Example: integer declarations and operations [int1.jl]

```
function integertest()
   a::UInt32=typemax(UInt32)
   b::UInt32=1
   c=a+b
   return a,c
end
x,y=integertest()
println(x)
println(y)
```

Base function typemax gives largest value
- typemin gives smallest

function “integertest” with no arguments is declared

variables a, b declared as unsigned 32-bit integers and given values

two integers are returned by the function

Base function println writes a line to standard output

Output:

```
$ 4294967295
$ 0
```

$2^{32} - 1$

$(2^{32} - 1 + 1) \bmod 2^{32}$

Try also with “Int32” instead of “UInt32”!

Example with an error [int2.jl]

Changing the function to (keep the rest of the previous example)

```
function integertest()
   a::UInt32=typemax(UInt32)
   b::UInt32=1
   b=a+1
   return a,b
end
```

Running gives this error message (+ more):

```
ERROR: LoadError: InexactError: trunc(UInt32, 4294967296)
```

Reason: My computer (and likely yours) is based on 64-bit architecture

- the constant “1” is then of type Integer64
- a+1 also is of type Integer64 (the “larger” of the two types involved)
- b is declared as UInt32 and cannot represent the value pf a+1

Integer types in Julia

`Int8, Int16, Int32, Int64, Int128`
`UInt8, UInt16, UInt32, UInt64, UInt128`

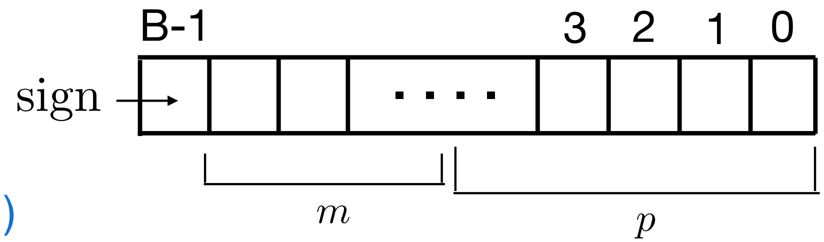
`Int` is the default integer type
- normally same as `Int64`

Bit representation of floating-point numbers

Arbitrary real-valued numbers cannot be represented by bits

- approximated by certain rational numbers; “floating-point numbers”

- p bits for “significand” (fraction, mantissa)
- m bits for exponent
- 1 sign bit



here bits b(i) are counted
from left (i=0) to right (i=p-1)

$$R = \text{sign} \times 2^e \sum_{i=0}^{p-1} b(i) 2^{-i} \quad \rightarrow \text{sign} \times 2^e \left(1 + \sum_{i=0}^{p-1} b(i) 2^{-(i+1)} \right)$$

1 ≤ significand < 2

The exponent can be positive or negative

- exactly how the exponent is stored is a bit subtle (we don’t need the details)

On most computers:

- single-precision (4 bytes); p=23, m=8 (precision about 7 decimals)
- double-precision (8 bytes); p=53, m=10 (precision about 16 decimals)
- some times 16-byte quadruple precision is available

Special values represented

+0,-0, +infinity, -infinity, “not a number” NaN

Example: floats, random numbers, arrays, multiple dispatch [randomarray.jl]

```
function makerandom(n::Int)
   r=Array{Float64}(undef,n)
   for i=1:n
      r[i]=rand()
   end
   return r
end
```

First method, Int argument
- array with n elements (undefined contents)
- one way to loop over values i
- i:th element assigned a random value in [0,1)

```
function makerandom(m::Float64)
   n=round(Int,m)
   r=Array{Float32}(undef,n)
   for i=1:n
      r[i]=rand()
   end
   return r
end
```

Second method, Float64 argument
- round to closest integer and convert to Int

In general, any number of methods can be used, as long as they can be uniquely identified by their arguments (more on functions later)

Two function declarations, same name, different argument types
- it's really one function with two methods
- the method that matches calling raguments is dispached

Code calling this function:

```
n=5
m=convert(Float64,n)
a=makerandom(n)
for i=1:n
   println(i,"  ",a[i])
end
a=makerandom(m)
for i=1:n
   println(i,"  ",a[i])
end
```

- converts integer n to 64-bit float

Output

```
$ 1  0.768629462884634
$ 2  0.2031804749902122
$ 3  0.1664474670812679
$ 4  0.5501970241421752
$ 5  0.4978716671303165
$ 1  0.5057016
$ 2  0.65821403
$ 3  0.2276439
$ 4  0.83020467
$ 5  0.84432185
```

Note, in second method: Float64 value is assigned to a Float32 variable; OK but of course some precision is lost

<u>Floating-point types in Julia</u>

`Float16, Float32, Float64`

Examples of matrix operations [matrix.jl]

Function to make a random n*n matrix

```
function randmatrix(n::Int)
   mat=Array{Float64}(undef,n,n)
   for j=1:n
      for i=1:n
         mat[i,j]=rand()
      end
   end
   return mat
end
```

matrix = 2-dimensional array

```
a=randmatrix(n)
b=randmatrix(n)
c=a*b
```

here * means actual matrix multiplication

```
for i=1:n
   println(a[i,:],"  ",b[i,:],"  ",c[i,:])
end
```

: means all elements

```
c=a.*b
```

point . before operator means element-by-element

```
b=inv(a)
```

Base function for matrix inversion

Notes on variable/function names, non-ascii symbols

Names are case-sensitive; “Var” is different from “var”

- customary to use lower case for variables and function names
- use upper case first letter for module and type names
- functions that change arguments end in “!” (I violate this rule…)

Names of variables and functions can contain Unicode characters
- in addition to the conventional ASCII characters

Example: [specialnames.jl]

```
function 笨蛋(γ)
    α=1
    β=1
    δ=α+β+γ
    return δ
end
println(笨蛋(2))
```

Depending on your editor/environment, it may be painful to enter characters

- in the REPL, Latex commands can be used, e.g., \delta<tab> for δ
- probably better to avoid using special characters in code

Elementary Mathematical Operations

from julialang.org

Expression	Name	Description
`+x`	unary plus	the identity operation
`-x`	unary minus	maps values to their additive inverses
`x + y`	binary plus	performs addition
`x - y`	binary minus	performs subtraction
`x * y`	times	performs multiplication
`x / y`	divide	performs division
`x ÷ y`	integer divide	x / y, truncated to an integer
`x \ y`	inverse divide	equivalent to `y / x`
`x ^ y`	power	raises `x` to the `y`th power
`x % y`	remainder	equivalent to `rem(x,y)`

x op y is really equivalent to op(x,y), i.e., op is a function with two arguments. Try in the REPL:

```
julia> +
+ (generic function with 190 methods)
```

same as div(x,y); ÷ is \div<tab> in the REPL

Updating ops: `+=` `-=` `*=` `/=` `\=` `÷=` `%=` `^=` `&=` `|=` `⊻=` `>>>=` `>>=` `<<=`

x += y is equivalent to x = x + y, etc.

Rounding functions

Function	Description	Return type
`round(x)`	round x to the nearest integer	`typeof(x)`
`round(T, x)`	round x to the nearest integer	T
`floor(x)`	round x towards `-Inf`	`typeof(x)`
`floor(T, x)`	round x towards `-Inf`	T
`ceil(x)`	round x towards `+Inf`	`typeof(x)`
`ceil(T, x)`	round x towards `+Inf`	T
`trunc(x)`	round x towards zero	`typeof(x)`
`trunc(T, x)`	round x towards zero	T

Conversion function

convert(T,x)

converts x to type T if possible

Functions related to division

Function	Description
`div(x,y)`, x÷y	truncated division; quotient rounded towards zero
`fld(x,y)`	floored division; quotient rounded towards `-Inf`
`cld(x,y)`	ceiling division; quotient rounded towards `+Inf`
`rem(x,y)`	remainder; satisfies `x == div(x,y)*y + rem(x,y)`; sign matches x
`mod(x,y)`	modulus; satisfies `x == fld(x,y)*y + mod(x,y)`; sign matches y

Sign related functions

Function	Description
`abs(x)`	a positive value with the magnitude of x
`abs2(x)`	the squared magnitude of x
`sign(x)`	indicates the sign of x, returning -1, 0, or +1
`signbit(x)`	indicates whether the sign bit is on (true) or off (false)
`copysign(x,y)`	a value with the magnitude of x and the sign of y
`flipsign(x,y)`	a value with the magnitude of x and the sign of x*y

Common math functions

Function	Description
`sqrt(x)`, √x	square root of x
`cbrt(x)`, ∛x	cube root of x
`hypot(x,y)`	hypotenuse of right-angled triangle with other sides of length x and y
`exp(x)`	natural exponential function at x
`expm1(x)`	accurate `exp(x)-1` for x near zero
`ldexp(x,n)`	x*2^n computed efficiently for integer values of n
`log(x)`	natural logarithm of x
`log(b,x)`	base b logarithm of x
`log2(x)`	base 2 logarithm of x
`log10(x)`	base 10 logarithm of x
`log1p(x)`	accurate `log(1+x)` for x near zero
`exponent(x)`	binary exponent of x
`significand(x)`	binary significand (a.k.a. mantissa) of a floating-point number x

Many special functions in package SpecialFunctions

Trig functions (radian args)

```
sin    cos    tan    cot    sec    csc
sinh   cosh   tanh   coth   sech   csch
asin   acos   atan   acot   asec   acsc
asinh  acosh  atanh  acoth  asech  acsch
sinc   cosc
```

Trig functions (degree args)

```
sind   cosd   tand   cotd   secd   cscd
asind  acosd  atand  acotd  asecd  acscd
```

Boolean Data Type and boolean operations

The type **`Bool`** is for variables with values **`true`** or **`false`**

- it uses 8 bits (even though 1 bit would be enough)
- Bool is a subset of Int (true=1, false=0)
- In most respects Bool is the same as Int8

Example:

```
function trueorfalse(b::Bool)
    println(b)
    println(b*1)
    println(b*2)
    println(b*true)
end
trueorfalse(true)
println()
trueorfalse(false)
```

[bool.jl]

Output:

```
true
1
2
true

false
0
0
false
```

Boolean ops:

x and y are of type boolean (expressions)

`!x` - negation

`x && y` - and (short-circuit; only evaluates y if x is true)

`x || y` - or (short-circuit; only evaluates y if x is false)