9c. Objects Allocating Memory

Martin Alfaro
PhD in Economics

INTRODUCTION

In the previous section, we introduced the fundamentals of memory allocations, highlighting that objects can be stored on either the heap or the stack. Furthermore, we emphasized the application of conventional terminology in programming and Julia, where **allocations exclusively refer to those on the heap**. This definition also gives rise to the expression that "an object allocates" when the object is stored on the heap.

The distinction isn't merely to economize on words. Rather, it reflects that heap allocations are the ones that matter when it comes to performance: they involve a more complex memory management, which can significantly hinder performance.

In fact, the close relationship between performance and heap allocations can be appreciated through the macros `@time` and `@btime`. To provide a comprehensive measure of performance, they not only return the total runtime of an operation, but additionally the heap allocations involved.

In the following, we initiate our analysis of memory allocation by categorizing objects into two groups: those that allocate and those that don't.

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

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

- Numbers
- Tuples
- Named Tuples
- Ranges

As these objects don't allocate, neither does creating, accessing, and operating on them. This is demonstrated below.

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

    x + y
end
```

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

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

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

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

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

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

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

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

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

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

ARRAYS AND SLICES DO ALLOCATE MEMORY

The most common object that allocates memory is arrays. These allocations occur not only when we create an array and assign it to a variable, but also when computations returning arrays are performed on the fly. The following example illustrates this point.

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

```
julia> @btime foo()
  13.714 ns (1 allocation: 80 bytes)
```

Slicing is another operation that creates an array, and therefore allocates. This is due to the default behavior of slicing, which returns a new copy rather than a view of the original object. The sole exception to this rule is when a single element is accessed, in which case no new allocation occurs.

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

foo(x) = x[1:2]                 # ONE allocation, since ranges don't allocate (but 'x[1:2]'
itself does)
```

```
julia> @btime foo($x)
  16.116 ns (1 allocation: 80 bytes)
```

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

foo(x) = x[[1,2]]                    # TWO allocations (one for '[1,2]' and another for
'x[[1,2]]' itself)
```

```
julia> @btime foo($x)
  31.759 ns (2 allocations: 160 bytes)
```

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

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

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

Other operations that involve array creation include array comprehensions and broadcasting. Remarkably, broadcasting even involves memory allocation when intermediate results are computed internally, but not explicitly returned. This specific case is demonstrated in "Broadcasting 2" below.

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

```
julia> @btime foo()
  13.514 ns (1 allocation: 80 bytes)
```

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

```
julia> @btime foo($x)
  15.916 ns (1 allocation: 80 bytes)
```

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

```
julia> @btime foo($x)
  21.242 ns (1 allocation: 80 bytes)
```