Memory Allocation III

CSE 351 Autumn 2023

Guest Instructor:

Ellis Haker

Teaching Assistants:

Afifah Kashif
Bhavik Soni
Cassandra Lam
Connie Chen
David Dai
Dawit Hailu
Ellis Haker
Eyoel Gebre
Joshua Tan
Malak Zaki
Naama Amiel
Nayha Auradkar
Nikolas McNamee
Pedro Amarante
Renee Ruan
Simran Bagaria
Will Robertson

Relevant Course Information

- HW21 due *Friday* (11/24)
- Lab 4 due Monday (11/27)
- Lab 5 released today, due 12/7 (last Thursday before final)

- Virtual section this week on memory allocation (videos)
- Support hour changes will be posted on Ed tonight

- Looking ahead
  - Final Dec. 11-13, regrade requests Dec. 17-18
  - Check your grades in Canvas as we go

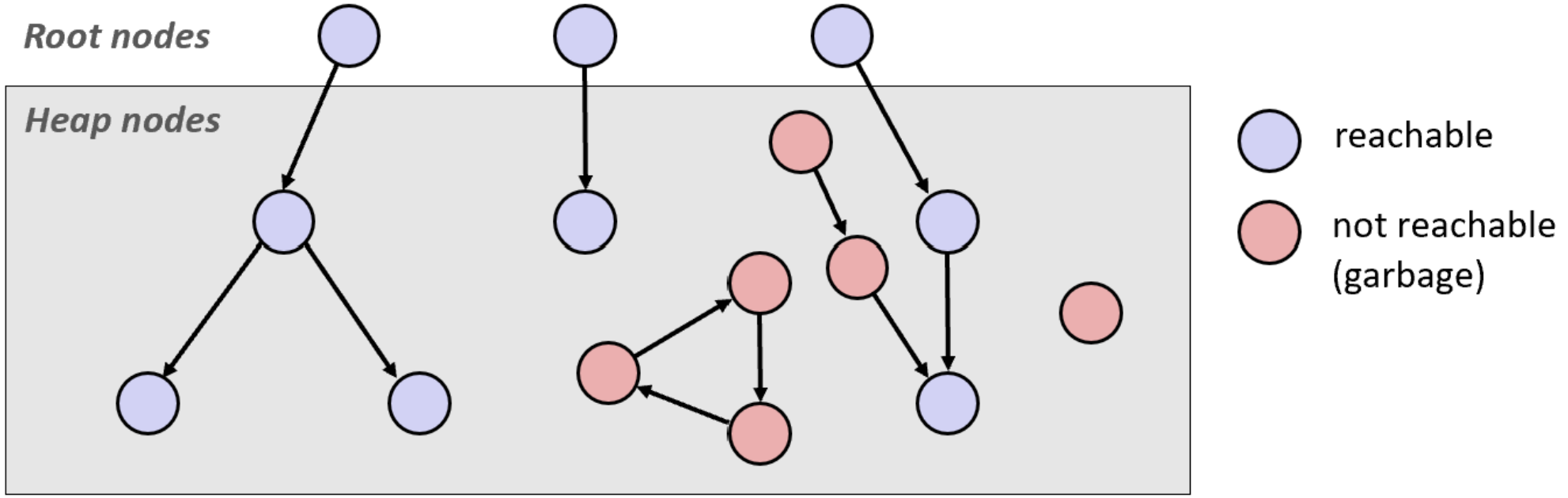
Memory Allocation III

Lesson Summary (1/3)

- **Garbage collection:** automatically freeing space on the heap when no longer needed
  - Part of an **implicit** memory allocator
  - Free any memory no longer reachable by the program
  - Done in many languages (Java, Python, etc.), but *not C!*
  - Why? – C gives us access to raw pointers
    - Pointer arithmetic makes it hard for the compiler to know where a block starts
    - Casting could "hide" a pointer
  - Garbage collectors for C exist, but they're inefficient, not part of the standard

Lesson Summary (2/3)

- **Mark-and-Sweep** is a common way of implementing a garbage collector
  - Store a **mark bit** for each heap block
  1. Start at **root nodes** (local/global variables that point to the heap)
  2. Recursively mark every heap location accessible by the root nodes
  3. Go through every heap block, free any unmarked blocks



Lesson Summary (3/3)

- Common malloc-specific bugs:
  - **Memory leak:** allocating space with malloc, but never freeing it
  - **Double-free:** freeing the same block twice
  - **Accessing a freed block:** using a block after it's been freed
  - **Wrong allocation size:** not allocating enough space for your data

- Any other memory-related bugs also apply here

Lesson Q&A

- Terminology:
  - Garbage collection: mark-and-sweep
  - Memory-related issues in C
- Learning Objectives:
  - Explain the tradeoffs between different allocator implementations, policies, and strategies.
  - Identify and debug issues such as memory leaks, incorrect pointer use, or buffer overflow in C programs.
- What lingering questions do you have from the lesson?

Memory Allocation III – Practice

Memory-Related Perils and Pitfalls in C

- Dereferencing a non-pointer
- Accessing a freed block
- Double-free
- Memory leak
- No bounds checking
- Reading uninitialized memory
- Referencing nonexistent variable
- Wrong allocation size

Find That Bug! (Slide 10)

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

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

8 bytes

9 bytes

if return address is overwritten

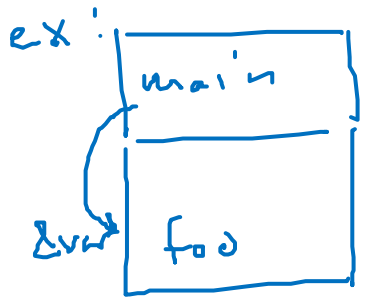
Error: no bounds checking **Program stop?** Y **Fix:** check bounds (fgets)

Find That Bug! (Slide 11)

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

    return &val;
}
```

returns address of local variable



when foo returns, its stack frame is deallocated, main will have a pointer to unallocated mem

Y in GCC – doesn't allow you to return address of local var, returns NULL instead, causes segfault if dereferenced

N elsewhere – returns address of var, but no segfault

Error: using nonexistent var **Program stop?** ? **Fix:** Allocate variable on the heap

Find That Bug! (Slide 12)

ints are ½ the size of pointers, so only N/2 elements allocated, only space for N/2 elements

p is an array of pointers to other arrays

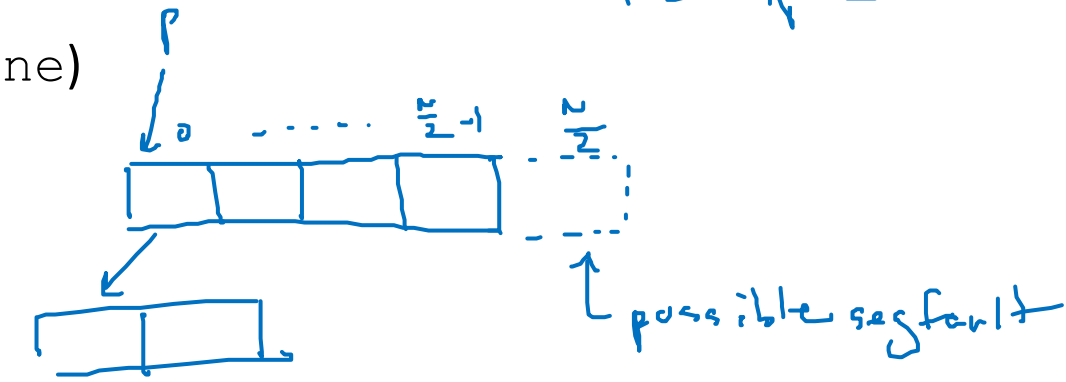
```
int** p;

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

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

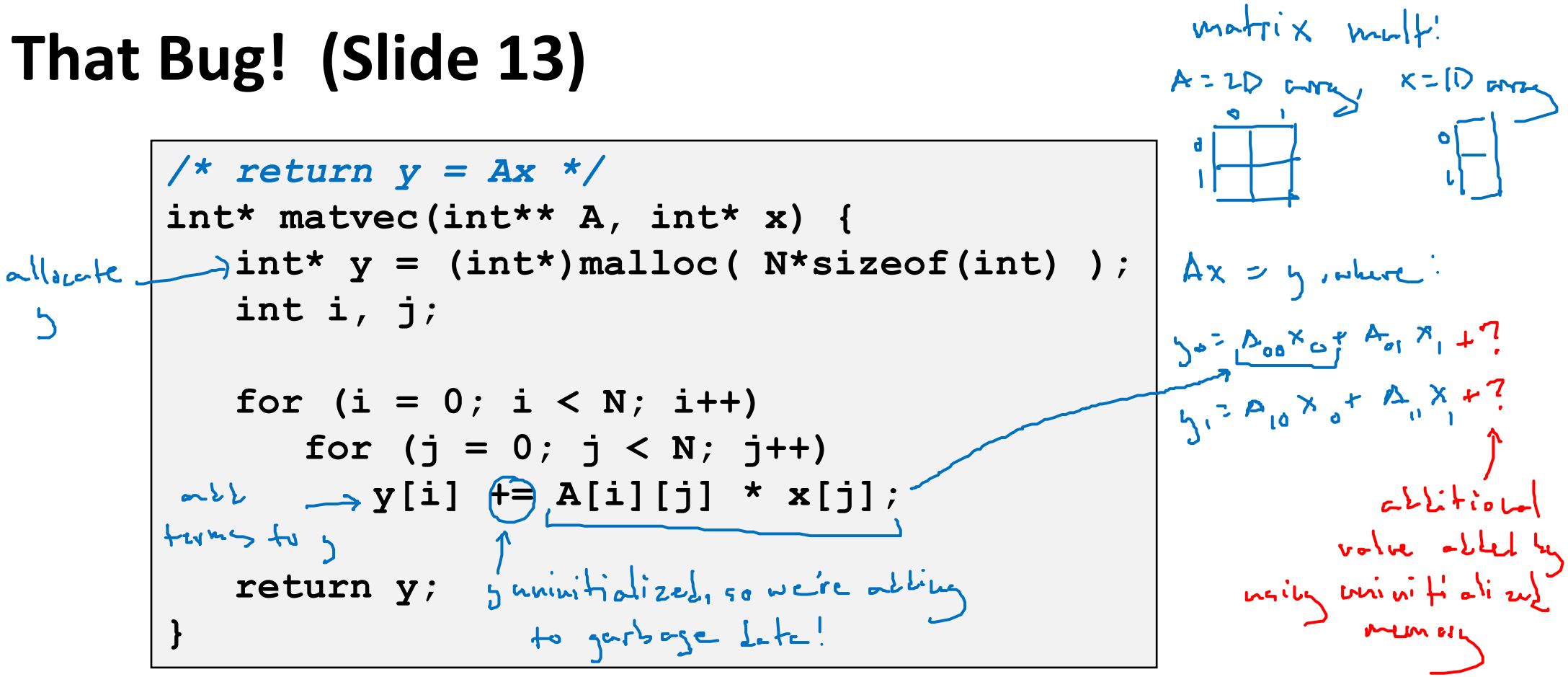
loop tries to access N elements in p, but we've only allocated space for N/2, could cause segfault when we go past allocated space

- N and M defined elsewhere (#define)



Error: wrong allocation size **Program stop?** Y **Fix:** p = malloc(N*sizeof(int*));

Find That Bug! (Slide 13)



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

Error: reading uninitalized mem **Program stop?** N **Fix:** Zero out memory (calloc)

Find That Bug! (Slide 14)

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

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

expects a pointer, but we passed in an int! scanf will dereference val, which does not contain a valid address - segfault

scanf arguments:
- format string
  - tells scanf how to parse input from terminal
- pointer variables
  - one for every format specifier in first arg
  - tells scanf where to store data it reads from terminal

Error: dereferencing non-pointer **Program stop?** Y **Fix:** Use &val instead

Find That Bug! (Slide 15)

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

   ...

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

free x

free x again

in GCC-4, aborts program

other compilers – undefined, can cause security vulnerabilities

Error: Double-free **Program stop?** ? **Fix:** free y in the last line, not x

Find That Bug! (Slide 16)

```
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]++;
```

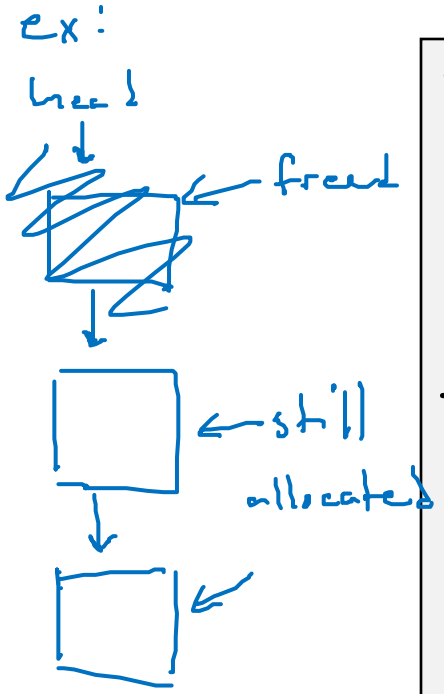
using x after it's free

if that block is marked as free - seg fault

also possible that y was allocated in that block

Error: accessing freed block **Program stop?** Y **Fix:** free x at the end instead

Find That Bug! (Slide 17)




```
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;
}
```

linked list

← only frees head! other nodes not free

loop or recursion

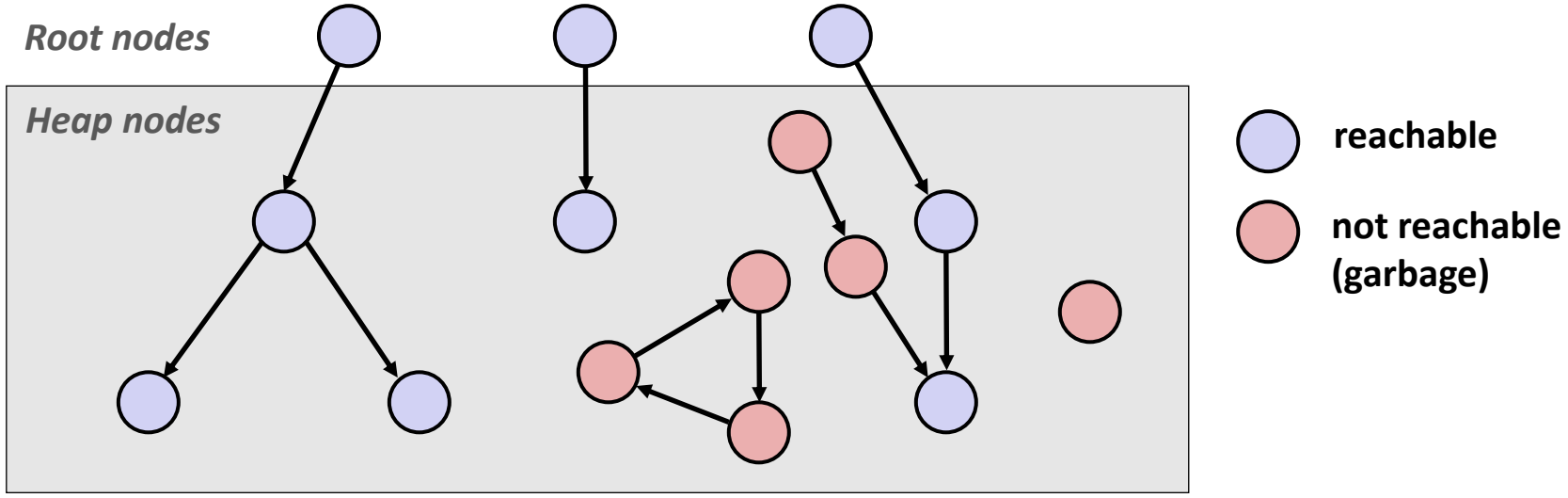
Error: memory leak **Program stop?** N **Fix:** free all allocated nodes

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
  - Nullifying a variable that is no longer in use can improve performance
- Example: Don't leave big data structures you're done with in a static field



Memory Allocation III – Context

Debugging

"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

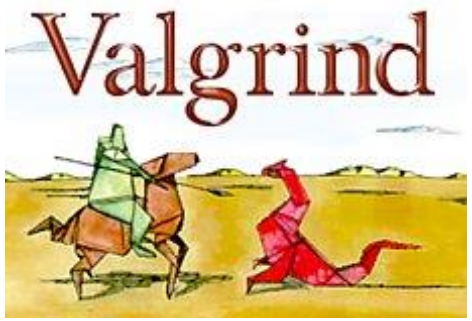
Quick Debugging Note

- 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

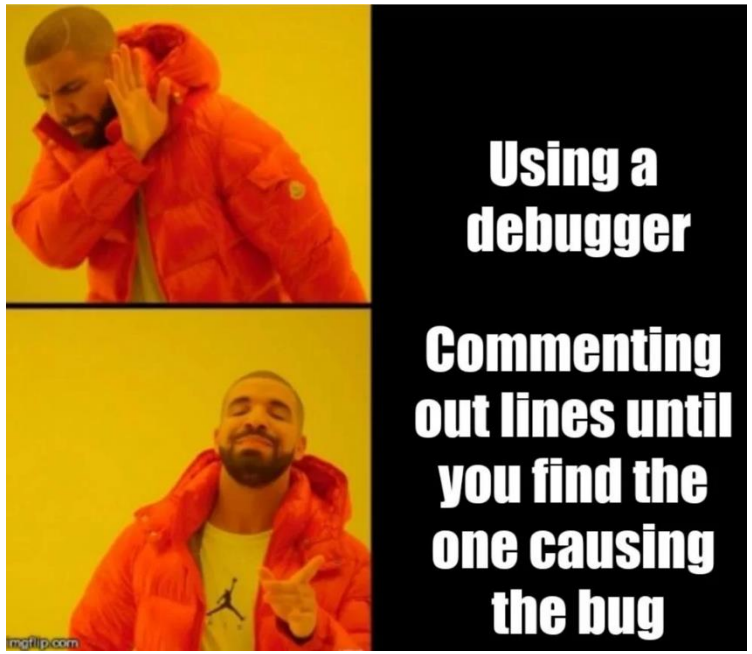
Dealing With Memory Bugs

- 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



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 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)
    - Long coding sessions and late nights are socially and culturally encouraged
      - Hackathons are explicitly designed this way!
    - 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 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 causing you to forget 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

- If you're distracted, feeling frustrated, 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, chat with friends, engage in hobbies, rest

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!

Discussion Question

- Discuss the following question(s) in groups of 3-4 students
  - I will call on a few groups afterwards so please be prepared to share out
  - Be respectful of others' opinions and experiences

- What are your go-to debugging strategies?
- What helps you when you're stuck on a particularly hard problem?

Group Work Time

- During this time, you are encouraged to work on the following:
  1) If desired, continue your discussion
  2) Work on the homework problems
  3) Work on the current lab

- Resources:
  - You can revisit the lesson material
  - Work together in groups and help each other out
  - Course staff will circle around to provide support