9c. Objects Allocating Memory

Martin Alfaro
PhD in Economics

INTRODUCTION

In the previous section, we outlined the basic ideas behind memory allocation, noting that objects may reside either on the stack (or even in CPU registers) or on the heap. We also adopted a common convention in programming language discussions: **memory allocations exclusively refer to those on the heap**. This convention isn't merely to economize on words. Rather, it highlights that heap allocations are the ones that meaningfully affect performance. They require more elaborate bookkeeping, can introduce latency, and often become a dominant source of overhead in performance-critical code.

Julia's own benchmarking tools reinforce this connection between performance and heap activity. Macros such as `@time` and `@btime` don't just measure execution time, but also report the number and size of heap allocations involved. This dual reporting encourages developers to think about performance not only in terms of speed, but also in terms of allocation patterns.

Given the importance of understanding when allocations occur, this section classifies objects according to whether they allocate on the heap or avoid allocation altogether. This distinction will guide our analysis of performance throughout the remainder of the chapter.

NUMBERS, TUPLES, NAMED TUPLES, AND RANGES DON'T ALLOCATE

We start by presenting objects that don't allocate memory. They include:

- Scalars (numbers)
- Tuples
- Named Tuples
- Ranges

As they don't allocate, neither does their creation, access, or manipulation. This is demonstrated below.

```
function foo()
    x = 1; y = 2

    x + y
end
```

```
julia> @btime foo()
  0.914 ns (0 allocations: 0 bytes)
```

```
function foo()
    tup = (1,2,3)

    tup[1] + tup[2] * tup[3]
end
```

```
julia> @btime foo()
  0.932 ns (0 allocations: 0 bytes)
```

```
function foo()
    nt = (a=1, b=2, c=3)

    nt.a + nt.b * nt.c
end
```

```
julia> @btime foo()
  0.835 ns (0 allocations: 0 bytes)
```

```
function foo()
    rang = 1:3

    sum(rang[1:2]) + rang[2] * rang[3]
end
```

```
julia> @btime foo()
  0.910 ns (0 allocations: 0 bytes)
```

ARRAYS AND THEIR SLICES DO ALLOCATE MEMORY

Arrays are among the most common heap-allocated objects in Julia. A new allocation occurs not only when you explicitly construct an array and assign it to a variable, but also whenever an expression implicitly produces a fresh array as part of its computation. The examples below illustrate both situations.

```
foo() = [1,2,3]
```

```
julia> @btime foo()
  13.000 ns (2 allocations: 80 bytes)
```

```
foo() = sum([1,2,3])
```

```
julia> @btime foo()
  7.938 ns (1 allocations: 48 bytes)
```

Slicing is another operation that results in memory allocation. By default, a slice produces a new array that copies the selected elements, instead of creating a lightweight view over the original data. This behavior ensures isolation between the slice and its source, but it also means that each slicing operation allocates fresh storage. The only exception occurs when a single element is accessed, in which case no allocations take place.

```
x      = [1,2,3]

foo(x) = x[1:2]                    # allocations only from 'x[1:2]' itself (ranges don't
allocate)
```

```
julia> @btime foo($x)
  13.405 ns (2 allocations: 80 bytes)
```

```
x      = [1,2,3]

foo(x) = x[[1,2]]                  # allocations from both '[1,2]' and 'x[[1,2]]' itself
```

```
julia> @btime foo($x)
  24.094 ns (4 allocations: 160 bytes)
```

```
x      = [1,2,3]

foo(x) = x[1] * x[2] + x[3]
```

```
julia> @btime foo($x)
  1.711 ns (0 allocations: 0 bytes)
```

Array comprehensions and broadcasting are two more constructs that result in fresh arrays. Notably, broadcasting also allocates memory for intermediate results computed on the fly, even when those values aren't explicitly returned. This behavior is demonstrated in the tab "Broadcasting 2" below.

```
foo()  = [a for a in 1:3]
```

```
julia> @btime foo()
  12.361 ns (2 allocations: 80 bytes)
```

```
x      = [1,2,3]
foo(x) = x .* x
```

```
julia> @btime foo($x)
  15.596 ns (2 allocations: 80 bytes)
```

```julia
x      = [1,2,3]
foo(x) = sum(x .* x)                # allocations from temporary vector 'x .* x'
```

```
julia> @btime foo($x)
  20.165 ns (2 allocations: 80 bytes)
```