Heap Heap Hooray: Memory Management

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

Task	Description	Tyler	Trevor
Heap Allocation Record	Implemented HeapHeader runtime structure to record information about heap allocations.	0.0	1.0
Heap Allocation Functions	Added runtime functions for heap allocation management (allocation, freeing, checking addresses), as well as modifying the reference counter.	0.2	0.8
Reference Counting Initialization Handling	Added functionality to ensure an object is initialized with a counter.	0.5	0.5
Reference Count Management	Implemented functionality in the compiler & runtime to track each object's aliases through its reference counter.	0.5	0.5
Setup SPARC Environment (QEMU, Jabberwocky)	Setting up SPARC on a local machine to run MiniJava compiler. We chose local over the Andrew server because we might need specific permissions in the future for gathering metrics.	1.0	0.0
Memory Tests	Designed and implemented MiniJava test cases to monitor memory management and reference counting GC.	0.5	0.5
Mark-and-Sweep	Stretch plan to implement the mark-and-sweep algorithm.	0.0	0.0

Basic Overview: Reference Counting

Objects are initialized with a reference count of 0

If a reference to an object is created

(Object aliased through assignment or formal parameter)

    The object's reference count will increment

If a reference to an object is destroyed

(Object alias leaves scope)

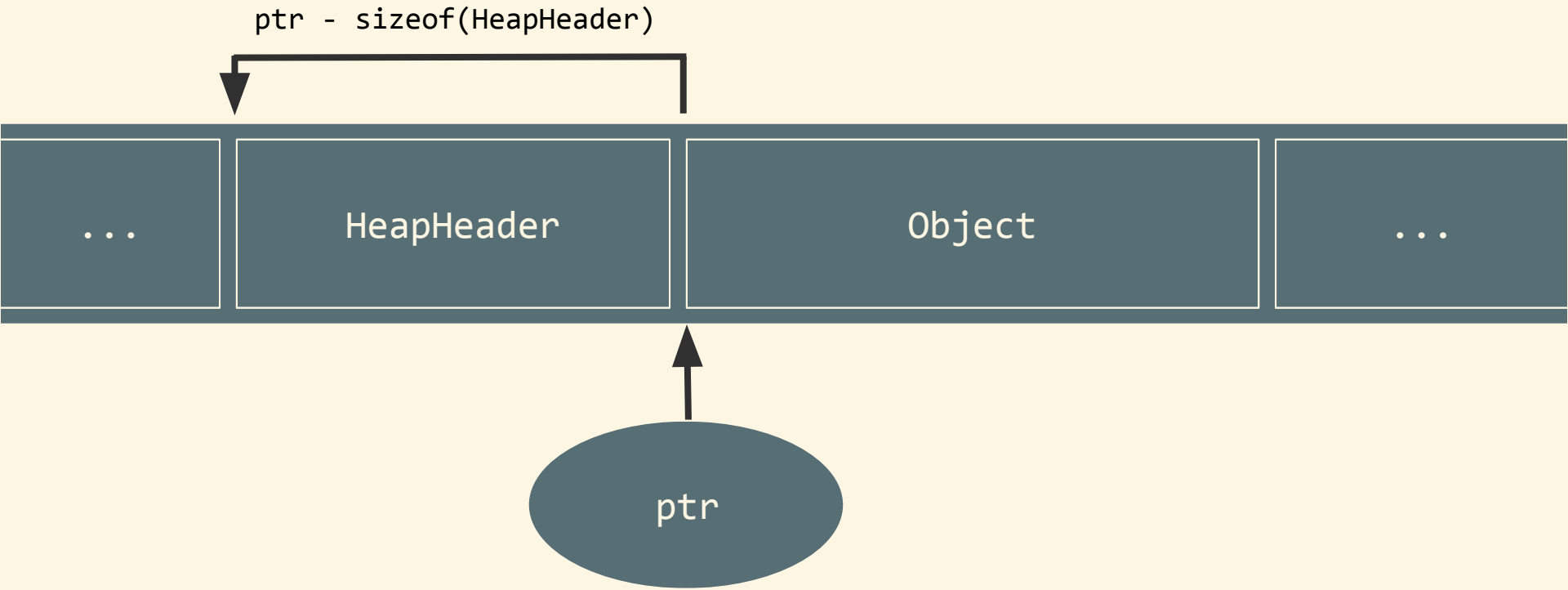
    The object's reference count will decrement

        If the object's reference count is now zero

            Reference count of child references are decremented

            Memory allocated for the object is now freed

Implementation: Reference Counting



Implementation: Reference Counting

```
// 32-bit "tag" to identify heap blocks from all other memory
#define HEAP_BLOCK_TAG 0x48424C4B // 'HBLK'

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

    // Intrusive, doubly-linked list
    struct HeapHeader* next; // at 0x4
    struct HeapHeader* prev; // at 0x8

    // Size of this allocation
    u32 size; // at 0xC

    // Mark bit (for mark-sweep GC)
    s32 marked : 1; // at 0x10
    // Reference count (for reference count GC)
    volatile s32 ref : 31; // at 0x10

    // Block data
    u8 data[]; // at 0x14
} HeapHeader;
```

Heap tag use case:

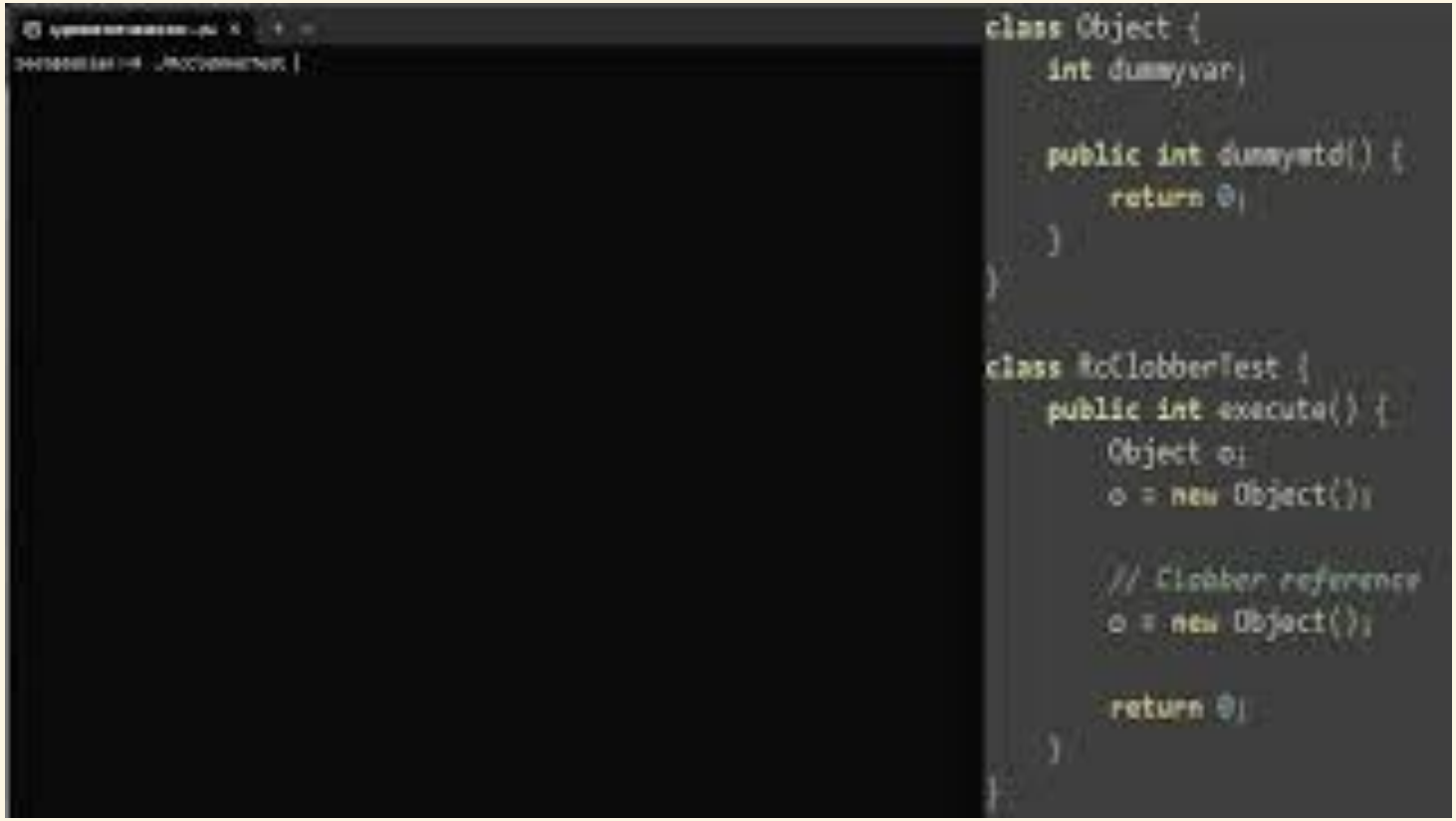
```
void refcount_decr_children(HeapHeader* header) {
    int i;
    void** ptr;
    HeapHeader* child;

    assert(header != NULL);

    // Search block data for pointers
    for (i = 0; i < header->size; i += sizeof(void*)) {
        // Current word of the block
        ptr = (char*)header->data + i;

        // Check for heap block header
        if (heap_is_header(*ptr)) {
            child = heap_get_header(*ptr);
            refcount_decrement(child);
        }
    }
}
```

Implementation: Reference Counting



```
class Object {
    int dummyvar;

    public int dummymtd() {
        return 0;
    }
}

class RcClobberTest {
    public int execute() {
        Object o;
        o = new Object();

        // Clobber reference
        o = new Object();

        return 0;
    }
}
```

```
root@debian:~# ./RcArgumentTest
[heap] alloc 0x23008 (size:4)
[refcount] increment 0x23008, now 1
[heap] alloc 0x23028 (size:4)
[refcount] increment 0x23028, now 1
[refcount] increment 0x23008, now 2
[refcount] increment 0x23028, now 2
[refcount] decrement 0x23008, now 1
[heap] alloc 0x23048 (size:4)
[refcount] increment 0x23048, now 1
[refcount] decrement 0x23028, now 1
[refcount] increment 0x23048, now 2
[refcount] decrement 0x23048, now 1
[refcount] decrement 0x23048, now 0
[refcount] free 0x23048
[refcount] decrement 0x23008, now 0
[refcount] free 0x23008
[refcount] decrement 0x23028, now 0
[refcount] free 0x23028
```

```
RcArgumentTest$fn$prologueEnd:
    mov %i1, %o0
    call runtime_ref_inc
    nop
    mov %i2, %o0
    call runtime_ref_inc
    nop
    mov %i1, %o0
    call runtime_ref_dec
    nop
    set 4, %o0
    call runtime_alloc_object
    nop
    mov %o0, %i1
    mov %i1, %o0
    call runtime_ref_inc
    nop
    mov %i2, %o0
    call runtime_ref_dec
    nop
    mov %i1, %i2
    mov %i2, %o0
    call runtime_ref_inc
    nop
    mov %i1, %o0
    call runtime_ref_dec
    nop
    mov %i2, %o0
    call runtime_ref_dec
    nop
    set 0, %i0
RcArgumentTest$fn$epilogueBegin:
```

```
class Main {
    public static void main(String[] a) {
        System.out.println(new RcArgumentTest().execute());
    }
}

class Object {
    int dummy;
}

class RcArgumentTest {
    public int execute() {
        Object a;
        Object b;

        a = new Object();
        b = new Object();
        return this.fn(a, b);
    }

    public int fn(Object c, Object d) {
        c = new Object();
        d = c;
        return 0;
    }
}
```

Milestone 3 Goals

- Mark-sweep GC as primary implementation
  - RC cannot collect cyclic (self-referencing) garbage
  - Traverses heap block graph by searching block data for pointers
- Copying GC as secondary implementation
  - Mark-sweep fragments the memory, which leaves some of it unusable
  - Copying defragments
- Run tests with all 3 GC implementations
  - Compare tests with vital metrics