Scientific Computing TU Berlin Winter 2021/22 © Jürgen Fuhrmann
Notebook 07

```
begin
    using PlutoUI
    using HypertextLiteral
    using StaticArrays
    using BenchmarkTools
end
```

Memory handling: allocations and all that

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.

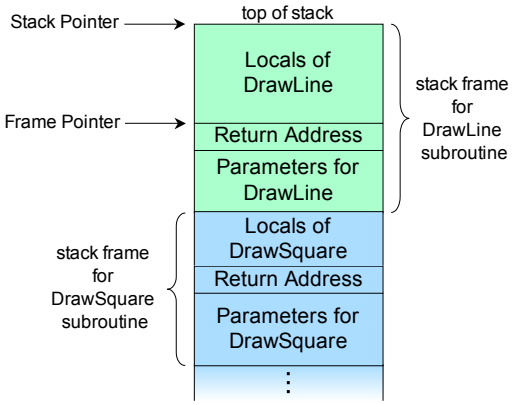
There are however details of the way data is stored which have a heavy impact on code performance and flexibility of code design.

The stack

- The stack is a memory region created when a program starts, which is implicitely passed to all functions subsequently called, providing memory space for storing local variables
- The name comes from the data structure behind. Besides of the memory it is characterized by a *stack pointer* which separated 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

- `DrawSquarw` 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 current pointer in the instruction stream is stored on the stack as the address to return to after finishing the call
- During execution, `DrawLine` ma 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"

Let us calculate Euler's number in a recursive manner:

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

```
function euler_sum_stack(n; e=1.0,k=1,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 ?

```
8.149037000748649e-13
```

```
abs(euler_sum_stack(15)-e)
```

Now let us try to become more accurate:

StackOverflowError:

1. **var"#euler_sum_stack#1"**(::Float64, ::Int64, ::Float64, ::typeof(Main.workspace#2.euler_sum_stack), ::Int64) @ *Other: 4*

```
euler_sum_stack(100_000)
```

Stack space is scarce!

When a program starts, it obtains from the system a fixed amount of memory for its stack. During program run it cannot be increased. It's size however can be configured before the program starts.

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++, this is an explicit operation (`malloc`, `new`)
- In Julia, the placement 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**

Allocations are expensive!

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

```
function normal_array()
    arr = [1.,2.0,3.0]
    return arr.^2
end
```

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

```
function static_array()
    arr = StaticArrays.SVector(1.,2,3)
    return arr.^2
end
```

```
BenchmarkTools.Trial: 10000 samples with 990 evaluations.
 Range (min … max):  46.794 ns …  1.175 μs  ┊ GC (min … max): 0.00% … 90.81%
 Time  (median):     48.536 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   55.287 ns ± 57.171 ns  ┊ GC (mean ± σ):  6.90% ±  6.39%
```


```
 Memory estimate: 224 bytes, allocs estimate: 2.
```

```
@benchmark normal_array()
```

```
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
 Range (min … max):  0.015 ns … 14.699 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     0.017 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   0.019 ns ±  0.147 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```


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

```
@benchmark static_array()
```

- We see a significant increase of runtime: an allocation can be several hundred times more expensive than 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 space passed to them which has been allocated before

```
a =  [1.0, 2.0, 3.0]
```

```
a=[1.0,2,3]
```

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

```
function functional_function(a)
    return a.^2
end
```

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

```
function mutating_function!(result,a)
    result.=a.^2
end
```

```
BenchmarkTools.Trial: 10000 samples with 994 evaluations.
 Range (min … max):  32.654 ns …  1.436 μs  ┊ GC (min … max): 0.00% … 94.16%
 Time  (median):     33.614 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   38.277 ns ± 49.413 ns  ┊ GC (mean ± σ):  6.13% ±  4.66%
```


```
 Memory estimate: 112 bytes, allocs estimate: 1.
```

```
@benchmark functional_function(a)
```

```
result =  [0.0, 0.0, 0.0]
```

```
result=zeros(3)
```

```
BenchmarkTools.Trial: 10000 samples with 998 evaluations.
 Range (min … max):  13.873 ns … 83.076 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     14.237 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   14.881 ns ±  3.913 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```


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

```
@benchmark mutating_function!(result,a)
```

How to release allocated memory ?

- In languages like C and C++, there are explicit statements for releasing memory allocated at the heap (`free`, `delete`)
- Julia has a *Garbage Collector* (GC) which keeps track of memory usage and frees memory once it is not needed anymore. It automatically runs between statements.