Memory Allocation III
CSE 351 Spring 2017

Instructor:
Ruth Anderson

Teaching Assistants:
Dylan Johnson
Kevin Bi
Linxing Preston Jiang
Cody Ohlsen
Yufang Sun
Joshua Curtis
Administrivia

- Homework 5 – Due Wed 5/31
- Lab 5 – Due Fri 6/2
  - Do not put this one off!!
  - Get started ASAP!
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

---

**Before mark**

**After mark**

**After sweep**

---

Arrows are NOT free list pointers

![Diagram showing Mark and Sweep process]

Mark bit set

free

free
Assumptions For a Simple Implementation

- Application can use functions to allocate memory:
  - \( b = \text{new}(n) \) 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:
  - \( \text{is_ptr}(p) \) determines whether \( p \) is a pointer to a block
  - \( \text{length}(p) \) returns length of block pointed to by \( p \), not including header
  - \( \text{get_roots}() \) returns all the roots

Non-testable Material
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

Non-testable Material
Sweep

- Sweeping using sizes in headers

```c
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)) {  
            free(p);
            // if not marked, but allocated
            // free the block
        }
        // adjust pointer to next block
        p += length(p);
    }
}
```

Non-testable Material

After mark

![Diagram showing the heap after marking with marked blocks highlighted and pointers adjusted.](image1)

After sweep

![Diagram showing the heap after sweeping with freed blocks highlighted and pointers adjusted.](image2)
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. Failing to Free Blocks
B. Misunderstanding pointer arithmetic
C. Off by one error
D. Freeing Blocks Multiple times
E. Referencing a pointer instead of the object it points to
F. Not checking the max string size
G. Interpreting something that is not a ptr as a ptr
H. Referencing Freed Blocks
I. Referencing nonexistent variables
J. Allocating the (possibly) wrong sized object
K. Reading uninitialized memory
Find That Bug! [1]

- The classic `scanf` bug
  ```c
  int val;
  ...
  scanf("%d", val);
  ```
Find That Bug! [2]

```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 N\times N matrix, x is N-sized vector (so product is vector of size N)
- N defined elsewhere (#define)
Find That Bug! [3]

```c
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)
Find That Bug! [4]

```c
int **p;

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

for (int i = 0; i <= N; i++) {
    p[i] = (int *)malloc( M * sizeof(int) );
}
```
Find That Bug! [5]

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

gets(s);  /* reads “123456789” from stdin */
```
Find That Bug! [6]

```c
int *search(int *p, int val) {
    while (p && *p != val)
        p += sizeof(int);

    return p;
}
```
Find That Bug! [7]

```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;
}
```

- In newer versions of C, the postfix decrement ‘--‘ has higher precedence than ‘*’ so -- happens first.
- In older versions of C, ‘--‘ and ‘*’ operators have same precedence and associate from right-to-left, so -- still happens first
Find That Bug! [8]

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

    return &val;
}
```
Find That Bug! [9]

\[
x = (\text{int}*)\text{malloc}( N \times \text{sizeof(int)} );
\]
<manipulate x>
\[
\text{free}(x);
\]

\[
y = (\text{int}*)\text{malloc}( M \times \text{sizeof(int)} );
\]
<manipulate y>
\[
\text{free}(x);
\]
Find That Bug! [10]

```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]++;
```

```c
void foo() {
    int *x = (int *) malloc( N * sizeof(int) );
    ...
    return;
}
```
Find That Bug! [12]

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