Advanced Topics from Scientific Computing

TU Berlin Winter 2023/24

Notebook 03

 Jürgen Fuhrmann

```
begin
using PlutoUI
using LinearAlgebra
using InteractiveUtils
import AbstractTrees
using BenchmarkTools
using StaticArrays
end
```

Some Julia specifics

Type system

- Concrete types
- Structs
- Functions, Methods and Multiple Dispatch
- Abstract types
  - The type tree
- The power of multiple dispatch
  - Missing type system features

Julia: compilation and Performance

Performance measurement

Memory handling: allocations and all that

- The stack
- The heap - the place for large amounts of data
- How to release allocated memory ?

Some performance hints

- Avoid global variables
- Avoid large numbers of allocations

Type system

- Julia is a strongly typed language
- Knowledge about the layout of a value in memory is encoded in its type
- Prerequisite for performance
- There are concrete types and abstract types
- See the Julia WikiBook for more

Concrete types

- Every value in Julia has a concrete type
- Concrete types correspond to computer representations of objects
- Inquire type info using `typeof()`

Built-in types

- Default types are deduced from concrete representations

`Int64`

```
1 typeof(10)
```

`Float64`

```
1 typeof(10.0)
```

`ComplexF64 (alias for Complex{Float64})`

```
1 typeof(3.0+3im)
```

`Bool`

```
1 typeof(false)
```

`String`

```
1 typeof("false")
```

`Vector{Float16} (alias for Array{Float16, 1})`

```
1 typeof(Float16[1,2,3])
```

```
Matrix{Int64} (alias for Array{Int64, 2})
```

```
1 typeof(rand(Int,3,3))
```

- One can initialize a variable with an explicitely given fixed type.

```
10
```

```
1 i::Int8=10
```

```
Int8
```

```
1 typeof(i)
```

```
5.0
```

```
1 x::Float16=5.0
```

```
Float16
```

```
1 typeof(x)
```

Structs

- Structs allow to define custom types

```
1 struct MyColor64
2     r::Float64
3     g::Float64
4     b::Float64
5 end
```

```
c =  MyColor64(0.1, 0.2, 0.3)
```

```
1 c=MyColor64(0.1,0.2,0.3)
```

```
MyColor64
```

```
1 typeof(c)
```

```
24
```

```
1 sizeof(c)
```

```
0.1
```

```
1 c.r
```

setfield!: immutable struct of type MyColor64 cannot be changed

1. **setproperty!**(::Main.var"workspace#4".MyColor64, ::Symbol, ::Int64) @ *Base.jl:38*
2. **top-level scope** @ *Local: 1* [inlined]

```
c.r=10
```

Mutable structs allow changing fields

```
mutable struct MMyColor64
     r::Float64
    g::Float64
    b::Float64
end
```

```
mc =  MMyColor64(0.1, 0.2, 0.3)
```

```
mc=MMyColor64(0.1,0.2,0.3)
```

0.5

```
mc.r=0.5
```

```
MMyColor64(0.4, 0.2, 0.3)
```

```
mc
```

(note that in this case Pluto's reactivity is tricked out ...)

Structs can be parametrized with types. This is similar to array types which are parametrized by their element types, and to C++ template classes.

```
struct MyColor{T}
    r::T
    g::T
    b::T
end
```

```
c2 =  MyColor(4.0f0, 25.0f0, 233.0f0)
```

```
c2=MyColor{Float32}(4.0,25,233)
```

```
MyColor(1, 2, 3)
```

```
MyColor(1,2,3)
```

MyColor{Float32}

```
typeof(c2)
```

12

```
sizeof(c2)
```

```
c3 =  MyColor(0x04, 0x19, 0xe9)
```

```
1 c3=MyColor{UInt8}(4,25,233)
```

```
MyColor{UInt8}
```

```
1 typeof(c3)
```

```
3
```

```
1 sizeof(c3)
```

Functions, Methods and Multiple Dispatch

- Functions can have different variants of their implementation depending on the types of parameters passed to them.
- These variants are called **methods**
- All methods of a function `f` can be listed by calling `methods(f)`
- The act of figuring out which method of a function to call depending on the type of parameters is called **multiple dispatch**.

```
1 test_dispatch(x)="general case: $(typeof(x)), x=$(x)";
```

```
1 test_dispatch(x::Float64)="special case Float64, x=$(x)";
```

```
1 test_dispatch(x::Int64)="special case Int64, x=$(x)";
```

```
"special case Int64, x=3"
```

```
1 test_dispatch(3)
```

```
"general case: Bool, x=false"
```

```
1 test_dispatch(false)
```

```
"special case Float64, x=3.0"
```

```
1 test_dispatch(3.0)
```

Here we defined a generic method which works for any variable type passed. In the case of `Int64`, `AbstractFloat` or `Float64` parameters, special cases are handeld by different methods of the same function. The compiler decides which method to call. This approach allows to specialize implementations dependent on data types, e.g. in order to optimize perfomance.

The `methods` function can be used to figure out which methods of a function exists.

3 methods for generic function **test_dispatch** from Main.var"workspace#4":

- test_dispatch(x::**Float64**) in Main.var"workspace#4" at /home/fuhrmann/Wias/teach/adscicomp/pluto/nb03-julia-types.jl#==#0468c2da-0955-11eb-271b-5d84d5d8343d:1
- test_dispatch(x::**Int64**) in Main.var"workspace#4" at /home/fuhrmann/Wias/teach/adscicomp/pluto/nb03-julia-types.jl#==#0cc7808a-0955-11eb-0b4d-ff491af88cf5:1
- test_dispatch(x) in Main.var"workspace#4" at /home/fuhrmann/Wias/teach/adscicomp/pluto/nb03-julia-types.jl#==#f5cc25e6-0954-11eb-179b-eddff99dd392:1

```
1 methods(test_dispatch)
```

Abstract types

- Abstract types label concepts which work for a several concrete types without regard to their memory layout etc.
- All variables with concrete types corresponding to a given abstract type (should) share a common interface
- A common interface consists of a set of functions with methods working for all types exhibiting this interface
- The functionality of an abstract type is implicitely characterized by the methods working on it
- This concept is close to "duck typing": use the "duck test" — "If it walks like a duck and it quacks like a duck, then it must be a duck" — to determine if an object can be used for a particular purpose
- When trying to force a parameter to have an abstract type,it ends up with having a conrete type which is compatible with that abstract type

The type tree

- Types can have subtypes and a supertype
- Concrete types are the leaves of the resulting type tree
- Supertypes are necessarily abstract
- There is only one supertype for every (abstract or concrete) type
- Abstract types can have several subtypes

```
[BigFloat, Float16, Float32, Float64]
```

```
1 subtypes(AbstractFloat)
```

- Concrete types have no subtypes

```
[]
```

```
subtypes(Float64)
```

```
AbstractFloat
```

```
supertype(Float64)
```

```
[BigFloat, Float16, Float32, Float64]
```

```
subtypes(AbstractFloat)
```

```
Real
```

```
supertype(AbstractFloat)
```

```
Number
```

```
supertype(Real)
```

```
Any
```

```
supertype(Number)
```

- "Any" is the root of the type tree and has itself as supertype

```
Any
```

```
supertype(Any)
```

We can use the `AbstractTrees` package to visualize the type tree. We just need to define what it means to have children for a type.

```
AbstractTrees.children(x::Type) = subtypes(x)
```

```
AbstractTrees.print_tree(Number)
```

```
Number
├─ Complex
└─ Real
   ├─ AbstractFloat
   │  ├─ BigFloat
   │  ├─ Float16
   │  ├─ Float32
   │  └─ Float64
   ├─ AbstractIrrational
   │  └─ Irrational
   ├─ FixedPoint
   │  ├─ Fixed
   │  └─ Normed
   ├─ Integer
   │  ├─ Bool
   │  ├─ Signed
   │  │  ├─ BigInt
   │  │  ├─ Int128
```

There are operators for testing type relationships

true

```
Float64<: Number
```

false

```
Float64<: Integer
```

false

```
isa(3,Float64)
```

true

```
isa(3.0,Float64)
```

Abstract types can be used for method dispatch as well

dispatch2 (generic function with 2 methods)

```
begin
    dispatch2(x::AbstractFloat)="$(typeof(x)) <:AbstractFloat, x=$(x)"
    dispatch2(x::Integer)="$(typeof(x)) <:Integer, x=$(x)"
end
```

"Bool <:Integer, x=false"

```
dispatch2(false)
```

"Float64 <:AbstractFloat, x=13.0"

```
dispatch2(13.0)
```

The power of multiple dispatch

- Multiple dispatch is one of the defining features of Julia
- Combined with the the hierarchical type system it allows for powerful generic program design
- New datatypes (different kinds of numbers, differently stored arrays/matrices) work with existing code once they implement the same interface as existent ones.
- In some respects C++ comes close to it, but for the price of more and less obvious code

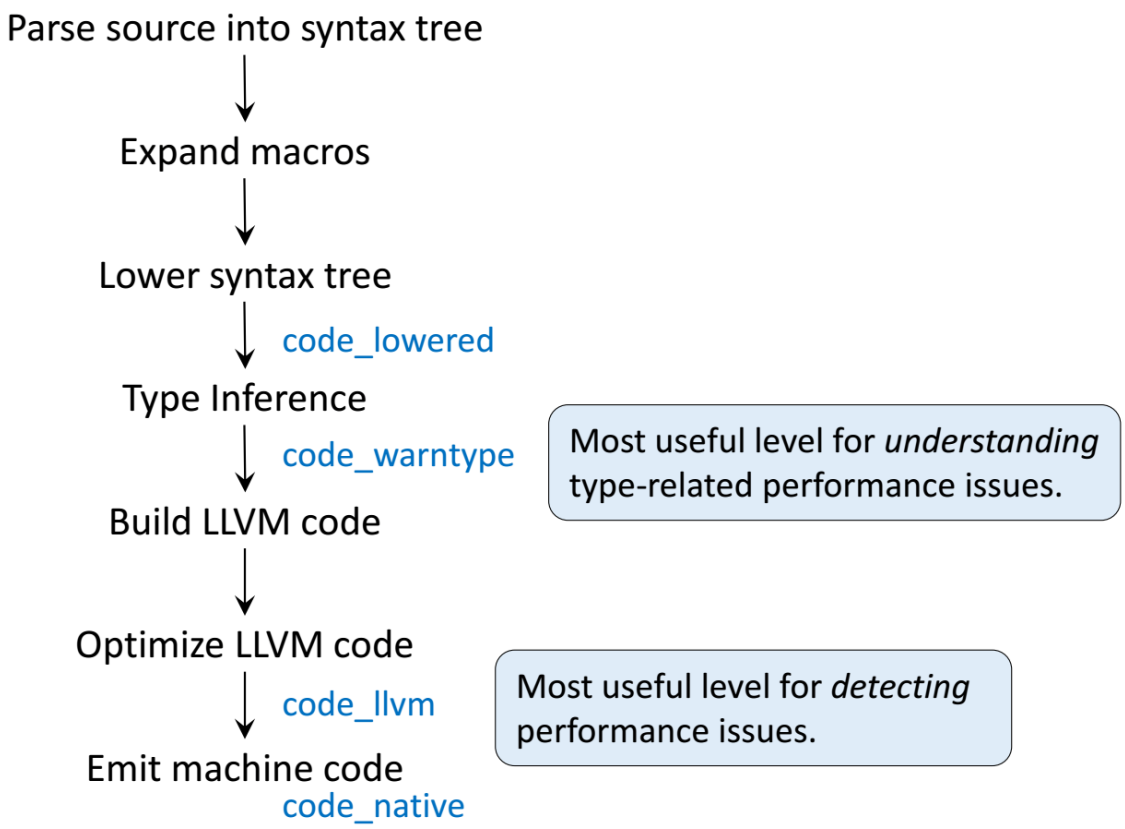
Missing type system features

- Formalization of interface definitions
- Possibility of multiple supertypes

Introduction of these and similar features is often discussed, but delayed at the moment in order to avoid breaking language changes.

Julia: compilation and Performance

- Just-in-time (JIT) or, more precisely, just-ahead of time (JAOT) compilation is another feature setting Julia apart
- Julia development started with this idea
- Julia uses the tools from the The LLVM Compiler Infrastructure Project to organize compilationof Julia code to machine code
- Tradeoff: startup time for code execution in interactive situations, however, since v1.9, machine code is cached on the disk for later reuse.
- Multiple steps: Parse the code, analyze data types etc.
- Intermediate results can be inspected using a number of macros (blue color in the diagram below)



From Introduction to Writing High Performance Julia by D. Robinson

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

```
1 g(x,y)=x+y
```

Call with integer parameter:

```
5
```

```
1 g(2,3)
```

Call with floating point parameter:

```
5.0
```

```
1 g(2.0,3.0)
```

The macro `@code_lowered` describes the abstract syntax tree behind the code

```
CodeInfo(
1 — %1 = x + y
└──      return %1
)
```

```
1 @code_lowered g(2,3)
```

```
CodeInfo(
1 — %1 = x + y
└──      return %1
)
```

```
1 @code_lowered g(2.0,3.0)
```

`@code_warntype` (with output to terminal) provides the result of type inference (detection of the parameter types and cooresponding choice of the translation strategy) according to the input:

```
1 @code_warntype g(2,3)
```

```
MethodInstance for Main.var"workspace#4".g(::Int64, ::Int64)
  from g(x, y) @ Main.var"workspace#4" ~/Wias/teach/adscicomp/pluto/nb03-julia-types.jl#==#2c750c8c-c0f4-4800-bdab-cbaf17b55bb7:1
Arguments
  #self#::Core.Const(Main.var"workspace#4".g)
  x::Int64
  y::Int64
Body::Int64
1 — %1 = (x + y)::Int64
└──      return %1
```

```
1 @code_warntype g(2.0,3.0)
```

```
MethodInstance for Main.var"workspace#4".g(::Float64, ::Float64)
  from g(x, y) @ Main.var"workspace#4" ~/Wias/teach/adscicomp/pluto/nb03-julia-types.jl#==#2c750c8c-c0f4-4800-bdab-cbaf17b55bb7:1
Arguments
  #self#::Core.Const(Main.var"workspace#4".g)
  x::Float64
  y::Float64
Body::Float64
1 — %1 = (x + y)::Float64
└──      return %1
```

`@code_llvm` prints a human readable rendering og the LLVM intermediate byte code representation:

```
1 @code_llvm g(2,3)
```

```
;  @ /home/fuhrmann/Wias/teach/adscicomp/pluto/nb03-julia-types.jl#==#2c750c8c-c0f4-4800-bdab-cbaf17b55bb7:1 within `g`
define i64 @julia_g_5359(i64 signext %0, i64 signext %1) #0 {
top:
; ┌ @ int.jl:87 within `+`
   %2 = add i64 %1, %0
; └
  ret i64 %2
}
```

```
1 @code_llvm g(2.0,3.0)
```

```
;  @ /home/fuhrmann/Wias/teach/adscicomp/pluto/nb03-julia-types.jl#==#2c750c8c-c0f4-4800-bdab-cbaf17b55bb7:1 within `g`
define double @julia_g_5384(double %0, double %1) #0 {
top:
; ┌ @ float.jl:408 within `+`
   %2 = fadd double %0, %1
; └
  ret double %2
}
```

Finally, `@code_native` prints the assembler code generated, which is a close match to the machine code sent to the CPU:

```
1 @code_native g(2,3)
```

```
    .text
    .file   "g"
    .globl  julia_g_5399                    # -- Begin function julia_g_5399
    .p2align    4, 0x90
    .type   julia_g_5399,@function
julia_g_5399:                           # @julia_g_5399
; ┌ @ /home/fuhrmann/Wias/teach/adscicomp/pluto/nb03-julia-types.jl#==#2c750c8c-c0f4-4800-bdab-cbaf17b55bb7:1 within `g`
    .cfi_startproc
# %bb.0:                                # %top
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
; │┌ @ int.jl:87 within `+`
   leaq    (%rdi,%rsi), %rax
; │└
    popq    %rbp
```

```
1 @code_native g(2.0,3.0)
```

```
    .text
    .file   "g"
    .globl  julia_g_5405                    # -- Begin function julia_g_5405
    .p2align    4, 0x90
    .type   julia_g_5405,@function
julia_g_5405:                               # @julia_g_5405
; ┌ @ /home/fuhrmann/Wias/teach/adscicomp/pluto/nb03-julia-types.jl#==#2c750c8
c-c0f4-4800-bdab-cbaf17b55bb7:1 within `g`
    .cfi_startproc
# %bb.0:                                    # %top
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
; │┌ @ float.jl:408 within `+`
    vaddsd  %xmm1, %xmm0, %xmm0
; │└
    popq    %rbp
```

We see that for the very same function, Julia creates different variants of executable code depending on the data types of the parameters passed. In certain sense, this extends the multiple dispatch paradigm to the lower level by automatically created methods.

```
1 using MethodAnalysis
```

```
Core.MethodInstance[
    1:  MethodInstance for Main.var"workspace#4".g(::Int64, ::Int64)
    2:  MethodInstance for Main.var"workspace#4".g(::Float64, ::Float64)
]
```

```
1 methodinstances(g)
```

1 method for generic function **g** from Main.var"workspace#4":

- g(x, y) in Main.var"workspace#4" at /home/fuhrmann/Wias/teach/adscicomp/pluto/nb03-julia-types.jl#==#2c750c8c-c0f4-4800-bdab-cbaf17b55bb7:1

```
1 methods(g)
```

Performance measurement

- Julia provides a number of macros to support performance testing.
- Performance measurement of the first invocation of a function includes the compilation step. If in doubt, measure timing twice.
- Pluto has the nice feature to indicate the execution time used below the lower right corner of a cell. There seems to be also some overhead hidden in the pluto cell handling which is however not measured.

- `@elapsed`: wall clock time used returned as a number.

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

```
1 f(n1,n2)= mapreduce(x->norm(x,2),+,[rand(n1) for i=1:n2])
2
```

```
0.001737091
```

```
1 @elapsed f(1000,1000)
2
```

- `@allocations` returns the number of allocations (available since Julia 1.9)

```
1001
```

```
1 @allocations f(1000,1000)
```

- `@allocated`: sum of memory allocated (including temporary) during the excution of the code. For storing intermediate and final calculation results, computer languages request memory from the operating system. This process is called allocation. Allocations as a rule are linked with lots of bookkeeping, so they can slow down code.

```
8136128
```

```
1 @allocated f(1000,1000)
```

`@time`: `@elapsed` and `@allocations` together, with output to the terminal. Be careful to time at least twice in order to take into account compilation time. In addition, the number of allocations is printed along with time spent for garbage collection. Garbage collection is the process of returning unused (temporary) memory to the system.

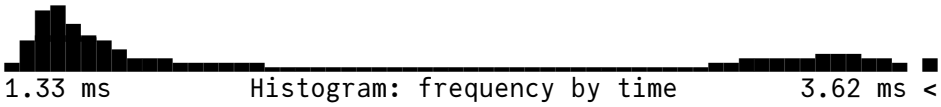
```
18255.742087111743
```

```
1 @time f(1000,1000)
```

```
0.001650 seconds (1.00 k allocations: 7.759 MiB)
```

- `@benchmark` from `BenchmarkTools.jl` creates a statistic over multiple samples in order to give a more reliable estimate.

```
BenchmarkTools.Trial: 2703 samples with 1 evaluation.
 Range (min … max):  1.334 ms …   3.950 ms  ┊ GC (min … max):  0.00% … 50.39%
 Time  (median):     1.507 ms               ┊ GC (median):     0.00%
 Time  (mean ± σ):   1.845 ms ± 741.690 μs  ┊ GC (mean ± σ):  16.74% ± 19.42%
```




```
Memory estimate: 7.76 MiB, allocs estimate: 1001.
```

```
1 @benchmark f(1000,1000)
```

- `@btime` prints the minimum elapsed time - this is the most realistic measure.

```
18252.649432788396
```

```
1         @btime f(1000,1000)
```

```
1.355 ms (1001 allocations: 7.76 MiB)
```

Memory handling: allocations and all that

All variable data of running computer code is stored in the main memory (RAM). This is true for almost any computer language.

There are however details of the way data is stored which have a heavy impact on code performance and flexibility of code design.

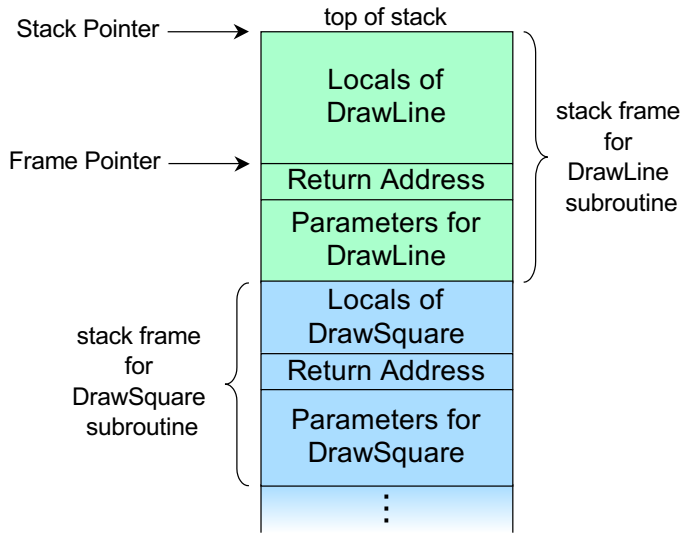
The stack

- The stack is a fixed size memory region made available when a program starts. It is implicitely passed to all functions subsequently called, providing memory space for storing local variables - that is variables declared in a function body.
- The name comes from the data structure used to manage the access to the region. Besides of the memory, a stack is characterized by a *stack pointer* which separates the unused space from the used one.
- When "putting data on the stack", these are copied to the position indicated by the stack pointer, and its value is increased accordingly
- "Removing data from the stack" just amounts to decreasing the stack pointer
- Any time a function is called, the current position in the instruction stream is stored in the stack as the return address, and the called function is allowed to work with the stack space following this storage location

```
function DrawLine(x0,y0,x1,y1)
  ... perform some drawing ...
end



function DrawSquare(x0,y0,a)
   xa=x0+a
   ya=y0+a
   DrawLine(x0,y0,xa,y0)
   DrawLine(xa,y0,xa,ya)
   DrawLine(xa,ya,x0,ya)
   DrawLine(x0,ya,x0,y0)
end
```



Drawing by R. S. Shaw, Public Domain

- `DrawSquare` takes space from the stack to store its local variables `xa` and `ya`.
- In the four calls to `DrawLine`, each time, the parameters are copied on the stack, and the current pointer in the instruction stream is stored on the stack as the address to return to after finishing the call
- During execution, `DrawLine` may put its own local variables on the stack and call other functions
- After returning from the call, everything on top of the local data of `DrawSquare` is "forgotten"

The advantage of this way of memory management consists in its simplicity, and therefore efficiency.

But ...

`euler_sum_stack (generic function with 1 method)`

```
function euler_sum_stack(n; e=1.0,k=1.0,kfac=1.0)
    if k<n
        kfac=kfac*k
        euler_sum_stack(n,e=e+1/kfac,k=k+1,kfac=kfac)
    else
        e
    end
end

```

Did we do the right thing ?

`2.7312660577649694e-8`

```
abs(euler_sum_stack(11)-e)
```

Now let us try to become more accurate:

```
StackOverflowError:

  1. NamedTuple  @  boot.jl:627  [inlined]
  2. NamedTuple  @  boot.jl:623  [inlined]
  3. var"#euler_sum_stack#1"(::Float64, ::Float64, ::Float64,
     ::typeof(Main.var"workspace#64".euler_sum_stack), ::Int64)  @  Other: 4
```

```
1 euler_sum_stack(100_000)
```

Stack space is scarce!

When a program starts, it obtains a fixed amount of memory for its stack from the operating from the system. During program execution it cannot be increased. As a rule, it's size however can be configured before the program starts. Operating systems have a mostly reasonable default for stack size.

The heap - the place for large amounts of data

- Chunks from free system memory can be reserved – "allocated" – on demand in order to provide memory space for data.
- Unlike the handling of the stack pointer, allocating memory is connected with lots of bookkeeping, so it is quite expensive.
- In languages like C, C++, allocation is a function call (`malloc`, `new`), though this might be hidden behind e.g. the constructor of a `std::vector`
- In Julia, the choice between stack and heap depends on the data type, though in principle the compiler could optimize allocations away if it knows that this is save
  - heap: Arrays, Dicts, mutable structs
  - stack: Numbers, Tuples, structs, Arrays from StaticArrays.jl, array views
  - also see this Discourse thread

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

```
1 function array_sum(x)
2     arr = [x,2.0x,3.0x,4x,5x,6x,7x,8x,9x,10x,11x]
3     return sum(arr)
4 end
```

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

```
function static_array_sum(x)
    arr = StaticArrays.SVector(x,2.0x,3.0x,4x,5x,6x,7x,8x,9x,10x,11x)
    return sum(arr)
end

```

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

```
function array_sum!(v,x)
    for i=1:length(v)
        v[i]=i*x
    end
    return sum(v)
end
```

```
BenchmarkTools.Trial: 10000 samples with 996 evaluations.
 Range (min … max):  25.864 ns …  1.947 μs  ┊ GC (min … max):  0.00% … 97.14%
 Time  (median):     27.753 ns              ┊ GC (median):     0.00%
 Time  (mean ± σ):   34.385 ns ± 91.005 ns  ┊ GC (mean ± σ):  14.58% ±  5.41%
```

```
 25.9 ns      Histogram: log(frequency) by time      66.5 ns <

 Memory estimate: 144 bytes, allocs estimate: 1.
```

```
@benchmark array_sum(20)
```

```
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
 Range (min … max):  1.059 ns … 5.498 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     1.096 ns             ┊ GC (median):    0.00%
 Time  (mean ± σ):   1.115 ns ± 0.084 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

```
 1.06 ns        Histogram: frequency by time       1.24 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

```
@benchmark static_array_sum(20)
```

```
const v =   [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

```
const v=zeros(11)
```

```
BenchmarkTools.Trial: 10000 samples with 999 evaluations.
 Range (min … max):   9.426 ns … 17.627 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     10.062 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   10.106 ns ±  0.386 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

```
 9.43 ns      Histogram: log(frequency) by time      11.3 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

```
@benchmark array_sum!(v,20)
```

Allocations are expensive!

- We see a significant increase of runtime due to an allocation: one can be more ten times as expensive as a floating point operation
- Avoiding allocations is an important step when optimizing Julia code
- One strategy is to work with tuples and SVectors which however must have their size fixed at compile time
- Alternatively, turn functions into mutating functions which work on memory space passed to them which has been allocated beforehand

How to release allocated memory ?

- In languages like C and C++, there are explicit statements for releasing memory allocated at the heap (`free`, `delete`). They can be hidden behind object destructors automatically called before returning from a function.
- Julia has a *Garbage Collector* (GC) which keeps track of memory usage and frees memory once it is not needed anymore. It automatically runs ("sweeps") between statements.

Some performance hints

Avoid global variables

```
myvec =
 [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
```

```
1 myvec=ones(Float64,1_000_000)
```

```
function mysum(v)
    x=0.0
    for i=1:length(v)
        x=x+v[i]
    end
    return x
end;
```

```
BenchmarkTools.Trial: 9353 samples with 1 evaluation.
 Range (min … max):  527.538 μs … 718.302 μs  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     529.075 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   531.939 μs ±   8.981 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

```
 528 μs        Histogram: log(frequency) by time        565 μs <

 Memory estimate: 16 bytes, allocs estimate: 1.
```

```
@benchmark mysum(myvec)
```

```
BenchmarkTools.Trial: 81 samples with 1 evaluation.
 Range (min … max):  61.300 ms … 71.318 ms  ┊ GC (min … max): 8.24% … 7.46%
 Time  (median):     61.779 ms              ┊ GC (median):    8.61%
 Time  (mean ± σ):   62.116 ms ±  1.318 ms  ┊ GC (mean ± σ):  8.90% ± 1.08%
```

```
 61.3 ms         Histogram: frequency by time        65.4 ms <

 Memory estimate: 91.54 MiB, allocs estimate: 4998980.
```

```
@benchmark begin
    x1=0.0
    for i=1:length(myvec)
        x1=x1+myvec[i]
    end
end
```

- Observation: both the begin/end block and the function do the same operation and calculate the same value. However the function is faster.
- The code within the begin/end clause works in the *global context*, whereas in `myfunc`, it works in the scope of a function. Julia is unable to dispatch on variable types in the global scope as they can change their type anytime. n the global context it has to put all variables into "boxes" tagged with type information allowing to dispatch on their type at runtime (this is by the way the default mode of Python). In functions, it has a chance to generate specific code for known types.
- Sometimes this is nevertheless hard to avoid - in this case, global constants can be made `const`, and since Julia v1.8, global variables can be typed.

Avoid the use of global variables

- Always Wrap performace critical code into functions
- This situation als occurs in the REPL.
- In fact it is anyway good coding style to separate out pieces of code into functions

Avoid large numbers of allocations

Three different ways of sum squares of matrix columns:

```
mymat = 10×100000 Matrix{Float64}:
 0.545357   0.739537  0.559041  0.289961   …  0.998465   0.704766   0.0768146
 0.648402   0.347477  0.669164  0.750804      0.206287   0.984684   0.390419
 0.919837   0.812439  0.153051  0.774273      0.236896   0.998815   0.644617
 0.271035   0.674481  0.64882   0.160063      0.733235   0.772355   0.64771
 0.18147    0.816421  0.677034  0.493633      0.909496   0.831408   0.0706894
 0.75007    0.318208  0.616359  0.474652   …  0.292198   0.100016   0.621373
 0.72749    0.310167  0.88076   0.286307      0.749029   0.70961    0.315238
 0.940119   0.106     0.172269  0.175492      0.499539   0.0790332  0.185407
 0.933641   0.737068  0.640129  0.0882112     0.669024   0.315119   0.526164
 0.0472359  0.857505  0.683675  0.306054      0.0454442  0.734238   0.197882
```

```
mymat=rand(10,100_000)
```

```
g1 (generic function with 1 method)
```

```
function g1(a)
    y=0.0
    for j=1:size(a,2)
        for i=1:size(a,1)
            y=y+a[i,j]^2
        end
    end
    y
end
```

```
g2 (generic function with 1 method)
```

```
function g2(a)
    y=0.0
    for j=1:size(a,2)
        y=y+mapreduce(z->z^2,+,a[:,j])
    end
    y
end
```

```
g3 (generic function with 1 method)
```

```
function g3(a)
    y=0.0
    for j=1:size(a,2)
        @views y=y+mapreduce(z->z^2,+,a[:,j])
    end
    y
end
```

```
true
```

```
g1(mymat)≈ g2(mymat) && g2(mymat)≈ g3(mymat)
```

```
BenchmarkTools.Trial: 9304 samples with 1 evaluation.
 Range (min … max):  527.557 μs … 767.752 μs  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     530.888 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   534.760 μs ±  12.231 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

```
 528 μs        Histogram: log(frequency) by time        587 μs <

 Memory estimate: 16 bytes, allocs estimate: 1.
```

```
@benchmark g1(mymat)
```

```
BenchmarkTools.Trial: 1494 samples with 1 evaluation.
 Range (min … max):  2.770 ms …   6.730 ms  ┊ GC (min … max):  0.00% …  0.00%
 Time  (median):     2.869 ms               ┊ GC (median):     0.00%
 Time  (mean ± σ):   3.343 ms ± 722.699 μs  ┊ GC (mean ± σ):  14.27% ± 16.13%
```

```
 2.77 ms         Histogram: frequency by time        4.57 ms <

 Memory estimate: 13.73 MiB, allocs estimate: 100001.
```

```
@benchmark g2(mymat)
```

```
BenchmarkTools.Trial: 9416 samples with 1 evaluation.
 Range (min … max):  514.567 μs … 760.522 μs  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     522.438 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   529.035 μs ±  24.569 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

```
 515 μs        Histogram: log(frequency) by time        660 μs <

 Memory estimate: 16 bytes, allocs estimate: 1.
```

```
@benchmark g3(mymat)
```

- Observation: g1 is the fastest implemetation, then comes g3 and then with significant distance g2.
- The difference between g2 and g1 is that each time we use a matrix slice `a[:,i]`, memory is allocated and data copied. Only then the mapreduce is employed, and the intermediate memory is garbage collected.
- The difference between g2 and g3 lies in the use of the `@views` macro which allows to avoid the creation of intermediae memory for matrix rows.

Avoid allocations of many small pieces of memory in "hot loops"

- Check your code for allocations of temporary memory
- If there are "many" allocations and their number grows proportionally with problem size, find the reason and modify your code to avoid them
- A couple of allocations (e.g. to pre-allocate memory for mutating functions) outside of hot loops is ok

There is more performance related information here.
