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 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 repeatedly creating new vectors is effectively bypassed.

The performance gains from pre-allocation can be substantial. Remarkably, this technique isn't exclusive to Julia, but rather represents a universal 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 the reliance on the heap.

Our presentation begins with a review of methods for initializing vectors, which serve as a prerequisite for implementing pre-allocations. 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 with the intention of 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 these values will display specific numbers if called in Julia, they're essentially arbitrary and meaningless, explaining why these values 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, for instance, your output matches the structure of another vector.

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.

```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)
  1.581 ms (100000 allocations: 85.45 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)
  1.623 ms (100000 allocations: 85.45 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)
  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 denoting **dummy variables**. They're variables that have a value, but aren't used or referenced anywhere in the code. In the context of a for-loop, the sole purpose of `_` is to satisfy the syntax requirements, which expects a variable to iterate over.
>
> The symbol `_` is arbitrary and any other could be used in its place. Throughout the website, we've consistently used `_` when our intention is to repeatedly compute the same operation.

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. Considering 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 comprehension offers a concise way to do so. In addition to array comprehension, there exists a closely related yet more efficient approach based on generators. This approach is covered in a subsequent section. At this point, you should only know that the method based on generators doesn't allocate. Furthermore, its syntax is similar to array comprehension, with the only difference that brackets `[]` are replaced with 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)
```

The demonstration uses `similar(x)` as an example, but 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 within for-loops and broadcasting, certain operations inherently require the creation of new vectors. In fact, these vectors can be generated even when the operation ultimately yields a scalar value. The following examples demonstrate this point.

```
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)
```

```
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)
```

When the outcome of a computation must be retained, allocating new vectors is an inevitable part of the process. This is most evident when the operation produces the final output. However, even the final output of an operation could serve as an intermediate step in a larger computation, which may involve another for-loop. These situations, in which one loop's work becomes part of another's, fall into the category of **nested for-loops**.

The following example illustrates how each iteration in the second for-loop generates a new vector for the intermediate result.

```
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)
```

```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)
```

Scenarios like this lead to unnecessary memory allocations, making them well-suited for pre-allocation of the intermediate result. By adopting this strategy, we can reuse the same vector across iterations, effectively bypassing the memory allocations stemming from creating a new vector multiple times.

To implement it, we need an in-place function that takes the output of the for-loop as one of the arguments. This function will eventually be called iteratively, updating its output in each iteration. There are two ways to implement this strategy, 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.

```julia
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)
```

```julia
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. Relative to the example above, this allows for an update through a simpler syntax, where `foo!` is defined in one 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 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, its applicability extends beyond this specific scenario.

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. If these two conditions are met, reusing the same pre-allocated vector outperforms a strategy based on a new vector for each iteration.

Next, we analyze a case where these conditions hold, even though `foo` isn't called in a for-loop as it'd be the case in a nested for-loop. The usefulness of a pre-allocation emerges because `output` demands a complex computation, making it convenient to split the calculation in several steps.

To illustrate the procedure, consider an operation where `temp` represents an intermediate variable to compute `output`. All the implementations below don't pre-allocate `temp`, and hence create a new vector in each iteration.

```julia
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)
```

```julia
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)
```

```julia
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)
```

In the following, we pre-allocate `temp`, although the scenario considered exhibits some differences relative to a nested for-loop. These differences result in some subtle aspects regarding their implementation.

First, as we're assuming that this function won't be called in a for-loop, the pre-allocation can be performed within the function, rather than defining `temp` as an argument of `foo`. Second, all the iterations occur within the same for-loop, making the broadcasting option more convenient. The consequences of these differences for the implementation are discussed below.

VIA A FOR-LOOP OR BROADCASTING

Pre-allocating `temp` and update its values via broadcasting is the simplest way for the scenario considered. In fact, this method doesn't require departing from the original syntax. On the contrary, the use a for-loop is more involved.

```
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)
```

```
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)
```

```
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 incurred only once rather than in each iteration.

VIA IN-PLACE FUNCTION

Given the features of the scenario considered, we can also implement the pre-allocation via an in-place function. We refer to it as `update_temp!`, which is defined outside the for-loop and updated in each iteration. An advantage of this approach is that we separate `update_temp!` from the for-loop, and hence we can focus on `update_temp!` if the operation is performance critical.

```
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)
```

```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
```

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

ADDING A NESTED FOR-LOOP

We know combine both cases, where `foo` requires an intermediate variable `temp` and then is called in another for-loop. In such a scenario, both `output` and `temp` needs to be initialized outside the functions and used as function arguments. The following example illustrates this by considering only `update_temp!` using 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)
```