Garbage Collection

- Learning Objectives
  - Explain the relationship between a language with explicit malloc/free and one that uses garbage collection.
  - Build a simple garbage collector for C.

With Power comes Responsibility

- Assignment 1 demonstrates that there are many ways you can introduce errors in the use of malloc/ free.
- Garbage collected languages (e.g., Java) are an alternative: programmers can create objects all they want, but they are not responsible for freeing them.
  - Instead, a garbage collector goes and finds all “live” objects and frees everything else.
  - Objects are “live” if they are reachable by some “live” variable.
- Malloc/free give you direct control over memory, but they also allow you to do bad/dangerous things.

Let’s build a Garbage Collector!

Overall Strategy: C Mark and Sweep

- Keep track of all memory allocations.
- Claim:
  - If we can find all the pointers to dynamically allocated structures, then we can mark all such structures as “live.”
  - We can safely free any allocated structures that are not “live.”
- Where might we find pointers to dynamically allocated data?
  - In variables
  - In dynamically allocated data (i.e., think of a linked list)
- Tricky part: how do we find all our variables?

Easy Part: Tracking Allocations

- `m61_malloc`
  - Must record all allocations.
- `m61_free`
  - Must find allocation being freed and remove it from the set of all allocations.
- Code is all in the `cs61-videos` repo in the `gc` directory. I'll talk through most of the code, but may skip a few helper functions.

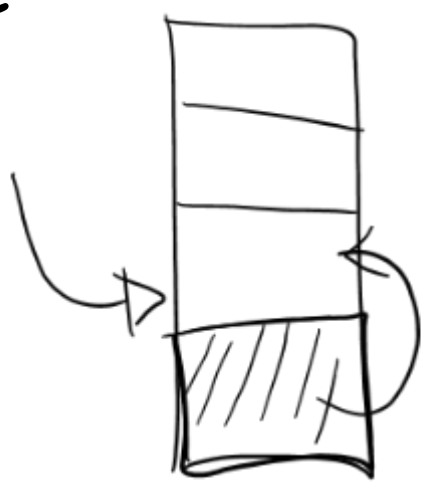
Data Structures

```
/* This is the information we keep for each allocation. */
typedef struct memregion {
    char *ptr;
    size_t sz;
    int mark;  /* Used to mark in-use regions. */
} memregion;

static memregion *mr; // Holds metadata for the region
static size_t nmr;    // Number of allocations currently
                      // in the region
static size_t mr_capacity; // Maximum number of allocations
                           // the region can hold
static void m61_mr_grow(void);  // Function that will grow
                                // the region
```

m61_malloc

```
void *
m61_malloc(size_t sz) {
    void *ptr;

    ptr = malloc(sz);
    if (ptr) {
        // keep track of allocated regions in `mr` array
        if (nmr == mr_capacity)
            m61_mr_grow();
        mr[nmr].ptr = ptr;
        mr[nmr].sz = sz;
        // We don't need to initialize mark as we always clear these
        // before we start gc.
        ++nmr;
    }
    return (ptr);
}
```

m61_free



```
void
m61_free(void* ptr) {
    if (ptr) {
        memregion *m;

        m = m61_find_mr(ptr);
        assert(m);
        assert(m->ptr == ptr);
        free(ptr);

        // Move the last entry in the mr array into the position
        // occupied by this allocation and then decrement the total
        // number of allocations.
        *m = mr[nmr - 1];
        --nmr;
    }
}
```

The tricky part

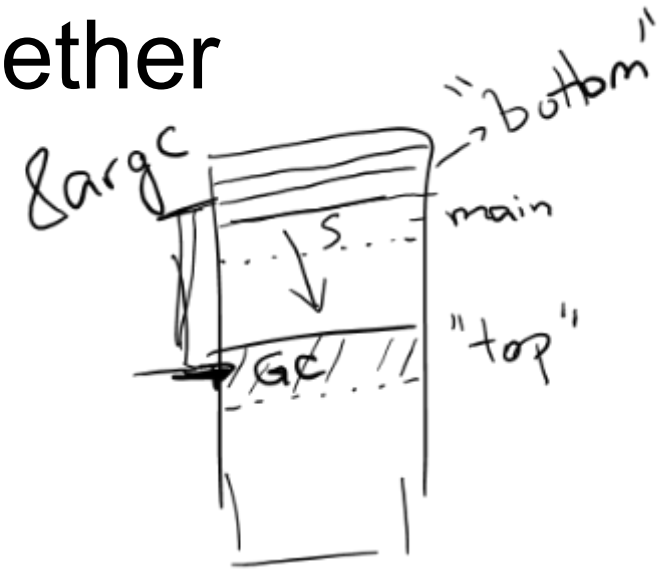
- Now we have a record of all the allocations – how do we find all the live variables that might reference memory?
- Let's start with a building block:
  - `m61_find_allocations`
  - Given a range of addresses, check each one and see if it falls into a dynamically allocated region.
  - If it does, mark that region as active.
  - Recurse: check that region to see if it also contains pointers to dynamically allocated regions.

m61_find_allocations

```
void
m61_find_allocations(char* base, size_t sz) {
    memregion *m;
    size_t i;
    void *ptr;

    for (i = 0; i + sizeof(void *) - 1 < sz; ++i) {
        memcpy(&ptr, &base[i], sizeof(void *));
        m = m61_find_mr(ptr);
        if (m && !m->mark) {
            m->mark = 1;
            m61_find_allocations(m->ptr, m->sz);
        }
    }
}
```

Putting it all together

```
void m61_gc(void) {
    char *stack_top;
    size_t i;

    stack_top = (char*) &stack_top;
    /* Clear all the mark bits.  */
    for (i = 0; i != nmr; ++i)
        mr[i].mark = 0;
    /* Mark all the allocations. */
    m61_find_allocations(stack_top, stack_bottom - stack_top);
    /* Anything not marked can be freed. */
    for (i = 0; i != nmr; ++i)
        if (!mr[i].mark) {
            m61_free(mr[i].ptr);
            --i;
        }
}
```



Shortcomings:

- Kind of dumb: if we see a value of 0 in memory, we shouldn't bother calling `m61_find_allocations`.
- Conservative: if we happen to store a value in memory that happens to look like a pointer into malloc'd memory, we'll treat it as such.
- Dangerous: doesn't look through the global space for pointers!
- Not as clever as it might be: we really ought to call the garbage collector any time malloc fails, so we can try again.

Wrapping Up

- C isn't a garbage collected language, but we have the power to create tools to make it behave like one.
- There is nothing magic about systems software – systems software are just programs that happen to know a bit more about what's going on than your typical application.
- Abstractions are useful, but understanding what is underneath those abstractions is also useful!