9c. Objects Allocating Memory

Martin Alfaro
PhD in Economics

INTRODUCTION

In the previous section, we introduced the fundamentals of memory allocations, noting that objects can be stored on either the heap or the stack. Furthermore, we introduced typical terminology, where **allocations exclusively refer to those on the heap**. This convention underlies the common expression that an object "allocates" when it's stored on the heap.

The distinction isn't merely to economize on words. Rather, it reflects a fundamental performance implication: heap allocations are the ones significantly impacting efficiency. They involve a more complex management process than the stack, thus potentially introducing significant overhead.

The intimate relationship between performance and heap allocations is even reflected in Julia's built-in benchmarking tools. Macros like `@time` and `@btime` report the total runtime of an operation, along with the heap allocations involved.

Considering the importance of memory allocations on the heap, the current section categorizes objects into 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 their creation, access, or manipulation. 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 THEIR SLICES DO ALLOCATE MEMORY

Arrays are among the most common objects that require memory allocation. This allocation occurs not only when an array is explicitly created and assigned to a variable, but also whenever a computation produces a new array as its result. The example below demonstrates this behavior.

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

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

Slicing is another operation that triggers memory allocation. 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 is when a single element is accessed, in which case no allocations take place.

```
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)
```

Array comprehensions and broadcasting are two additional operations that result in the creation of new arrays. Notably, broadcasting even allocates for intermediate results that are generated internally, even when those results aren't explicitly returned. This behavior is demonstrated in the tab "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)
```