9b. Stack vs Heap

Martin Alfaro
PhD in Economics

INTRODUCTION

Memory allocations are a fundamental process in computer programming that occurs whenever a new object is created. It involves setting aside a portion of the computer's Random Access Memory (RAM) to store the object's data. Conceptually, RAM is logically divided into two main areas: the stack and the heap. Importantly, these areas aren't physical locations, but rather conceptual models that govern how memory is managed.

The choice between stack and heap allocation has profound implications for performance. Allocating memory on the heap is a comparatively expensive operation. It requires a systematic search for an available block of memory, bookkeeping to track its status, and an eventual deallocation process to reclaim the space once it's no longer needed. This deallocation process is often handled by a system known as a garbage collector. Stack allocations, by contrast, are simpler and therefore more efficient.

The performance gap between stack and heap allocations can easily become a critical bottleneck if the operation is performed repeatedly. This disparity in performance explains the common convention in programming, including Julia, where **memory allocations will exclusively refer to heap allocations**. While stack allocations are technically a form of memory allocation, their speed and simplicity mean they rarely warrant the same level of scrutiny.

With this distinction in mind, the current section begins our exploration of memory allocations by taking a closer look at the mechanics of the stack and the heap.

STACK ALLOCATIONS

In Julia, typical objects stored on the stack include integers, numbers, characters, and small fixed-size collections like tuples.

Objects on the stack are characterized by having a fixed size, precluding the possibility of dynamically growing or shrinking in size. These characteristics make allocating and deallocating memory on the stack extremely efficient.

The primary limitation of the stack is its limited capacity, making it suitable only for objects with a few elements. Indeed, attempting to allocate more memory than the stack can accommodate will result in a "stack overflow" error, causing program termination. And, even if an object fits on the stack, allocating too many elements can significantly degrade performance. [1]

HEAP ALLOCATIONS

Common objects stored on the heap include arrays (such as vectors and matrices) and strings. Unlike the stack, the heap is designed to allocate and free memory in any order, allowing it to accommodate larger and more complex data structures. Thus, the heap can handle objects as large as the available RAM permits, which additionally could grow or shrink dynamically.

While the heap offers greater flexibility than the stack, its more complex memory management comes at the cost of slower performance. [2] Due to their reduced speed, Chapter 9 will outline approaches to minimizing heap allocations. Strategies for achieving this includes utilizing the stack whenever possible, and favoring mutation over the creation of new objects.

FOOTNOTES

[1.] There's no hard and fast rule about how many elements are "too many". Benchmarking is the only reliable way to determine the performance consequences for each particular case. As a rough guideline, objects with more than 100 elements will certainly suffer from poor performance, while those with fewer than 15 elements are likely to benefit from stack allocation.

[2.] To handle the memory of heap-allocated objects, Julia uses what's known as a *garbage collector*. This mechanism automatically identifies and frees memory no longer in use, which can be computationally expensive.