9h. Pre-Allocations

[Martin Alfaro](#)

PhD in Economics

INTRODUCTION

For-loops may entail the creation of new vectors during each iteration, resulting in repeated memory allocation. This dynamic allocation may be unnecessary, particularly if these vectors hold temporary intermediate results that don't need to be preserved for future use. In such situations, performance can be improved through the use of a technique known as pre-allocation.

Pre-allocation involves initializing a vector before the for-loop begins execution, which is then reused during each iteration to store temporary results. By allocating memory upfront and modifying it in place, the overhead associated with repeated vector creation is effectively bypassed.

The performance gains from pre-allocation can be substantial. Remarkably, this technique isn't exclusive to Julia, but rather represents an optimization strategy applicable across programming languages. Its effectiveness ultimately stems from prioritizing the mutation of pre-allocated memory over the creation of new objects, thereby minimizing assignments on the heap.

Our presentation begins with a review of methods for initializing vectors, which is a prerequisite for implementing a pre-allocation strategy. We then present two scenarios where pre-allocation proves advantageous, with special emphasis on its advantages within nested for-loops.

> **Remark**
>
> The review of methods for vector initialization will be relatively brief and centered on performance considerations. For a more detailed review, see the [section about vector initialization](#), as well as the sections on [in-place assignments](#) and [in-place functions](#).

INITIALIZING VECTORS

Vector initialization refers to the process of creating a vector for subsequently filling it with values. The process typically involves two steps: reserving space in memory, and populating that space with some initial values. An efficient approach to initializing a vector then involves performing only the first step, keeping whatever content is held in the memory address. Although Julia will display this content as numerical values, note that they're essentially arbitrary and meaningless, explaining why these values are referred to as `undef` (undefined).

There are two methods for initializing a vector with `undef` values. The first one is through a constructor, requiring the specification of length and element types. The second one is based on the function `similar(y)`, which creates a vector with the same type and dimension as another existing vector `y`. This approach is particularly useful when your output matches the structure of an input.

Below, we compare the performance of approaches to initializing a vector. In particular, we establish that working with `undef` values is faster than populating vectors with specific values. To starkly show the differences in execution time, we repeat the process of vector creation 100,000 times.

```julia
x           = collect(1:100)
repetitions = 100_000                       # repetitions in a for-loop

function foo(x, repetitions)
    for _ in 1:repetitions
        Vector{Int64}(undef, length(x))
    end
end

julia> @btime foo($x, $repetitions)
  2.498 ms (100000 allocations: 85.449 MiB)
```

```julia
x           = collect(1:100)
repetitions = 100_000                       # repetitions in a for-loop

function foo(x, repetitions)
    for _ in 1:repetitions
        similar(x)
    end
end

julia> @btime foo($x, $repetitions)
  3.045 ms (100000 allocations: 85.449 MiB)
```

```julia
x           = collect(1:100)
repetitions = 100_000                       # repetitions in a for-loop

function foo(x, repetitions)
    for _ in 1:repetitions
        zeros(Int64, length(x))
    end
end

julia> @btime foo($x, $repetitions)
  9.957 ms (100000 allocations: 85.449 MiB)
```

```
x          = collect(1:100)
repetitions = 100_000                       # repetitions in a for-loop

function foo(x, repetitions)
    for _ in 1:repetitions
        ones(Int64, length(x))
    end
end
```

```
julia> @btime foo($x, $repetitions)
  10.138 ms (100000 allocations: 85.449 MiB)
```

```
x          = collect(1:100)
repetitions = 100_000                       # repetitions in a for-loop

function foo(x, repetitions)
    for _ in 1:repetitions
        fill(2, length(x))                  # vector filled with integer 2
    end
end
```

```
julia> @btime foo($x, $repetitions)
  10.767 ms (100000 allocations: 85.449 MiB)
```

> **Remark**
>
> Recall that `_` is a convention adopted for **dummy variables**. These are variables with a value assigned, but without being used or referenced anywhere in the code. In the context of for-loops, the sole purpose of `_` is to satisfy the syntax requirements, which expects a variable to iterate over. The symbol `_` is purely a convention and any other one could be used instead.

APPROACHES TO INITIALIZING VECTORS

We can initialize `output` by passing it to the function as a keyword argument. This even enables the use of `similar(x)` with `x` a previous function's argument. Considering this approach, the following two implementations are equivalent.

```
function foo(x)
    output = similar(x)
    # <some calculations using 'output'>
end
```

```
function foo(x; output = similar(x))

    # <some calculations using 'output'>
end
```

When multiple variables of the same type need to be initialized, array comprehensions offer a concise solution. In comparison to this, a more efficient alternative is based on the so-called generators, which will be covered in a [subsequent section](). Although we haven't introduced generators yet, we present this approach because it avoids memory allocation and are therefore more performant. Moreover, the syntax for generators is identical to that of array comprehensions, with the only difference that square brackets `[]` are replaced by parentheses `()`.

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

function foo(x)
    a,b,c = [similar(x) for _ in 1:3]
    # <some calculations using a,b,c>
end
```

```julia
julia> @btime foo($x)
  80.810 ns (8 allocations: 320 bytes)
```

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

function foo(x)
    a,b,c = (similar(x) for _ in 1:3)
    # <some calculations using a,b,c>
end
```

```julia
julia> @btime foo($x)
  32.423 ns (3 allocations: 144 bytes)
```

Although the example uses `similar(x)`, note that the same principle applies to other initialization methods such as `Vector{Float64}(undef, length(x))`.

DESCRIBING THE TECHNIQUE

Julia's default implementation of broadcasting materializes outputs, leading to memory allocation when working with vectors. These allocations arise internally even when broadcasting is an intermediate operation and the final output yields a scalar value. Recall that [broadcasting essentially syntactic sugar for for-loops](). Therefore, these allocations also emerge in for-loops when the equivalent broadcasting operation is implemented.

To describe the technique, we'll consider a simple intuitive example. While the implementation in this example could be more efficiently approached with a reduction, the example is simple enough to show the mechanics as clear as possible.

Suppose we're assessing a worker during 30 days, for which we have information on daily performance. This performance is measured through scores normalized between 0 and 1. Assuming scores above 0.5 represent the target performance, the following code snippets store a vector indicating the days in which the goal was achieved.

```julia
nr_days            = 30
score              = rand(nr_days)


performance(score) = score .> 0.5
```

```julia
julia> @btime foo($x)
  42.906 ns (3 allocations: 96 bytes)
```

```julia
nr_days            = 30
score              = rand(nr_days)

function performance(score)
    target = similar(score)

    for i in eachindex(score)
        target[i] = score[i] > 0.5
    end

    return target
end
```

```julia
julia> @btime foo($x)
  32.670 ns (2 allocations: 304 bytes)
```

```julia
nr_days            = 30
score              = rand(nr_days)

function performance(score; target=similar(score))


    for i in eachindex(score)
        target[i] = score[i] > 0.5
    end

    return target
end
```

```julia
julia> @btime foo($x)
  31.688 ns (2 allocations: 304 bytes)
```

In many cases, the output just computed could actually serve as an intermediate step in another for loop. Situations where the output of one for-loop is the input of another are referred to as **nested for loops**.

The following example illustrates such a scenario. Suppose we have multiple workers, each with performance scores. Depending on the final computation we want, it may be unnecessary to explicitly store these scores. For instance, if our goal is simply to count the number of days on which performance met the target, we could apply a reduction using `sum` and avoid allocating additional storage altogether.

The situation changes, however, when we need to compute multiple summary statistics, or when the statistic of interest requires several intermediate calculations. In these cases, it's often more efficient to first store the vector of days on which the target was met. This avoids recomputing the same intermediate values multiple times, even if the stored values themselves are not directly meaningful.

To demonstrate such a case, we will compute the standard deviation of performance, expressed in units of the mean. This example highlights a nested for-loop scenario where the summary statistic requires multiple intermediate computations.

```julia
nr_days           = 30
scores            = [rand(nr_days), rand(nr_days), rand(nr_days)]  # 3 workers

performance(score) = score .> 0.5

function repeated_call(scores)
    stats = Vector{Float64}(undef, length(scores))

    for col in eachindex(scores)
      target     = performance(scores[col])
      stats[col] = std(target) / mean(target)
    end

    return stats
end
```

```julia
julia> @btime calling_foo_in_a_loop($x)
  409.717 ns (11 allocations: 368 bytes)
```

```julia
nr_days             = 30
scores              = [rand(nr_days), rand(nr_days), rand(nr_days)]    # 3 workers

function performance(score)
    target = similar(score)

    for i in eachindex(score)
        target[i] = score[i] > 0.5
    end

    return target
end

function repeated_call(scores)
    stats = Vector{Float64}(undef, length(scores))

    for col in eachindex(scores)
      target    = performance(scores[col])
      stats[col] = std(target) / mean(target)
    end

    return stats
end
```

```julia
julia> @btime calling_foo_in_a_loop($x)
  299.487 ns (8 allocations: 992 bytes)
```

```julia
nr_days             = 30
scores              = [rand(nr_days), rand(nr_days), rand(nr_days)]    # 3 workers

function performance(score; target = similar(score))

    for i in eachindex(score)
        target[i] = score[i] > 0.5
    end

    return target
end

function repeated_call(scores)
    stats = Vector{Float64}(undef, length(scores))

    for col in eachindex(scores)
      target    = performance(scores[col])
      stats[col] = std(target) / mean(target)
    end

    return stats
end
```

```julia
julia> @btime calling_foo_in_a_loop($x)
  299.885 ns (8 allocations: 992 bytes)
```

Cases like are strong candidates for pre-allocating intermediate results. By adopting this strategy, we can reuse the same vector across iterations, and hence avoid the repeated overhead of creating new vectors.

To implement this strategy, we require an in-place function that takes the for-loop's output as one of its arguments. In the example, this output will be the vector `target`. The in-place function will eventually be called iteratively, updating its output at each iteration. The updates can be implemented either through a for-loop or the operator `.=`.

```julia
nr_days = 30
scores  = [rand(nr_days), rand(nr_days), rand(nr_days)]


performance(score) = score .> 0.5

function repeated_call!(scores)
    stats = Vector{Float64}(undef, length(scores))

    for col in eachindex(scores)
        target      = performance(scores[col])
        stats[col] = std(target) / mean(target)
    end

    return stats
end
```

```julia
julia> @btime foo!($output, $x)
  419.763 ns (11 allocations: 368 bytes)
```

```julia
nr_days = 30
scores  = [rand(nr_days), rand(nr_days), rand(nr_days)]
target  = similar(scores[1])

performance!(target, score) = (@. target = score > 0.5)

function repeated_call!(target, scores)
    stats = Vector{Float64}(undef, length(scores))

    for col in eachindex(scores)
        performance!(target, scores[col])
        stats[col] = std(target) / mean(target)
    end

    return stats
end
```

```julia
julia> @btime calling_foo_in_a_loop($output, $x)
  265.741 ns (2 allocations: 80 bytes)
```

```julia
nr_days = 30
scores  = [rand(nr_days), rand(nr_days), rand(nr_days)]
target  = similar(scores[1])

function performance!(target, score)
    for i in eachindex(score)
        target[i] = score[i] > 0.5
    end
end

function repeated_call!(target, scores)
    stats = Vector{Float64}(undef, length(scores))

    for col in eachindex(scores)
        performance!(target, scores[col])
        stats[col] = std(target) / mean(target)
    end

    return stats
end
```

```julia
julia> @btime calling_foo_in_a_loop($output, $x)
  271.495 ns (2 allocations: 80 bytes)
```

> **Warning!** - Use of @. to update values
>
> When your goal is to update the values of a vector, recall that `@.` has to be *placed at the beginning* of the statement.
>
> ```julia
> # the following are equivalent and define a new variable
>     output  = @. 2  * x
>     output  =    2 .* x
> ```
>
> ```julia
> # the following are equivalent and update 'output'
> @. output  = 2  * x
>    output .= 2 .* x
> ```

Compared to a for-loop, the method using `.=` provides a simpler syntax. This is why, when the function `performance!` is simple enough as in our example, it's common to directly express the updates via broadcasting inside the inner for-loop. This possibility also enables implementing the update via a built-in in-place function. For instance, the function `map!` can be used with this purpose.

```julia
nr_days = 30
scores  = [rand(nr_days), rand(nr_days), rand(nr_days)]


function repeated_call!(scores)
    stats = Vector{Float64}(undef, length(scores))

    for col in eachindex(scores)
        target    = @. score > 0.5
        stats[col] = std(target) / mean(target)
    end

    return stats
end
```

```julia
julia> @btime foo!($output, $x)
  406.159 ns (11 allocations: 368 bytes)
```

```julia
nr_days = 30
scores  = [rand(nr_days), rand(nr_days), rand(nr_days)]
target  = similar(scores[1])

function repeated_call!(target, scores)
    stats = Vector{Float64}(undef, length(scores))

    for col in eachindex(scores)
        @. target  = scores[col] > 0.5
        stats[col] = std(target) / mean(target)
    end

    return stats
end
```

```julia
julia> @btime foo!($output, $x)
  270.358 ns (2 allocations: 80 bytes)
```

```julia
nr_days = 30
scores  = [rand(nr_days), rand(nr_days), rand(nr_days)]
target  = similar(scores[1])

function repeated_call!(target, scores)
    stats = Vector{Float64}(undef, length(scores))

    for col in eachindex(scores)
        map!(a -> a > 0.5, target, scores[col])
        stats[col] = std(target) / mean(target)
    end

    return stats
end
```

```julia
julia> @btime calling_foo_in_a_loop($output,$x)
  258.784 ns (2 allocations: 80 bytes)
```