Memory Allocation III
CSE 351 Spring 2019

Instructor:

Ruth Anderson

Teaching Assistants:

Gavin Cai
Jack Eggleston
John Feltrup
Britt Henderson
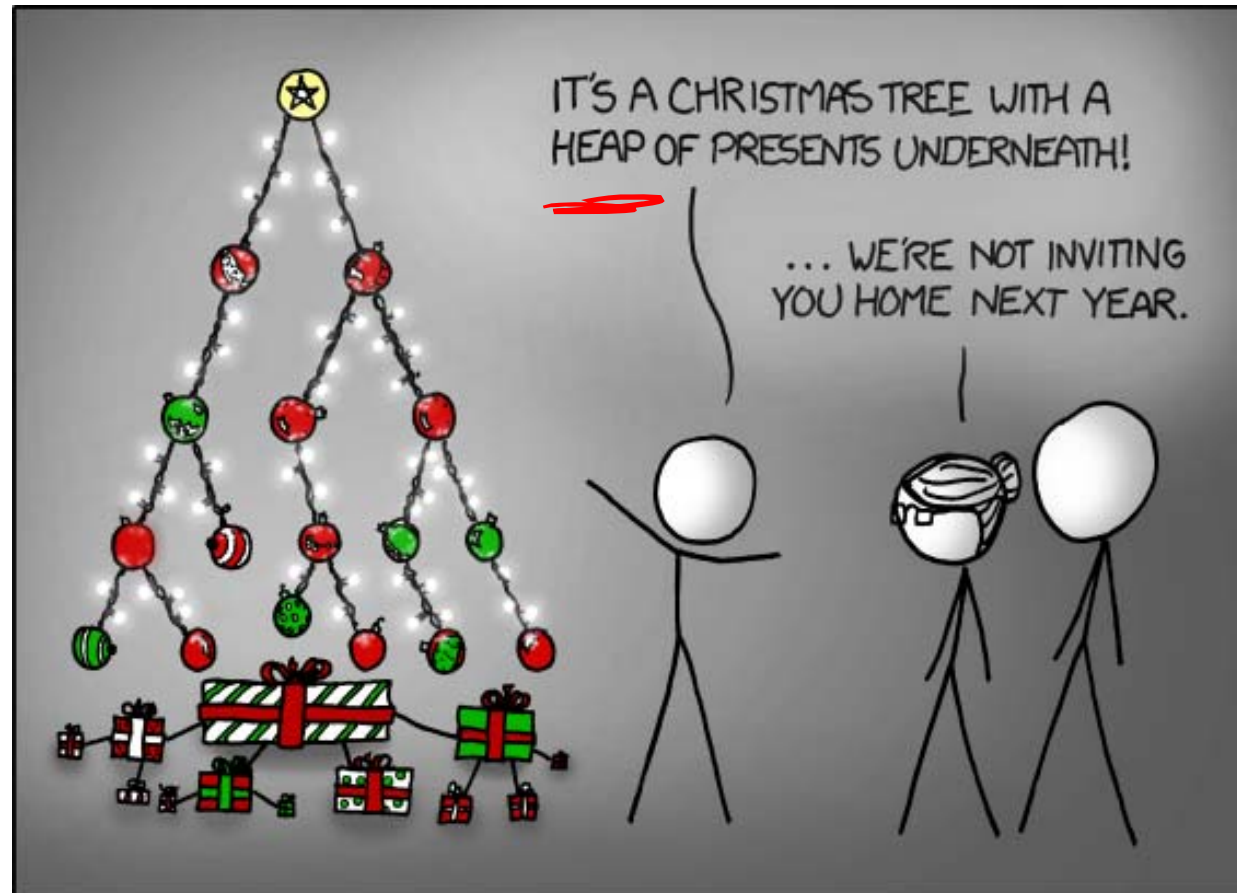Richard Jiang
Jack Skalitzky
Sophie Tian
Connie Wang
Sam Wolfson
Casey Xing
Chin Yeoh
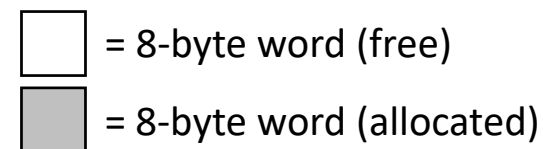

https://xkcd.com/835/

Administrivia

- ❖ Lab 5, due Friday (6/7)
  - Memory Allocation
  - Recommended that you watch the Lab 5 helper videos
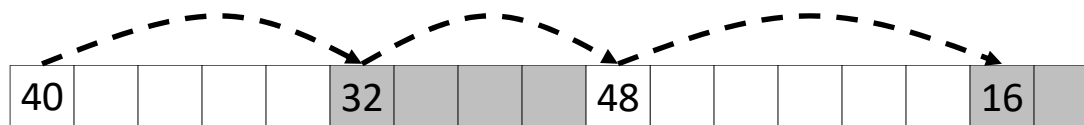 - Sunday 6/9 is last day Lab 5 may be submitted (if one late day is used)

- ❖ **Final Exam:** Wed, 6/12, 12:30-2:20 pm in KNE 130
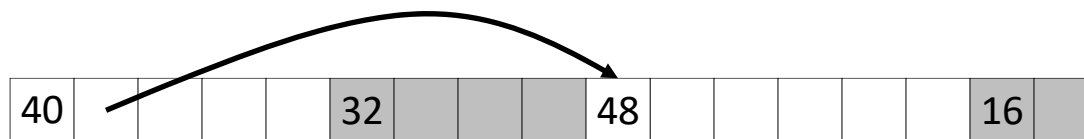
Keeping Track of Free Blocks

☐ = 8-byte word (free)

▨ = 8-byte word (allocated)

1) *Implicit free list* using length – links <u>all</u> blocks using math
   - No actual pointers, and must check each block if allocated or free

   | 40 | | | | | 32 | | | | 48 | | | | | 16 | |

2) *Explicit free list* among <u>only the free blocks</u>, using pointers

 | 40 | | | | 32 | | | | 48 | | | | | 16 | |

3) *Segregated free list*
   - Different free lists for different size "classes"

4) *Blocks sorted by size*
   - Can use a balanced binary tree (e.g. red-black tree) with pointers within each free block, and the length used as a key

Segregated List (SegList) Allocators

- ❖ Each *size class* of blocks has its own free list

- ❖ Organized as an <u>array of free lists</u>

Size class
(in bytes)

16

32

48-64

80-inf
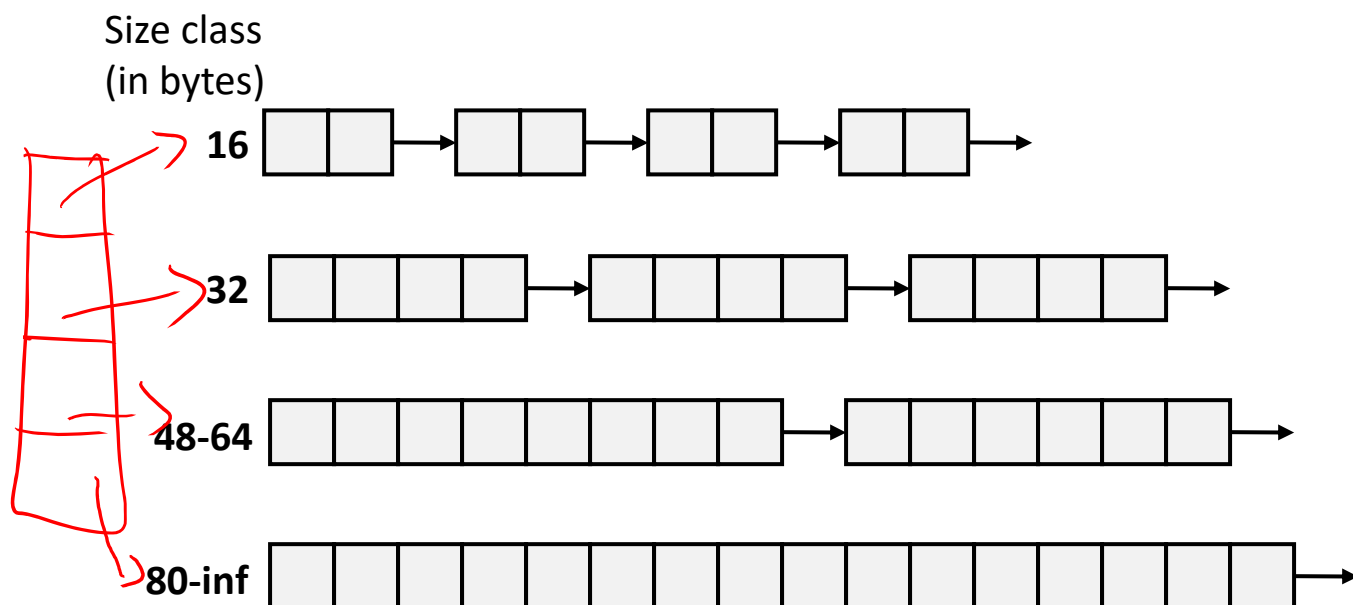
- ❖ Often have separate classes for each small size

- ❖ For larger sizes: One class for each two-power size

4

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! */
}                p is deallocated
```
stack

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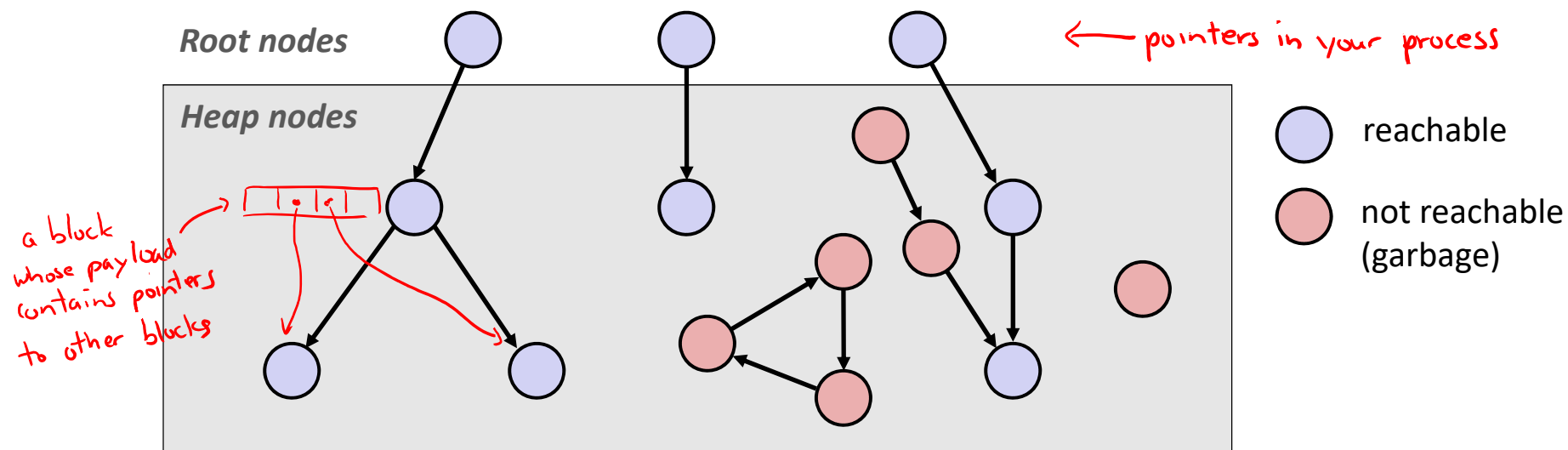
A node (block) is *reachable* if there is a path from any root to that node
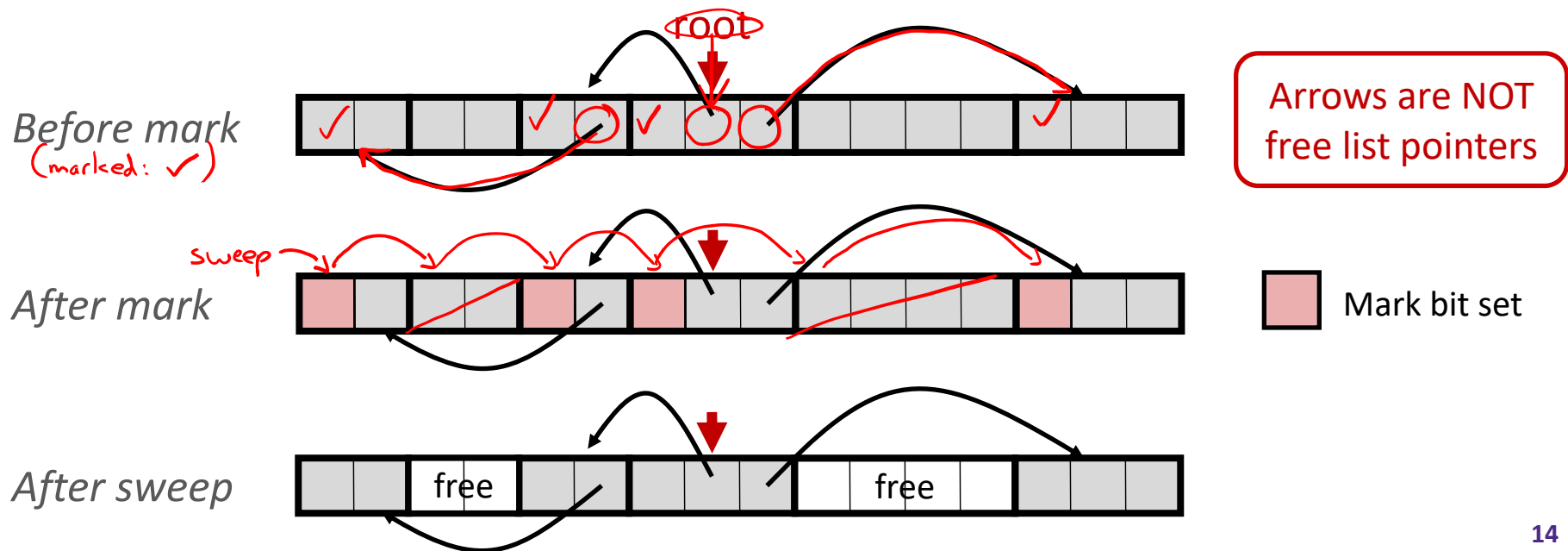Non-reachable nodes are *garbage* (cannot be needed by the application)

Garbage Collection

❖ Dynamic memory allocator can free blocks if there are <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 <u>header</u> 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]`)

❖ 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
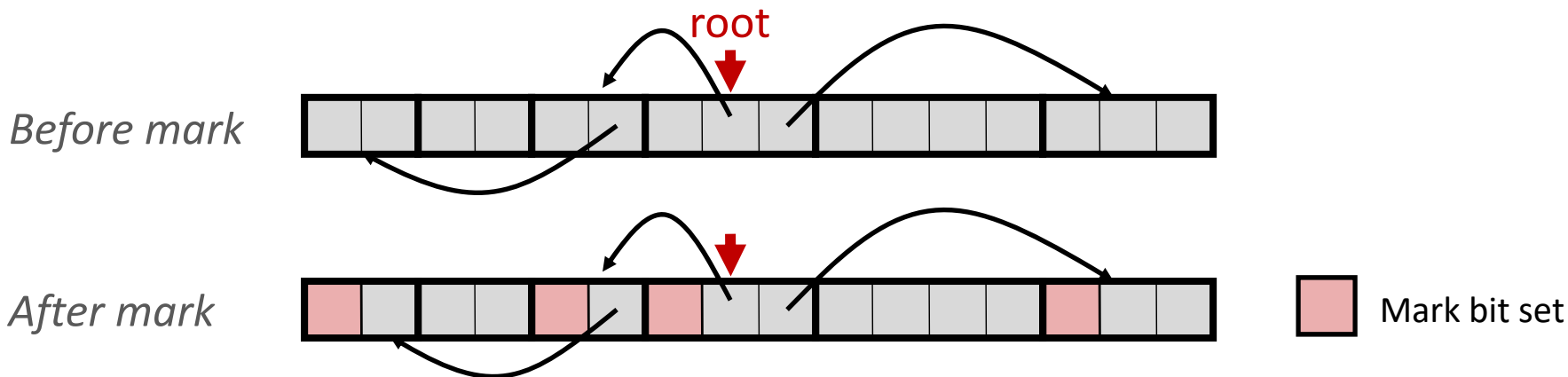
Mark

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

16

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

free   free
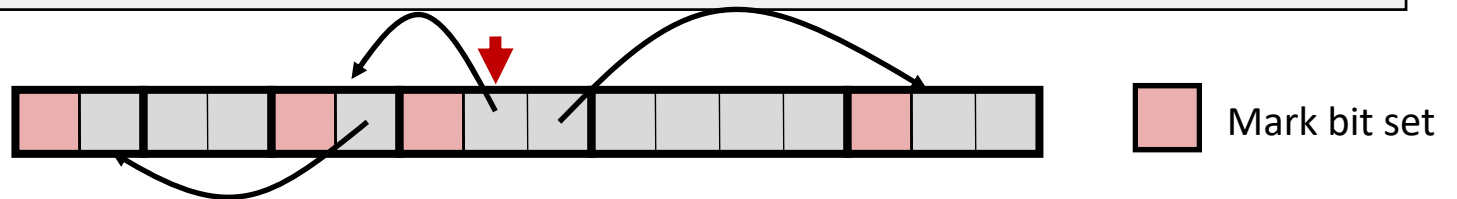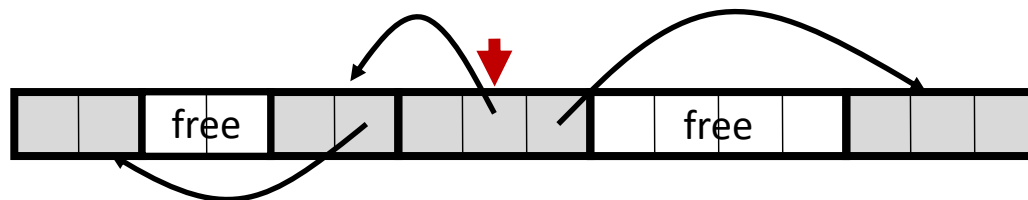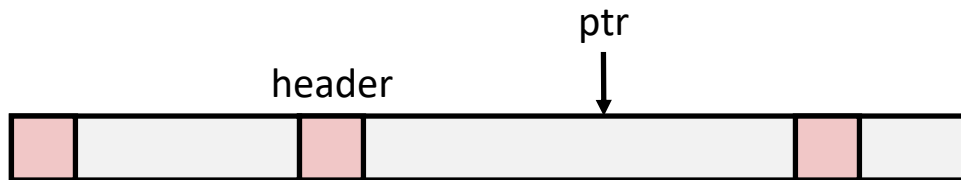
Conservative Mark & Sweep in C

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	24	Y	
B)	Freed block – access again	26	Y	
C)	Freed block – free again	25	Y	
D)	Memory leak – failing to free memory	27	N	
E)	No bounds checking	20	Y	
F)	Reading uninitialized memory	23	N	
G)	Referencing nonexistent variable	21	N	
H)	Wrong allocation size	22	Y	

Find That Bug! (Slide 20)

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

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

buffer overflow!

no bounds checking

Error Type: E

Prog stop Possible? Y

Fix: fgets(s, 8)

Find That Bug! (Slide 21)

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

↑ can't be in a register

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 22)

```
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 23)

```
/* 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 24)

❖ The classic `scanf` bug

  ▪ **int** scanf(**const char *** format)

```
int val;
...
scanf("%d", val);
```
← reads input, parses int, stores into <u>location</u> val

dereferencing
a non-pointer

segfault if val
does not contain
a valid address

Error
Type: A

Prog stop
Possible? Y

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

Find That Bug! (Slide 25)

```
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 26)

```
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 27)

```
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;
}
```
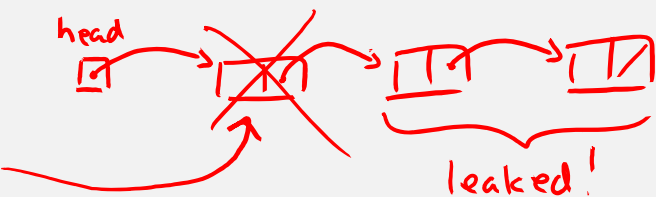
node: *(handwritten: val | next diagram)*

(handwritten: ↳ mallocs here)

(handwritten: ↳ only frees first node!)

(handwritten: head → ... leaked!)

(handwritten: how do you detect?)

memory leak

Error Type: D

Prog stop Possible? N

Fix: recursive/iterative free over list

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.)

❖ 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 …?

❖ 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

Root nodes

Heap nodes

reachable

not reachable (garbage)