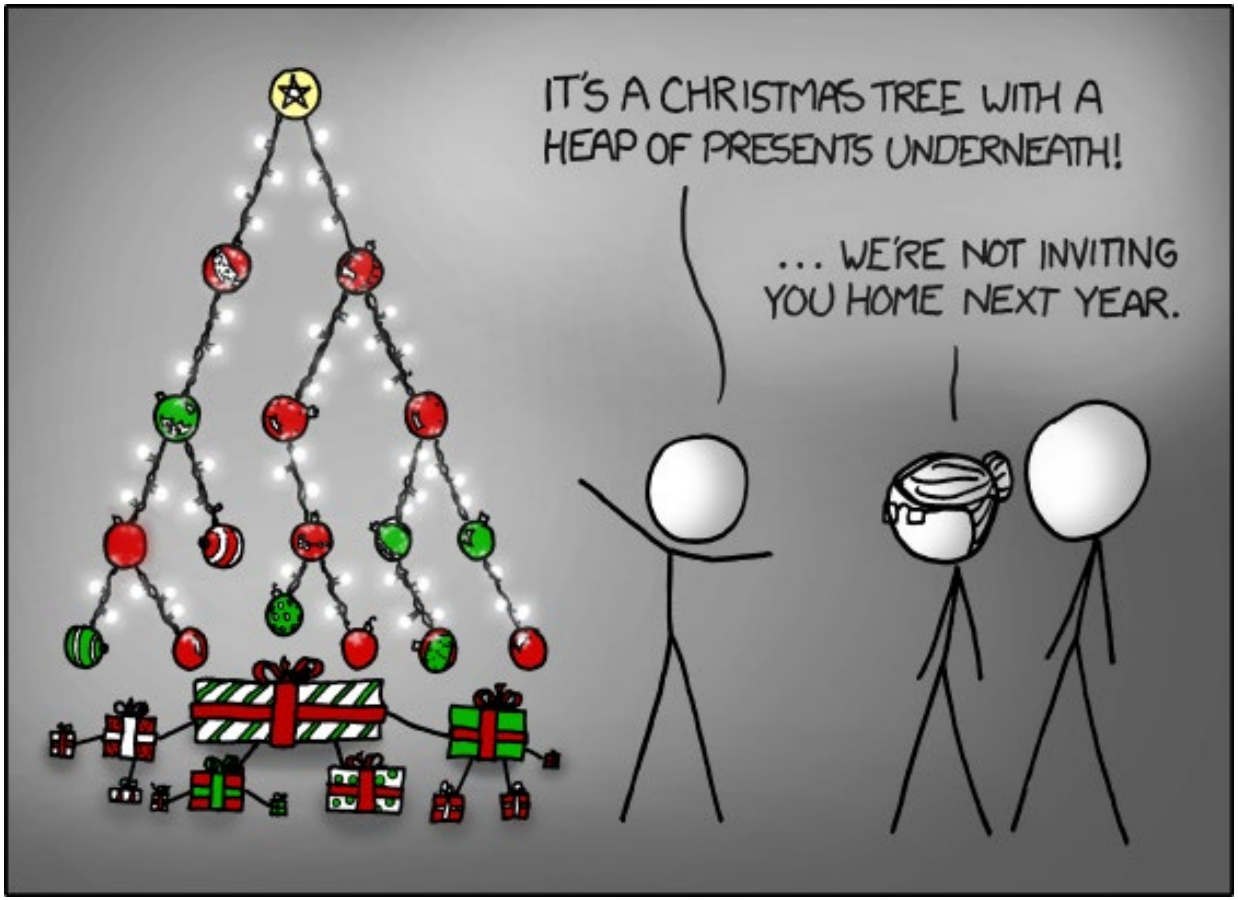
Memory Allocation III

CSE 351 Autumn 2024

Instructor:
Ruth Anderson

Teaching Assistants:
Alexandra Michael
Connie Chen
Chloe Fong
Chendur Jayavelu
Joshua Tan
Nikolas McNamee
Nahush Shrivatsa
Naama Amiel
Neela Kausik
Renee Ruan
Rubee Zhao
Samantha Dreussi
Sean Siddens
Waleed Yagoub



https://xkcd.com/835/

Relevant Course Information

- HW20 due Tonight, Monday (11/18) @ 11:59 pm
- HW21 due Wednesday (11/20) @ 11:59 pm
- Lab 4 due Friday (11/22) @ 11:59 pm
  - Cache parameter puzzles and code optimizations
- HW22 due Friday (11/22) @ 11:59 pm

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**

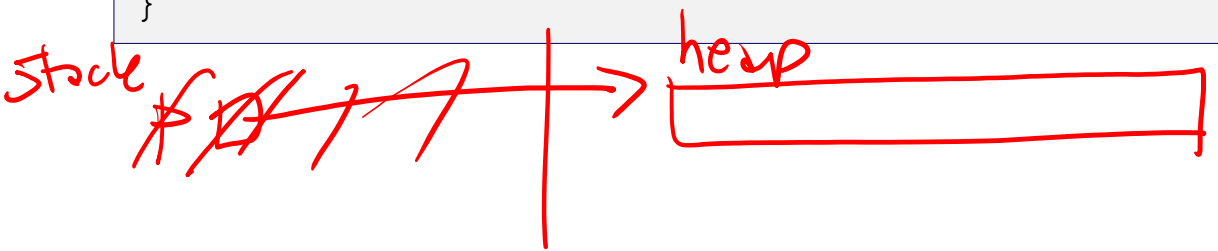
Reading Review

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

What is Garbage Collection?

- **Garbage Collection:** automatically freeing space on the heap when no longer needed
  - Part of an **implicit** memory allocator
    - application never explicitly frees memory!
  - Free any memory no longer reachable by the program's local variables
  - Runs periodically throughout the lifetime of your program

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



Which Languages have Garbage Collection?

- 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
  - Not part of the standard library
  - 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)

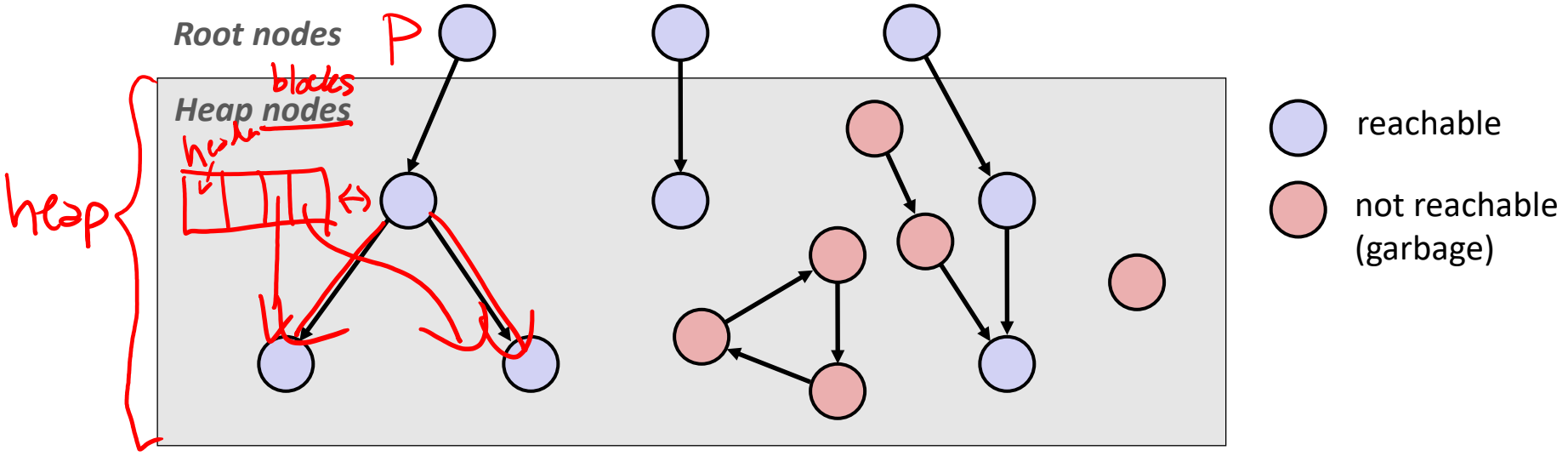
Garbage Collection

- 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 (heap) 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

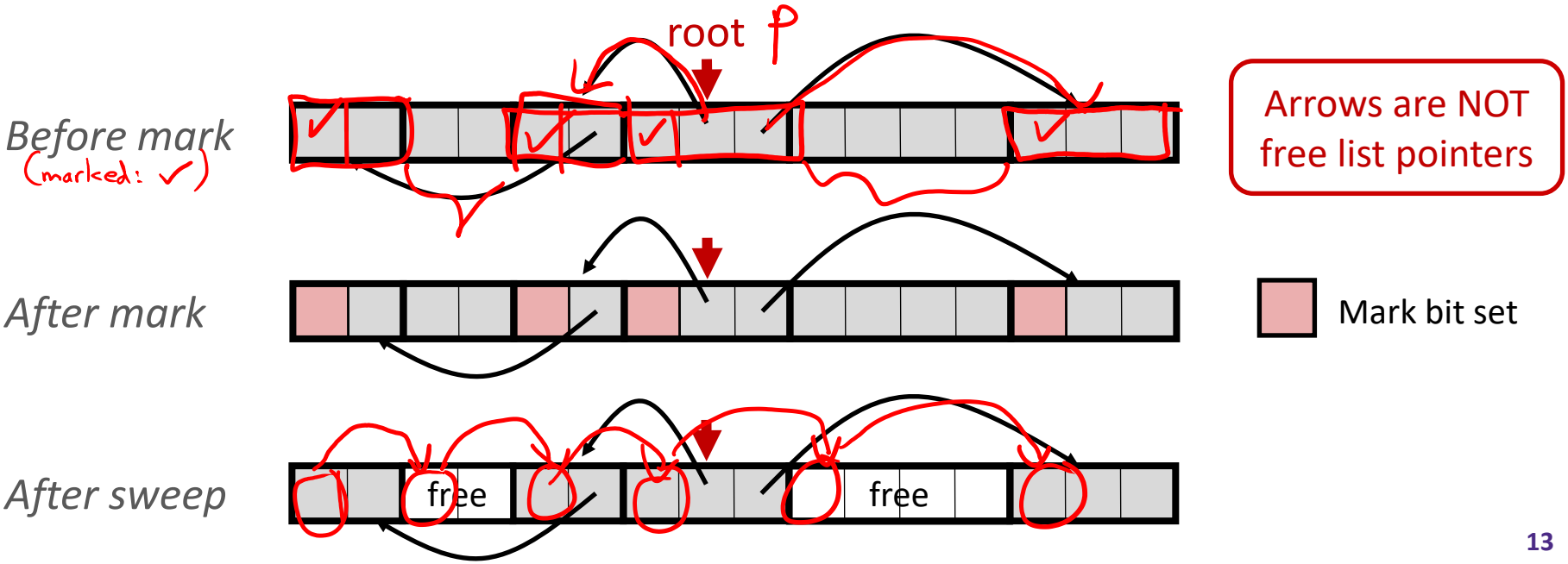
- Dynamic memory allocator can free blocks if there are no pointers to them
- How can it know what is a pointer and what is not?
- We'll make some *assumptions* about pointers:
  - Memory allocator can distinguish pointers from non-pointers
  - All pointers point to the start of a block in the heap
  - 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]`)

the magic that handles our assumptions!

- 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

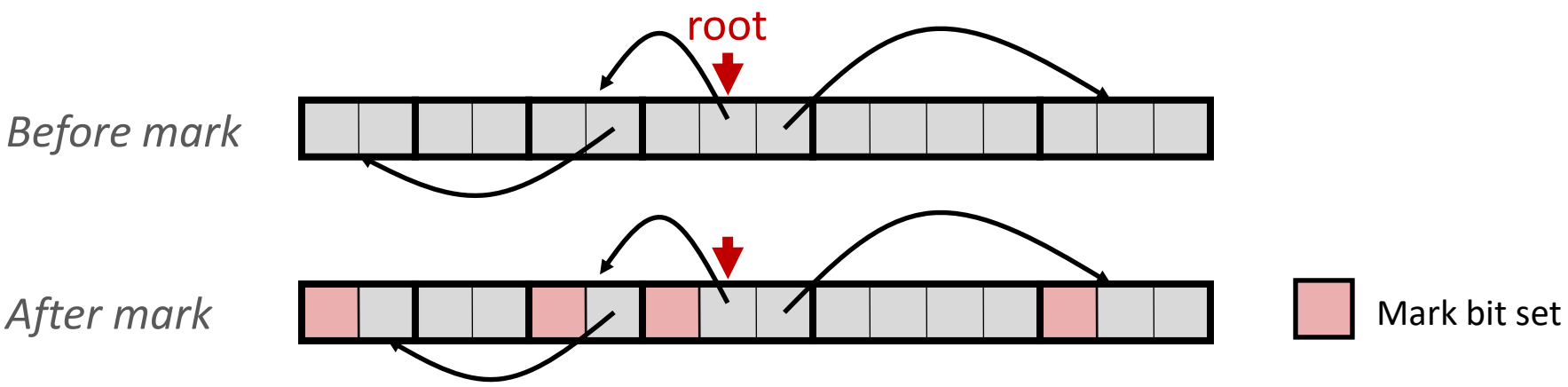
x = get_roots()
for p in x:
mark(p)

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;
}
```

avoids graph cycles and presumably already traversed



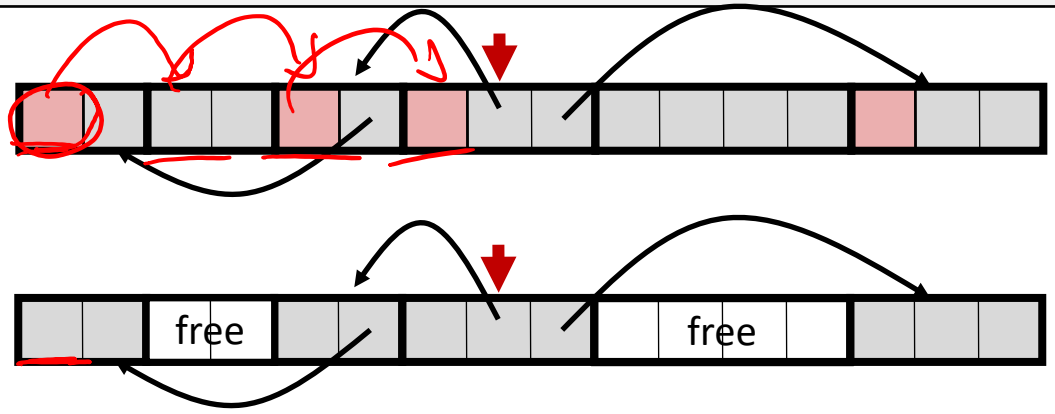
Sweep

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
   }
}
```

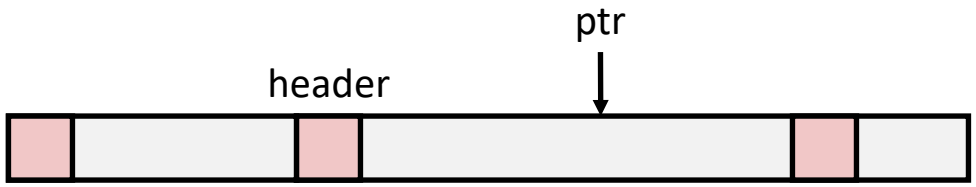
next block



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			

Q1: Find That Bug! (Slide 19)

```
char s[8];   //small buffer
int i;

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

No bounds checking

buffer overflow!

Error Type: E

Prog stop Possible? Y

Fix: fgets(s, 8)

Q2: Find That Bug! (Slide 20)

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

    return &val;
}

void bar() {
  int* addr = foo();
  *addr = 351;
}
```

can't be in a register

Recent versions of gcc will set a returned pointer to something on the stack = NULL. Dereferencing NULL would stop the program.

referencing nonexistent variables

valid address on the stack But

Error Type: G

Prog stop Possible? Y

Fix: pass-by-reference to foo or use malloc instead

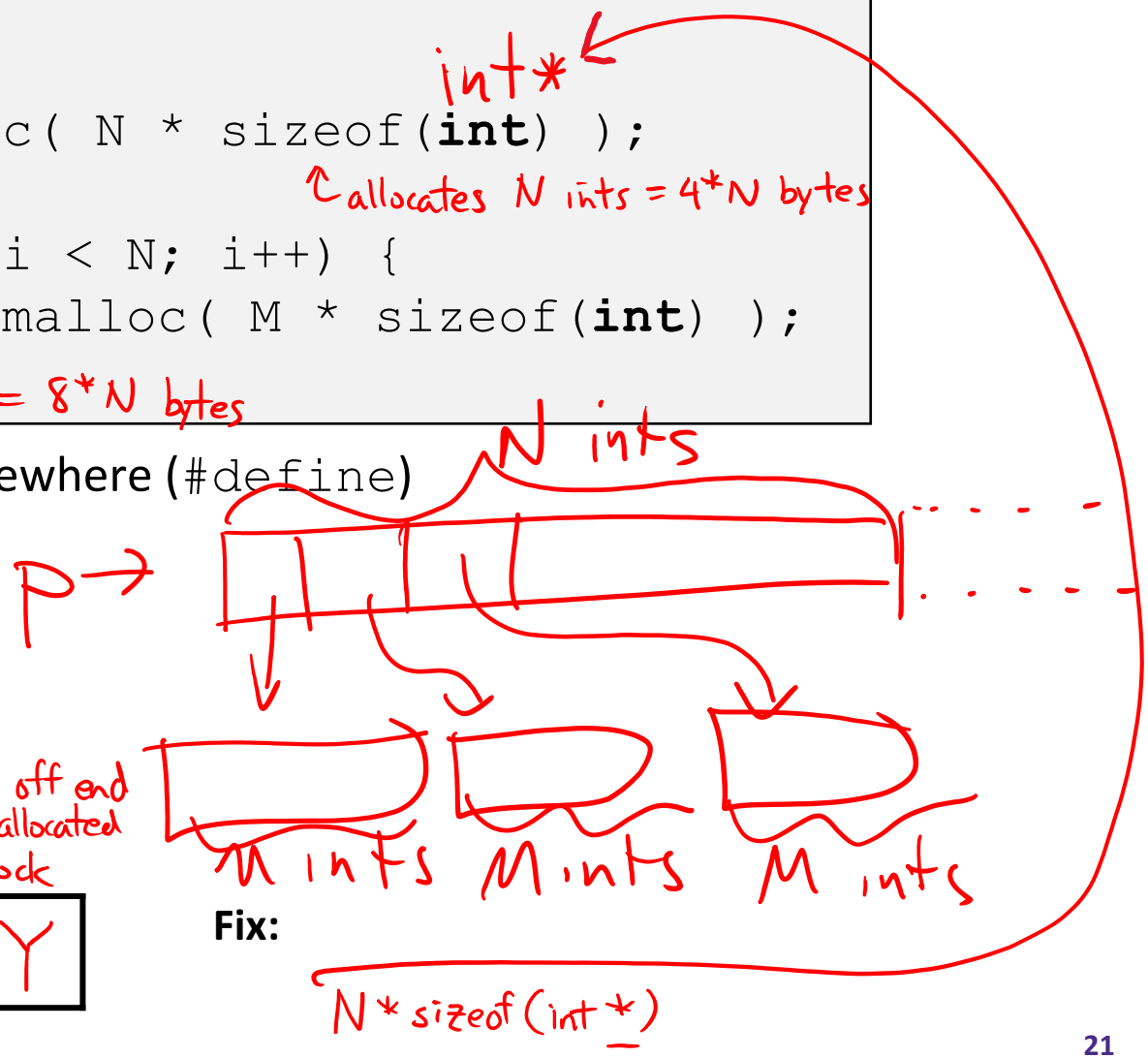
Q3: Find That Bug! (Slide 21)

```
int** p;

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

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

int* ↑ allocates N ints = 4*N bytes

↑ writes to N int* = 8*N bytes

- N and M defined elsewhere (#define)



wrong allocation size

runs off end of allocated block

Error Type: H

Prog stop Possible? Y

Fix: N * sizeof(int*)

Q4: Find That Bug! (Slide 22)

```
/* 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;
}
```

y[i] = y[i] + A[i][j] * x[j];
↑ reads uninitialized values!

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

reading uninitialized memory

just using uninitialized values – runs fine but get weird results

Error Type: F

Prog stop Possible? N

Fix: calloc(N, sizeof(int))

Q5: Find That Bug! (Slide 23)

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

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

address of int

reads input, parses int, stores into location val

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

segfault if val does not contain a valid address

dereferencing a non-pointer

Error Type: A

Prog stop Possible? Y

Fix: `scanf("%d", &val);`

Q6: Find That Bug! (Slide 24)

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

    ...

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

free again

undefined behavior (some systems will segfault)

Error Type: C

Prog stop Possible? Y

Fix: free (y) ← probably a typo

Q7: Find That Bug! (Slide 25)

```
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]++;
```

Freed Block: Access Again

undefined behavior

Error Type: B

Prog stop Possible? Y

Fix: free(x) later (at bottom)

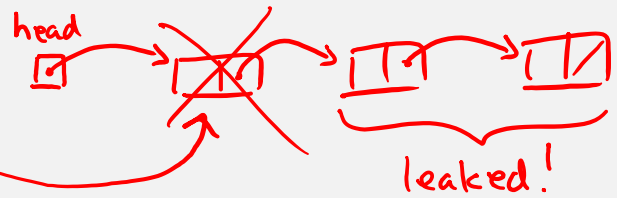
(Not in Ed) Find That Bug! (Slide 26)

```
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;
}
```

node: val next

mallocs here

only frees first node!



how do you detect?

memory leak

Error Type: D

Prog stop Possible? N

Fix: recursive/iterative free over list

Don't Just Stare at Your Code to Debug

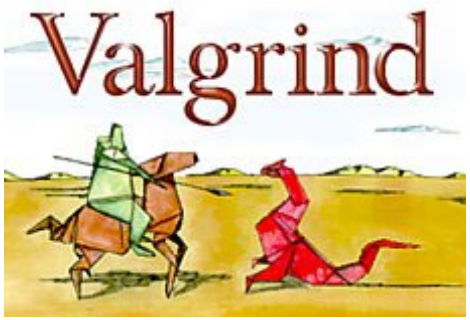
- Staring at code until you think you spot a bug is generally *not* an effective way to debug!
  - Of course it looks logically correct to you – you wrote it!
  - Language like C doesn't abstract away memory – it's part of your program state that you need to keep track of
    - Your code will only get 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

Better Debugging Strategies

- 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 seg faults as a first step
- 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