# Heap Heap Hooray: Improving Memory Management

Tyler Gutowski, Trevor Schiff,
Dr. Ryan Stansifer (client)

| Task | Tyler | Trevor |
|---|---|---|
| Analysis of the Garbage Collector Handbook | 0.5 | 0.5 |
| Evaluation of pros & cons | 1.0 | 0.0 |
| Examination of open-source projects | 0.0 | 1.0 |
| Defining objectives and scope | 0.2 | 0.8 |
| Determining vital metrics | 1.0 | 0.0 |
| Assessment of project feasibility | 0.5 | 0.5 |
| Identification of prospective challenges | 0.0 | 1.0 |
| Selection of a suitable garbage collection algorithm | 0.8 | 0.2 |
| Development of high-level design | 0.8 | 0.2 |
| Development of detailed design documents | 0.0 | 1.0 |

| Algorithm | Pro | Cons |
| --- | --- | --- |
| Reference Counting | Simple<br>Real-time reclamation | Counter costs memory<br>Stack variables are inefficient |
| Mark-And-Sweep | GC operations only occur in GC cycles | All memory must be searched during cycles<br>Memory fragmented |
| Copying | Defragments memory | Only a portion of memory is usable |
| Generational | Young is collected frequently<br>Fast GC cycles | Tuning required<br>Only a portion of memory is usable |
| Incremental | No stutters | GC is slower |
| Concurrent/Parallel | No GC cycle stutters | Impractical |

# Basic Overview: Reference Counting

Assume objects are initialized with a reference count of 0

If a reference to an object is created

    The object’s reference count will increment

If a reference to an object is destroyed

    The object’s reference count will decrement

        If the object’s reference count is now zero

            Memory allocated for the object is now freed

# Basic Overview: Mark-and-Sweep

```
Assume each object initialized has a “mark” bit
If an attempt to allocate a new object fails
    For each root object (stack locals, registers)
        For each child heap object
            Mark the object
    For each object on the heap
        If the object isn’t marked
            Free the object
```

# Vital Metrics

- Throughput
  - Allocations per second
  - Reclaims per second
- Pause Times
  - Maximum pause time
  - Average pause time
  - Pause time frequency
- Memory Overhead
  - Total memory allocated
  - Total memory available
- CPU Overhead
  - Resources used by application
  - Resources used by garbage collector

# Feasibility

- Milestone 1 completely completed
- Milestone 2 is definitely feasible
- Make sure to realistically plan for Milestone 2
- Primary and secondary goals
  1. Reference Counting
  2. Mark-and-Sweep
- Wait until progress on Milestone 2 to begin planning Milestone 3

# Design

- Main focus on reference counting
  - Tally all live references to a given block
  - Free block when ref count is decremented to zero (unreachable)
- Mark-sweep GC as secondary implementation
  - RC cannot collect cyclic (self-referencing) garbage
  - Traverses heap block graph by searching block data for pointers
- Implemented in MiniJava runtime, in C
  - Compiler inserts calls during codegen phase
- Header created at start of every heap-allocated block
  - For reference counting: ref count
  - For mark-sweep: size, next block, mark flag

# Heap Block

```
// 32-bit "tag" to identify heap blocks from all other memory
#define HEAP_BLOCK_TAG 'HBLK'

typedef struct HeapBlock {
    // Block identification
    u32 tag;                // at 0x0

    // Next block in list of runtime allocations
    struct HeapBlock* next; // at 0x4
    // Size of this allocation
    size_t size;            // at 0x8

    // Mark bit (for mark-sweep GC)
    s32 marked : 1;         // at 0xC
    // Reference count (for reference count GC)
    s32 ref : 31;           // at 0xC

    // Block user data begins at offset 0x10 . . .
} HeapBlock;
```

# Milestone 2 Plans

- Compiler integration
  - Generate calls to runtime GC functions
- Runtime integration
  - Implement reference counting
    - Increment, decrement functions
  - Begin mark-sweep implementation if time allows
    - Mark, sweep functions
- Performance analysis