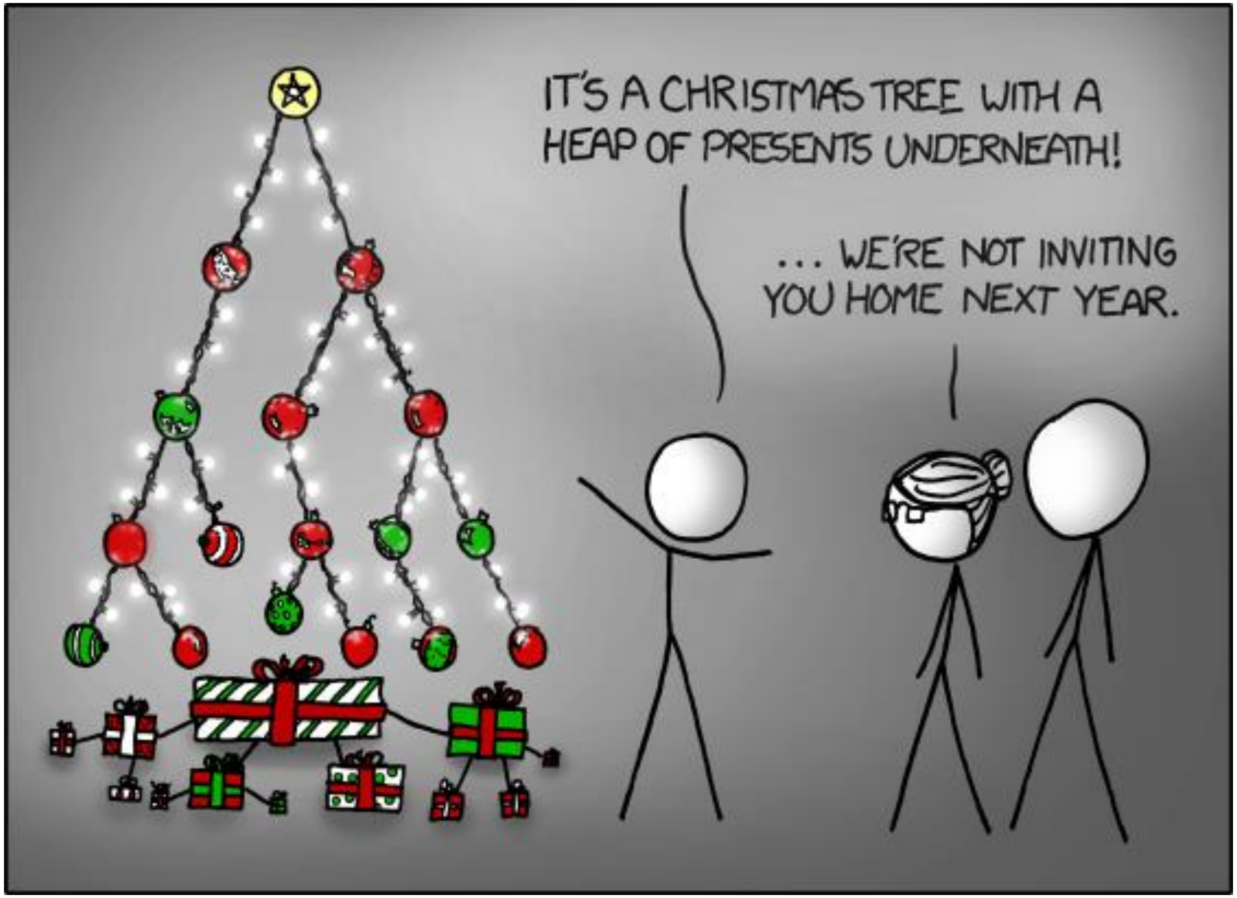
CSE 351 Autumn 2022

Instructor:

Justin Hsia

Teaching Assistants:

Angela Xu
Arjun Narendra
Armin Magness
Assaf Vayner
Carrie Hu
Clare Edmonds
David Dai
Dominick Ta
Effie Zheng
James Froelich
Jenny Peng
Kristina Lansang
Paul Stevans
Renee Ruan
Vincent Xiao



https://xkcd.com/835/

Relevant Course Information

- hw25 due Wednesday (12/7)
- Lab 5 due next Friday (12/9)
  - Recommended that you watch the Lab 5 helper videos
- No readings for next week's lectures!
- **Final Exam:** 12/12-14
  - Similar to midterm; Gilligan's Island Rule in effect
  - Final review section on 12/8
  - Review Session: Fri, 12/9, evening (time TBD) on Zoom
    - More info to be released on Ed Discussion

Lab 5 Hints

- Struct pointers can be used to access field values, even if no struct instances have been created – just reinterpreting the data in memory
- Pay attention to boundary tag data
  - Size value + 2 tag bits – when do these need to be updated and do they have the correct values?
  - The `examine_heap` function follows the implicit free list searching algorithm – don't take its output as "truth"
- Learn to use and interpret the trace files for testing!!!
- A special heap block marks the end of the heap

Explicit List Summary

- Comparison with implicit list:
  - Block allocation is linear time in number of ***free*** blocks instead of ***all*** blocks
    - ***Much faster*** when most of the memory is full
  - Slightly more complicated allocate and free since we need to splice blocks in and out of the list
  - Some extra space for the links (2 extra pointers needed for each free block)
    - Increases minimum block size, leading to more internal fragmentation
- Most common use of explicit lists is in conjunction with *segregated free lists*
  - Keep multiple linked lists of different size classes, or possibly for different types of objects

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 are we willing to tolerate?

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 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**

Reading Review

- Terminology:
  - Garbage collection: mark-and-sweep
  - Memory-related issues in C

- Questions from the Reading?

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 memory

```
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 and C++
  - However, cannot necessarily collect all garbage

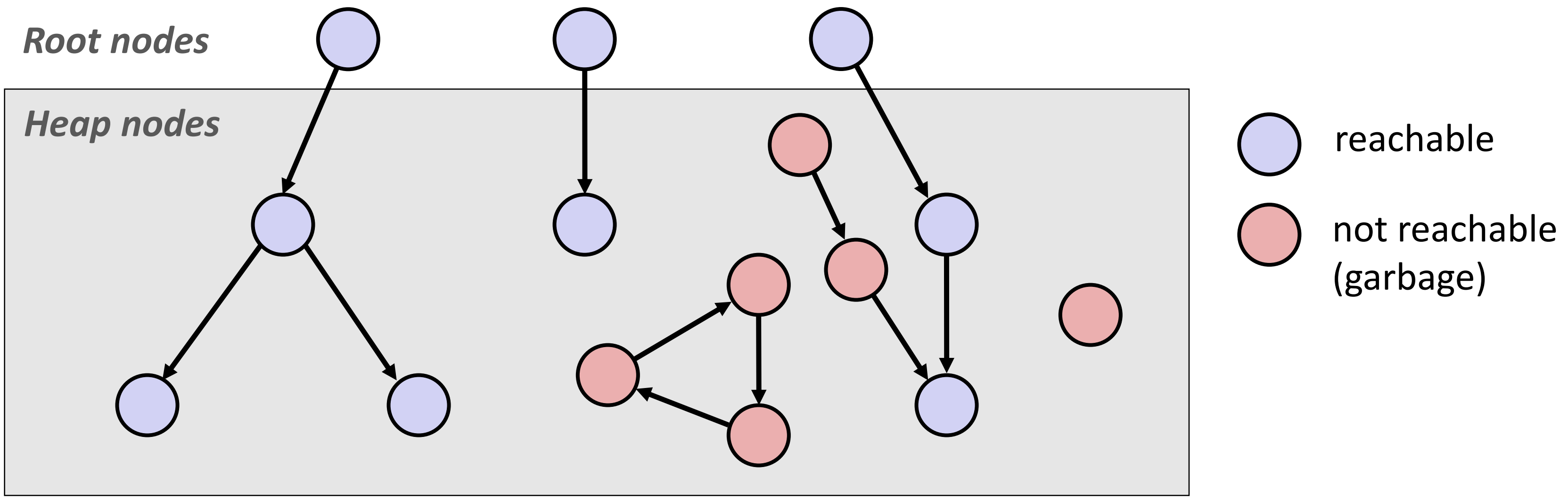
Garbage Collection

- 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?

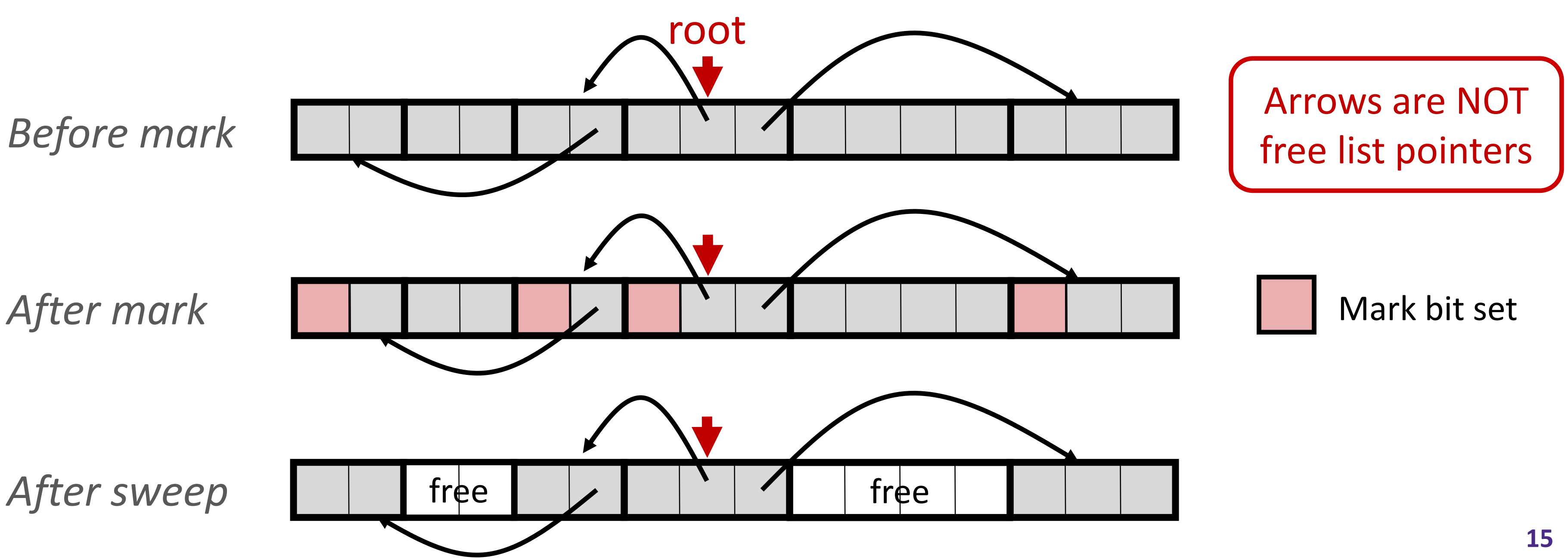
- We'll make some *assumptions* about pointers:
  - Memory allocator can distinguish pointers from non-pointers
  - All pointers point to the start of a block in the heap
  - Application cannot hide pointers (*e.g.*, by coercing them to a `long`, and then back again)

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]`)

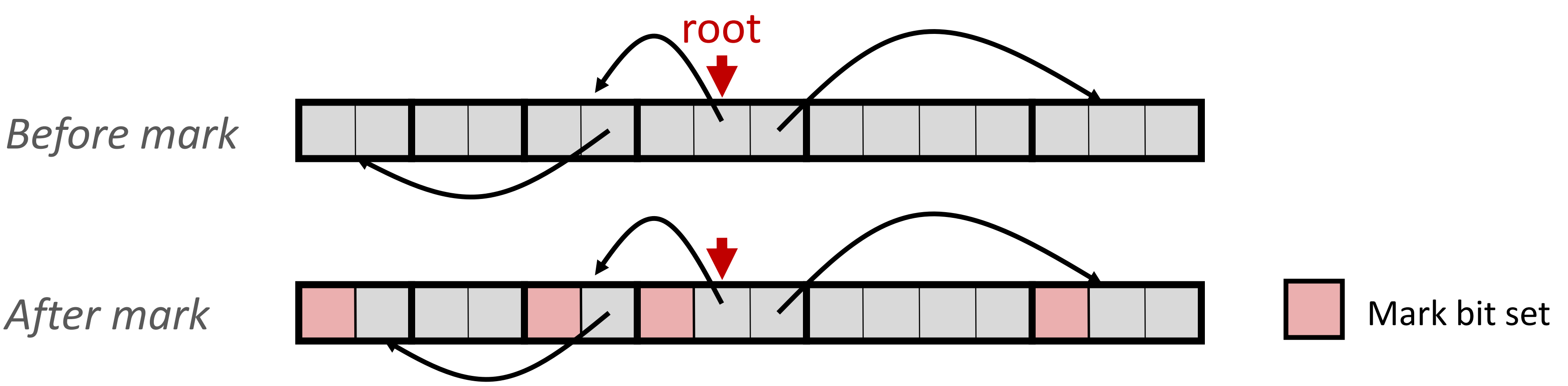
- Functions used by the garbage collector:
  - `is_ptr(p)` determines whether `p` is a pointer to a block
  - `length(p)` returns length of block pointed to by `p`, not including header
  - `get_roots()` returns all the roots

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;
}
```

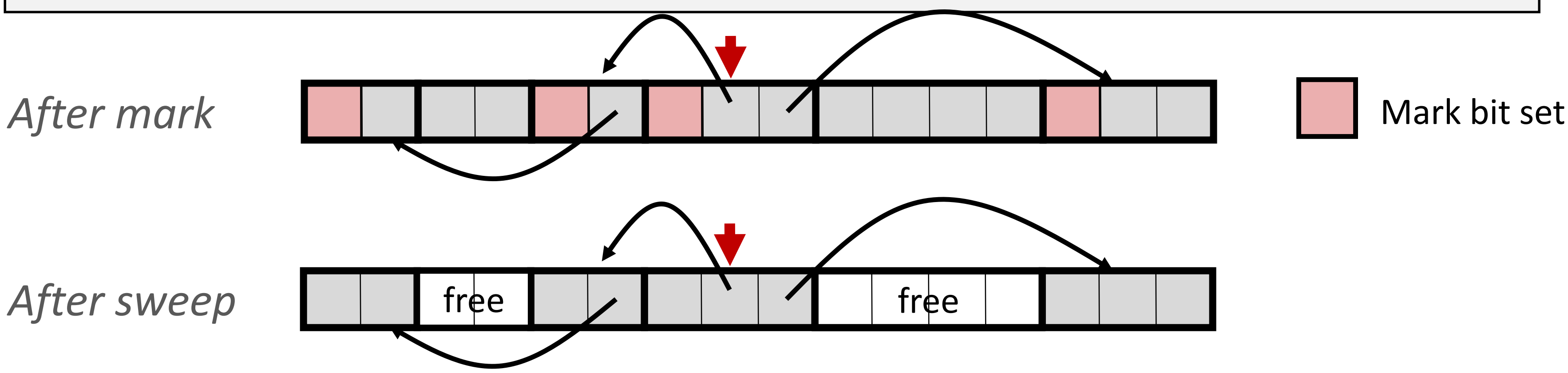


Sweep

Non-testable Material

- Sweep using sizes in headers

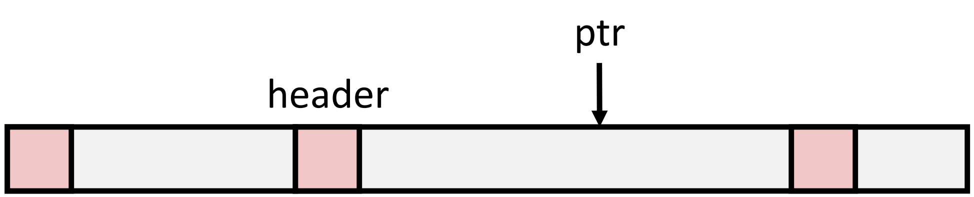
```
ptr sweep(ptr p, ptr end) {              // ptrs to start & end of heap
    while (p < end) {                    // while not at end of heap
        if (markBitSet(p))               // check if block is marked
            clearMarkBit(p);             // if so, reset mark bit
        else if (allocateBitSet(p))      // 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-Related Perils and Pitfalls in C

		Slide	Program stop possible?	Fixes:
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! (Slide 21)

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

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

Error Type:

Prog stop Possible?

Fix:

Find That Bug! (Slide 22)

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

   return &val;
}
```

Error Type:

Prog stop Possible?

Fix:

Find That Bug! (Slide 23)

```
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! (Slide 24)

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

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

Find That Bug! (Slide 25)

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

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

Error Type:

Prog stop Possible?

Fix:

Find That Bug! (Slide 26)

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

   ...

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

Error Type:

Prog stop Possible?

Fix:

Find That Bug! (Slide 27)

```
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]++;
```

Error Type: ☐

Prog stop Possible? ☐

Fix:

Find That Bug! (Slide 28)

```
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;
}
```

Error Type:		Prog stop Possible?		Fix:

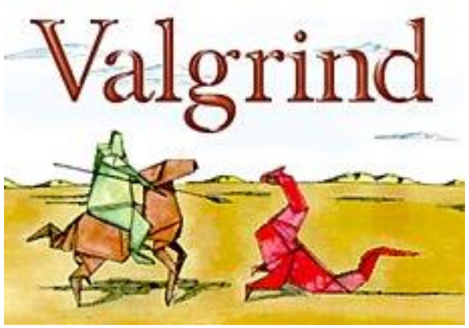
Quick Debugging Note

- Staring at code until you think you spot a bug is generally *not* an effective way to debug!
  - Of course it looks logically correct to you – you wrote it!
  - Language like C doesn't abstract away memory – it's part of your program state that you need to keep track of
    - Your code will only get longer and more complicated in the future: there's too much to try to keep track of mentally

- Instead, start with bad/unexpected behavior to guide your search
  - Memory bugs/"errors" can be especially tricky because they often don't result in explicit errors or program stoppages

Dealing With Memory Bugs

- Make use of all of the tools available to you:
  - Pay attention to compiler warnings and errors
  - Use debuggers like GDB to track down runtime errors
    - Good for bad pointer dereferences, bad with other memory bugs
  - `valgrind` is a powerful debugging and analysis utility for Linux, especially good for memory bugs
    - Checks each individual memory reference at *runtime* (*i.e.*, only detects issues with parts of code used in a specific execution)
    - Can catch many memory bugs, including bad pointers, reading uninitialized data, double-frees, and memory leaks

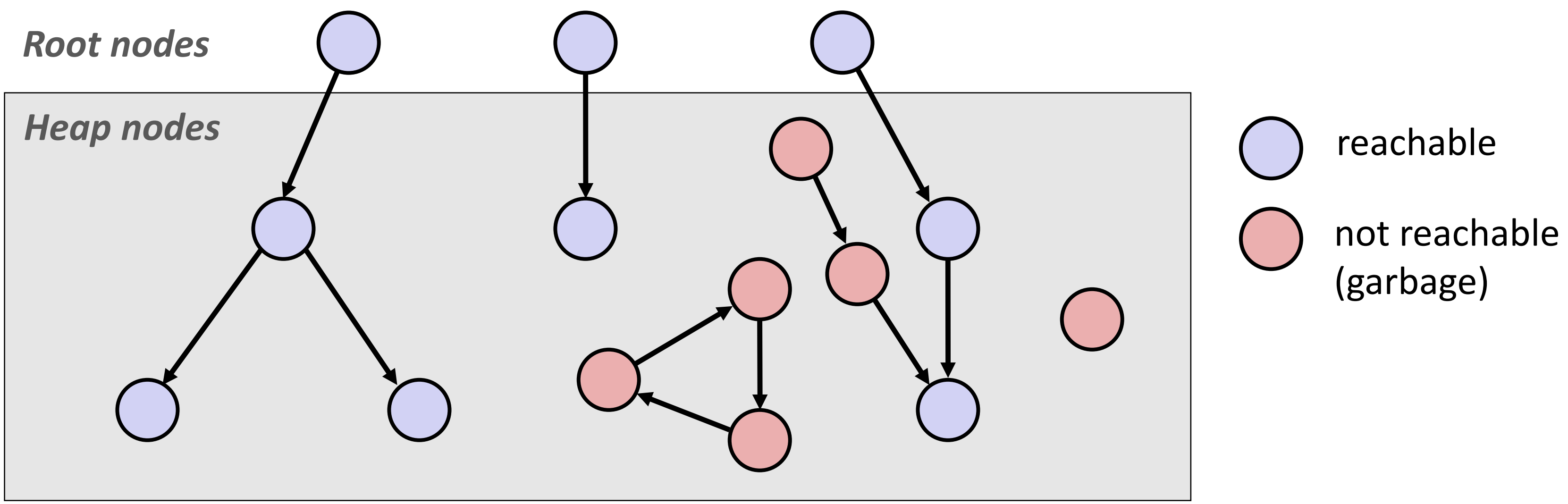


What about Java or ML or Python or …?

- 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?

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



Debugging, Revisited

"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."

– *Memoirs of a Computer Pioneer*
by Maurice Wilkes

Debugging Strategies

- You've got to find what works best for you
- Try a lot – your debugging technique should grow over time and some techniques will work better for different domains
  - Print debugging
  - Using a debugger
  - Visualizations
  - Generating thorough test cases/suites
  - Including sensible checks throughout your program
  - etc.
- But this isn't what we're here to talk about now…

Supporting Yourself While Debugging

- This is also a learning process!
- Why is this necessary (and difficult)?
  - CS actively encourages prolonged periods of mental concentration
    - Easy to tune everything else out when you remain immobile just a few feet from your screen (and screens are getting bigger)
    - Programmers describe sometimes being "in the zone"
    - Long coding sessions and late nights are socially and culturally encouraged
    - Hackathons are designed this way and also encourage you to ignore your bodily needs
    - Tech companies entice you to stay at work with free food and amenities

Supporting Yourself While Debugging

- This is also a learning process!
- Why is this necessary (and difficult)?
  - When your code doesn't work, it can evoke a lot of different negative emotions
  - A heightened emotional state can impede your thinking ability and scope, which can cause you to spiral
    - Can interact with imposter syndrome, stereotype threat, and other self-esteem issues
  - As your mood drops, this can also manifest physically in your body – bad posture, feeling "tense," delaying attending to your needs or forgetting to altogether

Supporting Yourself While Debugging

- **Mindfulness:** "The practice of bringing one's attention in the present moment"
  - Lots of different definitions and nuance, but we'll stick with this broad definition and not the wellness craze

- While debugging, try to be *mindful* of your emotional and physical state as well as your current approach
  - Are you focused on the task at hand or distracted?
  - Am I calm and/or rested enough to be thinking "clearly?"
  - How is my posture, breathing, and tenseness?
  - Do I have any physical needs that I should address?
  - What approach am I trying and why? Are there alternatives?

Supporting Yourself While Debugging

- Try: set a timer for <your interval of choice> (*e.g.*, 15 minutes) to evaluate your state and approach
  - Like the system timer your OS uses for context switching!
- If you're distracted, feeling negative emotions, tense, or need to address something, ***take a break!***
  - You will often find that you'll make a discovery while on a break or at least recover from setbacks
  - Breaks also vary wildly by individual and situation
    - Make sure that you actually feel rested afterward
    - *e.g.*, make tea, work out, do chores, watch a show/movie, play games, chat with friends, make art

Supporting Yourself

- There are few guarantees for support, besides the support that you can give yourself
  - Get comfortable in your own skin and stand up for yourself
  - Can also find support from peers, mentors, family, friends
- Your wellbeing is much more important than your assignment grade, your GPA, your degree, your pride, or whatever else is pushing you to finish *right now*
- Don’t attach too much of your self-worth to programming and debugging
  - There’s so much more that makes you a wonderful and worthwhile human being!