The Hardware/Software Interface

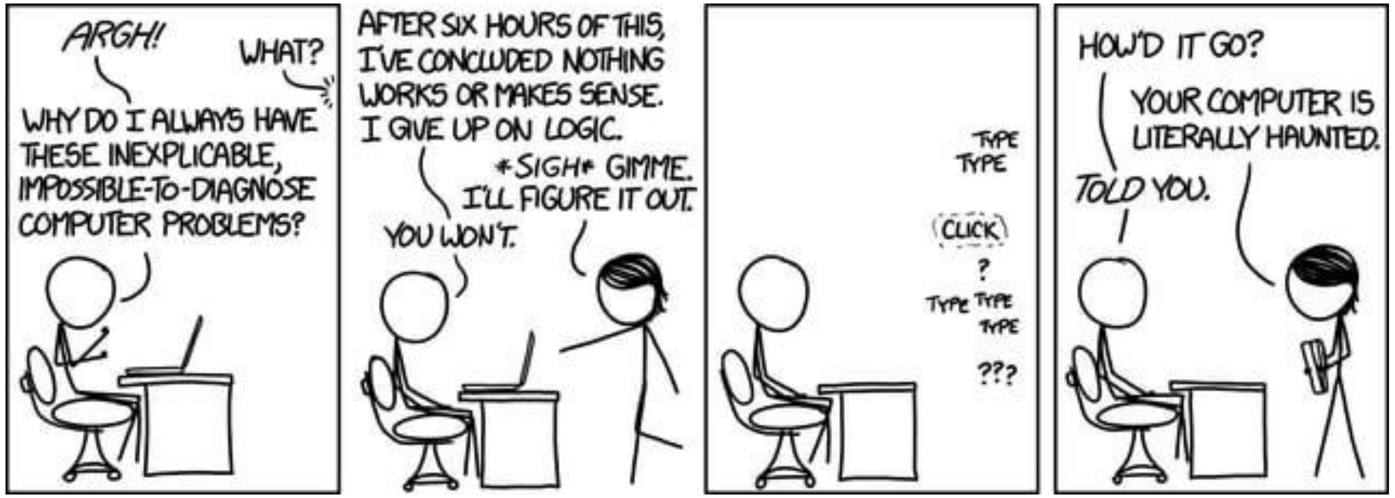
Memory Allocation III

Instructors:

Amber Hu, Justin Hsia

Teaching Assistants:

Anthony Mangus, Grace Zhou, Jiuyang Lyu, Kurt Gu, Mendel Carroll, Naama Amiel, Rose Maresh, Violet Monserate, Divya Ramu, Jessie Sun, Kanishka Singh, Liander Rainbolt, Ming Yan, Pollux Chen, Soham Bhosale


ARGH!
WHAT?
WHY DO I ALWAYS HAVE THESE INEXPLICABLE, IMPOSSIBLE-TO-DIAGNOSE COMPUTER PROBLEMS?
AFTER SIX HOURS OF THIS, I'VE CONCLUDED NOTHING WORKS OR MAKES SENSE. I GIVE UP ON LOGIC.
SIGH GIMME. I'LL FIGURE IT OUT.
YOU WON'T.
TYPE TYPE
CLICK
?
TYPE TYPE TYPE
???
HOW'D IT GO?
YOUR COMPUTER IS LITERALLY HAUNTED.
TOLD YOU.

https://xkcd.com/1316/

Relevant Course Information

- HW 21 due Wed (11/19)
- HW 22 due Fri (11/21)
- Lab 4 due Fri (11/21)
- Lab 5 released today
  - Due Dec. 4 (Thu)

- Looking ahead
  - Finals review session on Dec. 5 (Fri)
    - Last day of classes
  - Final exam on Dec. 10 (Wed)

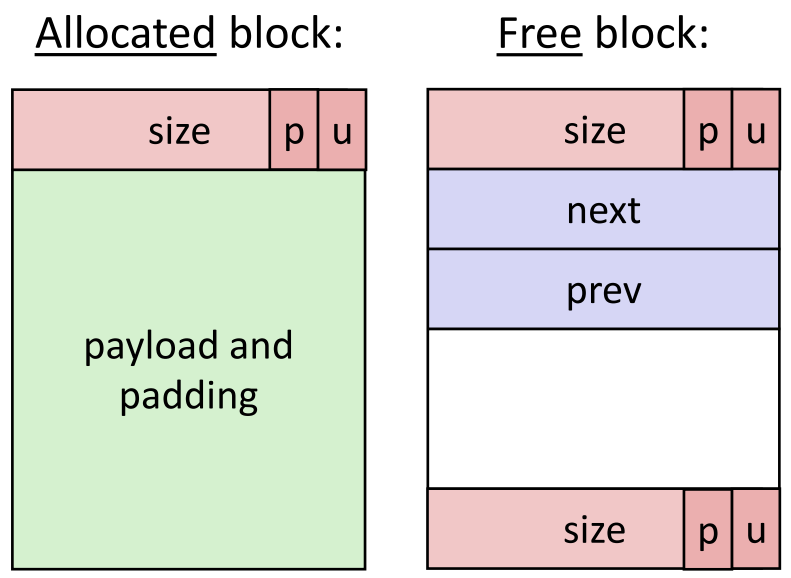
Lecture Outline (1/4)

- **Lab 5 Debugging Practice**
- Garbage Collection
- Memory-Related Issues in C
- Debugging

Lab 5 Boundary Tags

- Boundary Tags
  - Headers and footers are 8 bytes wide
  - Bit 0 (LSB) is *used?* (u)
  - Bit 1 is *preceding-used?* (p)

- Allocated blocks do not have a footer
  - Can determine coalescing with preceding block from your own header's *preceding-used?* bit

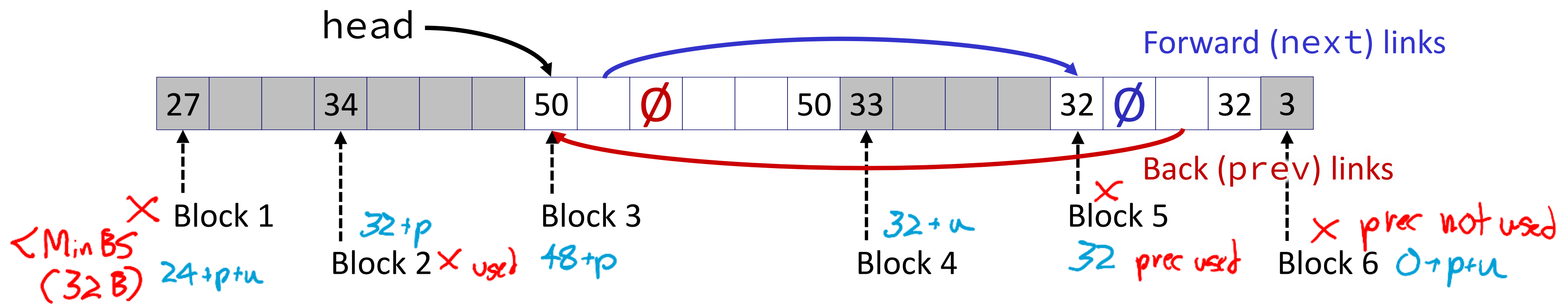


Polling Question

□ = 8-byte word (free)

▨ = 8-byte word (allocated)

- Below is a snapshot of the heap after our buggy Lab 5 implementation runs a sequence of `malloc` and `free` calls. For each of the 6 blocks shown, identify if there is an issue with its current state.
  - Explicit free list with 8-byte boundary tags
    - Allocated block boundary tags: <u>header</u>
    - Free block boundary tags: <u>header</u>, <u>footer</u>
    - Tags: preceding-used? (bit 1), is-used? (bit 0)
  - Alignment: 8



Lecture Outline (2/4)

- Lab 5 Debugging Practice
- **Garbage Collection**
- Memory-Related Issues in C
- Debugging

Garbage Collection

- Automatic memory reclamation of heap-allocated storage
  - The application never explicitly frees memory, but performance usually suffers a bit
  - 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

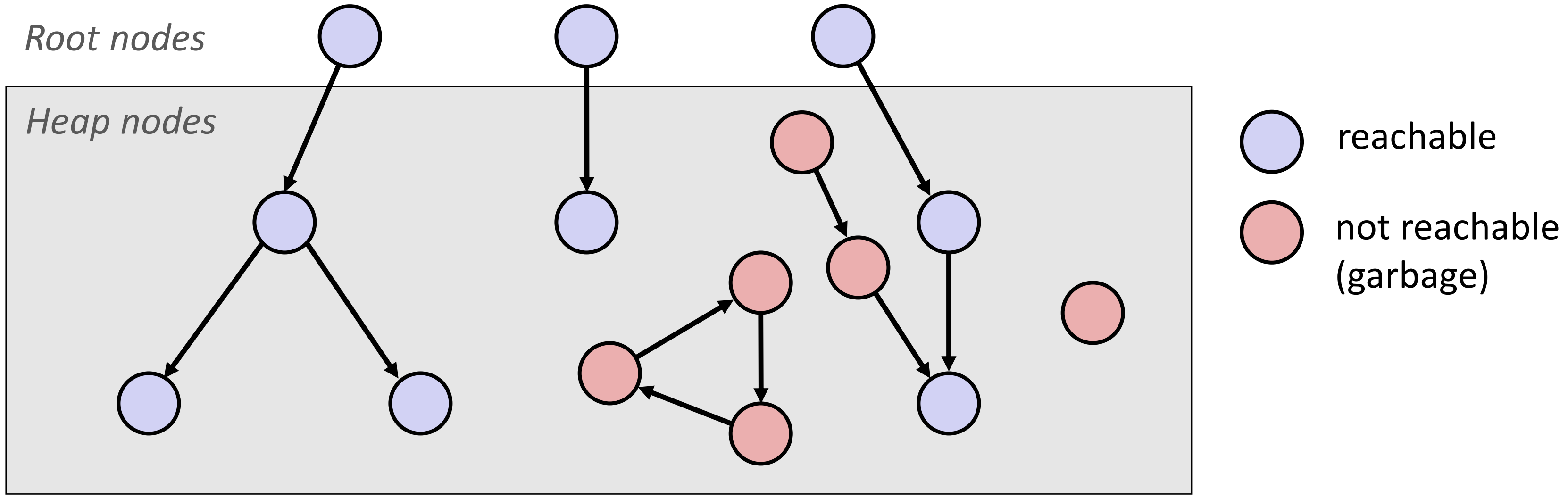
Garbage Collection: How?

- How does the memory allocator know when memory can be deallocated?
  - In general, we cannot know what is going to be used in the future since it depends on control flow (*e.g.*, conditionals)
  - But, we can tell that certain blocks cannot be used if they are *unreachable*:

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

Memory as a Graph (Review)

- We view memory as a *directed graph*:
  - Each allocated heap block is a node; each pointer is an edge
  - Pointers that live outside the heap and point into the heap are called ***root*** nodes



  - 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 in C (Review)

- Memory allocator can deallocate blocks if there are no pointers to them
- We'll make some *assumptions* about pointers:
  - Memory allocator can distinguish pointers from non-pointers ha!
  - 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

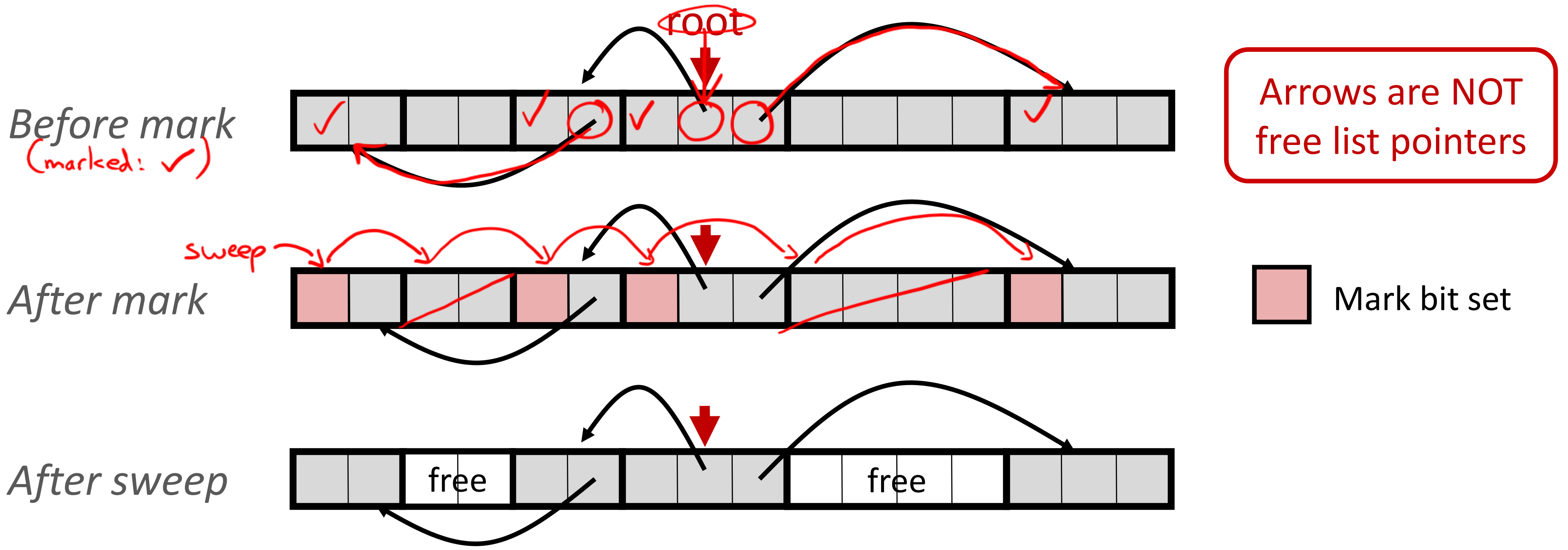
Classical Garbage Collection Algorithms

- **Mark-and-sweep collection** (McCarthy, 1960)
- Reference counting (Collins, 1960)
- Copying collection (Minsky, 1963)
- Generational Collectors (Lieberman and Hewitt, 1983)

- For more information:
  - Jones, Hosking, and Moss, *The Garbage Collection Handbook: The Art of Automatic Memory Management*, CRC Press, 2012.
  - Jones and Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory*, John Wiley & Sons, 1996.

Mark-and-Sweep Collecting (Review)

- Can build on top of `malloc/free` package
- Run garbage collector when you "run out of space" or on periodic cues:
  - 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



Implementation Pseudocode

Non-testable Material

- Garbage collector algorithm:

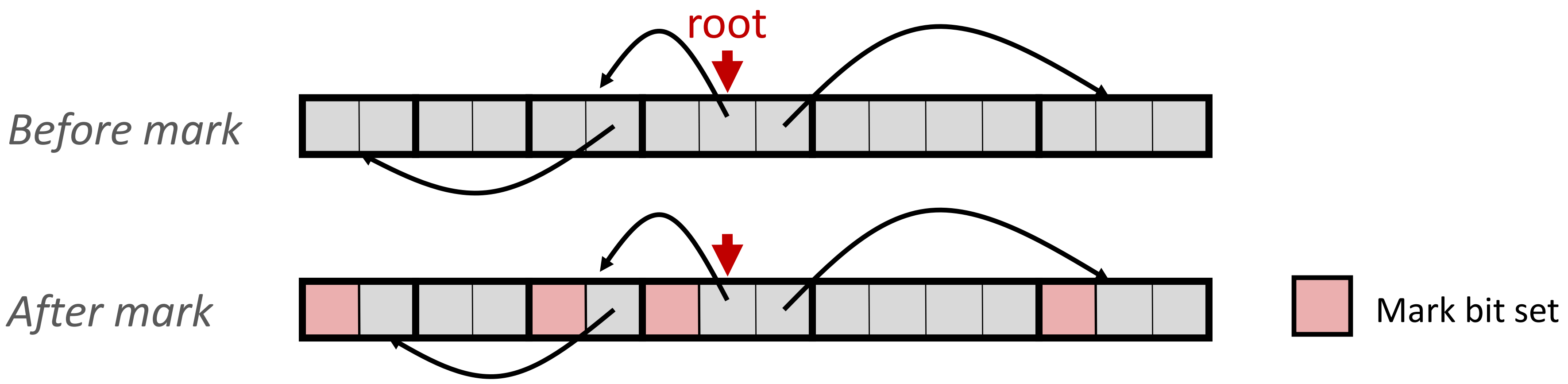
```
ptr heap_start;  // beginning of heap
ptr brk;         // end of heap

void mark_and_sweep() {
   x = get_roots();           // returns all root nodes
   for p in x:                // mark step
      mark(p);
   sweep(heap_start, brk);    // sweep step
}
```

Mark

Non-testable Material

- Mark using depth-first traversal of the memory graph:

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



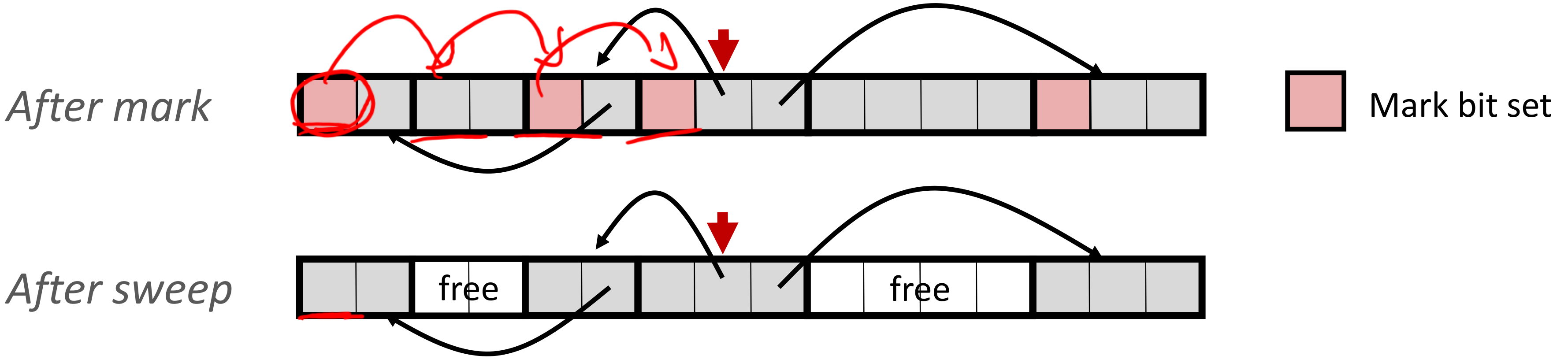
Sweep

Non-testable Material

- Sweep using sizes in headers:

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

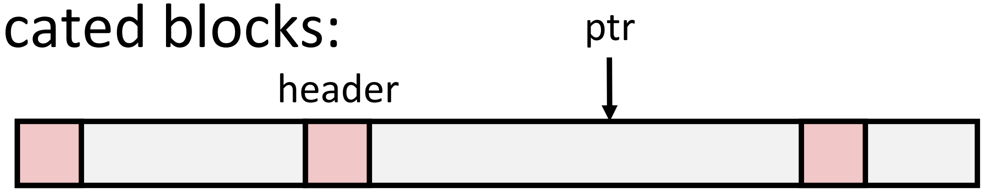
next block



Does Mark-and-Sweep Work?

Non-testable Material

- `is_ptr` determines if a word is a pointer to start of an allocated block in the heap
  - In C: Usually can't tell what's a pointer vs. not and pointers can point into the middle of allocated blocks:



  - In Java: Everything that is not a primitive type is a reference that points to the starting address of an object (*i.e.*, the start of an allocated block)

- Application cannot hide pointers from `is_ptr` and `get_roots`
  - In C: Can freely cast things to change data types
  - In Java: Stricter casting according to inheritance

Lecture Outline (3/4)

- Lab 5 Debugging Practice
- Garbage Collection
- **Memory-Related Issues in C**
- Debugging

Memory-Related Issues in C (Review)

- General pointer issues:
  - Bad/incorrect pointer arithmetic
  - Dereferencing a non-pointer
  - Improper or lack of bounds checking on input or array indices
- General memory issues:
  - Reading/using a deallocated variable
  - Reading/using uninitialized/mystery memory
- Heap-specific issues:
  - Memory leak – failing to free memory
  - Trying to free a freed block ("double free")
  - Trying to access a freed block

Debugging with Memory

- 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

- Memory bugs/"errors" can be especially tricky because they often don't result in explicit errors or program stoppages

Find That Bug! Example

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

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

Error:
Bad pointer arithmetic
Dereferencing a non-pointer
Bad bounds checking
Using a deallocated variable
Using uninitialized memory
Memory leak
Double free
Accessing a freed block

Program stop?
Yes
Sometimes
No

Fix:

Find That Bug! Example Solution

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

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

fgets(stdin, 8, s);

Error:
- Bad pointer arithmetic
- Dereferencing a non-pointer
- **Bad bounds checking**
- Using a deallocated variable
- Using uninitialized memory
- Memory leak
- Double free
- Accessing a freed block

Program stop?
- Yes
- **Sometimes**
- No

e.g., stack smashing

Fix: Check bounds with `fgets`

Find That Bug! Exercise #1

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

   return &val;
}
```

Error:

Bad pointer arithmetic
Dereferencing a non-pointer
Bad bounds checking
Using a deallocated variable
Using uninitialized memory
Memory leak
Double free
Accessing a freed block

Program stop?

Yes
Sometimes
No

Fix:

Find That Bug! Exercise #1 Solution

```
int* foo() {
   int* valp = ~~0;~~ malloc(sizeof(int));
   *valp = 0;
   return ~~&val;~~ valp;
}
```

Error:

Bad pointer arithmetic
Dereferencing a non-pointer
Bad bounds checking
Using a deallocated variable
Using uninitialized memory
Memory leak
Double free
Accessing a freed block

Program stop?

Yes
Sometimes
No

Depends on if value changed on Stack and what Caller does with address/value

Fix: Use `malloc/calloc` to allocate space on the Heap instead

Find That Bug! Exercise #2

```
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:
Bad pointer arithmetic
Dereferencing a non-pointer
Bad bounds checking
Using a deallocated variable
Using uninitialized memory
Memory leak
Double free
Accessing a freed block

Program stop?
Yes
Sometimes
No

Fix:

Find That Bug! Exercise #2 Solution

```
int** p;
                                     int*
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:
- Bad pointer arithmetic
- Dereferencing a non-pointer
- [✓ **Bad bounds checking**]
- Using a deallocated variable
- Using uninitialized memory
- Memory leak
- Double free
- Accessing a freed block

Program stop?
- Yes
- [✓ **Sometimes**]
- No

Segfault if runs into protected area of the Heap.

Fix: Use correct allocation size: `N*sizeof(int*)`

Find That Bug! Exercise #3

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

   ...

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

Error:
- Bad pointer arithmetic
- Dereferencing a non-pointer
- Bad bounds checking
- Using a deallocated variable
- Using uninitialized memory
- Memory leak
- Double free
- Accessing a freed block

Program stop?
- Yes
- Sometimes
- No

Fix:

Find That Bug! Exercise #3 Solution

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

   ...

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

Error:
- Bad pointer arithmetic
- Dereferencing a non-pointer
- Bad bounds checking
- Using a deallocated variable
- Using uninitialized memory
- Memory leak
- [✓ **Double free**]
- Accessing a freed block

Program stop?
- Yes
- [✓ **Sometimes**]
- No

Undefined behavior, but gcc crashes.

Fix: Fix typo: second one is `free(y)`

Find That Bug! Exercise #4

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

   ...

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

Error:
- Bad pointer arithmetic
- Dereferencing a non-pointer
- Bad bounds checking
- Using a deallocated variable
- Using uninitialized memory
- Memory leak
- Double free
- Accessing a freed block

Program stop?
- Yes
- Sometimes
- No

Fix:

Find That Bug! Exercise #4 Solution

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

   ...

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

Error:
- Bad pointer arithmetic
- Dereferencing a non-pointer
- Bad bounds checking
- Using a deallocated variable
- Using uninitialized memory
- Memory leak
- Double free
- **Accessing a freed block**

Program stop?
- Yes
- **Sometimes**
- No

Undefined behavior

Fix: Move `free(x)` to later (below your last use)

Find That Bug! Exercise #5

```
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 linked list
      ...
   free(head);  // free linked list
}
```

Error:
- Bad pointer arithmetic
- Dereferencing a non-pointer
- Bad bounds checking
- Using a deallocated variable
- Using uninitialized memory
- Memory leak
- Double free
- Accessing a freed block

Program stop?
- Yes
- Sometimes
- No

Fix:

Find That Bug! Exercise #5 Solution

```
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 linked list
      ...
   free(head);  // free linked list
}
```

Error:
Bad pointer arithmetic
Dereferencing a non-pointer
Bad bounds checking
Using a deallocated variable
Using uninitialized memory
Memory leak
Double free
Accessing a freed block

Program stop?
Yes
Sometimes
No

Fix:
(1) Recurse on `head->next`, then free `head`, OR
(2) Iterate through list by storing `head->next`, then free `head`

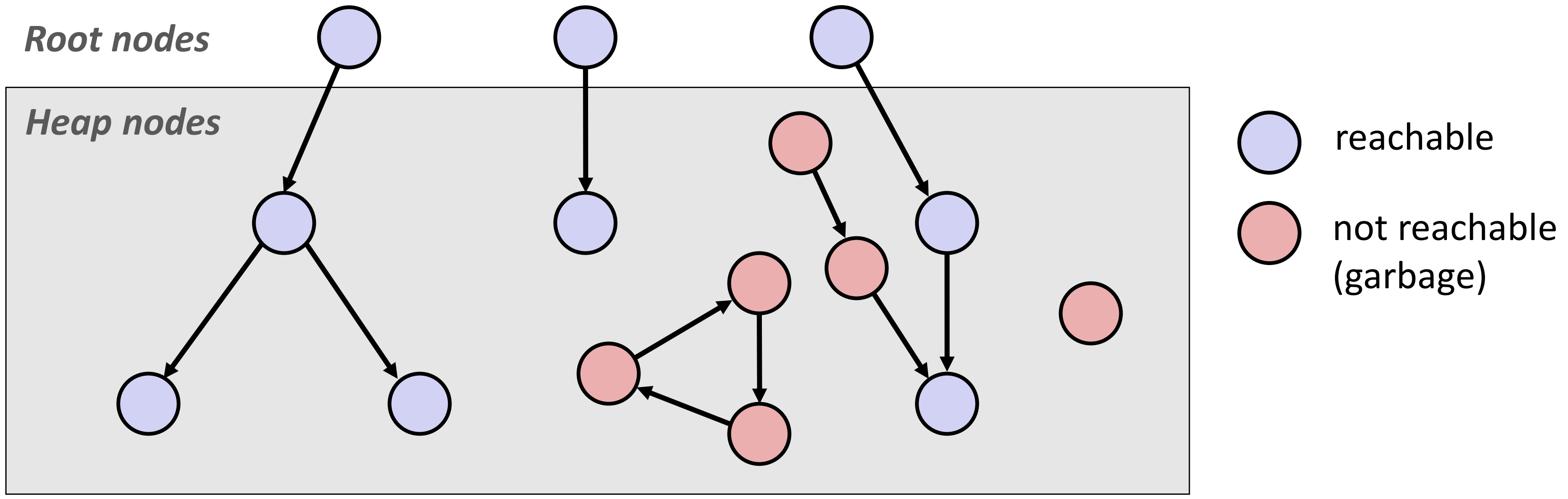
What about Java or Python or Rust or…?

- 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 Garbage Collection

- Not because of forgotten `free` — we have garbage collection!
- 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



Lecture Outline (4/4)

- Lab 5 Debugging Practice
- Garbage Collection
- Memory-Related Issues in C
- **Debugging**

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."

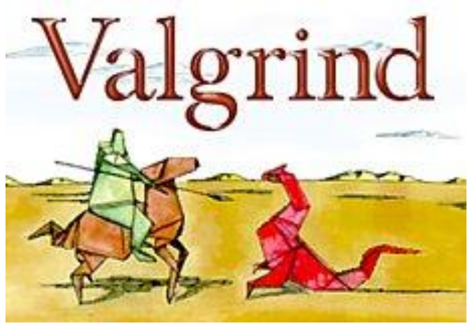
– *Memoirs of a Computer Pioneer* by Maurice Wilkes

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
    - You won't be able to just trace through it in your head from beginning like in CSE 12x
- Instead, start with bad/unexpected behavior to guide your search
  - This is why we like code that crashes early
  - Search bottom-up and not top-down (exhaustive search will take forever)
  - *e.g.*, use `backtrace` on segfaults as a first step

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
  - Including sensible checks throughout your program
  - … and more!
- But this isn’t what we’re here to talk about now…

Supporting Yourself While Debugging: Difficulties

- This is also a learning process! Necessary and sometimes 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 encourage you to ignore your bodily needs
  - Tech companies entice you to stay at work with free food and amenities
- Struggling with your code can evoke a lot of negative emotions
  - A heightened emotional state can impede your thinking ability and scope
    - Can interact with impostor syndrome, stereotype threat, and other self-esteem issues
  - A bad mood can manifest physically, *e.g.*, bad posture, feeling “tense”

Supporting Yourself While Debugging: Mindfulness

- **Mindfulness:** "The practice of bringing one's attention to 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: Taking Breaks

- Try: set a timer for <your interval of choice> (*e.g.*, 15 minutes) to evaluate your state and approach
- 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.*, get a beverage, exercise, 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!

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

- Reflect on the last time you were frustrated or weary while debugging, whether for this class or another. What was the issue? How did you go about solving it? Is there something you would try differently?

Summary (1/3)

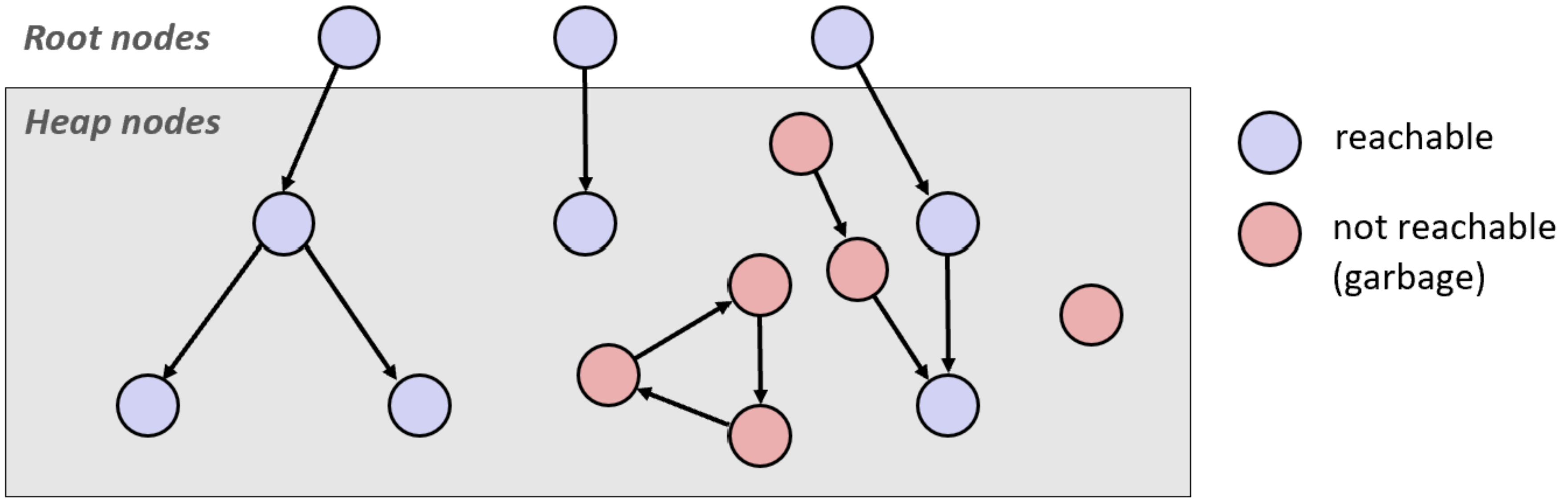
- **Garbage Collection:** automatically freeing space on the heap when no longer needed
  - Part of an **implicit** memory allocator
  - Free any memory no longer reachable by the program's local variables
  - Runs periodically throughout the lifetime of your program

- Done in many languages (Java, Python, etc.), but not *C!*
  - Why not? – C's flexibility comes at a cost
    - Hard to tell what is a pointer and what isn't (casting)
    - Pointers don't always point to the beginning of blocks (pointer arithmetic)
  - Garbage collectors for C exist, but they don't catch everything
    - Not part of the standard library

Summary (2/3)

- **Mark-and-Sweep** is a common garbage collection algorithm
  - Stores a **mark bit** for each heap block

1. Start at root nodes (all variables in scope – global variables, stack variables, etc.)
2. Mark all "reachable" heap blocks (from all root nodes)
3. Look through all heap blocks in order, **free any unmarked blocks**



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

- **Debug smarter, not harder**
  - Start from the symptoms and work backwards
    - *e.g.*, use backtrace on a segmentation faults
  - GDB is your friend – helpful for Lab 5 and beyond!