9e. Pre-Allocations

Martin Alfaro
PhD in Economics

INTRODUCTION

In many scenarios, for-loops entail the creation of new vectors at every 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. They serve as 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 creation, 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 involves performing only the first step, keeping whatever content is held in the memory address at the moment of creation. Although Julia will display these contents as numerical values, note that they're essentially arbitrary and meaningless, explaining why they're referred to as `undef` (undefined).

There are two methods for initializing a vector with `undef` values. The first one requires specifying the type and length of the array. Its syntax closely resembles the creation of new vectors. 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 show that working with `undef` values is faster than populating the vector with specific values. To starkly show these differences, we create a vector with 100 elements and repeat the procedure 100,000 times.

```
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)
  1.581 ms (100000 allocations: 85.45 MiB)
```

```
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)
  1.623 ms (100000 allocations: 85.45 MiB)
```

```
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)
  7.530 ms (100000 allocations: 85.45 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)
  4.674 ms (100000 allocations: 85.45 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)
  4.877 ms (100000 allocations: 85.45 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 expect a variable to iterate over. The symbol `_` is purely a convention and any other symbol could be used instead.

APPROACHES TO INITIALIZING VECTORS

We can initialize `output` by passing it to the function as a keyword argument. This enables the use of `similar(x)`, where `x` is a previous function's argument. Given this, the following two implementations turn out to be 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. A more efficient alternative is based on the so-called generators, which will be covered in a subsequent section. Although we haven't introduced this approach yet, we present it because generators are generally preferred. At this point, all you should know is that generator-based methods avoid memory allocation and are therefore more performant. Their syntax is almost identical to that of array comprehensions, with the only difference being that square brackets `[]` are replaced by parentheses `()`.

```
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)
  49.848 ns (4 allocations: 320 bytes)
```

```
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)
  35.348 ns (3 allocations: 240 bytes)
```

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

PRE-ALLOCATING VECTORS IN NESTED FOR-LOOPS

When working with vectors in for-loops or through broadcasting, certain operations inevitably create new vectors. This may occur even when the operation ultimately yields a scalar value. The following examples demonstrate this point.

```
x = rand(100)

foo(x) = sum(2 .* x)                    # 2 .* x implicitly creates a temporary vector
```

```
julia> @btime foo($x)
  36.779 ns (1 allocation: 896 bytes)
```

```julia
x = rand(100)

function foo(x)
    output = similar(x)                # you need to create this vector to store the results

    for i in eachindex(x)
        output[i] = 2 * x[i]
    end

    return output
end
```

```
julia> @btime foo($x)
  45.416 ns (1 allocation: 896 bytes)
```

Furthermore, when the outcome of a computation must be preserved, allocating new vectors becomes an inevitable part of the process. In many cases, however, final outputs could actually serve themselves as intermediate steps within a larger computation, which may itself involve 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 how each iteration in the second for-loop generates a new vector for the intermediate result.

```julia
x = rand(100)

foo(x) = 2 .* x

calling_foo_in_a_loop(x) = [sum(foo(x)) for _ in 1:100]
```

```
julia> @btime calling_foo_in_a_loop($x)
  6.433 μs (101 allocations: 88.38 KiB)
```

```julia
x = rand(100)

function foo(x; output = similar(x))
    for i in eachindex(x)
        output[i] = 2 * x[i]
    end

    return output
end

calling_foo_in_a_loop(output,x) = [sum(foo(x)) for _ in 1:100]
```

```
julia> @btime calling_foo_in_a_loop($x)
  6.160 μs (101 allocations: 88.38 KiB)
```

Cases like this lead to unnecessary memory allocations, making them 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. This function will eventually be called iteratively, updating its output at each iteration. There are two ways to implement this method, and we analyze each separately in the following.

VIA A FOR-LOOP

The first approach defines an in-place function that updates the values of `output` through a for-loop.

```
x      = rand(100)
output = similar(x)

function foo!(output,x)
    for i in eachindex(x)
        output[i] = 2 * x[i]
    end

    return output
end
```

```
julia> @btime foo!($output, $x)
  5.100 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)
output = similar(x)

function foo!(output,x)
    for i in eachindex(x)
        output[i] = 2 * x[i]
    end

    return output
end

calling_foo_in_a_loop(output,x) = [sum(foo!(output,x)) for _ in 1:100]
```

```
julia> @btime calling_foo_in_a_loop($output, $x)
  1.340 μs (1 allocation: 896 bytes)
```

VIA BROADCASTING

The second option relies on the operator `.=` to update a vector's values. Compared to the previous example, this strategy enables updates with a simpler syntax, allowing `foo!` to be defined in a single line.

```
x      = rand(100)
output = similar(x)

foo!(output,x) = (output .= 2 .* x)
```

```
julia> @btime foo!($output, $x)
  5.800 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)
output = similar(x)

foo!(output,x) = (@. output = 2 * x)
```

```
julia> @btime foo!($output, $x)
  5.400 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)
output = similar(x)

foo!(output,x) = (@. output = 2 * x)

calling_foo_in_a_loop(output,x) = [sum(foo!(output,x)) for _ in 1:100]
```

```
julia> @btime calling_foo_in_a_loop($output,$x)
  1.320 μs (1 allocation: 896 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.

```
# the following are equivalent and define a new variable
   output  = @. 2  * x
   output  =    2 .* x
```

```
# the following are equivalent and update 'output'
@. output  = 2  * x
   output .= 2 .* x
```

PRE-ALLOCATIONS FOR INTERMEDIATE STEPS

So far, our discussion has centered around the benefits of pre-allocating vectors in nested for-loops. However, such benefits extend beyond that specific setting.

Broadly speaking, pre-allocating proves useful when: ***i***) the vector serves an intermediate result that feeds into another operation, and ***ii***) the intermediate result is computed inside a for-loop. When both conditions are met, reusing a pre-allocated vector consistently outperforms creating a new vector at each iteration.

To illustrate, consider a function `foo` that returns `output`, whose computation requires an intermediate variable `temp`. The key feature here is that, even though `foo` itself won't be called inside a nested for-loop, the computation of `output` is complex enough that it's convenient to split it into multiple steps.

If `temp` isn't pre-allocated, a new vector is created at each iteration as shown below.

```
x = rand(100)

function foo(x; output = similar(x))
    for i in eachindex(x)
        temp      = x .> x[i]
        output[i] = sum(temp)
    end

    return output
end
```

```
julia> @btime foo($x)
  14.700 μs (201 allocations: 11.81 KiB)
```

```
x = rand(100)

foo(x) = [sum(x .> x[i]) for i in eachindex(x)]
```

```
julia> @btime foo($x)
  14.600 μs (201 allocations: 11.81 KiB)
```

```
x = rand(100)

function foo(x)
    temp   = [x .> x[i] for i in eachindex(x)]
    output = sum.(temp)
end
```

```
julia> @btime foo($x)
  15.100 μs (202 allocations: 12.69 KiB)
```

Next, we demonstrate different strategies that pre-allocate `temp`. The implementation of the pre-allocation differs in subtle but important ways from the nested-loop case.

First, because `foo` won't be eventually called in a for-loop, the pre-allocation of `temp` can be performed within the function itself. This means we don't need to define `temp` as an argument of `foo`. Second, since all iterations occur within a single for-loop, broadcasting becomes a particularly convenient option for updating `temp`.

VIA A FOR-LOOP OR BROADCASTING

The simplest strategy to avoid the allocations of `temp` is to pre-allocate it and update its values via broadcasting. By contrast, updating `temp` through a for-loop represents a more involved approach. The difference can be observed below.

```julia
x = rand(100)

function foo!(x; output = similar(x), temp = similar(x))
    for i in eachindex(x)
        for j in eachindex(x)
            temp[j] = x[j] > x[i]
        end
        output[i] = sum(temp)
    end

    return output
end
```

```
julia> @btime foo!($x)
  1.850 μs (2 allocations: 1.75 KiB)
```

```julia
x = rand(100)

function foo!(x; output = similar(x), temp = similar(x))
    for i in eachindex(x)
        temp     .= x .> x[i]
        output[i] = sum(temp)
    end

    return output
end
```

```
julia> @btime foo!($x)
  1.660 μs (2 allocations: 1.75 KiB)
```

```julia
x = rand(100)

function foo!(x; output = similar(x), temp = similar(x))
    for i in eachindex(x)
        @. temp      = x > x[i]
           output[i] = sum(temp)
    end

    return output
end
```

```
julia> @btime foo!($x)
  1.660 μs (2 allocations: 1.75 KiB)
```

Note that the two allocations observed are due to the creation of `temp` and `output`, which are unavoidable.

VIA IN-PLACE FUNCTION

Another way to avoid the memory allocations of `temp` is to pre-allocate it via an in-place function. We'll refer to this function as `update_temp!`, which is defined outside the for-loop and executed at each iteration. The advantage of this approach is modularity: by separating the update logic from the

for-loop, we can focus on optimizing `update_temp!` independently if its performance becomes critical. This design also improves code readability and maintainability.

```
x = rand(100)

function update_temp!(x, temp, i)
    for j in eachindex(x)
       temp[j] = x[j] > x[i]
    end
end

function foo!(x; output = similar(x), temp = similar(x))
    for i in eachindex(x)
        update_temp!(x, temp, i)
        output[i] = sum(temp)
    end

    return output
end
```

```
julia> @btime foo!($x)
  1.790 μs (2 allocations: 1.75 KiB)
```

```
x = rand(100)

update_temp!(x, temp, i) = (@. temp = x > x[i])

function foo!(x; output = similar(x), temp = similar(x))
    for i in eachindex(x)
           update_temp!(x, temp, i)
           output[i] = sum(temp)
    end

    return output
end
```

```
julia> @btime foo!($x)
  1.680 μs (2 allocations: 1.75 KiB)
```

ADDING A NESTED FOR-LOOP

Finally, let's study a scenario that combines the two situations discussed earlier: a function `foo` that requires an intermediate variable `temp`, and the repeated invocation of `foo` inside an outer for-loop. In this case, we're layering the complexity of intermediate pre-allocation with the demands of a nested loop structure.

In this combined setting, both `output` and `temp` must be initialized outside the function and passed in as arguments. This ensures that no vector is re-created at each iteration of the inner or outer for-loop. The following example illustrates an implementation that avoids unnecessary allocations by updating `update_temp!` via broadcasting.

```julia
x      = rand(100)



update_temp!(x, temp, i) = (@. temp = x > x[i])

function foo!(x; output = similar(x), temp = similar(x))
    for i in eachindex(x)
           update_temp!(x, temp, i)
           output[i] = sum(temp)
    end
    return output
end

calling_foo_in_a_loop(x) = [foo!(x) for _ in 1:1_000]
```

```
julia> @btime calling_foo_in_a_loop($x)
  1.734 ms (2001 allocations: 1.72 MiB)
```

```julia
x      = rand(100)
output = similar(x)
temp   = similar(x)

update_temp!(x, temp, i) = (@. temp = x > x[i])

function foo!(x, output, temp)
    for i in eachindex(x)
           update_temp!(x, temp, i)
           output[i] = sum(temp)
    end
    return output
end

calling_foo_in_a_loop(x, output, temp) = [foo!(x, output, temp) for _ in 1:1_000]
```

```
julia> @btime calling_foo_in_a_loop($x, $output, $temp)
  1.666 ms (1 allocation: 7.94 KiB)
```