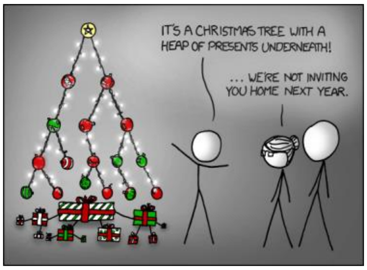
Memory Allocation III
CSE 351 Winter 2018

Instructor:
Mark Wyse

Teaching Assistants:
Kevin Bi
Parker DeWilde
Emily Furst
Sarah House
Waylon Huang
Vinny Palaniappan



https://xkcd.com/835/

Administrivia

- Homework 5 due tonight
- Lab 5 due Saturday (3/10)
  - Recommended that you watch the Lab 5 helper videos
- **Final Exam:** Wed, March 14 @ 2:30pm in KNE 110

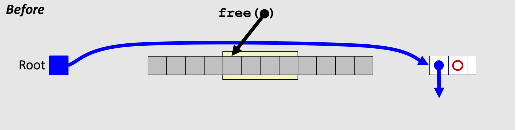
Coalescing in Explicit Free Lists

	Case 1	*Case 2*	*Case 3*	*Case 4*
	Allocated	Allocated	Free	Free
Block being freed →				
	Allocated	Free	Allocated	Free

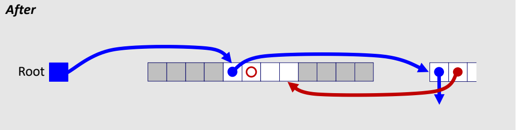
- Neighboring free blocks are *already part of the free list*
  1) Remove old block from free list
  2) Create new, larger coalesced block
  3) Add new block to free list (insertion policy)
- How do we tell if a neighboring block if free?

Freeing with LIFO Policy (Case 1)

Boundary tags not shown, but don't forget about them!

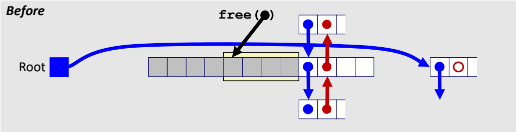


- Insert the freed block at the root of the list

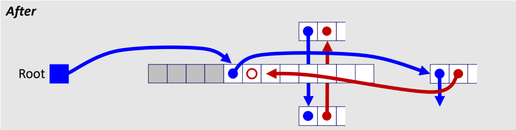


Freeing with LIFO Policy (Case 2)

Boundary tags not shown, but don't forget about them!

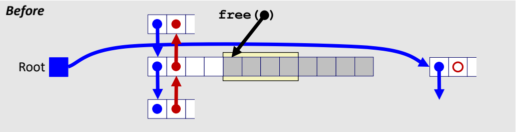


- Splice *successor* block out of list, coalesce both memory blocks, and insert the new block at the root of the list

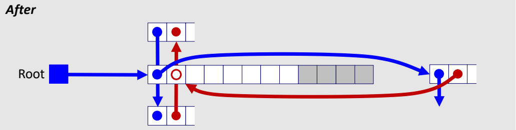


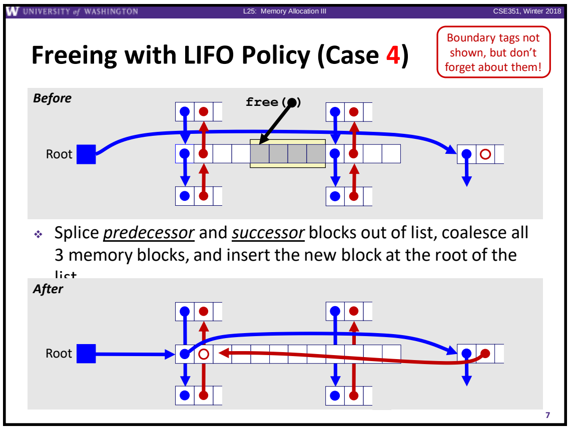
Freeing with LIFO Policy (Case 3)

Boundary tags not shown, but don't forget about them!



- Splice *predecessor* block out of list, coalesce both memory blocks, and insert the new block at the root of the list

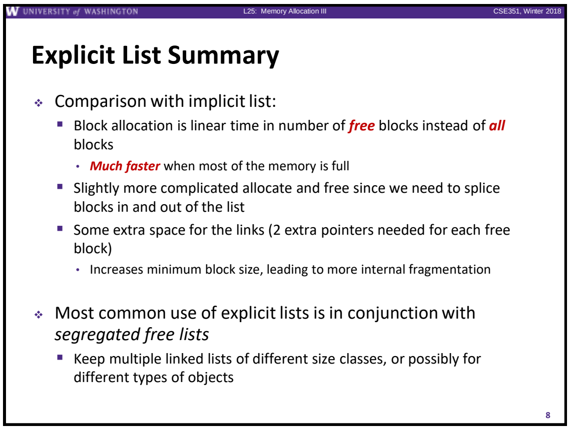




Freeing with LIFO Policy (Case 4)

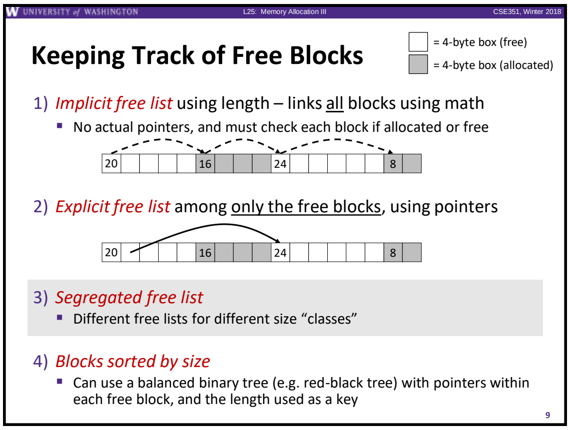
Boundary tags not shown, but don't forget about them!

- Splice *predecessor* and *successor* blocks out of list, coalesce all 3 memory blocks, and insert the new block at the root of the list



Explicit List Summary

- Comparison with implicit list:
  - Block allocation is linear time in number of ***free*** blocks instead of ***all*** blocks
    - ***Much faster*** when most of the memory is full
  - Slightly more complicated allocate and free since we need to splice blocks in and out of the list
  - Some extra space for the links (2 extra pointers needed for each free block)
    - Increases minimum block size, leading to more internal fragmentation
- Most common use of explicit lists is in conjunction with *segregated free lists*
  - Keep multiple linked lists of different size classes, or possibly for different types of objects



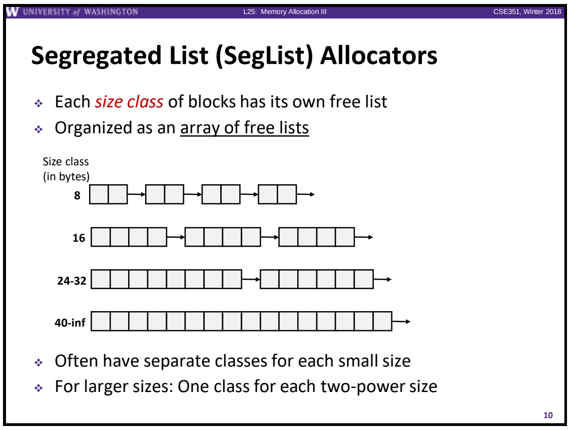
Keeping Track of Free Blocks

☐ = 4-byte box (free)
▨ = 4-byte box (allocated)

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

2) *Explicit free list* among only the free blocks, using pointers

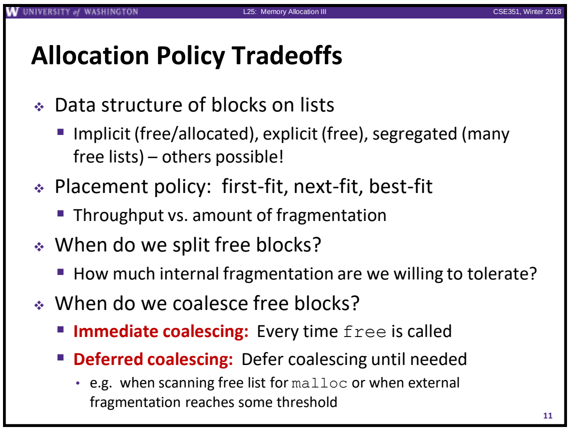
3) *Segregated free list*
   - Different free lists for different size "classes"
4) *Blocks sorted by size*
   - Can use a balanced binary tree (e.g. red-black tree) with pointers within each free block, and the length used as a key



Segregated List (SegList) Allocators

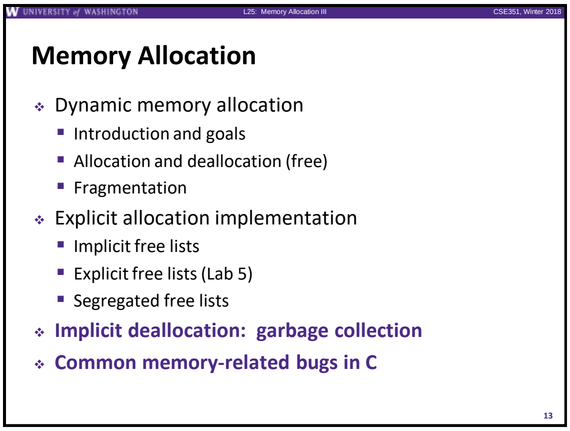
- Each *size class* of blocks has its own free list
- Organized as an array of free lists

- Often have separate classes for each small size
- For larger sizes: One class for each two-power size



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



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! */
}
```

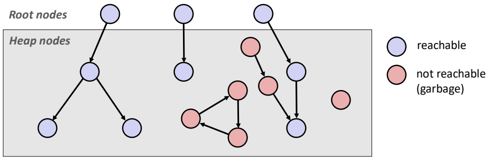
- Common in implementations of functional languages, scripting languages, and modern object oriented languages:
  - Lisp, Racket, Erlang, ML, Haskell, Scala, Java, C#, Perl, Ruby, Python, Lua, JavaScript, Dart, Mathematica, MATLAB, many more…
- Variants ("conservative" garbage collectors) exist for C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

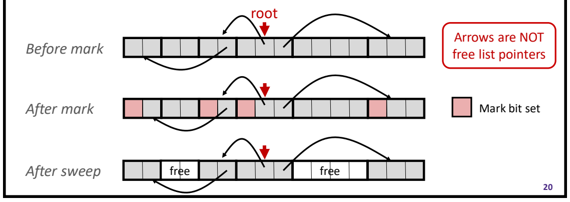
- Dynamic memory allocator can free blocks if there are no pointers to them
- How can it know what is a pointer and what is not?
- We'll make some *assumptions* about pointers:
  - Memory allocator can distinguish pointers from non-pointers
  - All pointers point to the start of a block in the heap
  - Application cannot hide pointers
    (e.g. by coercing them to an `int`, 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



Arrows are NOT free list pointers

Memory-Related Perils and Pitfalls in C

		Slide	Prog stop Possible?	Security Flaw?
A)	Bad order of operations			
B)	Bad pointer arithmetic			
C)	Dereferencing a non-pointer			
D)	Freed block – access again			
E)	Freed block – free again			
F)	Memory leak – failing to free memory			
G)	No bounds checking			
H)	Off-by-one error			
I)	Reading uninitialized memory			
J)	Referencing nonexistent variable			
K)	Wrong allocation size			

Find That Bug! (Slide 26)

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

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

Error Type: ☐ **Prog stop Possible?** ☐ **Security flaw Possible?** ☐ **Fix:**

Find That Bug! (Slide 27)

```
/* 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;
}
```

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

Error Type: ☐ **Prog stop Possible?** ☐ **Security flaw Possible?** ☐ **Fix:**

Find That Bug! (Slide 28)

```
int **p;

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

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

- `N` and `M` defined elsewhere (`#define`)

Error Type: ☐ **Prog stop Possible?** ☐ **Security flaw Possible?** ☐ **Fix:**

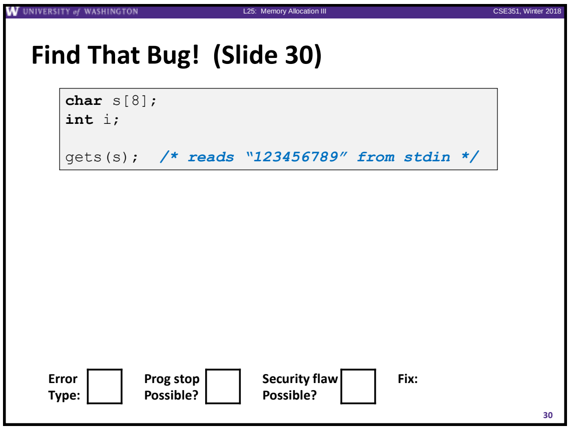
Find That Bug! (Slide 29)

```
int **p;

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

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

Error Type: ☐ **Prog stop Possible?** ☐ **Security flaw Possible?** ☐ **Fix:**

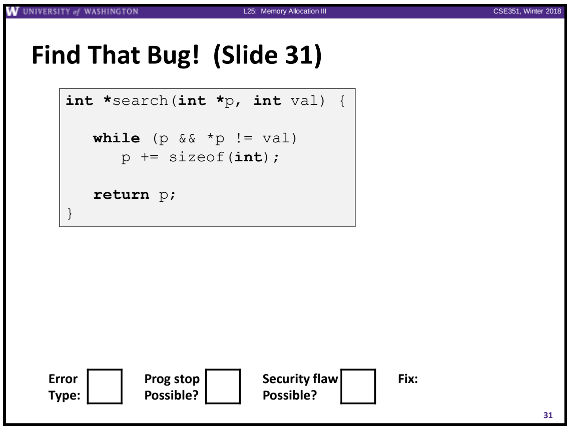


Find That Bug! (Slide 30)

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

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

Error Type: ☐ Prog stop Possible? ☐ Security flaw Possible? ☐ Fix:



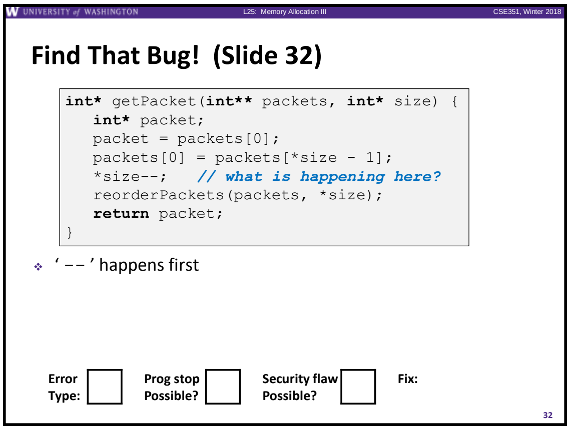
Find That Bug! (Slide 31)

```
int *search(int *p, int val) {

   while (p && *p != val)
      p += sizeof(int);

   return p;
}
```

Error Type: ☐ Prog stop Possible? ☐ Security flaw Possible? ☐ Fix:

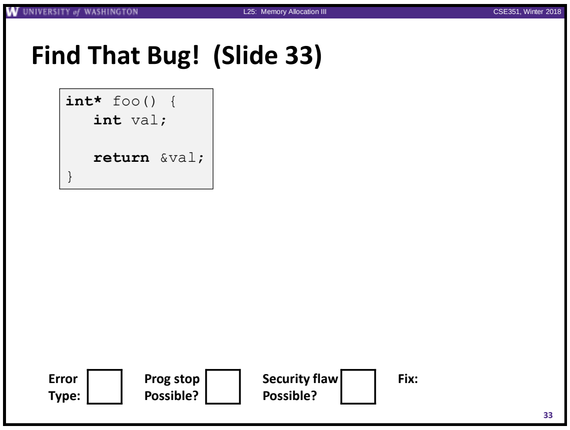


Find That Bug! (Slide 32)

```
int* getPacket(int** packets, int* size) {
   int* packet;
   packet = packets[0];
   packets[0] = packets[*size - 1];
   *size--;   // what is happening here?
   reorderPackets(packets, *size);
   return packet;
}
```

- ‘--’ happens first

Error Type: ☐ Prog stop Possible? ☐ Security flaw Possible? ☐ Fix:

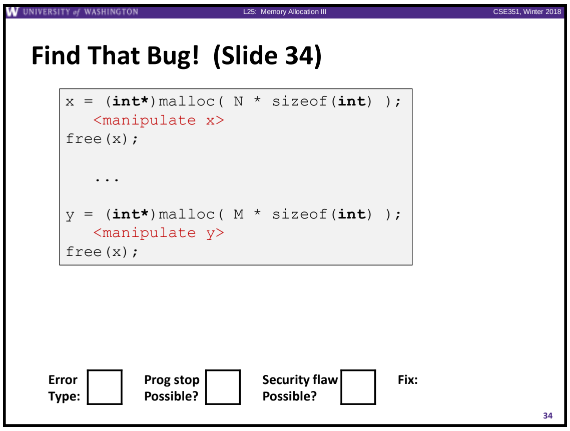


Find That Bug! (Slide 33)

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

   return &val;
}
```

Error Type: ☐ Prog stop Possible? ☐ Security flaw Possible? ☐ Fix:



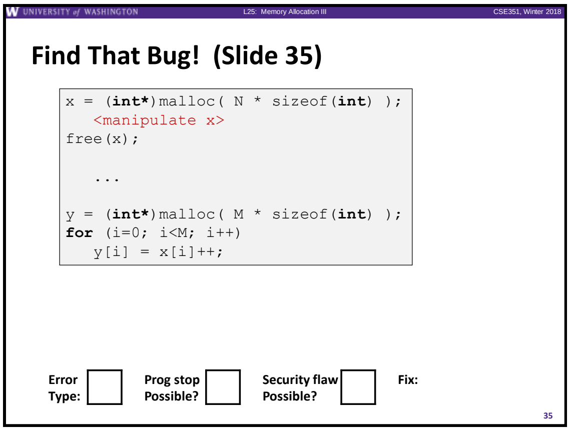
Find That Bug! (Slide 34)

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

   ...

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

Error Type: ☐ Prog stop Possible? ☐ Security flaw Possible? ☐ Fix:



Find That Bug! (Slide 35)

```
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]++;
```

Error Type: ☐ Prog stop Possible? ☐ Security flaw Possible? ☐ Fix:

Find That Bug! (Slide 36)

```
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;
}
```

Error Type: ☐ **Prog stop Possible?** ☐ **Security flaw Possible?** ☐ **Fix:**

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