Intro to value ownership

A program is nothing without data, and all modern programming languages store data in one of two places: the call stack and the heap (also sometimes in CPU registers, but we won't get into that here). However, each language reads and writes data a bit differently—sometimes very differently. So in the following sections, we'll explain how Mojo manages memory in your programs and how this affects the way you write Mojo code.

> **Note**
>
> For an alternate introduction to ownership in Mojo, check out our two-part blog post: What ownership is really about: a mental model approach, and Deep dive into ownership in Mojo.

Stack and heap overview

In general, all modern programming languages divide a running program's memory into four segments:

- Text. The compiled program.
- Data. Global data, either initialized or uninitialized.
- Stack. Local data, automatically managed during the program's runtime.
- Heap. Dynamically-allocated data, managed by the programmer.

The text and data segments are statically sized, but the stack and heap change size as the program runs.

The *stack* stores data local to the current function. When a function is called, the program allocates a block of memory—a *stack frame*—that is exactly the size required to store the function's data, including any *fixed-size* local variables. When another function is called, a new stack frame is pushed onto the top of the stack. When a function is done, its stack frame is popped off the stack.

Notice that we said only "*fixed-size* local values" are stored in the stack. Dynamically-sized values that can change in size at runtime are instead stored in the heap, which is a much larger region of memory that allows for dynamic memory allocation. Technically, a local variable for such a value is still stored in the call stack, but its value is a fixed-size pointer to the real value on the heap. Consider a Mojo string: it can be any length, and its length can change at runtime. So the Mojo `String` struct includes some statically-sized fields, plus a pointer to a dynamically-allocated buffer holding the actual string data.

Another important difference between the heap and the stack is that the stack is managed automatically—the code to push and pop stack frames is added by the compiler. Heap memory, on the other hand, is managed by the

programmer explicitly allocating and deallocating memory. You may do this indirectly—by using standard library types like `List` and `String`—or directly, using the `UnsafePointer` API.

Values that need to outlive the lifetime of a function (such as an array that's passed between functions and should not be copied) are stored in the heap, because heap memory is accessible from anywhere in the call stack, even after the function that created it is removed from the stack. This sort of situation—in which a heap-allocated value is used by multiple functions—is where most memory errors occur, and it's where memory management strategies vary the most between programming languages.

Memory management strategies

Because memory is limited, it's important that programs remove unused data from the heap ("free" the memory) as quickly as possible. Figuring out when to free that memory is pretty complicated.

Some programming languages try to hide the complexities of memory management from you by utilizing a "garbage collector" process that tracks all memory usage and deallocates unused heap memory periodically (also known as automatic memory management). A significant benefit of this method is that it relieves developers from the burden of manual memory management, generally avoiding more errors and making developers more productive. However, it incurs a performance cost because the garbage collector interrupts the program's execution, and it might not reclaim memory very quickly.

Other languages require that you manually free data that's allocated on the heap. When done properly, this makes programs execute quickly, because there's no processing time consumed by a garbage collector. However, the challenge with this approach is that programmers make mistakes, especially when multiple parts of the program need access to the same memory—it becomes difficult to know which part of the program "owns" the data and must deallocate it. Programmers might accidentally deallocate data before the program is done with it (causing "use-after-free" errors), or they might deallocate it twice ("double free" errors), or they might never deallocate it ("leaked memory" errors). Mistakes like these and others can have catastrophic results for the program, and these bugs are often hard to track down, making it especially important that they don't occur in the first place.

Mojo uses a third approach called "ownership" that relies on a collection of rules that programmers must follow when passing values. The rules ensure there is only one "owner" for a given value at a time. When a value's lifetime ends, Mojo calls its destructor, which is responsible for deallocating any heap memory that needs to be deallocated.

In this way, Mojo helps ensure memory is freed, but it does so in a way that's deterministic and safe from errors such as use-after-free, double-free and memory leaks. Plus, it does so with a very low performance overhead.

Mojo's value ownership model provides an excellent balance of programming productivity and strong memory safety. It only requires that you learn some new syntax and a few rules about how to share access to memory within your program.

But before we explain the rules and syntax for Mojo's value ownership model, you first need to understand value semantics.