Memory Allocation III

CSE 351 Summer 2020

Instructor:

Porter Jones

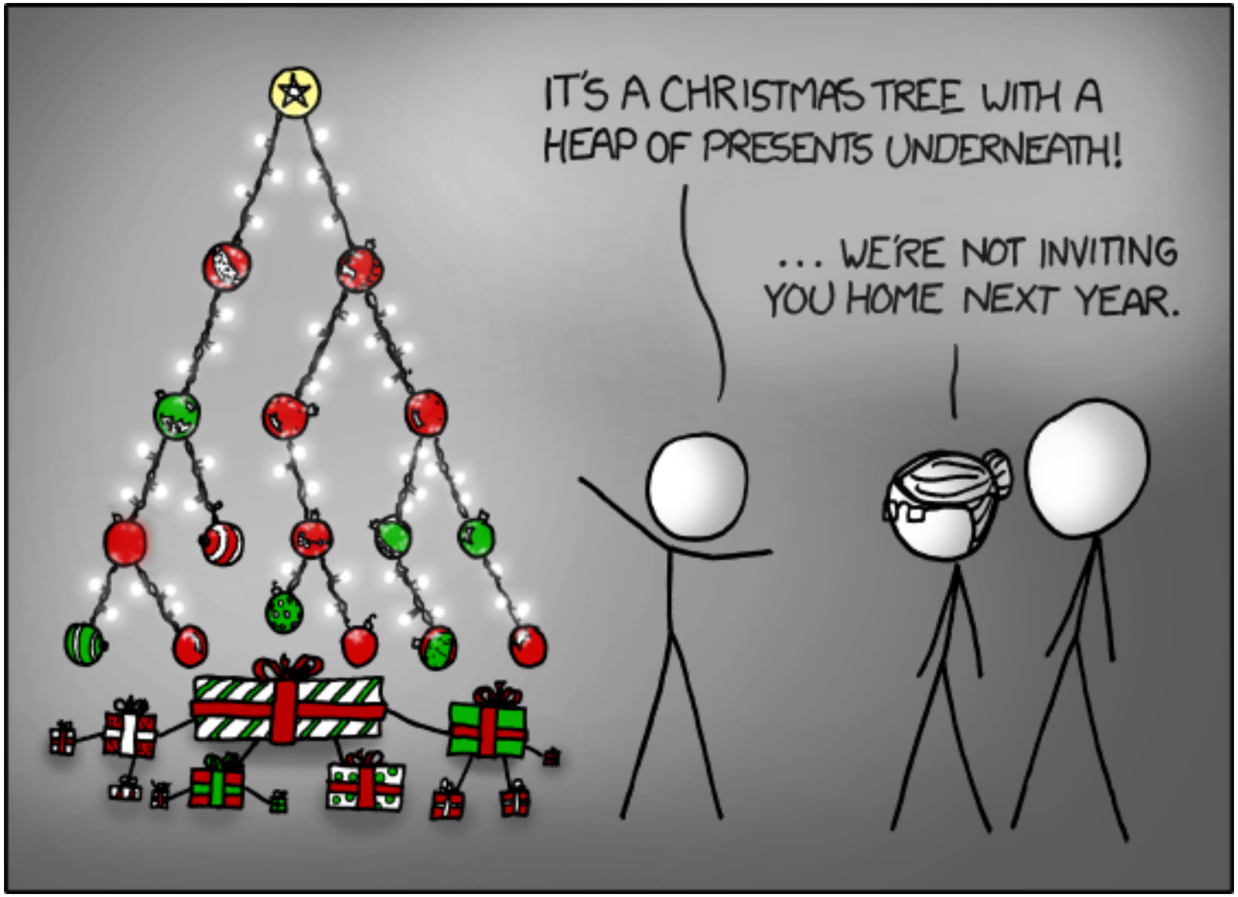
Teaching Assistants:

Amy Xu

Callum Walker

Sam Wolfson

Tim Mandzyuk



https://xkcd.com/835/

Administrivia

- Questions doc: https://tinyurl.com/CSE351-8-17

- hw19 is optional
  - Can complete it at any point before the quarter ends
  - Practice with virtual memory concepts
- hw22 due Wednesday (8/19) – 10:30am
  - Helpful for Lab 5!
- hw23 due Monday (8/24) – 10:30am
  - Won’t cover material until Wed this week

- Section Thursday is TA’s Choice & time for questions
  - See cool applications of 351 material and ask your TAs questions!

Administrivia

- Lab 5 due last day of quarter (Friday 8/21)
  - ***Cutoff is Saturday 8/22 @11:59pm (only one late day can be used!)***
  - The most significant amount of C programming you will do in this class – combines lots of topics from this class: pointers, bit manipulation, structs, examining memory
  - Understanding the concepts *first* and efficient *debugging* will save you lots of time
  - Can be difficult to debug so please start early and use OH
  - Light style grading
  - hw22 will help get you started!

- Unit Summary 3 due last day of quarter (Friday 8/21)
  - ***Cutoff is Saturday 8/22 @11:59pm (only one late day can be used!)***

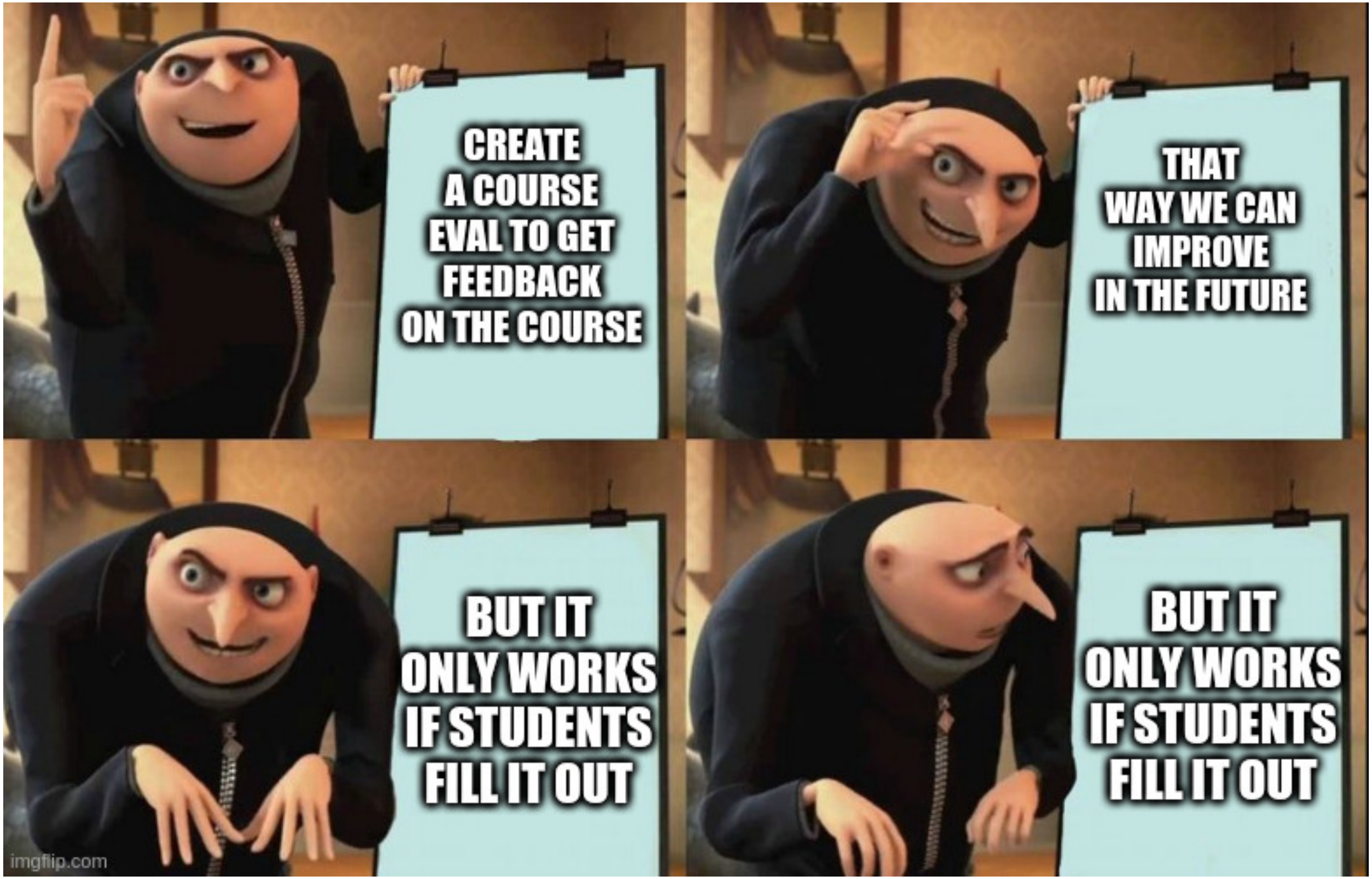
Course Evaluations

- Course Evals are Open!
  - You should have received an email with a link last night
  - They close at 11:59pm Friday (8/21) night!

- Course Evals are helpful for several reasons
  - They are helpful for improving the course content and organization in future quarters
  - They let myself and the course staff receive more feedback from you all and improve our teaching practices

- Because they are helpful and we want feedback from everyone, I will be spamming you with reminder emails so that you fill them out!
  - Get ready for a lot of reminders…

Course Evaluation Reminder Meme



Course Evaluation Reminder Meme

- **Me:** That's probably enough, you've already sent them `(unsigned int) 0xFFFFFFFF` reminders
  +1 = 0 reminders (overflow)
- **Me to me:** One more email never hurt anybody

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

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! */
}
```

- 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

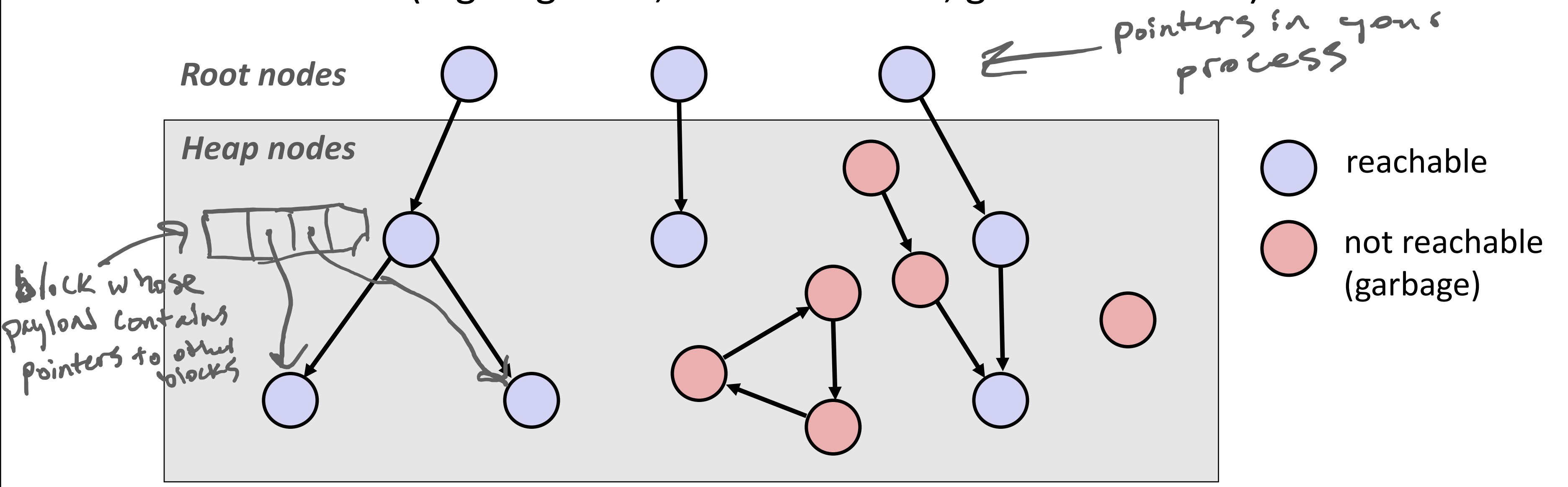
Garbage Collection

- 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?

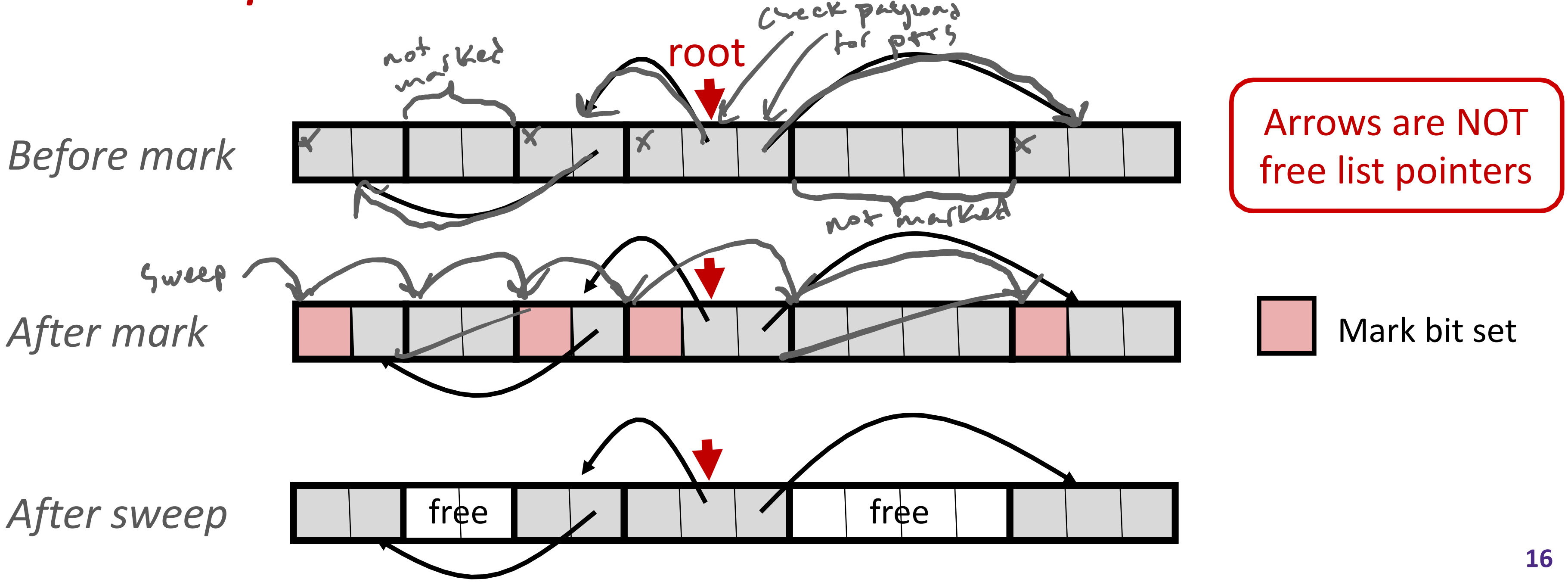
- We'll make some *assumptions* about pointers:
  - Memory allocator can distinguish pointers from non-pointers
  - All pointers point to the start of a block in the heap
  - Application cannot hide pointers
(*e.g.* by coercing them to a `long`, and then back again)

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
  - ***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]`)

magical functions that handle 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

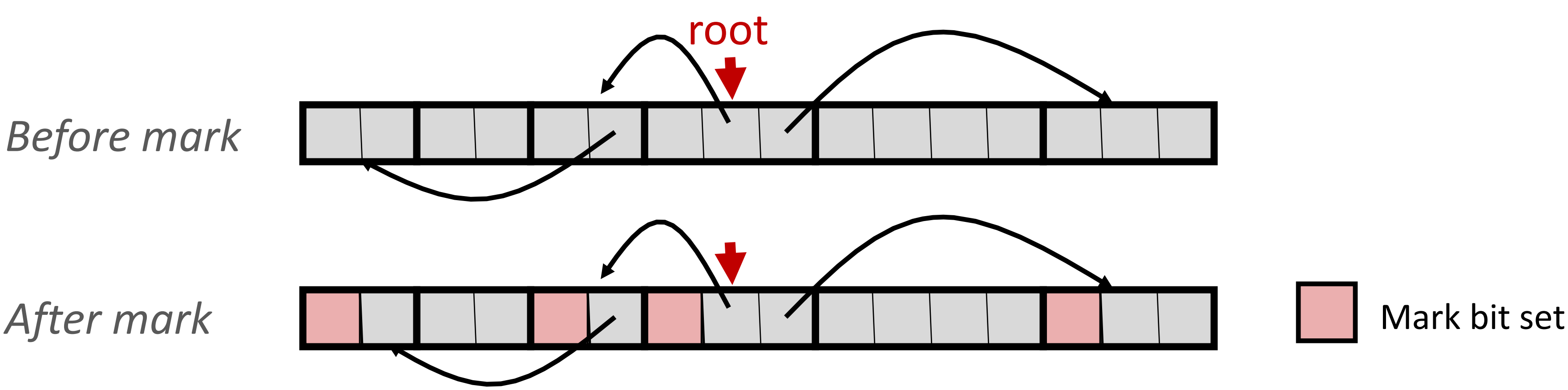
x = get_roots();
for p in x:
mark(p)

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

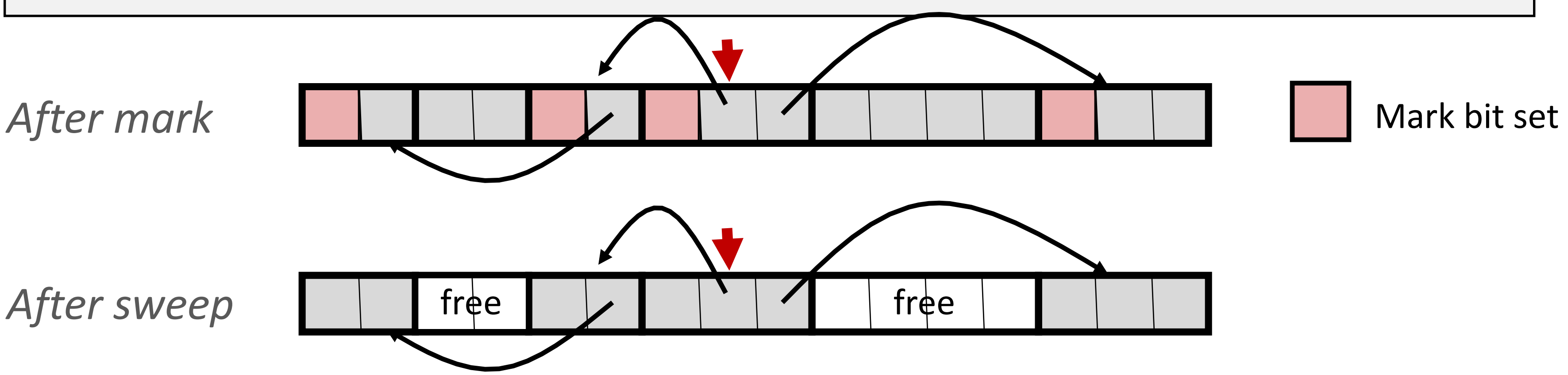


Sweep

Non-testable Material

- Sweep using sizes in headers

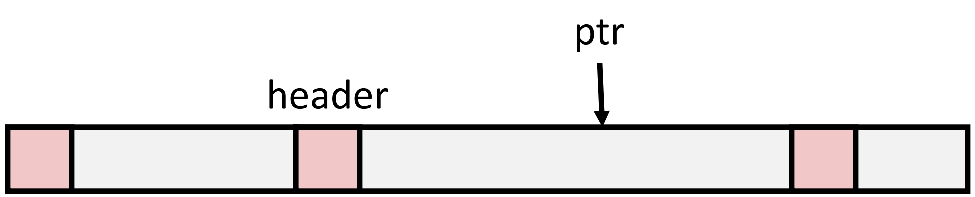
```
ptr sweep(ptr p, ptr end) {          // ptrs to start & end of heap
   while (p < end) {                 // while not at end of heap
      if (markBitSet(p))             // check if block is marked
         clearMarkBit(p);            // if so, reset mark bit
      else if (allocateBitSet(p))   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



Conservative Mark & Sweep in C

Non-testable Material

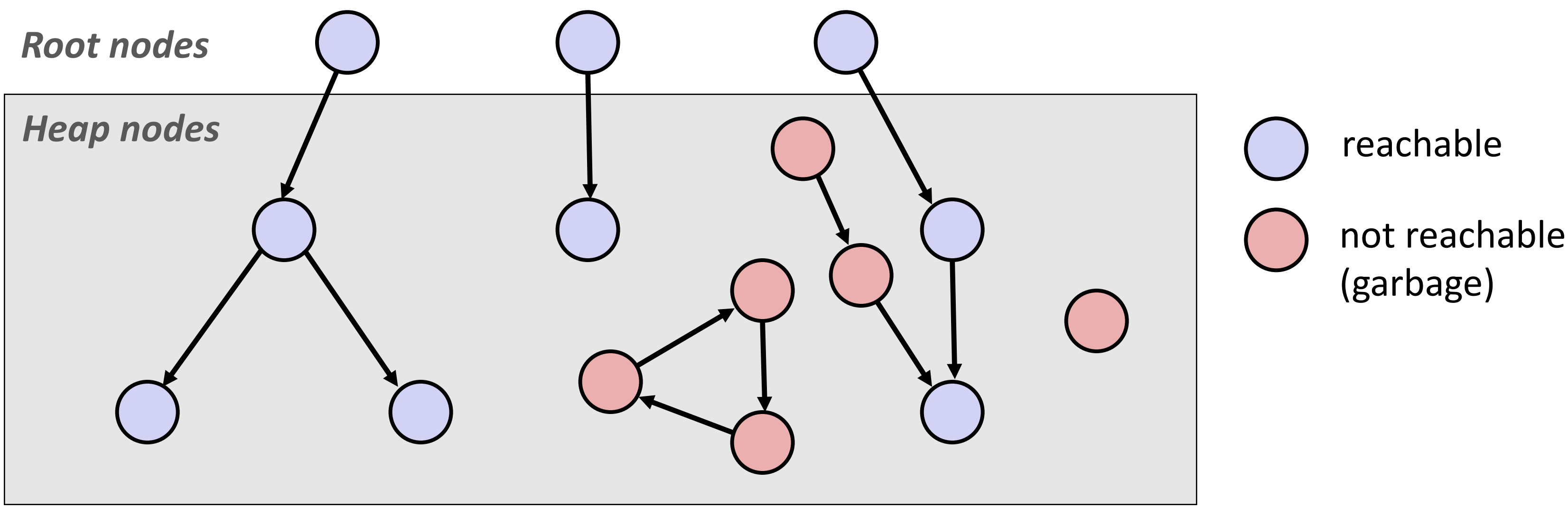
- Would mark & sweep work in C?
  - `is_ptr` determines if a word is a pointer by checking if it points to an allocated block of memory
  - But in C, pointers can point into the middle of allocated blocks (not so in Java)
    - Makes it tricky to find all allocated blocks in mark phase



  - 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 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



Memory-Related Perils and Pitfalls in C

		Slide	Program stop possible?	Fixes:
A)	Dereferencing a non-pointer	27	Y	scanf(..., &val)
B)	Freed block – access again	29	Y	free(x) later
C)	Freed block – free again	28	Y	free(y)
D)	Memory leak – failing to free memory	30	N	free all nodes
E)	No bounds checking	23	Y	fgets
F)	Reading uninitialized memory	26	N	calloc
G)	Referencing nonexistent variable	24	N	malloc
H)	Wrong allocation size	25	Y	sizeof(int *)

Find That Bug! (Slide 23)

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

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

no bounds checking

buffer overflow!

Error Type: E

Prog stop Possible? Y

Fix: fgets(s, 8)

Polling Question [Alloc III]

- Which error is this?
  - http://pollev.com/pbjones

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

deallocated on return (stack frame gone)

A. Dereferencing a non-pointer

B. Reading uninitialized Memory

C. Returning/referencing a non-existent variable

D. Returning the wrong type

Find That Bug! (Slide 25)

```
int **p;

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

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

writes N*8 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 26)

```
/* 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];
         y[i] = y[i] + A[i][j] * x[j]
   return y;          ↑ not initialized
}
```

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

Uninitialized memory   just writes garbage values

Error Type: F   **Prog stop Possible?** N   **Fix:** calloc(N * sizeof(int))

Find That Bug! (Slide 27)

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

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

Should be of type int*

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

dereferencing non-pointer

Val might not contain valid pointer

Error Type: A

Prog stop Possible? Y

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

Find That Bug! (Slide 28)

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

   ...

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

free again

Could it result in seg fault

probably was a typo

Error Type: C

Prog stop Possible? Y

Fix: free(y)

Find That Bug! (Slide 29)

```
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

Error Type: B

undefined behavior

Prog stop Possible? Y

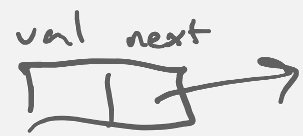
Fix: free(x) later (at the bottom)

Find That Bug! (Slide 30)

```
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 other nodes

only frees first node in list!

memory leak

Error Type: D

hard to detect

Prog stop Possible? N

Fix: recursive/iterative free over list

Dealing With Memory Bugs

Non-testable Material

- 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?

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		

Find That Bug! (Slide 35)

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

   return &val;
}
```

Error Type: | Prog stop Possible? | Fix: