9h. Pre-Allocations

Martin Alfaro
PhD in Economics

INTRODUCTION

For-loops may involve 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 to store temporary results before the for-loop begins execution, which is then reused during each iteration. 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 mutations 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.

INITIALIZING VECTORS

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.

Vector initialization refers to the process of creating a vector that subsequently will be filled 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 involves performing only the first step, keeping whatever content is in the memory address. These values held in memory are referred to as `undef` (undefined). Although Julia will display them as numerical values, they're essentially arbitrary and meaningless.

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 the output sought 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)
  2.005 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)
  2.062 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)
  9.002 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
        ones(Int64, length(x))
    end
end
```

```
julia> @btime foo($x, $repetitions)
  9.764 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
        fill(2, length(x))                   # vector filled with integer 2
    end
end
```

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

Remark

Recall that `_` isn't a keyword, but rather a convention widely adopted by programmers to denote **dummy variables**. These are variables included solely to meet syntactical requirements, but are never actually used or referenced within the code. In our example, the inclusion of `_` is because for-loops must always include a variable to iterate over. While any other symbol could be used, `_` signals programmers that its value can be safely ignored.

INITIALIZING VECTORS IN FUNCTIONS

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

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

```julia
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. Nonetheless, a more efficient alternative is based on the so-called generators, which are the lazy counterpart of array comprehensions. This reduces memory allocations by initializing and assigning the vectors element-wise, while an array comprehension first creates a vector holding all the initialized vectors.

```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> @btime foo($x)
  54.235 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> @btime foo($x)
  21.595 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

To describe how pre-allocation works, we'll consider a typical scenario where it proves to be advantageous. This setup involves a **nested for-loop**, in which the output of a for-loop serves as an intermediate input for another for-loop. The example will rely on explicit for-loops, but also on constructs that internally compute the operation via a for-loop (e.g., broadcasting).

Let's first describe the inner for-loop that will be eventually embedded in an outer for-loop. Suppose we're assessing a worker's performance over a 30-day period, with daily scores recorded on a scale from 0 to 1. Defining successful performance as any score above 0.5, the following code snippets generate vectors indicating the days on which the goal was achieved.

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

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

```
julia> @btime performance($score)
  29.186 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> @btime performance($score)
  24.548 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> @btime performance($score)
  26.479 ns (2 allocations: 304 bytes)
```

Now consider that the output of this operation (namely, the vector of days in which the target was met) represents an intermediate step in another for-loop. For instance, suppose we have multiple workers, each with recorded performance scores. The goal is to summarize each worker's information through a summary statistic. In particular, we consider the ratio of the standard deviation to the mean, which expresses variability in units of the average.

In practice, this statistic requires computing both the mean and the standard deviation. Below, we show that computing each statistic via reductions isn't efficient. The reason is that, while reductions avoid memory allocations, they require computing the days on which the target was met twice. Instead, it's more efficient to compute and store this vector, which we then reuse as an input for computing the mean and the standard deviation.

The result highlights a more general principle: when the same intermediate input is required for multiple outputs, the cost of allocating memory for the intermediate vector tends to be lower than its repeated computation. In the example, the reduction is implemented via lazy broadcasting, which internally relies on reduction techniques. The rest implement the result via explicit or implicit for-loops, with the intermediate input stored in a vector.

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



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

    for col in eachindex(scores)

      stats[col] = std(@~ scores[col] .> 0.5) / mean(@~ scores[col] .> 0.5)
    end

    return stats
end
```

```
julia> @btime repeated_call($scores)
  364.949 ns (2 allocations: 80 bytes)
```

```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> @btime repeated_call($scores)
  274.988 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> @btime repeated_call($scores)
  197.941 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> @btime repeated_call($scores)
  188.021 ns (8 allocations: 992 bytes)
```

Once we establish that storing the intermediate vector is more performant, our approach will necessarily involve memory allocation. However, the methods described above allocate more than required: during each iteration, a new vector is created for each worker to store the days in which the target was met. This intermediate vector doesn't need to be preserved for future use. Indeed, because it is stored in a local variable, it is discarded once the iteration concludes.

Such cases are strong candidates for pre-allocating intermediate results. By defining a single vector `target`, we can reuse it across iterations to compute the days when the target was met for each worker. This strategy incurs the overhead of creating the vector only once, after which it is reused for every worker.

The implementation requires defining the vector `target` before the execution of the outer for-loop. During each iteration, we then mutate target using an in-place function, which we call `performance!`. This function can update the contents either with a standard for-loop or by applying the broadcasting 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> @btime repeated_call($scores)
  259.300 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> @btime repeated_call!($target, $scores)
  193.557 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> @btime repeated_call!($target, $scores)
  171.647 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.

```
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> @btime repeated_call($scores)
  431.740 ns (23 allocations: 608 bytes)
```

```
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> @btime repeated_call!($target, $scores)
  190.480 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> @btime repeated_call!($target, $scores)
  167.516 ns (2 allocations: 80 bytes)
```