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/CPU registers. 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. Such usage isn't merely to economize on words. Rather, it reflects a fundamental performance implication: heap allocations are the ones that significantly impact efficiency. They involve a more complex management process, thus potentially introducing significant overhead.

The close relationship between performance and heap allocations is even reflected in Julia's benchmarking tools. Macros like `@time` and `@btime` report not only the total runtime of an operation, but also the heap allocations involved.

Considering the importance of heap allocations, 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:

- 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 objects that require memory allocation. The more straightforward cases where this allocation occurs is when an array is explicitly created and assigned to a variable, or when a computation returns a new array. The example below demonstrates this case.

```
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 triggers memory allocation. The reason is 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]                    # 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 additional operations that result in the creation of new arrays. Notably, broadcasting even allocates for intermediate results computed on the fly that 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)
```

```
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)
```