Lecture 18 - Space: Allocation and Memory (optional) (last updated: 01/26/23)

In this course, we learned about memory, pointers, and allocation through the lens of the C language. Even though C's memory model make C programming notoriously cumbersome and error-prone, it also brings us much closer to understanding how program memory works from the perspective of operating systems and computer hardware. After all, these concepts exist in all computer programs; other languages simply do a better job of these low-level concerns away from programmers, at the cost of C's precision and acuity.

In this optional lecture, we will dive deeper into the subject of memory and see how some other languages avoid the memory hazards that C exposes programmers to.

Allocation

Almost all languages support some kind of heap allocation, even if they don't call it that. For example, objects in object-oriented languages like Java, Python, and JavaScript are all heap-allocated, as are closures in functional languages like Haskell and OCaml, and strings in string-manipulating languages like Bash and AWK. Though implementation details vary widely between these languages, under the hood they are all fundamentally doing the same thing as malloc(): allocating some buffer for storing data, to be freed at a later time.

That memory does not appear out of thin air; a memory allocator usually searches among available space for a contiguous range of memory to allocate, or asks for more space from the operating system if no more space is immediately available. The allocation process can be very complex and time-consuming, and the design of specific allocation algorithms is beyond the scope of this lecture.

However, not all allocators have to be complex. For example, here is the implementation of malloc() and free() for an extremely simple "bump allocator":

```
char _heap[2 << 16];        // Some global buffer to allocate memory from.
char *_free_ptr = _heap;    // Keep track of what we've already allocated.

void *malloc(size_t size) {
    void *ptr = _free_ptr;
    _free_ptr += size;      // Ensure we don't allocate this memory again.

    if (_free_ptr > _heap + sizeof(_heap))
        return NULL;        // We ran out of space!

    return ptr;             // Return pointer to within _heap.
}

void free(void *p) {
    // Do nothing to reclaim space; if we run out, we run out!
}
```

Because bump allocators do not meaningfully free() individually allocated chunks of memory, they cannot be used indefinitely without occasionally freeing all allocated memory at once:

```
void free_all(void) {
    _free_ptr = _heap;      // All previously allocated memory is freed now.
}
```

Though bump allocators are too simplistic and limiting for general-purpose use, they turn out to be perfectly fine for particular kinds of workloads that rapidly allocate lots of variable-size chunks, before freeing them all at once.

The point here is that allocators are nothing magical; malloc() and free() are just regular functions that happen to play the pivotal role of providing the C programming language (and other languages built on top of C) with a heap. These functions just happen to be part of the C standard library, and can be replaced. Large software companies like Google and Meta even maintain their own malloc() and free(), tailored to their specific workload needs.

Some additional considerations when designing and comparing allocators:

- Some programs require allocated memory to be aligned to a particular size; that is, the address of the allocated memory should be a multiple of that size. Real-world allocators need to account for these alignment needs.

- Allocators usually need to maintain metadata, such as the size and alignment of each allocated chunk, or the location and size of available space.

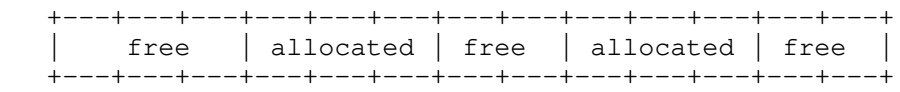
  Allocators need to store this metadata in a location that is convenient to access, and in a space-efficient format. For instance, some allocators store the chunk size in the few bytes before the chunk address returned by malloc(); other allocators may store this metadata in a separate table.

  Many allocators also make use of the free space to store metadata. For example, some allocators use free space to build a "free list", a linked list to help malloc() quickly find available free space.

- Allocators cannot always make the most efficient use out of available space; this wasted space is known as "fragmentation".

  There are two types of fragmentation: internal and external. Internal fragmentation occurs when larger chunk sizes are allocated than requested, usually to satisfy alignment needs. The unused space in these chunks is wasted.

  External fragmentation occurs because the layout of existing chunks in the heap does not allow the remaining space to be reused for subsequent allocations. For example, suppose the heap looks like this:



  There are 7 bytes of available space, at most 3 bytes can be allocated, because there are no contiguous chunks of free space larger than 3 bytes.

Automatic garbage collection

Because C requires programmers to manage memory manually, C programs are prone to memory leaks. Informally speaking, a memory leak refers to memory that is no longer useful, but is never freed. Since what constitutes "useful" largely depends on the specific programming using that memory, we will adopt a stricter definition of a memory leak: heap memory is leaked if it is no longer reachable. Specifically, a chunk of memory is reachable if:

- it is a global or local (stack) variable; or
- it is pointed to by another piece of reachable memory.

This notion of reachability might make more sense if we think of memory as a graph, where chunks of memory are nodes and pointers form edges.

Most modern programming languages rely on garbage collectors (GCs) to automatically free unreachable memory, to free programmers from the burden of manually managing memory. The most common type of GC is "tracing" GC, where the collector periodically pauses the program to scan local and global variables for heap pointers, and identifying reachable chunks of memory by recursively following (i.e., "tracing") those pointers. Any memory that isn't reachable is automatically freed and reclaimed by the allocator for reuse. This most basic form of tracing GC is known as mark and sweep GC. Most modern GCs use more sophisticated collection algorithms to reduce the pause times and fragmentation that naive collectors and allocators are prone to.

GC (ca. 1960s) actually predates the C language (ca. 1970s); so why didn't C include GC? Because C permits arbitrary pointer arithmetic, the language makes it very difficult for collectors to determine what is and isn't a pointer. After all, pointers are just numbers that sometimes refer to valid memory locations, and programs can do whatever they want to numbers.

For instance, consider the following contrived allocation function, collam(), which returns the bitwise complement of a heap-allocated pointer:

```
void *collam(size_t size) {
    unsigned long p = (unsigned long) malloc(size);
    return (void *) ~p;
}
```

In memory, the "pointer" collam() it returns looks nothing like memory address it refers to, but you can still "dereference" it by flipping the bits again:

```
int *p = collam(sizeof(int));
*(int *) ~(unsigned long) p = 42;           // Write 42 to ~p
assert(*(int *) ~(unsigned long) p == 42);  // Read 42 from ~p
```

When the collector comes across p on the stack, it will see a number referring to a completely different memory address than what p actually "points" to, preventing it from correctly inferring the reachability of the memory allocated by collam().

Is this a gross misuse of C? Almost certainly. But even though C programmers probably shouldn't code like this, most still expect this to work. And there are certainly legitimate uses for seemingly dangerous pointer arithmetic in C (e.g., arrays!), which is part of what makes the language well-suited for writing low-level code. For instance, collectors themselves are typically written in C (or C++), to support the kind of pointer manipulation involved in

garbage collection.

However, that low-level flexibility makes it difficult for the language to safely provide high-level language features. Higher-level, garbage-collected languages like Java and Haskell usually avoid these kinds of headaches by simply forbidding arbitrary pointer arithmetic, and placing more guard rails on memory allocations and accesses.

Reference counting

Though C itself does not have tracing garbage collection, many C libraries and programs use another form of GC called reference counting. Reference counting makes it easier to determine when a heap-allocated object should be freed, which allows the program to distribute the responsibility of freeing heap objects across different parts of the program. This technique is useful when multiple data structures share pointers to the same heap object.

With reference counting, heap objects include a "reference count" that keep track of how many pointers point to that object. For example, here is a simple struct definition and constructor for a reference-counted integer object:

```
struct Int {
    int data;
    size_t refs;
};

struct Int *makeInt(int i) {
    struct Int *p = malloc(sizeof(struct Int));
    p->data = i;
    p->refs = 1;
    return p;
}
```

alloc_int() returns a pointer to a heap-allocated integer alongside its reference count. Using that pointer, we can access that integer data embedded in the struct Int, like a regular pointer, using derefInt(), while we use the dupInt() and dropInt() methods to maintain the reference count:

```
int derefInt(struct Int *p) {
    return p->data;
}

void dupInt(struct Int *p) {
    p->refs++;
}

void dropInt(struct Int *p) {
    if (--p->refs == 0)
        free(p);
}
```

dupInt() should be called when one part of the program shares a struct Int pointer with another part of the program, while dropInt() should be called when one part of the program has finished using a struct Int. As long as all parts of the program consistently do so, it will not leak any struct Int objects.

For example, consider the putMax() function, which saves a pointer to the largest struct Int in a global variable that can retrieved using getMax():

```
struct Int *maxInt = NULL;

void putMax(struct Int *p) {
    if (maxInt == NULL) {
        maxInt = p;
    } else if (p->data > maxInt->data) {
        dropInt(maxInt);                // No longer need the old maxInt
        maxInt = p;
    } else {
        dropInt(p);                     // No longer need p
    }
}

int getMax(void) {
    if (maxInt == NULL)
        return 0;
    else
        return maxInt->data;
}
```

Note that putMax() calls dropInt() on the struct Int object that it isn't saving to the maxInt variable, which may be different depending on the relative size of the integers in the old maxInt and the argument p.

Now, we can write other functions that allocate and use struct Int objects without worrying about who is specifically responsible for free()ing those objects:

```
void foo(void) {
    struct Int *p = makeInt(42);    // Initialize ref count to 1

    dupInt(p);                      // Increment ref count to 2 before
                                    // sharing with putMax()
    putMax(p);

    // Do something with p...

    dropInt(p);                     // Decrement ref count +
                                    // free() p if ref count is 0

    // We can still use getMax() without worrying about memory errors!
}
```

foo() increments the reference count of p before sharing it with putMax(). After putMax() returns, foo() doesn't know whether putMax() is actually done using p. There are two possibilities:

- If putMax() is also done using p, because maxInt->data was greater than p->data, then it should have also called dropInt() before returning. This restores the reference count back to 1, reflecting the fact that foo() holds the only remaining reference to the object.

  When foo() is done using p, it calls dropInt() too, decrementing the reference count to 0 and automatically free()ing p.

- If putMax() is not done using p, because p->data was greater than maxInt->data, then it should have left p's reference counter intact at 2,

reflecting the fact that both foo() and maxInt hold references to the same struct Int object.

When foo() is done using p, it calls dropInt() to decrement p's reference count, but does not free() p due to its non-zero reference count. In this scenario, the reference count prevents foo() from turning maxInt into a dangling pointer.

When used consistently, this kind of pointer-sharing scheme helps C programs avoid memory leaks and errors. For instance, it is used in the Linux kernel to manage the lifetimes of resources shared between processes. Many modern systems languages like C++, Swift, and Rust also support reference counting in lieu of tracing GC, and provide facilities to automatically insert dup() and drop() calls where appropriate, freeing the programmer from the burden of manual memory memory management.