Wissenschaftliches Programmieren

Meik Hellmund

Meik.Hellmund@math.uni-leipzig.de

https://www.math.uni-leipzig.de/~hellmund/teaching/julia.html

- Montags 11:15 HS 5
- A-Woche Donnerstag 13:15 HS 5



Vectors, matrices, arrays

- `Array{T,N}` is an N-dimensional array (an element is addressed by N indices) with elements of type `T`.
- Two aliases for the most important cases:
  - `Vector{T}` = `Array{T,1}`
  - `Matrix{T}` = `Array{T,2}`

```
julia> Vector{Float64} === Array{Float64,1} && Matrix{Float64} === Array{Float64,2}
true
```

- Vectors are written with square brackets and commas. The element type is determined automatically:

```
julia> v = [3, 4, 88]
3-element Vector{Int64}:
  3
  4
 88
```

```
julia> v = [3, 4, 5.1]
3-element Vector{Float64}:
 3.0
 4.0
 5.1
```

Element type

- For numeric types, a common type is selected and all elements are converted (*promotion*).

```
julia> v = [3//7, 4, 2im]
3-element Vector{Complex{Rational{Int64}}}:
 3//7 + 0//1*im
 4//1 + 0//1*im
 0//1 + 2//1*im
```

- If there is no such common type, we get an array of type `Any`.

```
julia> v = [33, "33", 1.2]
3-element Vector{Any}:
 33
   "33"
  1.2
```

- This works, but is much less efficient: the elements are stored as pointers to boxed objects of arbitrary type. For fast numerics we want arrays with a concrete element type like `Float64`.

Some basic functions

```
julia> v = [3//7, 4, 2im];
julia> length(v)
3
julia> ndims(v)
1
julia> size(v)
(3,)
julia> eltype(v)
Complex{Rational{Int64}}
julia> isempty(v)
false
julia> empty!(v)
Complex{Rational{Int64}}[]
```

```
julia> m = [2 3
            7 8]
2×2 Matrix{Int64}:
 2  3
 7  8
julia> for f in [length, eltype, ndims, size]
           println("$f = $(f(m))")
       end
length = 4
eltype = Int64
ndims = 2
size = (2, 2)
```

- `size()` always returns a tuple, even for vectors.
- Functions are objects, too: we can put them in a vector and iterate over it.

Indexing

- Indices start at 1 ('1-based', like in Fortran, Matlab, Mathematica, R, ...).
- `end` denotes the last index.

```
julia> v = [2, 3, 4.]
3-element Vector{Float64}:
 2.0
 3.0
 4.0
julia> v[1], v[3], v[end]
(2.0, 4.0, 4.0)
julia> v[1] = 100;
julia> v
3-element Vector{Float64}:
 100.0
   3.0
   4.0
```

- Read & write access outside the array is an error:

```
julia> v[5]
ERROR: BoundsError: attempt to access
  3-element Vector{Float64} at index [5]
julia> v[5] = 100
ERROR: BoundsError: attempt to access
  3-element Vector{Float64} at index [5]
```

Changing the length of a vector

- One-dimensional arrays can grow and shrink at both ends.
- Naming convention: functions with a `!` modify their arguments.

```
julia> v = [1, 2, 3];
julia> push!(v, 4);
julia> append!(v, [5, 6])
6-element Vector{Int64}:
 1
 2
 3
 4
 5
 6
julia> x = pop!(v)
6
```

```
julia> v = [100., 3., 4.];
julia> pushfirst!(v, 77)
4-element Vector{Float64}:
  77.0
 100.0
   3.0
   4.0
julia> x = popfirst!(v)
77.0
```

- Start with an empty vector of a given type:

```
julia> v = Float64[]
Float64[]
```

Constructors

```
julia> zeros(2, 3)
2×3 Matrix{Float64}:
 0.0  0.0  0.0
 0.0  0.0  0.0
julia> ones(Int8, 3)
3-element Vector{Int8}:
 1
 1
 1
julia> rand(2, 2)
2×2 Matrix{Float64}:
 0.359047  0.843602
 0.955374  0.417118
```

```
julia> v = zeros(Int, (2,3,2))
2×3×2 Array{Int64, 3}:
[:, :, 1] =
 0  0  0
 0  0  0

[:, :, 2] =
 0  0  0
 0  0  0
julia> v = fill(1, (5,2));
julia> w = similar(v)
5×2 Matrix{Int64}:
 140659028293168  140659028293168
 140659028293168  140659028293168
 ...
```

- The size can be given as several arguments or as a tuple.
- `similar(v)` creates an *uninitialised* array of the same type and size as `v`.
  Same for `Matrix{Float64}(undef, 2, 2)`. Contents: whatever was in memory before.

Comprehensions

- A very compact way to create arrays, close to the mathematical set notation:

$$\{i^2 \mid i \in \{1, \ldots, 10\}\}$$

```
julia> [i^2 for i in 1:10]'
1×10 adjoint(::Vector{Int64}) with eltype Int64:
 1  4  9  16  25  36  49  64  81  100
```

- Several iteration variables give a multi-dimensional array:

```
julia> [i+3j for i=0:3, j=1:3]
4×3 Matrix{Int64}:
 3  6   9
 4  7  10
 5  8  11
 6  9  12
```

- ...unless a filter `if <condition>` is added.
  Then the result is always a vector.

```
julia> [(a,b,c) for a in 1:100, b in 1:100,
                    c in 1:100 if a^2 + b^2 == c^2
                                  && a<=b && gcd(a,b)==1]
16-element Vector{Tuple{Int64, Int64, Int64}}:
 (3, 4, 5)
 (5, 12, 13)
 ...
```

Matrix literals

- Space (or `;;`) separates columns, newline (or `;`) separates rows.

```
julia> v1 = [2, 3, 4]
3-element Vector{Int64}:
 2
 3
 4
```

```
julia> v2 = [2 3 4]
1×3 Matrix{Int64}:
 2  3  4
```

```
julia> v3 = [2; 3; 4;;]
3×1 Matrix{Int64}:
 2
 3
 4
```

- A vector, a row matrix and a column matrix are three *different* objects!

```
julia> m = [2 4
            6 7]
2×2 Matrix{Int64}:
 2  4
 6  7
julia> m == [2 4; 6 7] == [2; 6;; 4; 7]
true
```

```
julia> [2; 3; 4;; 5; 6; 7;;;
        0; 2; 3;; 5; 6; 7]
3×2×2 Array{Int64, 3}:
[:, :, 1] =
 2  5
 3  6
 4  7

[:, :, 2] =
 0  5
 2  6
 3  7
```