## Composite types

The constructor ‘struct’ for creating a composite type named System

```
struct System
   size::Int
   temp::Float64
   conf::Array{Int,1}
end
```

These are the **fields** of System

An object of type system can now be created, e.g.,

```
sys=System(a,b,c)
```

where a,b,c must match the field types of System

The fields are accessed as: `sys.size, sys.temp, sys.conf`

There is a function fieldnames() that returns the field names

sys can be passed as an argument to a function like any object

A struct is an unmutable object

- but in sys the array field is still mutable (can be changed in a function)

There is also `mutable struct`

Example in struct.jl

## More about mutable/unmutable, variable bindings

A variable in julia is bound to (refers to, points to) a value

var ⟶ value

- var is a memory address
- value is stored at that address

- var2 = var means that var2 will point to the same value as var

when an **unmutable** object is changed (e.g., var=var+1)

- 'value' may not change, but var points to another address with new value

when a **mutable** object changes

- the address does not change but the contents of that address change

An array is an example of a mutable object

- the binding is to the first memory address where the array is stored

Of relevance to how arguments are passed (from Julia doc):

Julia function arguments follow a convention sometimes called "pass-by-sharing", which means that values are not copied when they are passed to functions. Function arguments themselves act as new variable bindings (new locations that can refer to values), but the values they refer to are identical to the passed values. Modifications to mutable values (such as an array) made within a function will be visible to the caller.

## More about functions

```
function func(a,b,c)
. . .
return d,e
end
```

without return, the last evaluated expression is returned

`return` or `return nothing` returns object 'nothing'

### Single-expression function

```
func(arguments) = expression
func(a,b,c) = a+b-c
```

expression can be multiple statements between begin … end

Functions are objects that can be assigned, passed to other functions, etc

```
func2=func
somefunction(func,…)
```

Read about: optional arguments, Varargs (arbitrary number of arguments), keywords…

### Anonymous function

Example from Julia documentation

```
julia> map(x -> x^2 + 2x - 1, [1, 3, -1])
3-element Vector{Int64}:
  2
 14
 -2
```

map(function,collection) is a Base function, performs function on each element of collection

## Modules

Can be used to organize codes

- make modules with functions and data structures for specifc tasks
- variables and functions can be exported to code block using the module

```
module ModName
...
export vari1, func1
...
end
```

```
using .ModName
```

if module declared in same file

Even functions/data not exported can be accessed:

```
ModName.vari2
Modname.func2
```

Those exported do not need ModName

```
include("modname.jl")
using .ModName
```

if in a different file

- include() inserts the contents of the file

. before module name required if the module is not installed as a package

- only make a package if you have developed a stable module

Example in module.jl, to be used with main.jl

## Using modules available in the “community”

**Packages (which may involve several modules) that are registered can be added with the REPL package manager**

**Information about the registry and all its packages available here**

https://github.com/JuliaRegistries/General

**You can register your own package if you make something useful!**

**There is a search function, but just googling “Julia whatyouwant” may be better**

**Example: after googling “Julia integration” I quickly found QuadGK**

https://juliapackages.com/p/quadgk

**Installation in the REPL package manager (“]” at the Julia prompt)**

```
(@v1.6) pkg> add QuadGK
    Updating registry at `~/.julia/registries/General`
   Resolving package versions...
   Installed QuadGK — v2.4.1
    Updating `~/.julia/environments/v1.6/Project.toml`
  [1fd47b50] + QuadGK v2.4.1
    Updating `~/.julia/environments/v1.6/Manifest.toml`
  [1fd47b50] + QuadGK v2.4.1
Precompiling project...
  1 dependency successfully precompiled in 3 seconds (136 already precompiled, 1 skipped during auto
 due to previous errors)
```

**Now we can integrate functions of one variable:**

```
julia> using QuadGK

julia> integral, err = quadgk(x -> exp(-x^2), 0, 1, rtol=1e-8)
(0.746824132812427, 7.887024366937112e-13)
```

## The “let” block; a simple way to store values in functions

We often want to store the “internal state” of some function
without having to pass that state as an argument

For example, rand() can be called without any argument
- but clearly there must be some internal state that is somehow saved

References to data (pointers) can be permanently saved in “let” blocks
- functions defined inside a let block can access these pointers

Example, part of letblock.jl (random number generator, inside a module)

```
let
    r = Ref(convert(UInt64,1))
    global function ran64()
       r[]=r[]*a+c
    end
end
```

r is a reference (pointer) to an unsigned integer
- the value at r is accessed by r[]
- would be r[i] for element i of a 1-dim array

Why not just use r declared in the global scope?
- for efficiency, avoid using global variables

The function must be declared global to make it accessible outside let-end
- global function objects are treated as constants, not slowing things down
- the integers a and c are declared as constants before let

The let block is a local hard scope, many other uses (see Julia doc)