Memory Allocation III
CSE 351 Spring 2018

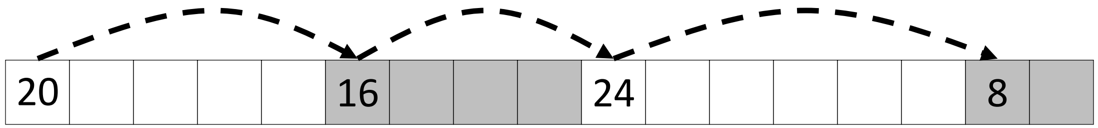
(original source unknown)

Keeping Track of Free Blocks

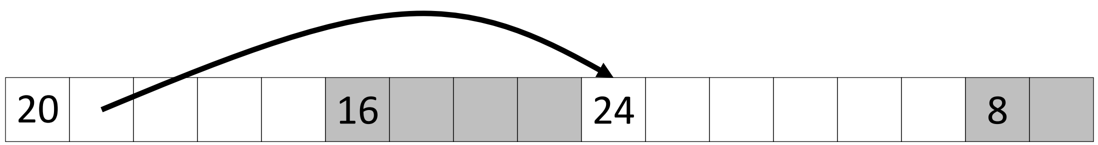
= 4-byte box (free)

= 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

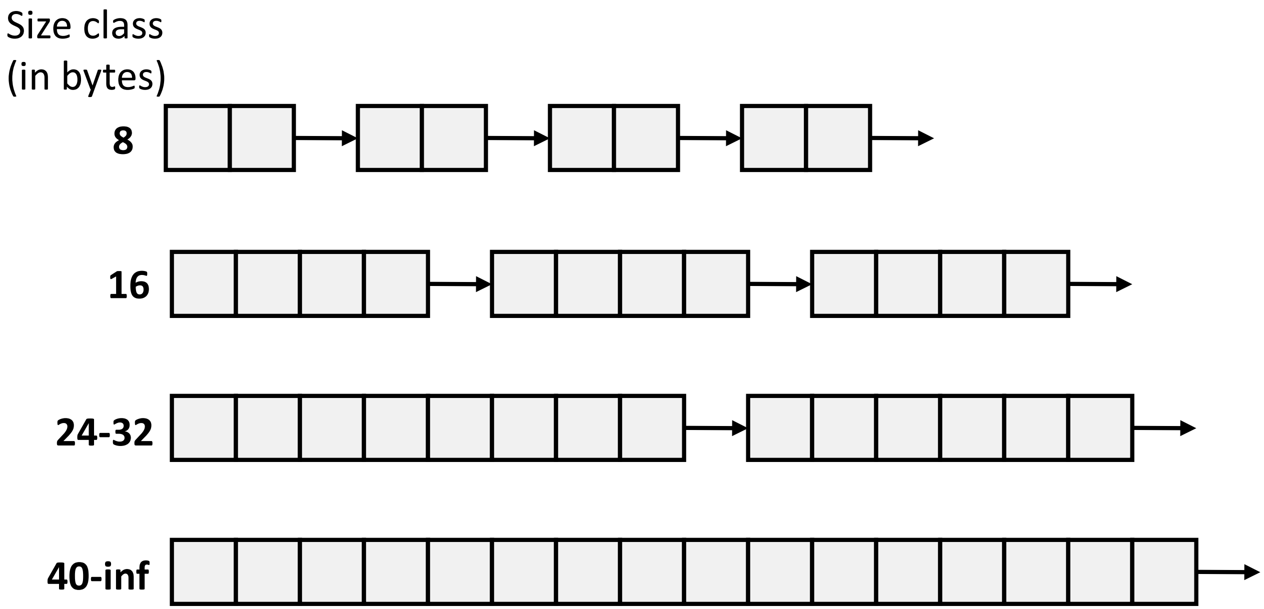


3) *Segregated free list*
   - Different free lists for different size "classes"

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

SegList Allocator

- Have an array of free lists for various size classes

- To allocate a block of size n:
  - Search appropriate free list for block of size $m \geq n$
  - If an appropriate block is found:
    - [Optional] Split block and place free fragment on appropriate list
  - If no block is found, try the next larger class
    - Repeat until block is found
- If no block is found:
  - Request additional heap memory from OS (using `sbrk`)
  - Place remainder of additional heap memory as a single free block in appropriate size class

SegList Allocator

- Have an array of free lists for various size classes

- To free a block:
  - Mark block as free
  - Coalesce (if needed)
  - Place on appropriate class list

SegList Advantages

- Higher throughput
  - Search is log time for power-of-two size classes
- Better memory utilization
  - First-fit search of seglist approximates a best-fit search of entire heap
  - *Extreme case:* Giving every block its own size class is no worse than best-fit search of an explicit list
  - Don’t need to use space for block size for the fixed-size classes

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 [explicitly] free memory?
  - And couldn't mess up and free it too early?
- Do you free objects in Java?

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

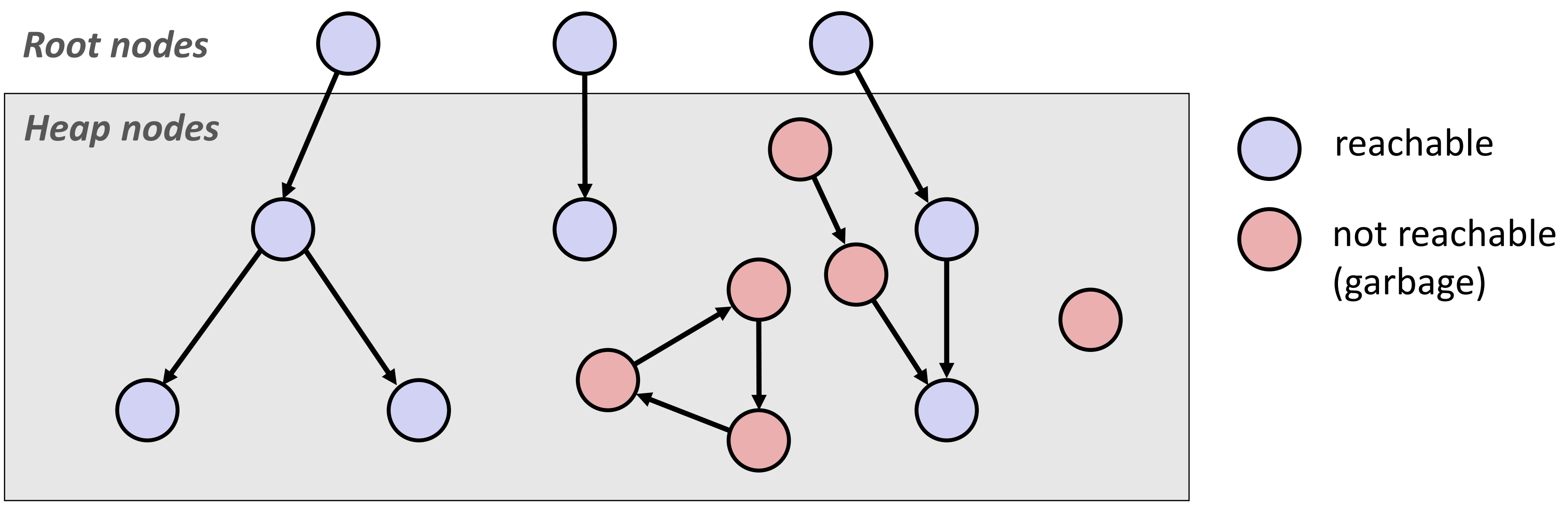
Garbage Collection

- 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

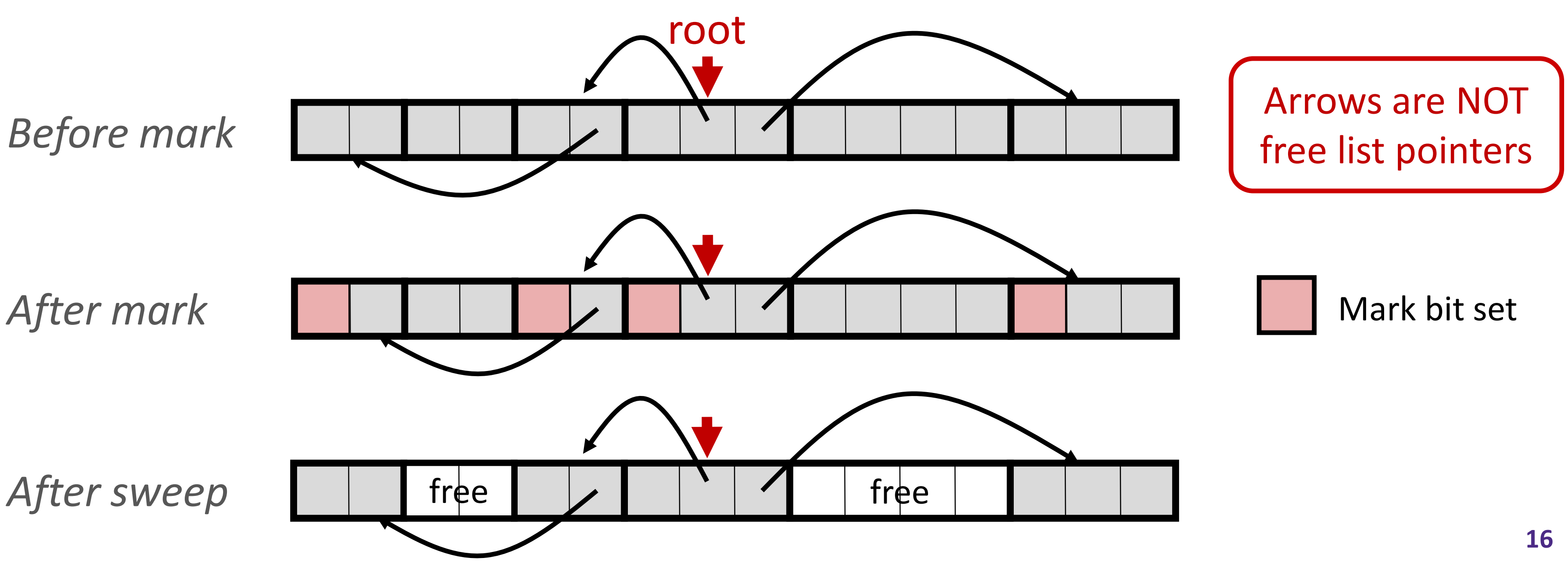
- Dynamic memory allocator can free blocks if there are no pointers to them
- How can it know what is a pointer and what is not?
- We'll make some *assumptions* about pointers:
  - Memory allocator can distinguish pointers from non-pointers
  - All pointers point to the start of a block in the heap
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



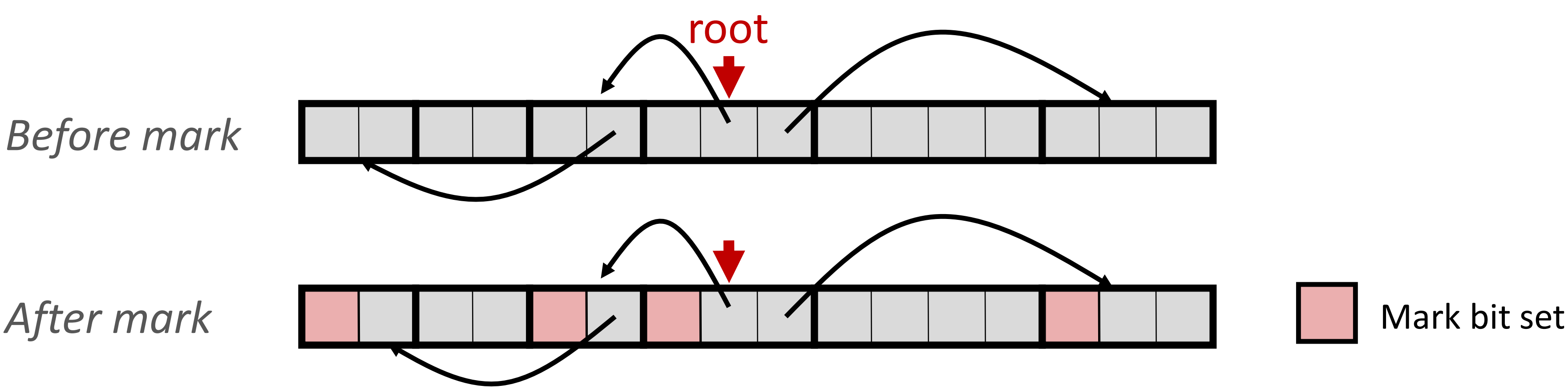
Assumptions For a Simple Implementation

- 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

- Mark using depth-first traversal of the memory graph
  - Start recursive marking from all the roots (`get_roots()`)

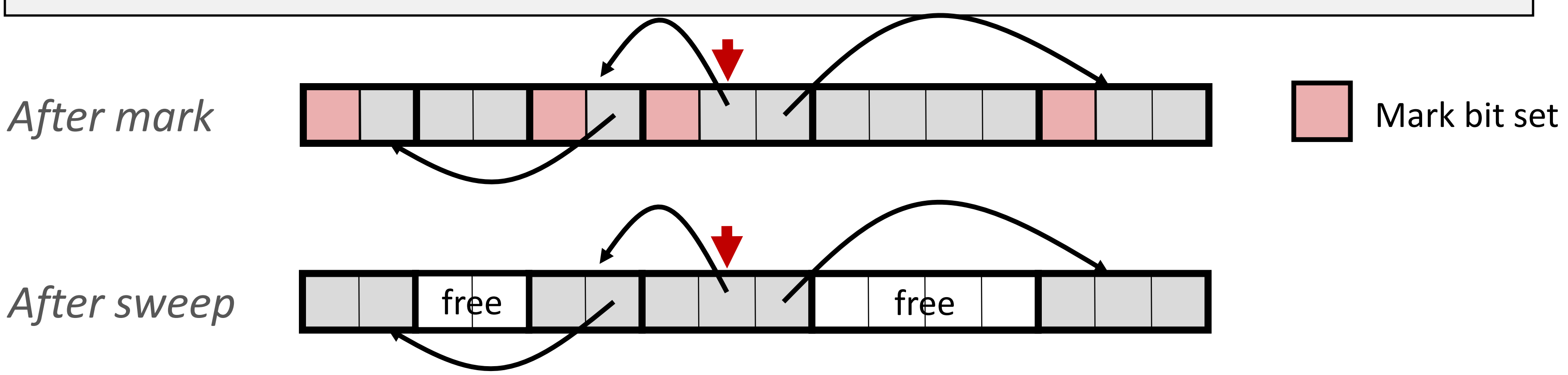
```
ptr mark(ptr p) {                        // p: some word in a heap block
   if (!is_ptr(p))    return;            // do nothing if not pointer
   if (markBitSet(p)) return;            // check if already marked
   setMarkBit(p);                        // set the mark bit
   for (i=0; i<length(p);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



Sweep

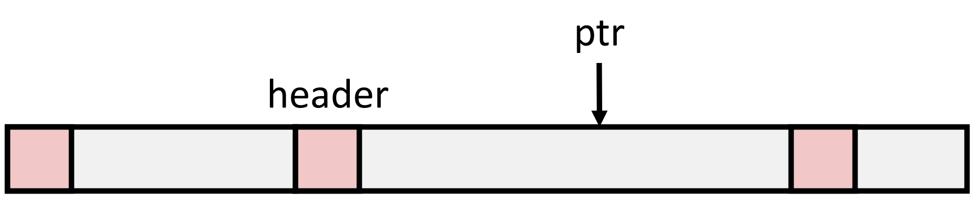
- Sweep using sizes in headers

```
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

- 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

A. **Misunderstanding pointer arithmetic**
B. **Off by one error**
C. **Using a pointer instead of the object it points to (or reverse)**
D. **Not checking the max string size**
E. **Interpreting something that is not a pointer as a pointer**
F. **Failing to free blocks**
G. **Accessing freed blocks or deallocated stack pointers**
H. **Freeing blocks multiple times**
I. **Allocating the (possibly) wrong sized object**
J. **Reading uninitialized memory**
K. **…**

So "what happens"?

- Unlike in "safe" aka "managed" languages, a C program with even one of these (or other errors) can, when executed, do *anything*
- Compiler, especially with higher optimization levels, assumes no such errors exist, and does not worry about what the assembly code might do otherwise
- Therefore, in practice, *debugging*, can involve your 351-level thinking
  - C is a "high-level language" only when it isn't buggy

Dereferencing Bad Pointers

- The classic `scanf` bug

```
int val;

...

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

- Causes `scanf` to interpret contents of `val` as an address!
  - Best case: program terminates immediately due to segmentation fault
  - Worst case: contents of **`val`** correspond to some valid read/write area of virtual memory, causing **`scanf`** to overwrite that memory, with disastrous and baffling consequences much later in program execution

Reading Uninitialized Memory

- Wrongly assuming that heap data is initialized to zero

```
/* 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;
}
```

Overwriting Memory

- Allocating the (possibly) wrong sized object

```
int **p;

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

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

Overwriting Memory

- Off-by-one error

```
int **p;

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

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

Overwriting Memory

- Not checking the max string size

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

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

- Basis for classic buffer overflow attacks
  - Lab 3

Overwriting Memory

- Misunderstanding pointer arithmetic

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

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

   return p;
}
```

Referencing Stack Variables Too Late

- Forgetting that local variables disappear when a function returns (call-stack space reused by subsequent calls)

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

- This will never "mess up" foo, but rather some other code later (if we're lucky, the caller real soon)

Freeing Blocks Multiple Times

- Nasty!

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

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

- What does the free list look like?

Freeing Blocks Multiple Times, Part 2, 3

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

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

void foo(int *y) {
  …
  free(y);
}
```

Referencing Freed Blocks

- Evil!

```
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]++;
```

Failing to Free Blocks (Memory Leaks)

- Slow, silent, long-term killer!

```
void foo() {
   int *x = (int *)malloc(N*sizeof(int));
   ...
   return;
}
```

Failing to Free Blocks (Memory Leaks)

- Freeing only part of a data structure

```
struct list {
   int val;
   struct list *next;
};

foo() {
   struct list *head =
      (struct list *)malloc( sizeof(struct list) );
   head->val = 0;
   head->next = NULL;
   <create and manipulate the rest of the list>
    ...
   free(head);
   return;
}
```

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 (**`x.f = null`**)
- Example: Don't leave big data structures you're done with in a static field