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



- Fixed-length bit patterns $\Longrightarrow$ range of values is finite.
- 2 sorts: *Signed* and *Unsigned*: machine models for $\mathbb{Z}$ and $\mathbb{N}$, resp.

Unsigned integers `UInt`

- rarely used in scientific computing
- used in low-level programming for binary data and memory addresses
- are written as hexadecimal numbers: with prefix `0x` and digits `0-9a-f`
- `UInt`s are simple binary numbers with n=8, 16, 32, 64 or 128 bits length and a value range of

$$0 \leq x < 2^n$$

```
julia> for t in subtypes(Unsigned) print("$t ") end
UInt128 UInt16 UInt32 UInt64 UInt8

julia> z = UInt64(32); @show z;
z = 0x0000000000000020
julia> x = 0x00fe
0x00fe
julia> @show typeof(x) Int(x) bitstring(x);
typeof(x) = UInt16
Int(x) = 254
bitstring(x) = "0000000011111110"

julia> x = 0x5a
0x5a
julia> @show typeof(x) Int(x) bitstring(x);
typeof(x) = UInt8
Int(x) = 90
bitstring(x) = "01011010"
julia> Char(x)
'Z': ASCII/Unicode U+005A (category Lu: Letter, uppercase)
```

```
julia> for t in subtypes(Signed) print("$t ") end
BigInt Int128 Int16 Int32 Int64 Int8

julia> Int===Int64
true

julia> typemin(Int), typemax(Int)
(-9223372036854775808, 9223372036854775807)
```

- value range of n-bit integer:

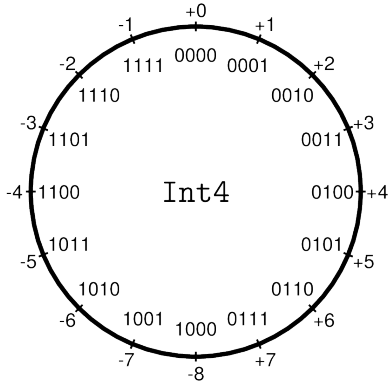
$$-2^{n-1} \leq x < 2^{n-1}$$

Integers

- value range of n-bit integer:

$$-2^{n-1} \leq x < 2^{n-1}$$

- Negative integers: *Two's complement* representation
- $x \Rightarrow -x$ corresponds to: *flip all bits, then add 1*




```
julia> bitstring(5)
"00000000000000000000000000000000000000000000000000000000000000101"
julia> bitstring(-5)
"11111111111111111111111111111111111111111111111111111111111111011"
```

```
julia> x = 2^62 - 10 + 2^62
9223372036854775798

julia> x + 20
-9223372036854775798
```

Integer arithmetic is arithmetic modulo 2^n. There are no overflow errors/warnings.

- The operations `+,-,*` have the usual exact arithmetic of $\mathbb{Z}_{2^n}$.
- Division always creates a floating point number, `40/5 = 8.0`
- Powers $a^n, n \in \mathbb{N}$ are also exact *modulo* 2^n.

```
julia> 0^0, 2^62, 2^63, 2^64
(1, 4611686018427387904, -9223372036854775808, 0)
```

Integer Division

- The functions `div(a,b)`, `rem(a,b)`, and `divrem(a,b)` compute the *quotient* of integer division, the associated *remainder*, or both as a tuple, respectively.
- For `div(a,b)`, the operator form is `a ÷ b` (input: `\div<TAB>`), and for `rem(a,b)`, the operator form is `a % b`.
- By default, division is *rounded toward zero*, resulting in the remainder having the same sign as the dividend a.
- All 3 functions allow another rounding mode as third argument. See `?RoundingMode`.

```
julia> rem(-10, 3)
-1
julia> rem(-10, 3, RoundDown)
2
julia> mod(-10,3)
2
```

- `mod(a,b)` is equivalent to `rem(a,b, RoundDown)`.
- For all rounding modes we have `div(a, b, RoundingMode) * b + rem(a, b, RoundingMode) = a`.

Arbitrary-precision Integer Arithmetic

- If necessary, numerical constants get larger types:

```
julia> z = 10; typeof(z)
Int64
julia> z = 10_000_000_000_000_000_000; typeof(z)
Int128
julia> z = 10_000_000_000_000_000_000_000_000_000_000_000_000_000; typeof(z)
BigInt
```

- The type **BigInt** allocates as much storage as necessary – and is much slower.

```
julia> 3^300      # this is mod 2^64 !
4157753088978724465
julia> BigInt(3)^300
136891479058588375991326027382088315966463695625337436471480190078368997177499076593800206155688941388250484440597994042813512732765695774566001
```

Exponentiation by Squaring

Julia uses for the calculation of integer powers x^n the method of *exponentiation by squaring*. So, x^{23} is calculated with 7 multiplications:

$$x^{23} = x \cdot \left(x \cdot \left(x \cdot (x^2)^2\right)^2\right)^2$$

```
function pow_by_sq(x, n)

  ????

end
```

Recursion, tail recursion and iteration

Example 1: $S(n) = \sum_{i=0}^{n} i = 0 + 1 + \cdots + n$

recursive definition:

$$S(n) = \begin{cases} 0, & \text{if } n = 0 \\ S(n-1) + n, & \text{otherwise} \end{cases}$$

```
function S(n)
    if n == 0 return 0 end
    return S(n-1) + n
end
```

```
S(3)
3 + S(2)
3 + 2 + S(1)
3 + 2 + 1 + S(0)
3 + 2 + 1 + 0
3 + 2 + 1
3 + 3
6
```

recursion is time- and memory-consuming

make it *tail-recursive* using accumulator variable

```
S(n) = S_a(0, n)

function S_a(a, n)
    if n == 0 return a end
    return S_a(a+n, n-1)
end
```

```
S(3) = S_a(0, 3)
       S_a(3, 2)
       S_a(5, 1)
       S_a(6, 0)
       6
```

Recursion, tail recursion and iteration

tail-recursive using accumulator variable

```
function S_a(a, n)
    if n == 0 return a end
    return S_a(a+n, n-1)
end

S(n) = S_a(0, n)
```

iterative variant

```
function S(n)
    a= 0; k = n   #  for loop invariant
    while k>0
        a = a + k       # or:   a += k
        k = k - 1       # or:   k -= 1
    end
    return a
end
```

loop invariant $I = a + \sum_{i=0}^{k} i$

- The *loop invariant* is unchanged after each loop iteration.
- Start values: $a = 0, k = n \quad \Longrightarrow I = S(n)$
- End values: $k = 0 \quad \Longrightarrow a = S(n)$
- helpful in non-trivial examples

Exponentiation by Squaring

Julia uses for the calculation of integer powers x^n the method of *exponentiation by squaring*. So, x^{23} is calculated with 7 multiplications:

$$x^{23} = x \cdot \left(x \cdot \left(x \cdot (x^2)^2\right)^2\right)^2$$

Complexity:

- Naive method: $O(n)$ multiplications for x^n.
- Exponentiation by squaring: $O(\log n)$ multiplications.

```
function pow_by_sq(x, n)

  ????

end
```

Exponentiation by Squaring

- Recursive understanding of the algorithm
  - either write power as square: $x^{22} = (x^{11})^2$
  - or split a factor: $x^{23} = x \cdot x^{22}$

$$p(x,n) = \begin{cases} 1, & \text{if } n = 0 \\ [p(x,n/2)]^2, & \text{if } n \text{ even} \\ x p(x,n-1), & \text{otherwise} \end{cases}$$

---

add accumulator

$$p(x,n) = p_a(1,x,n)$$

$$p_a(a,x,n) = \begin{cases} a & \text{if } n = 0 \\ p_a(a,x,n/2)^2 & \text{if } n \text{ even} \\ p_a(ax,x,n-1) & \text{otherwise} \end{cases}$$

make tail-recursive

$$p_a(a,x,n) = \begin{cases} a & \text{if } n = 0 \\ p_a(a,x^2,n/2) & \text{if } n \text{ even} \\ p_a(ax,x,n-1) & \text{otherwise} \end{cases}$$

Exponentiation by Squaring

$$p(x, n) = p_a(1, x, n)$$

$$p_a(a, x, n) = \begin{cases} a & \text{if } n = 0 \\ p_a(a, x^2, n/2) & \text{if } n \text{ even} \\ p_a(ax, x, n-1) & \text{otherwise} \end{cases}$$

```
function pow(x, n)
    a = 1
    while n ≥ 1
        if iseven(n)
            x = x * x
            n = n ÷ 2
        else
            a = a * x
            n = n - 1
        end
    end
    return a
end
```

Exponentiation by Squaring

```
function pow(x, n)
    u = x; k = n  # only for loop inv.
    a = 1
    while k ≥ 1
        if iseven(k)
            u = u * u
            k = k ÷ 2
        else
            a = a * u
            k = k - 1
        end
    end
    return a
end
```

Correctness proof by loop invariant:

$I = au^k = x^n$

- The *loop invariant* is unchanged after each loop iteration.
- Start values:
  $a = 1, u = x, k = n \quad \Longrightarrow I = x^n$
- End values: $k = 0 \quad \Longrightarrow a = x^n$