

Memory Allocation III

CSE 351 Spring 2020

Instructor:

Ruth Anderson

Teaching Assistants:

Alex Olshanskyy

Rehaan Bhimani

Callum Walker

Chin Yeoh

Diya Joy

Eric Fan

Edan Sneh

Jonathan Chen

Jeffery Tian

Millicent Li

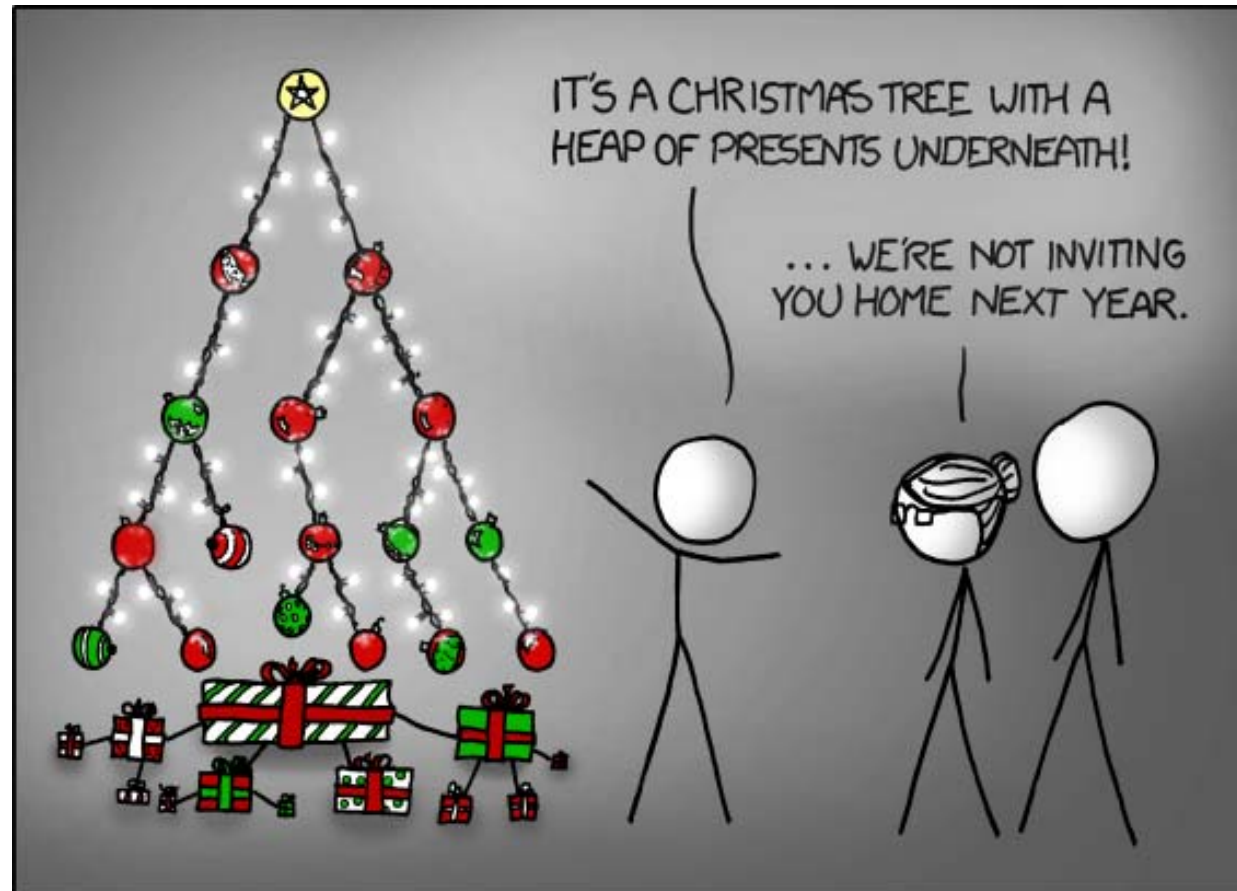
Melissa Birchfield

Porter Jones

Joseph Schafer

Connie Wang

Eddy (Tianyi) Zhou

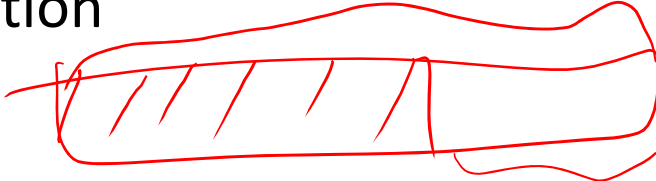


<https://xkcd.com/835/>

Administrivia

- ❖ Unit Summary #3 – due TONIGHT Friday (5/29)
- ❖ Lab 5 (on Mem Alloc) due the last day of class (6/05)
 - Understanding the concepts *first* and efficient *debugging* will save you lots of time
 - Light style grading
 - Can be submitted at most ONE day late. (Sun 6/07)
 - hw22 due Monday (6/01) – Do EARLY, will help with Lab 5
- ❖ **You must log on with your @uw google account to access!!**
 - **Google doc** for 11:30 Lecture: <https://tinyurl.com/351-05-29A>
 - **Google doc** for 2:30 Lecture: <https://tinyurl.com/351-05-29B>

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?
- ❖ 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

we've assumed this up to now

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**

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! */  
}
```

stack → *heap*
p is deallocated

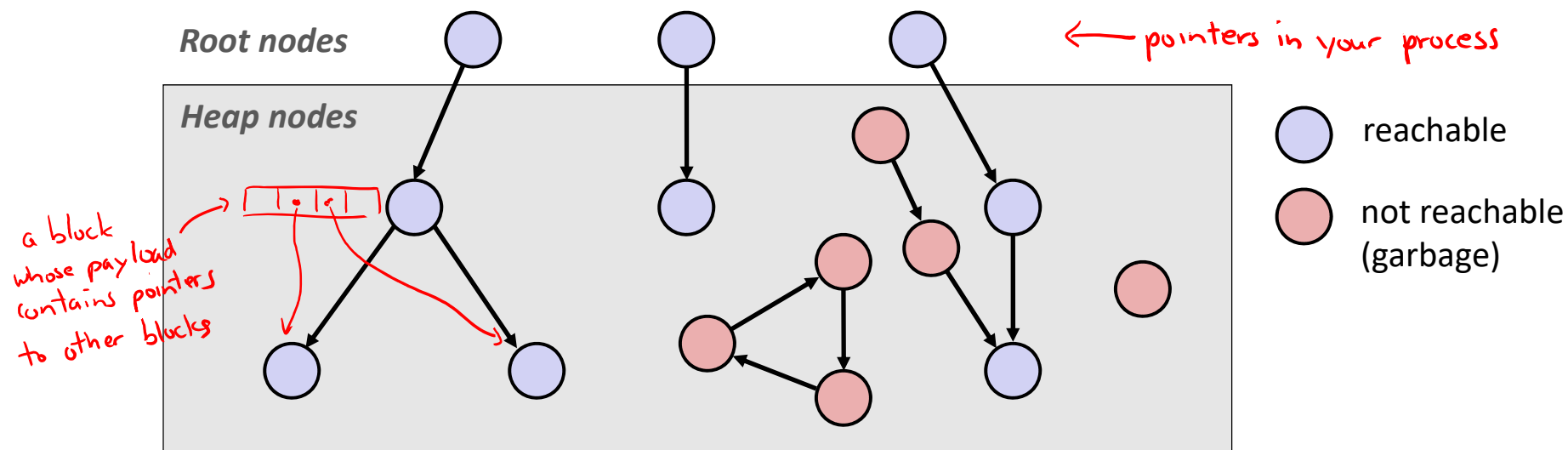
- ❖ Common in implementations of functional languages, scripting languages, and modern object oriented languages:
 - Lisp, Racket, Erlang, ML, Haskell, Scala, Java, C#, Perl, Ruby, Python, Lua, JavaScript, Dart, Mathematica, MATLAB, many more...
- ❖ Variants (“conservative” garbage collectors) exist 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

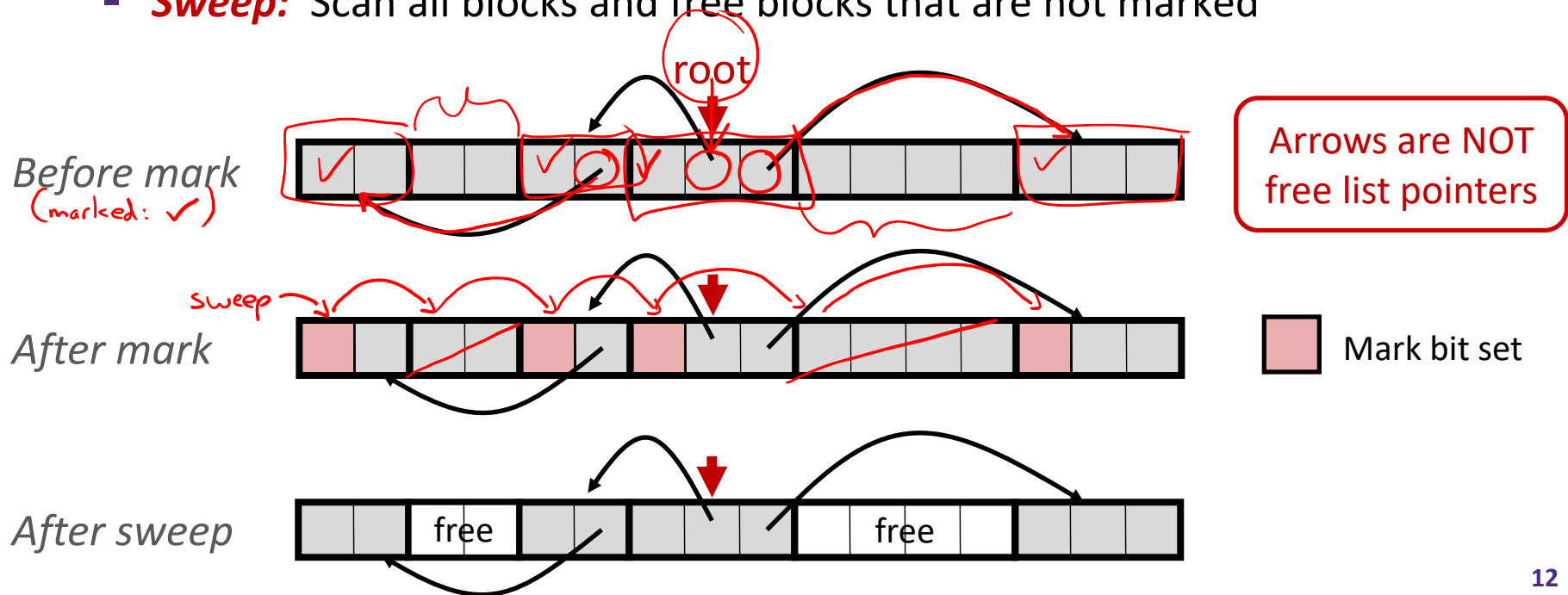
- ❖ Dynamic memory allocator can free blocks if there are no pointers to them
- ❖ How can it know what is a pointer and what is not?
- ❖ We'll make some *assumptions* about pointers:
 - Memory allocator can distinguish pointers from non-pointers
 - All pointers point to the start of a block in the heap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

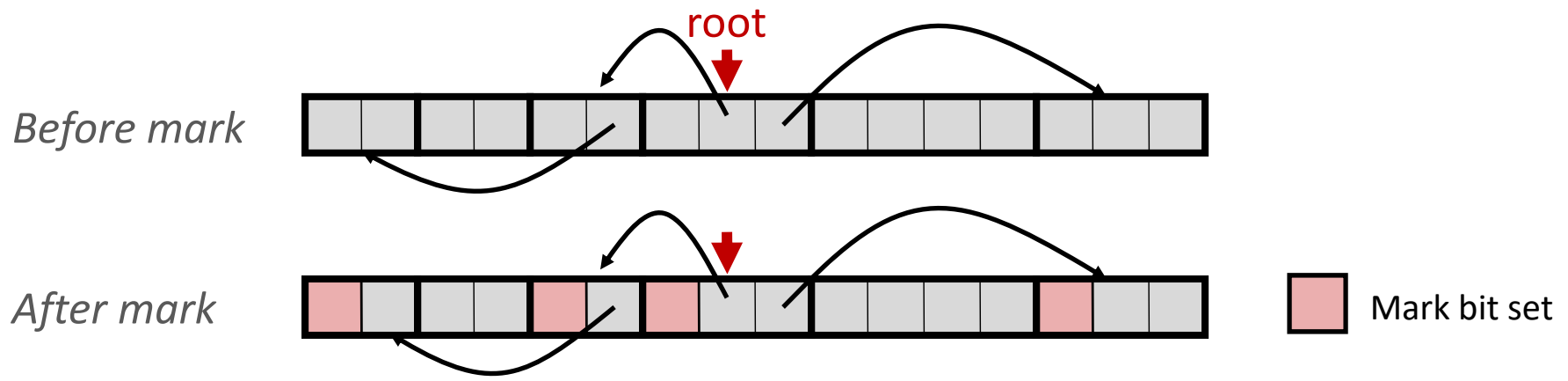
Non-testable
Material

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

```

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

↑ avoids graph cycles and presumably already traversed



Non-testable
Material

Sweep

❖ Sweep using sizes in headers

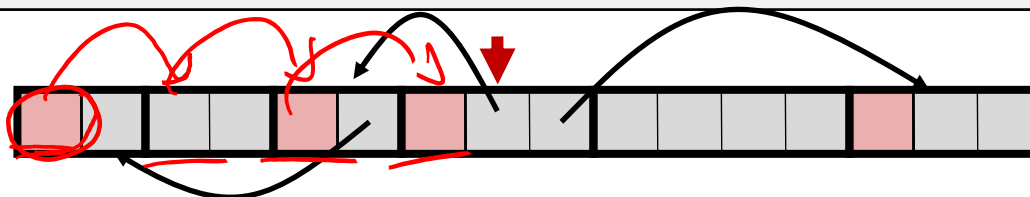
```

ptr sweep(ptr p, ptr end) {
    while (p < end) {
        if (markBitSet(p))
            clearMarkBit(p);
        else if (allocateBitSet(p))
            free(p);
        p += length(p);
    }
}
    
```

// ptrs to start & end of heap
// while not at end of heap
// check if block is marked
// if so, reset mark bit
// if not marked, but allocated
// free the block
// adjust pointer to next block

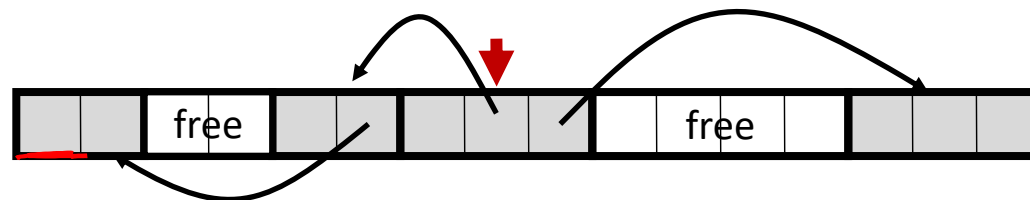
next
block →

After mark



Mark bit set

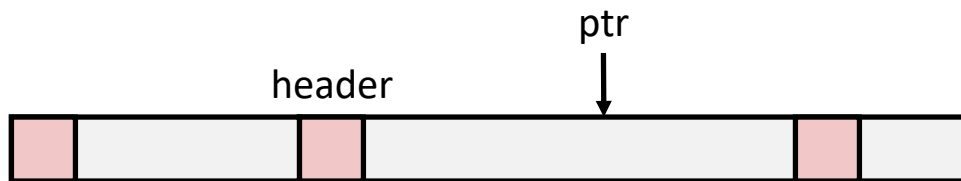
After sweep



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 | 22 | Y | <code>scanf(..., <u>l</u>val)</code> |
| B) | Freed block – access again | 24 | Y | <code>free(x)</code> later |
| C) | Freed block – free again | 23 | Y | <code>free(y)</code> |
| D) | Memory leak – failing to free memory | 25 | N | free all nodes |
| E) | No bounds checking | 18 | Y | <code>fgets</code> |
| F) | Reading uninitialized memory | 21 | N | <code>calloc</code> |
| G) | Referencing nonexistent variable | 19 | N | <code>malloc</code> |
| H) | Wrong allocation size | 20 | Y | <code>sizeof (int *)</code> |

Find That Bug! (Slide 18)

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

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

lab 3

no bounds checking

Error Type: E

Prog stop Possible? Y

buffer overflow!

Fix: fgets(s, 8)

Polling Question [Alloc III]

- ❖ Which error is this?
 - <http://pollev.com/rea>

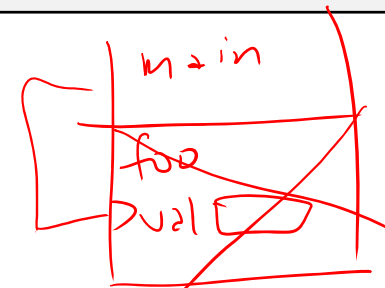
```
int* foo() {  
    int val = 0;  
    . . .  
    return &val;  
}
```

~~A. Dereferencing a non-pointer~~

~~B. Reading uninitialized Memory~~

C. Referencing a non-existent variable

~~D. Returning the wrong type~~



Find That Bug! (Slide 20)

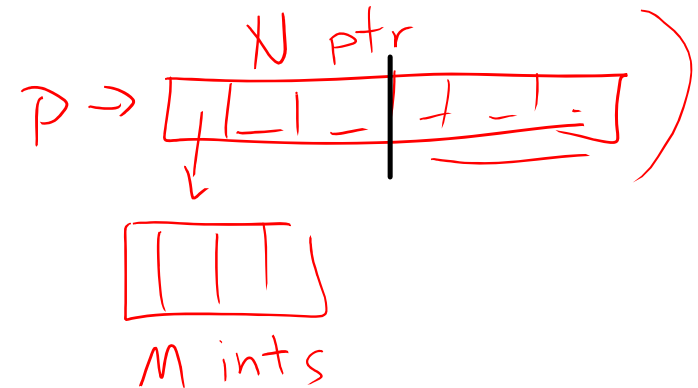
```

int **p;

p = (int **)malloc( N * sizeof(int) );
                    ↑ allocates N ints = 4*N bytes

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

Find That Bug! (Slide 21)

```

/* 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 garbage!

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

reading uninitialized memory

Error Type: **F**

*just using garbage values
- runs fine but get weird results*

Prog stop Possible? **N**

Fix: *calloc (N, sizeof(int))*

Find That Bug! (Slide 22)

❖ The classic scanf bug

▪ `int scanf(const char *format, ...)`

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

← reads input, parses int, stores into location val

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

dereferencing
a non-pointer

Error
Type: A

segfault if val
does not contain
a valid address

Prog stop
Possible? Y

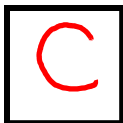
Fix: `scanf("%d", &val);`

Find That Bug! (Slide 23)

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

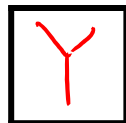
free again

Error
Type:



undefined behavior
(some systems will segfault)

Prog stop
Possible?



Fix:

free(y)

↑ probably a typo

Find That Bug! (Slide 24)

```
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]++;
```

access freed memory

undefined
behavior

Error
Type:

B

Prog stop
Possible?

Y

Fix:


free(x) later
(at bottom)

Find That Bug! (Slide 25)

```

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: 

mallocs here (pointing to the `malloc` call)

only frees first node! (pointing to the `free(head)` call)

leaked! (under a diagram of a linked list where the first node is crossed out and the rest are under a bracket)

head (pointing to the first node in the diagram)

memory leak

Error Type: D

Prog stop Possible? N

Fix: recursive/iterative free over list

how do you detect?

Non-testable
Material

Dealing With Memory Bugs

- ❖ 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 has been reallocated in the interim
 - Referencing freed blocks

Dealing With Memory Bugs (cont.)

Non-testable
Material

- ❖ Some `malloc` implementations contain checking code
 - 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 ML 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?

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

```
free(p);  
p = NULL;
```

