Memory Allocation III
CSE 351 Autumn 2016

Instructor:
Justin Hsia

Teaching Assistants:
Chris Ma
Hunter Zahn
John Kaltenbach
Kevin Bi
Sachin Mehta
Suraj Bhat
Thomas Neuman
Waylon Huang
Xi Liu
Yufang Sun

https://xkcd.com/835/
Administrivia

- Homework 4 due today @ 11:45pm
- Lab 5 due Dec. 9 @ 11:45pm
  - New Lab 5 videos on website!

- **Final Exam:** Tue, Dec. 13 @ 12:30pm in Kane 120
  - Review Session: Sun, Dec. 11 @ 1:30pm in EEB 105
  - Cumulative (midterm clobber policy applies)
  - You get TWO double-sided handwritten 8.5×11” cheat sheets
    - Recommended that you reuse or remake your midterm cheat sheet
  - Reference sheet on website & passed out today
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:
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

![Diagram showing mark and sweep process]

**Before mark**

**After mark**

**After sweep**

Arrows are NOT free list pointers
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

```c
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;
}
```

Before mark

After mark

Mark bit set

Non-testable Material
Sweep

- Sweep using sizes in headers

```c
ptr sweep(ptr p, ptr end) {
    while (p < end) {
        if (markBitSet(p)) {
            clearMarkBit(p);
        } else if (allocateBitSet(p)) {
            free(p);
        }
        p += length(p);
    }
}
```

**Non-testable Material**

After mark

Mark bit set

After sweep

free

free
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

<table>
<thead>
<tr>
<th></th>
<th>Slide</th>
<th>Prog stop Possible?</th>
<th>Security Flaw?</th>
</tr>
</thead>
<tbody>
<tr>
<td>A)</td>
<td>Bad order of operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B)</td>
<td>Bad pointer arithmetic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C)</td>
<td>Dereferencing a non-pointer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D)</td>
<td>Freed block – access again</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E)</td>
<td>Freed block – free again</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F)</td>
<td>Memory leak – failing to free memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G)</td>
<td>No bounds checking</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H)</td>
<td>Off-by-one error</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I)</td>
<td>Reading uninitialized memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J)</td>
<td>Referencing nonexistent variable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K)</td>
<td>Wrong allocation size</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Find That Bug!  (Slide 12)

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

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

Error Type: [ ]
Prog stop Possible? [ ]
Security flaw Possible? [ ]
Fix: [ ]
Find That Bug! (Slide 13)

```c
/* 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 NxN 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: □ □ □ □ □
int **p;

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

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

- N and M defined elsewhere (#define)
int **p;

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

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

Error Type: □ Prog stop □ Security flaw □ Fix: □
Find That Bug! (Slide 16)

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

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

Error Type:  
Prog stop Possible?  
Security flaw Possible?  
Fix:
Find That Bug! (Slide 17)

```c
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 18)

```c
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;
}
```

- ‘--’ and ‘*’ operators have same precedence and associate from right-to-left, so -- happens first.
Find That Bug! (Slide 19)

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

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

Find That Bug! (Slide 20)

```c
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 21)

```c
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 22)

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;
}
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