Memory Allocation III
CSE 351 Winter 2020

Instructor:

Ruth Anderson

Teaching Assistants:

Jonathan Chen
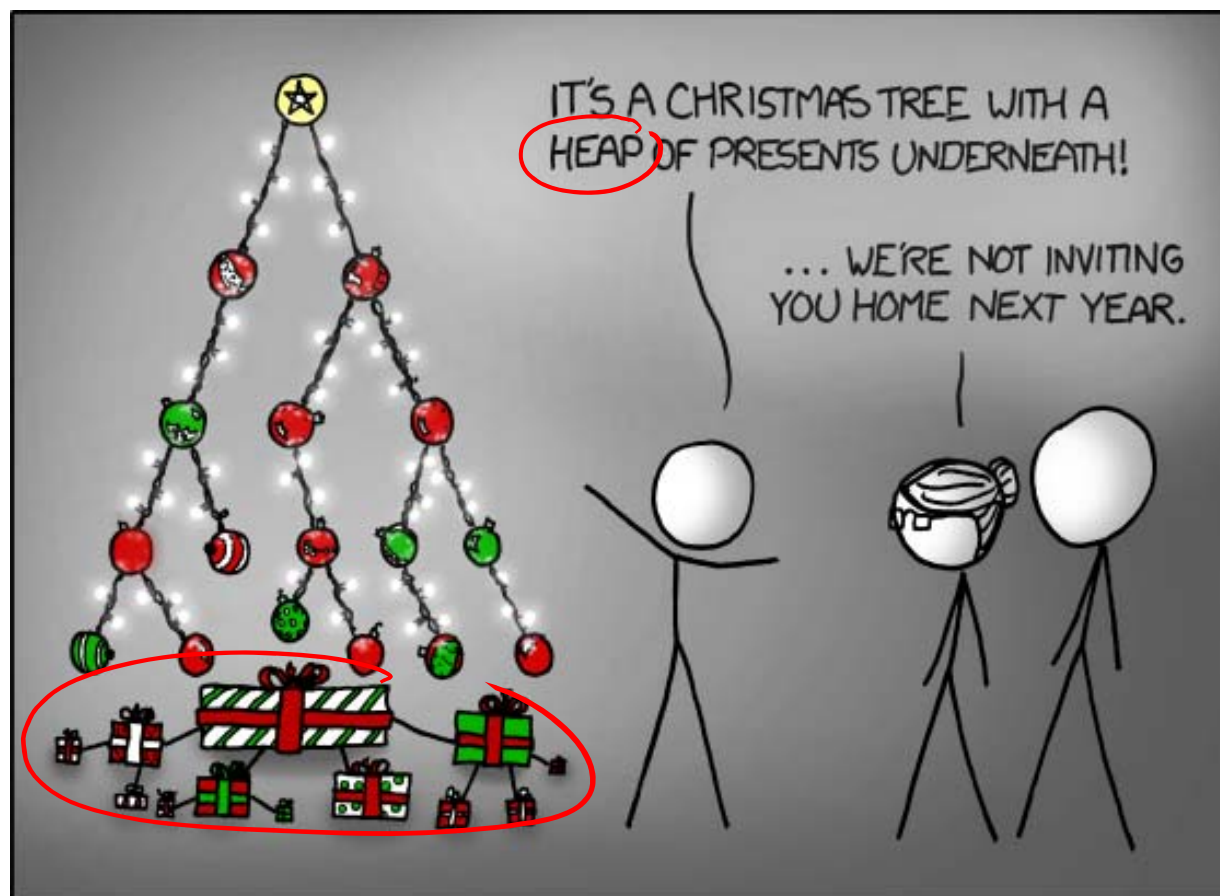Justin Johnson
Porter Jones
Josie Lee
Jeffery Tian
Callum Walker
Eddy (Tianyi) Zhou

https://xkcd.com/835/

Administrivia

- ❖ hw22 due Wednesday (3/11) – Do EARLY, will help with Lab 5

- ❖ Lab 5 (on Mem Alloc) due the last day of class (3/13)
  - Can only use 1 late day (hard deadline Sun 3/15)
  - Light style grading

- ❖ **Final Exam:**
  Wed, March 18, 2:30-4:20pm in CSE2 G20

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?   *we've assumed this up to now*
  - **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

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);        heap
    return;   /* p block is now garbage! */
}
```
stack →
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++
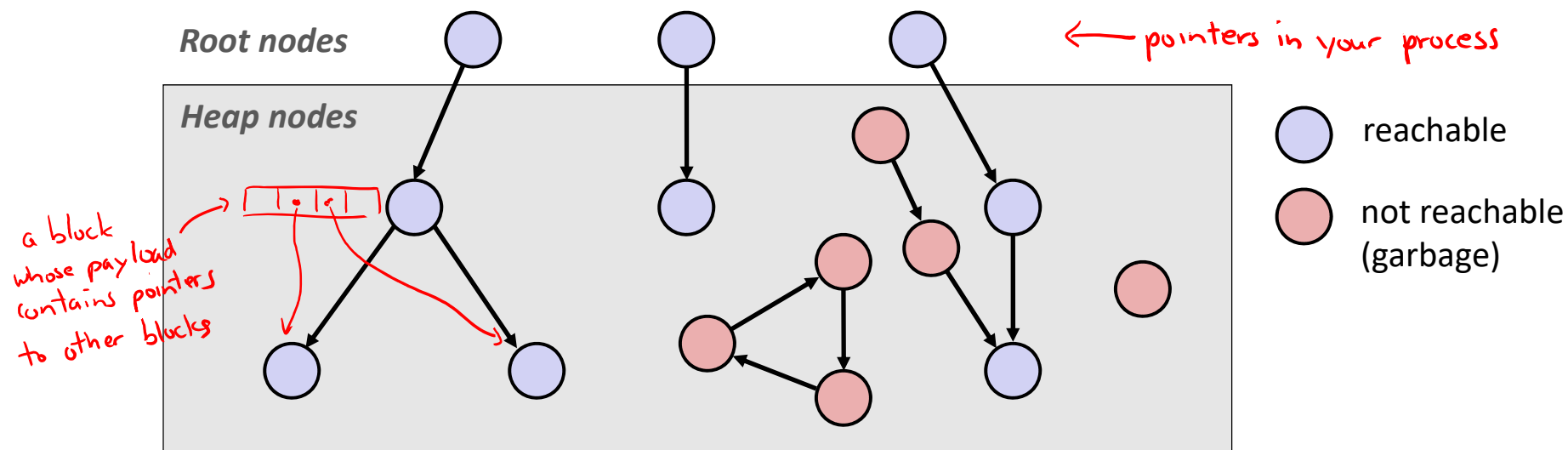
- However, cannot necessarily collect all garbage

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)

Root nodes

Heap nodes

← pointers in your process

a block whose payload contains pointers to other blocks

reachable

not reachable (garbage)

A node (block) is *reachable* if there is a path from any root to that node
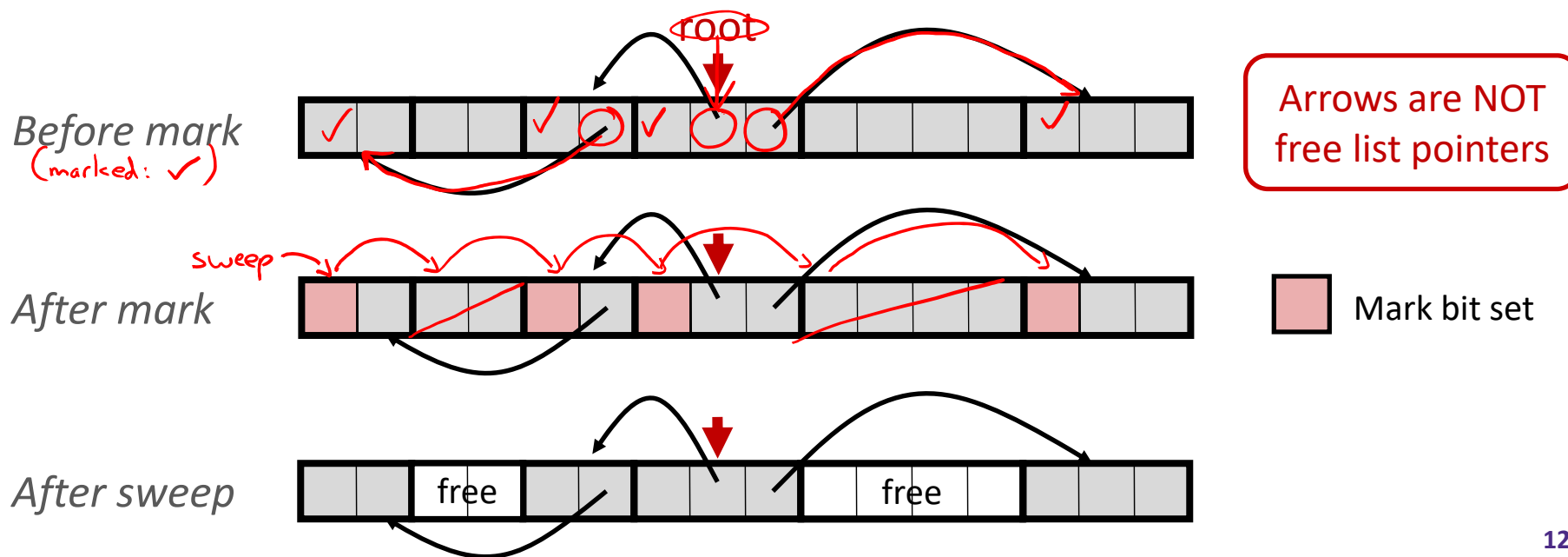Non-reachable nodes are *garbage* (cannot be needed by the application)

Garbage Collection

❖ Dynamic memory allocator can free blocks if there are <u>no pointers to them</u>

❖ 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

- ❖ **<u>Mark-and-sweep collection</u>** (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

Before mark
(marked: ✓)

After mark

After sweep

Arrows are NOT
free list pointers

■ Mark bit set

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
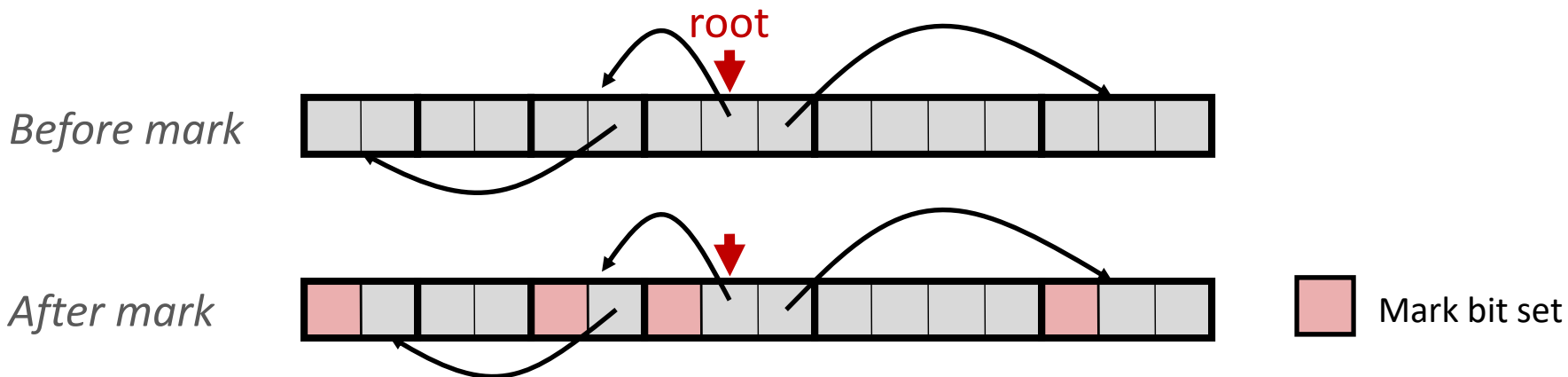
Mark

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

root

Before mark

After mark

Mark bit set

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
    }
}
```
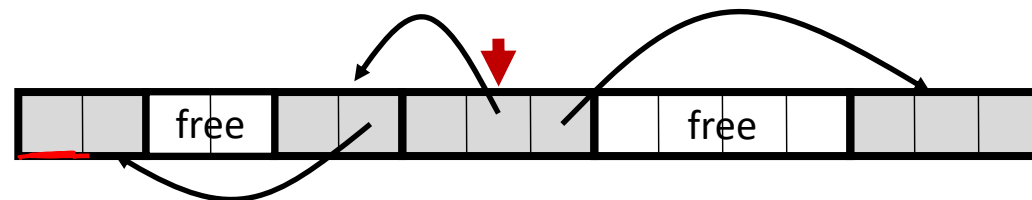
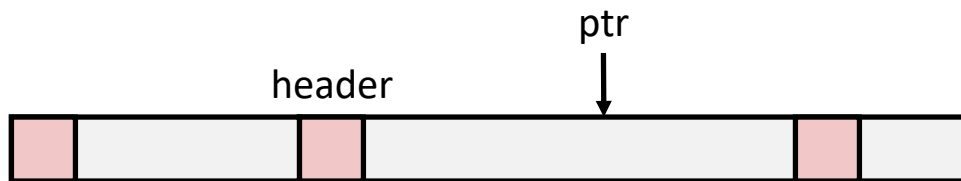*next block* →
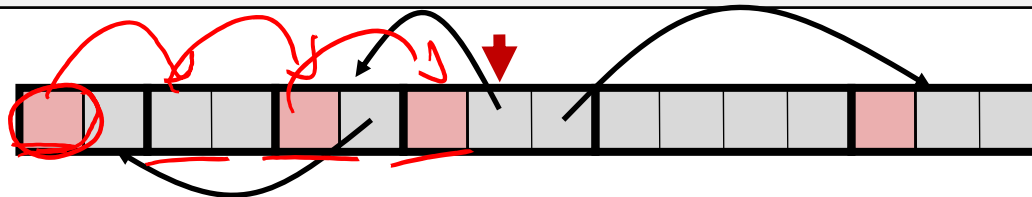
After mark   ▢ Mark bit set

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

ptr

header

▪ There are ways to solve/avoid this problem in C, but the resulting garbage collector is <u>conservative</u>:

• Every reachable node correctly identified as reachable, but some unreachable nodes might be incorrectly marked as reachable

▪ In <u>Java</u>, all pointers (*i.e.* references) point to the starting address of an object structure – the start of an allocated block

16

Memory-Related Perils and Pitfalls in C

		Slide	Program stop possible?	Fixes:
A)	Dereferencing a non-pointer	**22**	Y	scanf(... , &val)
B)	Freed block – access again	**24**	Y	free(x) later
C)	Freed block – free again	**23**	Y	free(y)
D)	Memory leak – failing to free memory	**25**	N	free all nodes
E)	No bounds checking	**18**	Y	fgets
F)	Reading uninitialized memory	**21**	N	calloc
G)	Referencing nonexistent variable	**19**	N	malloc
H)	Wrong allocation size	**20**	Y	sizeof (int *)

Find That Bug! (Slide 18)

```
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)
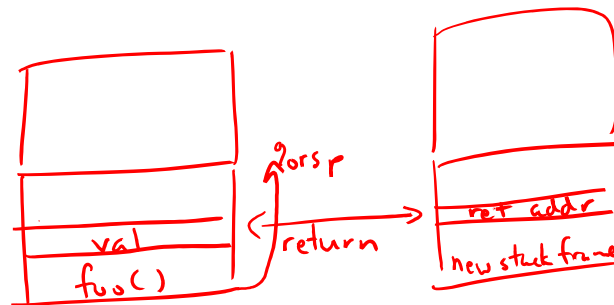
Find That Bug! (Slide 19)

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

    return &val;
}
```

↑ can't be in a register

2 or sp

val
foo()

return

ret addr
new stack frame

referencing
nonexistent
variables

valid address
on the stack

Error Type: G   **Prog stop Possible?** N   **Fix:** pass-by-reference to foo or use malloc instead

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)

Error Type: [H]   wrong allocation size

Prog stop Possible? [Y]   runs off end of allocated block

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

just using garbage values
- runs fine but get weird results

Error Type: F   **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 <u>location</u> val

segfault if val
does not contain
a valid address

dereferencing
a non-pointer

Error Type: A   **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   undefined behavior
(some systems will segfault)

Error Type: C   **Prog stop Possible?** Y   **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:

val next

↳ mallocs here

↳ only frees first node!

head

leaked!

memory leak

how do you detect?

Error
Type: D

Prog stop
Possible? N

Fix: recursive/iterative
free over list

25

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

free (p);
p = NULL;

❖ Example: Don't leave big data structures you're done with in a static field

Root nodes

Heap nodes

reachable

not reachable (garbage)
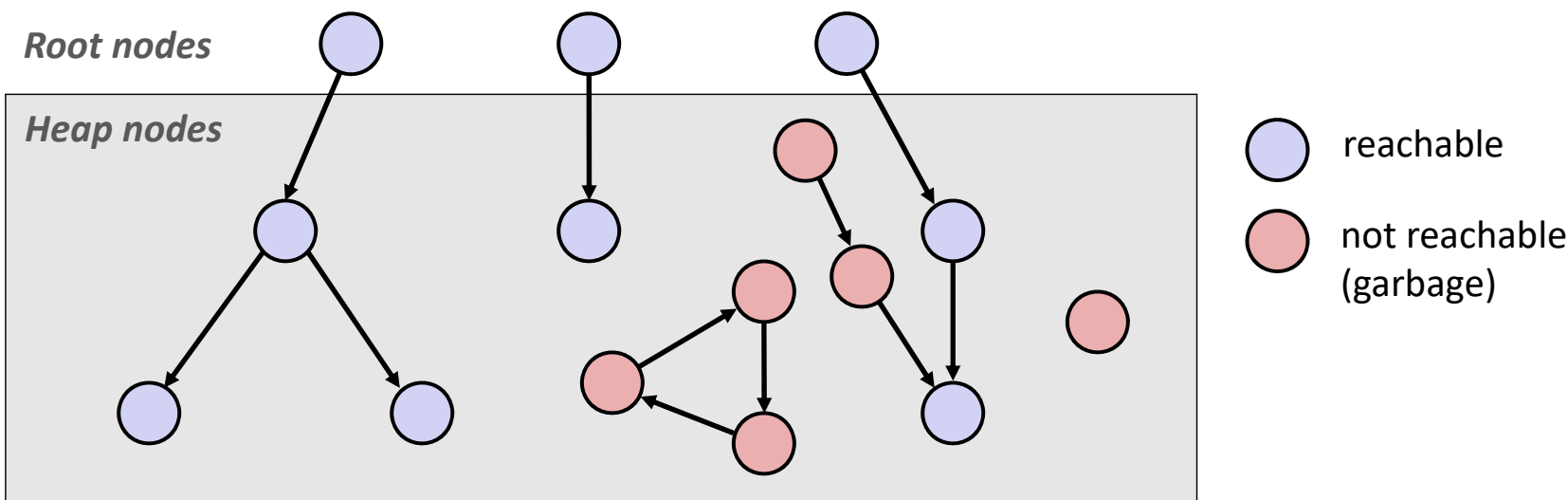
Freeing with LIFO Policy (Explicit Free List)

	Predecessor Block	**Successor Block**	**Change in Nodes in Free List**	**Number of Pointers Updated**
Case 1	Allocated	Allocated		
Case 2	Allocated	Free		
Case 3	Free	Allocated		
Case 4	Free	Free		