Notes 2: Memory and scope

Chris Paciorek

2025-01-23

Table of contents

Introduction

This document is the second of a set of notes, this document focusing on memory, storage of objects, and variable scope. The notes are not meant to be particularly complete in terms of useful functions (Google and LLMs can now provide that quite well), but rather to introduce the language and consider key programming concepts in the context of Julia.

Given that, the document heavily relies on demos, with interpretation in some cases left to the reader.

Memory use

Mutable objects

Let's see if Julia behaves as we would expect if we try to change objects in different ways.

```
x = "hello"
x[1] = "a"

x = [3.1, 2.1]
x[2] = 5.5

const tmpVar = [3.1, 2.1]
tmpVar[2] = 5.5
tmpVar = "foo"

x = (3, 5, "hello")
x[2] = 7

const tmpVarTuple = (3, 5, "hello")
tmpVarTuple = (5, 9)
```

Be careful as `const` objects cannot be deleted or reassigned.

One nice aspect of this is that you can define a variable without fear that it will be used in some other way.

Modifying objects in place

Use of `<function_name>!()` indicates the function operates on the inputs in place and modifies arguments (non-black box execution).

```
t = ['a', 'b', 'c'];
push!(t, 'd')
```

```
4-element Vector{Char}:
 'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
 'b': ASCII/Unicode U+0062 (category Ll: Letter, lowercase)
 'c': ASCII/Unicode U+0063 (category Ll: Letter, lowercase)
 'd': ASCII/Unicode U+0064 (category Ll: Letter, lowercase)
```

```
pop!(t)
```

```
'd': ASCII/Unicode U+0064 (category Ll: Letter, lowercase)
```

```
push(t, 'e')
```

```
UndefVarError: `push` not defined
```

```
x = Dict("test" => 3, "tmp" => [2.1, 3.5], 7 => "weird")
```

```
Dict{Any, Any} with 3 entries:
  7      => "weird"
  "test" => 3
  "tmp"  => [2.1, 3.5]
```

```
pop!(x, "tmp");
push!(x, 'b' => 3);
x
```

```
Dict{Any, Any} with 3 entries:
  7      => "weird"
  "test" => 3
  'b'    => 3
```

> **Exercise**
>
> See if you can create a function that does not have "!" at the end of the name that modifies an input argument.

Memory use and copying

We can use `===` to see if objects are identical. For mutable objects this involves looking at whether the data is stored at the same place in memory.

```
a = "banana"
```

```
"banana"
```

```
b = "banana"
```

```
"banana"
```

```
a === b
```

```
true
```

```
a   b
```

```
true
```

```
a = [1, 2, 3];
b = a;
c = [1, 2, 3];

a === b
```

```
true
```

```
a === c
```

```
false
```

```
a == c
```

```
true
```

```
a = [1, 2, [4,7]]
```

```
3-element Vector{Any}:
 1
 2
  [4, 7]
```

```
c = [1, 2, [3]]
```

```
3-element Vector{Any}:
 1
 2
  [3]
```

```
c[3] = a[3]
```

```
2-element Vector{Int64}:
 4
 7
```

```
a === c
```

```
false
```

```
a[3] === c[3]
```

```
true
```

Aliasing

This avoids copying but can be dangerous. The behavior is like Python, but not like R.

```
a = [1, 2, 3]
```

```
3-element Vector{Int64}:
 1
 2
 3
```

```
b = a
```

```
3-element Vector{Int64}:
 1
 2
 3
```

```
a[1] = 99
```

```
99
```

```
b
```

```
3-element Vector{Int64}:
 99
  2
```

```
3
```

Using copies rather than aliases

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

```
3-element Vector{Int64}:
 1
 2
 3
```

```
y = x[:]
```

```
3-element Vector{Int64}:
 1
 2
 3
```

```
pop!(x)
```

```
3
```

```
y
```

```
3-element Vector{Int64}:
 1
 2
 3
```

> Question
>
> It would be useful to know if there's a way to make a copy of an array without having to know its dimension.

Objects in memory

If we want to see the memory address of an object, you can use `pointer_from_objref`.

In which of these cases do you expect that the same object in memory is referenced?

```
a = [1, 2, 3]
b = a
c = [1, 2, 3]
pointer_from_objref(a)
pointer_from_objref(b)
pointer_from_objref(c)

tmp = [4,7]
a = [1, 2, tmp, [4,7]]
```

```
pointer_from_objref(tmp)
pointer_from_objref(a[3])
pointer_from_objref(a[4])
```

What happens with `pointer_from_objref` on immutable objects?

```
x = (3,5)
pointer_from_objref(x)
```

```
ERROR: pointer_from_objref cannot be used on immutable objects
```

```
y = "hello"
```

```
"hello"
```

```
pointer_from_objref(y)
```

```
Ptr{Nothing} @0x00007fb5ec837a78
```

There's a bit more info in the help for `pointer_from_objref`.

Pass by reference

Julia uses pass by reference. If you pass a mutable object into a function and modify it, that affects the state of the object outside of the function; no local copy of the object is made. This is efficient in terms of copying and memory use, but it does not following functional programming principles.

```
function array_modifier(x)
    push!(x, 12)
    return Nothing
end
```

```
array_modifier (generic function with 1 method)
```

```
y = [1, 2, 3]
```

```
3-element Vector{Int64}:
 1
 2
 3
```

```
array_modifier(y)
```

```
Nothing
```

```
y
```

```
4-element Vector{Int64}:
  1
  2
  3
 12
```

We should instead define the function as `array_modifier!` to be consistent with Julia's syntax.

```
function array_modifier!(x)
    push!(x, 12)
    return Nothing
end
```

```
array_modifier! (generic function with 1 method)
```

```
y = [1, 2, 3]
```

```
3-element Vector{Int64}:
 1
 2
 3
```

```
array_modifier!(y)
```

```
Nothing
```

```
y
```

```
4-element Vector{Int64}:
  1
  2
  3
 12
```

Using tuples to prevent side effects

If we use a tuple as input to a function, we don't have to worry about the input being modified; any attempt at modification will throw an error.

Example

Consider this function, modified from a Python function that was modified from an R function that I was looking at with a student who wanted to reduce the memory use of his code.

```
function fastcount(xvar, yvar)
    naline = isnan.(xvar)
    naline[isnan.(yvar)] .= 1
    localx = xvar[:]
    localy = yvar[:]
    localx[naline] .= 0
    localy[naline] .= 0
    useline = .!naline
    # Rest of code...
end
```

```
fastcount (generic function with 1 method)
```

```julia
using Random, Distributions
n = 20;
x = rand(Normal(), n);
y = rand(Normal(), n);
x[[3, 5]] .= NaN;
y[[1, 7]] .= NaN;
fastcount(x, y);
```

> **Exercise**
>
> Determine all the places where additional memory is allocated (including for any temporary arrays).

Arrays and pointers

If we have an array made of numbers all of the same type, the values can be stored contiguously in memory.

```julia
n = Int(1e7);
x = randn(Float32, n);
sizeof(x)
```

```
40000000
```

```julia
typeof(x)
```

```
Vector{Float32} (alias for Array{Float32, 1})
```

What about an array where the elements are not all the same type?

> **Exercise**
>
> What does the following code indicate about how arrays of heterogeneous elements are stored? And how many bytes is a pointer?

```julia
x = [1.3, 2.5, 7.4, "hello"]
```

```
4-element Vector{Any}:
 1.3
 2.5
 7.4
  "hello"
```

```julia
sizeof(x)
```

```
32
```

```julia
typeof(x)
```

```
Vector{Any} (alias for Array{Any, 1})
```

```julia
devs = randn(n);
x[1] = devs;
sizeof(x)
```

```
32
```

```julia
pointer_from_objref(devs)
```

```
Ptr{Nothing} @0x00007fb5e9058010
```

```julia
pointer_from_objref(x[1])
```

```
Ptr{Nothing} @0x00007fb5e9058010
```

```julia
devs[1]
```

```
-0.9381735894936654
```

```julia
x[1][1]
```

```
-0.9381735894936654
```

```julia
devs[1] = 3.0
```

```
3.0
```

```julia
x[1][1]
```

```
3.0
```

> Exercise
>
> Check your understanding by creating an array of matrices where each of the individual matrices are just pointers to the same underlying matrix. Modify the underlying matrix. Modify one of the matrices. See what happens.
> Side note: suppose `x` is a matrix. What's the difference between `[x, x]`, `[x x]`, and `[x; x]`?

Scope

Lexical scoping

Julia uses lexical scoping, which means that lookup of non-local variables occurs in the scope in which a function is defined, not the scope from which it was called. This means that code is easier to reason about (where the behavior of a function doesn't depend on where it is called from) and is modular.

Exercise: Lexical scoping

Experiment with the following cases and make sure you understand how the lookup / scoping is working. Predict the result **before** running the code.

Case 1: Will the code print 3 or 7?

```
# Case 1
x = 3
function f2()
    print(x)
end

function f()
    x = 7
    f2()
end

f() # what will happen?
```

Case 2: Will the code print 3, 7, or 100?

```
x = 3
function f2()
    print(x)
end

function f()
    x = 7
    f2()
end

x = 100
f() # what will happen?
```

Case 3: Will the code print 3, 7, or 100?

```
x = 3
function f()
    function f2()
        print(x)
    end
    x = 7
    f2()
end

x = 100
```

```
f() # what will happen?
```

Case 4: Will the code print 3 or 100 or produce an error?

```
x = 3
function f()
    function f2()
        print(x)
    end
    f2()
end

x = 100
f() # what will happen?
```

Closures

Here's a somewhat tricky scoping example:

```
y = 100
function fun_constructor()
    y = 10
    function g(x)
        return x + y
    end
    return g
end

## fun_constructor() creates functions
myfun = fun_constructor()
myfun(3)
```

> **Exercise: Lexical scoping**
>
> Try to understand what is going on with `fun_constructor`. What do you expect `myfun(3)` to return? Where is `myfun` defined?
> Extra: modify `fun_constructor` in such a way that you can determine if `g` can modify `y` in the enclosing scope.

The above is an example of a *closure*, a useful concept in functional programming that provides functionality similar to object-oriented programming. `y` is "bound" up/captured in the enclosing scope of `g`/`myfun`.

Global and local scopes

Global and local variables

We can access global variables from within functions via Julia's scoping rules, as seen previously.

To modify global variables, we need to use `global`.

```
x = 100
```

```
100
```

```
function test()
  global x
  println(x)
  x = 3;
  return nothing
end
```

```
test (generic function with 1 method)
```

```
test()
```

```
100
```

```
print(x)
```

```
3
```

This is like Python. Also note the difference in behavior compared to being able to modify the captured variable in the closure without any explicit syntax.

Interestingly, this doesn't work to be able to access both a local and global variable of the same name.

```
y = 100

function test()
  println(y)
  y = 3;
  return nothing
end

test()
```

```
ERROR: UndefVarError: `y` not defined
```

Note that use of `global` in Section 8 (Looping and Counting) of Think Julia seems incorrect/unnecessary.

Modules

You can isolate code from your working context using `module`.

```
x = 0;

module testmod
  x = 99;
end

testmod.x
```

```
99
```

```
x
```

```
0
```

Each module has its own global scope. And each code block has its own local scope (as we saw with the for loop in Notes 1).

```
for i in 1:3
  tmp = i*7
end

print(tmp)
```

```
ERROR: UndefVarError: `tmp` not defined
```

```
print(i)
```

```
ERROR: UndefVarError: `tmp` not defined
```

Scoping gets rather more complicated.

The use of `using` adds variables from a package/module to the current scope.

```
A = rand(3, 3);
eigvals(A)
```

```
ERROR: UndefVarError: `eigvals` not defined
```

```
A = rand(3, 3);
using LinearAlgebra
eigvals(A)
```

```
3-element Vector{Float64}:
 -0.12046036270814879
  0.14753045325528416
  0.6235226428412464
```

Let blocks

You can also use `let` to create a new scope:

```julia
x = 0
```

```
0
```

```julia
let x = 5
  print(x)
end
```

```
5
```

```julia
print(x)
```

```
0
```