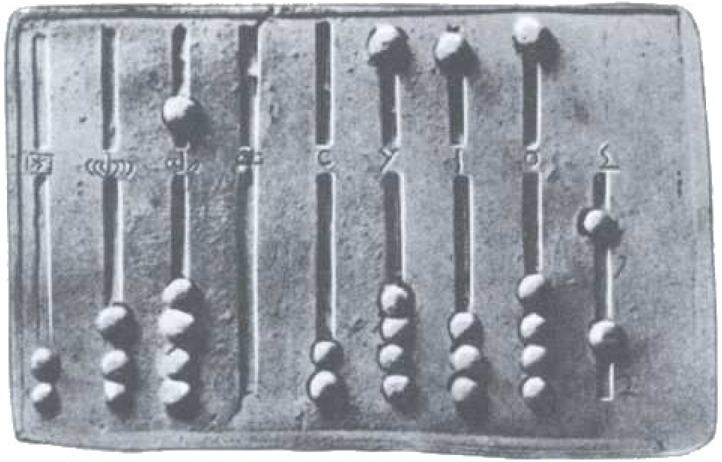
Ancient Math / Modern C++

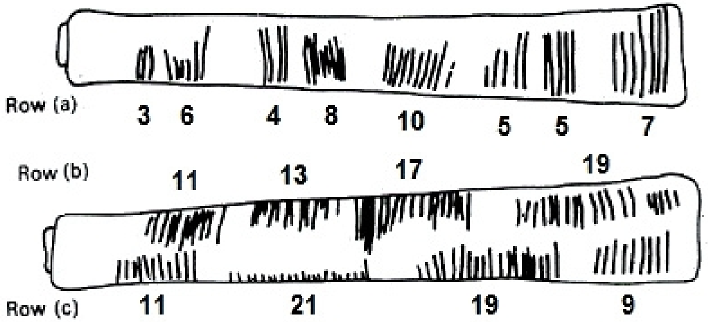
Petter Holmberg – StockholmCpp XXIII – June 2019



Natural numbers

Ishango bone (D.R. Of Congo, c. 18.000 BC – 20.000 BC)

Natural numbers



Ishango bone (D.R. Of Congo, c. 18.000 BC – 20.000 BC)

Natural numbers, prehistoric definition

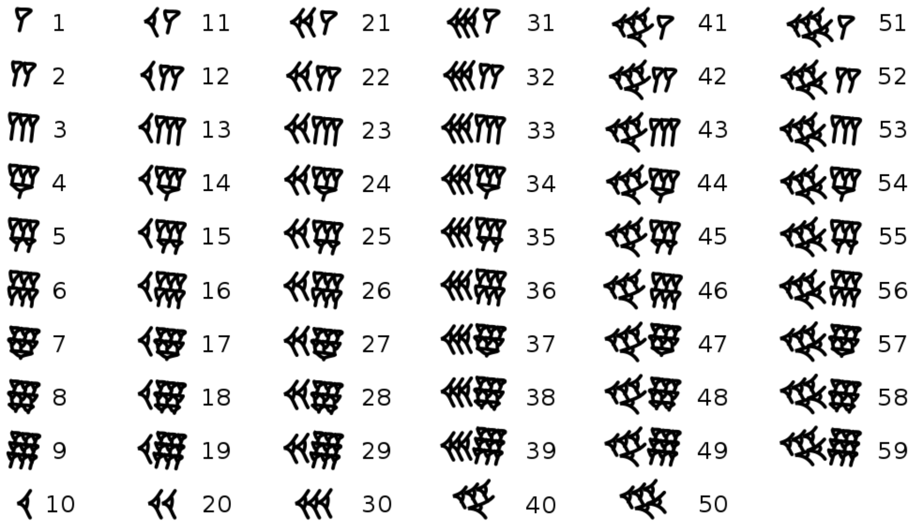
Defined as: $\left(\cdot, \cdot\cdot, \cdot\cdot\cdot, ...\right)$

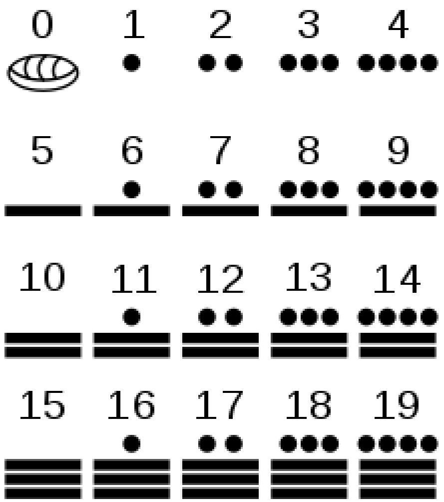
Written as (today): $1, 2, 3, ...$ $\mathbb{N}$

Natural numbers as a string of dots

```
#include <string>

struct PaleolithicNatural
{
    std::string k{“.”};

    PaleolithicNatural() = default;

    // Constructor for the Stone Age person
    PaleolithicNatural(std::string const& x)
       : k(x.length(), '.')
    {}
};
```

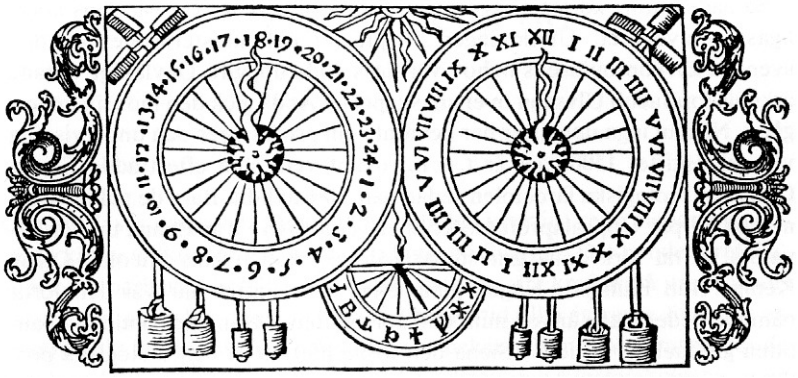
A better representation





Left: Babylonian numerals (c. 2.000 BC)
Right: Mayan numerals (c. 1000 BC)

A better representation



Astronomical clock (Uppsala Cathedral, 1506 - 1702)

Highly composite numbers

$$\frac{24}{1}=24, \frac{24}{2}=12, \frac{24}{3}=8, \frac{24}{4}=6$$

$$\frac{24}{6}=4, \frac{24}{8}=3, \frac{24}{12}=2, \frac{24}{24}=1$$

$$\frac{60}{1}=60, \frac{60}{2}=30, \frac{60}{3}=20, \frac{60}{4}=15, \frac{60}{5}=12, \frac{60}{6}=10$$

$$\frac{60}{10}=6, \frac{60}{12}=5, \frac{60}{15}=4, \frac{60}{20}=3, \frac{60}{30}=2, \frac{60}{60}=1$$

Rational numbers

Defined as: $\left(p,q\right),\left\{p,q\in\mathbb{Z}\right\},q\neq 0$

Written as: $\dfrac{p}{q}$ $\mathbb{Q}$

Rational numbers is the most important example of a ***Field*** *in mathematics!*

Rational numbers

```
struct Rational
{
    int p = 0;
    int q = 1;

    Rational() = default;

    Rational(int x)
        : p{x}, q{1}
    {}

    Rational(int x0, int x1)
        //[[expects: x1 != 0]] // In anticipation of C++20 contracts
        : p{x0}, q{x1}
    {}
};
```

Relational operators

```
operator==
operator!=
operator<
operator>=
operator>
operator<=
```

Rational numbers, equality

$$\frac{p_0}{q_0} = \frac{p_1}{q_1} \Leftrightarrow p_0 q_1 = p_1 q_0$$

Rational numbers, equality

$$\frac{p_0}{q_0} = \frac{p_1}{q_1} \Leftrightarrow p_0 q_1 = p_1 q_0$$

- Enables equational reasoning
- Enables reasoning about laws of arithmetic
- Enables linear search (`std::find`)

Rational numbers, equality

```
constexpr auto operator==(Rational x0, Rational x1) -> bool
{
    return x0.p * x1.q == x1.p * x0.q;
}

constexpr auto operator!=(Rational x0, Rational x1) -> bool
    //[[ensures ret: !(ret == (x0 == x1))]] // Complement
{
    return !(x0 == x1);
}
```

Dijkstra's nomenclature

For the pronunciation of the relations I propose the following:

for "$x = y$" read "x equals y"
for "$x \neq y$" read "x differs from y"
for "$x > y$" read "x exceeds y"
for "$x \geq y$" read "x is at least y"
for "$x < y$" read "x is less than y"
for "$x \leq y$" read "x is at most y" .

EWD768 - "Largely on Nomenclature" - http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD768.PDF

Equivalence relation

$x \sim x$	*(reflexive)*
$x \sim y \quad \Leftrightarrow \quad y \sim x$	*(symmetric)*
$x \sim y \quad \wedge \quad y \sim z \quad \Rightarrow \quad x \sim z$	*(transitive)*

Assert that `operator==` *is an equivalence relation!*

Rational numbers, total ordering

$$\frac{p_0}{q_0} < \frac{p_1}{q_1} \Leftrightarrow p_0 q_1 < p_1 q_0$$

Rational numbers, total ordering

$$\frac{p_0}{q_0} < \frac{p_1}{q_1} \Leftrightarrow p_0 q_1 < p_1 q_0$$

- Enables fundamental mathematical algorithms (*abs*, *gcd* etc.)
- Enables sorting (used in `std::set, std::map` etc.)
- Enables binary search (`std::lower_bound` etc.)

Rational numbers, total ordering

```
constexpr auto operator<(Rational x0, Rational x1) -> bool
{
    return x0.p * x1.q < x1.p * x0.q;
}

constexpr auto operator>=(Rational x0, Rational x1) -> bool
    //[[ensures ret: !(ret == (x0 < x1))]] // Complement
{ return !(x0 < x1); }

constexpr auto operator>(Rational x0, T Rational x1) -> bool
    //[[ensures ret: !(ret == (x0 <= x1))]] // Converse
{ return x1 < x0; }

constexpr auto operator<=(Rational x0, Rational x1) -> bool
    //[[ensures ret: !(ret == (x0 > x1))]] // Complement of converse
{ return !(x1 < x0); }
```

Weak ordering

$x \prec y \quad \wedge \quad y \prec z \quad \Rightarrow \quad x \prec z$ *(transitive)*

Exactly one of the below holds:

(I) $(x \prec y)$

(II) $(y \prec x)$

(III) $(\boldsymbol{x} \sim \boldsymbol{y})$ *(weak trichotomy)*

Total ordering

$x < y \quad \wedge \quad y < z \quad \Rightarrow \quad x < z$ *(transitive)*

Exactly one of the below holds:

(I) $(x < y)$

(II) $(y < x)$

(III) $(\boldsymbol{x = y})$ *(trichotomy)*

Arithmetic operators

```
operator+
operator*
operator-
operator/
```

What should this print?

```
print(“Sweden” + “Cpp”)
```

What should this print?

```
print(“Sweden” + “Cpp”)

>>> SwedenCpp
```

What should this print?

```
print(“Sweden” * 3)
```

What should this print?

```
print(“Sweden” * 3)

>>> SwedenSwedenSweden
```

What should this print?

```
print(“Sweden” * “Cpp”)
```

What should this print?

```
print(“Sweden” * “Cpp”)

>>> SwedenCpp
```

Which one is correct?

```
print(“Sweden” + “Cpp”)

print(“Sweden” * “Cpp”)
```

Which one is correct?

~~`print("Sweden" + "Cpp")`~~

```
print("Sweden" * "Cpp")
```

C++ `std::string` got it wrong!

So did Java `String`…
...and Python `string`…
...and JavaScript…
...and Go…
…and Swift…
...

Commutativity of addition

$x + y = y + x$

- *Holds for* $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$
- *Holds for polynomials*
- *Holds for vectors and matrices*
- ...

Will these print the same string?

```
print(“Sweden” + “Cpp”)

print(“Cpp” + “Sweden”)
```

Commutativity of multiplication

$x \cdot y = y \cdot x$

- *Holds for $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$*
- *Holds for polynomials (regular and scalar multiplication)*
- *Holds for vectors (scalar multiplication)*
- ***Does not hold for matrices or quaternions***

String concatenation

```
print(“Sweden” * 3)

print(“Sweden” * “Cpp”)

print(“Sweden” “Cpp”)
```

Associativity

$$(x + y) + z = x + (y + z)$$

$$(x \cdot y) \cdot z = x \cdot (y \cdot z)$$

Associativity

“This “ “is “ “ a” “sentence ”

“consisting “ “of “ “many “ “ words.”

Associativity

(“This “ “is “ “ a” “sentence ”)

(“consisting “ “of “ “many “ “ words.”)

Semigroups

$\{\ T,\ \circ\ \}$ *where:*

$(x \circ y) \circ z = x \circ (y \circ z)$ *(associative)*

`{ int, + }` `{ int, * }`

`{ int, std::min }`

`{ bool, & }` `{ bool, | }` `{ bool, ^ }`

`{ std::string, + }` { ~~`double, +`~~ } { ~~`double, *`~~ }

Monoids

*{ T, ∘, **e** } where:*

$(x \circ y) \circ z = x \circ (y \circ z)$ *(associative)*

$\boldsymbol{x \circ e = e \circ x = x}$ ***(identity element)***

```
{ int, +, 0 }            { int, *, 1 }

{ int, std::min, std::numeric_limits<int>::max() }

{ bool, &, true }        { bool, |, false }       { bool, ^, false }

{ std::string, +, "" }
```

~~{ double, +, 0.0 }~~ ~~{ double, *, 1.0 }~~

Distributivity

$$x \cdot (y + z) = x \cdot y + x \cdot z$$

$$(y + z) \cdot x = y \cdot x + z \cdot x$$

Semirings

$\{ T, +, 0, \cdot, 1 \}$ *where:*

$\{ T, +, 0 \}$ *(commutative monoid)*

$\{ T, \cdot, 1\}$ *(monoid)*

$0 \neq 1$

$x \cdot (y + z) = x \cdot y + x \cdot z$ *(distributivite)*

$(y + z) \cdot x = y \cdot x + z \cdot x$ *(distributivite)*

`{ int, +, 0, *, 1 }`   `{ bool, |, false, &, true }`

`{ int, std::min, std::numeric_limits<int>::max(), +, 0 }`

Rational numbers, addition

$$\frac{p_0}{q_0} + \frac{p_1}{q_1} = \frac{p_0 q_1 + q_0 p_1}{q_0 q_1}$$

Rational numbers, additive identity

$$e = \frac{0}{1}$$

$$\frac{p_0}{q_0} + \frac{0}{1} = \frac{p_0 \cdot 1 + q_0 \cdot 0}{q_0 \cdot 1} = \frac{p_0}{q_0}$$

$$\frac{0}{1} + \frac{p_1}{q_1} = \frac{0 \cdot q_1 + 1 \cdot p_1}{1 \cdot q_1} = \frac{p_1}{q_1}$$

Rational numbers, addition

```
constexpr auto
operator+(Rational x0, Rational x1) -> Rational
{
    return {x0.p * x1.q + x0.q * x1.p, x0.q * x1.q};
}
```

Rational numbers, multiplication

$$\frac{p_0}{q_0} \cdot \frac{p_1}{q_1} = \frac{p_0 p_1}{q_0 q_1}$$

Rational numbers, multiplicative identity

$$e = \frac{1}{1}$$

$$\frac{p_0}{q_0} \cdot \frac{1}{1} = \frac{p_0 \cdot 1}{q_0 \cdot 1} = \frac{p_0}{q_0}$$

$$\frac{1}{1} \cdot \frac{p_1}{q_1} = \frac{1 \cdot p_1}{1 \cdot q_1} = \frac{p_1}{q_1}$$

Rational numbers, multiplication

```
constexpr auto
operator*(Rational x0, Rational x1) -> Rational
{
    return {x0.p * x1.p, x0.q * x1.q};
}
```

Rational numbers, subtraction

$$\frac{p_0}{q_0} - \frac{p_1}{q_1} = \frac{p_0 q_1 - q_0 p_1}{q_0 q_1}$$

Groups

$\{ T, \circ, e \}$ where:

$(x \circ y) \circ z = x \circ (y \circ z)$	*(associative)*
$x \circ e = e \circ x = x$	*(identity element)*
$\boldsymbol{x \circ x^{-1} = x^{-1} \circ x = e}$	***(cancellation)***

{ int, +, 0 }   ~~{ int, *, 1 }~~

~~{ int, std::min, std::numeric_limits<int>::max() }~~

~~{ bool, &, true }~~   ~~{ bool, |, false }~~   ~~{ bool, ^, false }~~

~~{ std::string, +, “” }~~   ~~{ double, +, 0.0 }~~   ~~{ double, *, 1.0 }~~

Additive groups enable -

{ T, +, 0 } where:

$\boldsymbol{x + y = y + x}$	***(commutative)***
$(x + y) + z = x + (y + z)$	*(associative)*
$x + 0 = 0 + x = x$	*(identity element)*
$\boldsymbol{x + (-x) = -x + x = 0}$	***(cancellation)***
$\boldsymbol{x - y = x + (-y)}$	***(subtraction)***

```
T operator-(T const& x) {
    return additive_inverse(x);
}

T operator-(T const& x, T const& y) {
    return x + (-y);
}
```

Rational numbers, subtraction

$$-\left(\frac{p}{q}\right)=\frac{-p}{q}$$

$$\frac{p_0}{q_0}-\frac{p_1}{q_1}=\frac{p_0 q_1 - q_0 p_1}{q_0 q_1}=\frac{p_0}{q_0}+\left(-\left(\frac{p_1}{q_1}\right)\right)$$

Rational numbers, subtraction

```
constexpr auto
operator-(Rational x) -> Rational
{
    return {-x.p, x.q};
}
```

```
constexpr auto
operator-(Rational x0, Rational x1) -> Rational
{
    return x0 + (-x1);
}
```

Rational numbers, generalizing subtraction

```
constexpr auto
operator-(Rational x) -> Rational
{
    return {-x.p, x.q};
}
```

```
template <typename G>
constexpr auto
operator-(G const& x0, G const& x1) -> G
{
    return x0 + (-x1);
}
```

Rational numbers, generalizing subtraction

```
constexpr auto
operator-(Rational x) -> Rational
{
    return {-x.p, x.q};
}

#define AdditiveGroup typename // In anticipation of C++20 concepts

template <AdditiveGroup G>
constexpr auto
operator-(G const& x0, G const& x1) -> G
{
    return x0 + (-x1);
}
```

Rings

{ T, +, 0, ·, 1 } where:

{ T, +, 0, ·, 1 } *(semiring)*

{ T, +, 0 } ***(additive group)***

```
{ int, +, 0, *, 1, - }

{ Rational, +, {0, 1} , *, {1, 1}, - }
```

Integral domains

$\{ T, +, 0, \cdot, 1 \}$ *where:*

$\{ T, +, 0, \cdot, 1 \}$ *(ring)*

$\boldsymbol{x \cdot y = y \cdot x}$ ***(commutative)***

$\boldsymbol{x \cdot y = 0 \Rightarrow (x = 0 \lor y = 0)}$ ***(no zero divisors)***

```
{ int, +, 0, *, 1, - }

{ Rational, +, {0, 1} , *, {1, 1}, - }
```

Rational numbers, division

$$\frac{\dfrac{p_0}{q_0}}{\dfrac{p_1}{q_1}} = \frac{p_0\, q_1}{q_0\, p_1}$$

Undefined if: $p_1 = 0$

Rational numbers, division

```
constexpr auto
operator/(Rational x0, Rational x1) -> Rational
    //[[expects: x1 != Rational{0, 1}]]
{
    return {x0.p * x1.q, x0.q * x1.p};
}
```

Fields

{ T, +, 0, ·, 1 } where:

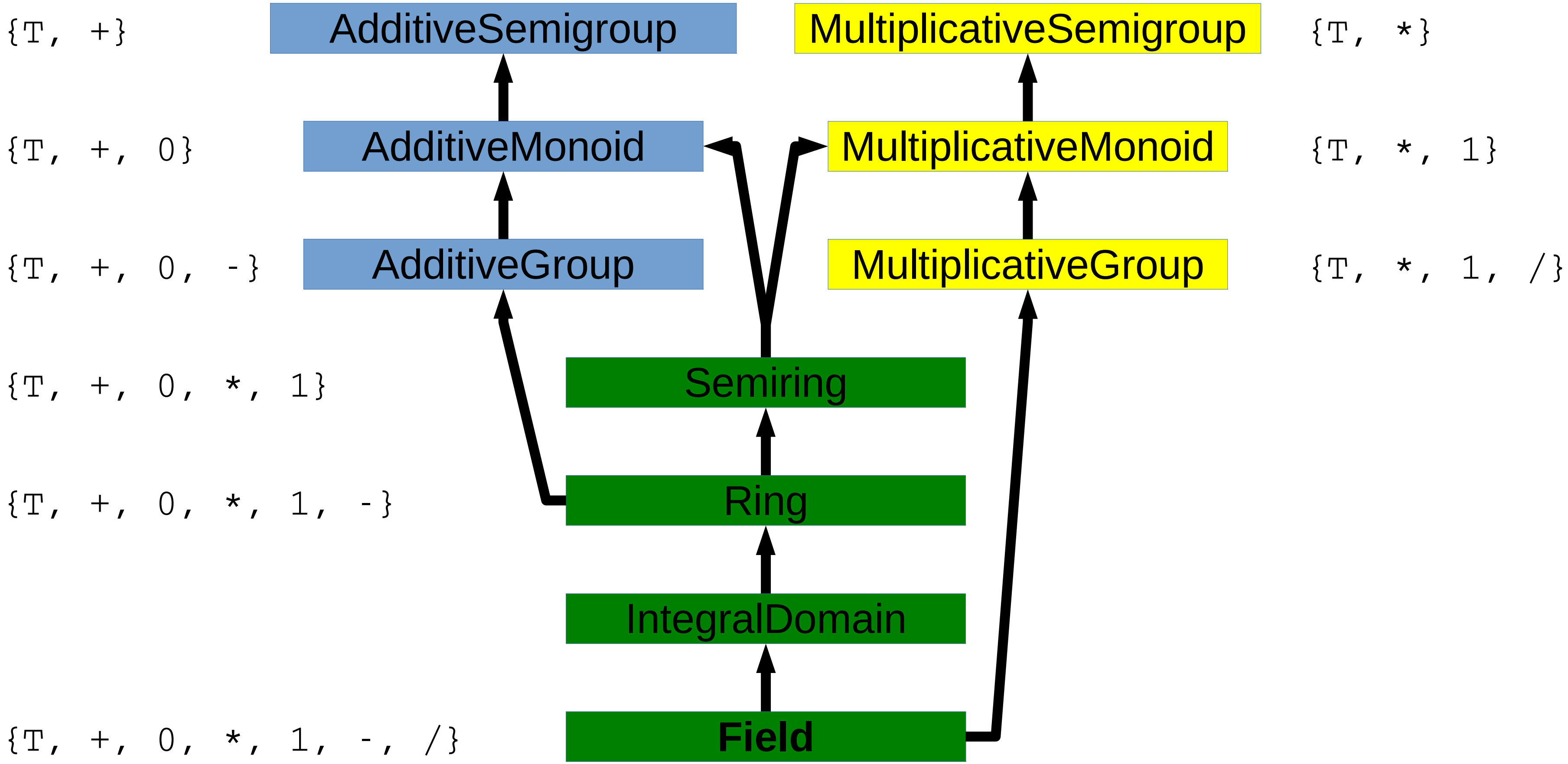
{ T, +, 0, ·, 1 } *(integral domain)*

$(x \neq 0) \Rightarrow x \cdot x^{-1} = x^{-1} \cdot x = 1$ ***(cancellation)***

$(y \neq 0) \Rightarrow x / y = x \cdot y^{-1}$ ***(division)***

```
T operator/(T const& x, T const& y)
    //[[expects: y != T{0}]]
{
    return x * multiplicative_inverse(y);
}
```

Algebraic concepts



Generalizing rational numbers

```
struct Rational
{
    int p = 0;
    int q = 1;

    Rational() = default;

    Rational(int x)
        : p{x}, q{1}
    {}

    Rational(int x0, int x1)
        //[[expects: x0 != 0]]
        : p{x0}, q{x1}
    {}
};
```

Generalizing rational numbers

```
template <IntegralDomain I>
struct Rational
{
    I p = 0;
    I q = 1;

    Rational() = default;

    Rational(I const& x)
        : p{x}, q{1}
    {}

    Rational(I const& x0, I const& x1)
        //[[expects: x1 != 0]]
        : p{x0}, q{x1}
    {}
};
```

Generalizing rational numbers

```
template <IntegralDomain I>
struct Rational
{
    I p = 0;
    I q = 1;

    Rational() = default;

    Rational(I const& x)
        : p{x}, q{1}
    {}

    Rational(I const& x0, I const& x1)
        //[[expects: x1 != 0]]
        : p{x0}, q{x1}
    {}
};
```

Generalizing rational numbers

```
template <IntegralDomain I>
struct Rational
{
    I p = Zero<I>;
    I q = One<I>;

    Rational() = default;

    Rational(I const& x)
        : p{x}, q{One<I>}
    {}

    Rational(I const& x0, I const& x1)
        //[[expects: x1 != Zero<I>]]
        : p{x0}, q{x1}
    {}
};
```

Type function, additive identity

```
template <AdditiveSemigroup S>
struct zero_tf
{
    static S const value = S{0}; // Default, other types can specialize
};

template <AdditiveSemigroup S>
S const Zero = zero_tf<S>::value; // Variable template (C++14)
```

Type function, multiplicative identity

```
template <MultiplicativeSemigroup S>
struct one_tf
{
    static S const value = S{1}; // Default, other types can specialize
};

template <MultiplicativeSemigroup S>
S const One = one_tf<S>::value; // Variable template (C++14)
```

Rational numbers, Zero and One

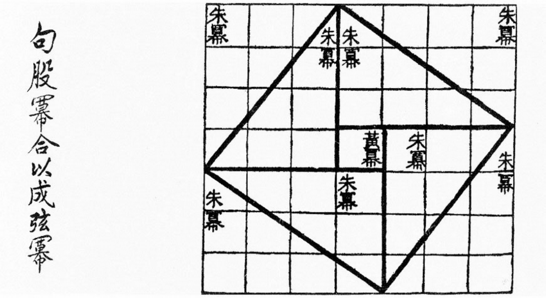
```
template <IntegralDomain I> // Specialization for Zero<Rational<I>>
struct zero_tf<Rational<I>> { static Rational<I> const value; };

template <IntegralDomain I> // Definition of Zero<Rational<I>>
Rational<I> const zero_tf<Rational<I>>::value =
Rational<I>{}; // Default constructor returns {Zero<I>, One<I>}



template <IntegralDomain I> // Specialization for One<Rational<I>>
struct one_tf<Rational<I>> { static Rational<I> const value; };

template <IntegralDomain I> // Definition of One<Rational<I>>
Rational<I> const one_tf<Rational<I>>::value =
Rational<I>{One<I>, One<I>};
```

Building on rational numbers

https://github.com/petter-holmberg/talks/blob/master/math.cpp

Geometry

Vector2D (Vector Space over the Field Rational<I>)
Point2D (Affine Space over Vector2D)
Line2D (using the Field Rational<I>)



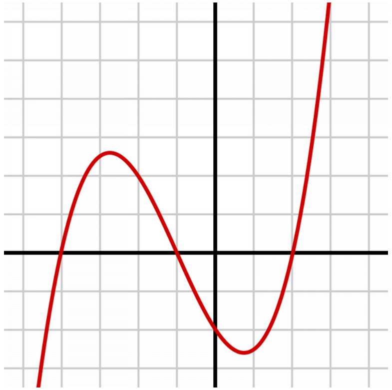
Zhoubi Suanjing (China, 500–200 BC)

Building on rational numbers

https://github.com/petter-holmberg/talks/blob/master/math.cpp

Algebra, Calculus

`Polynomial` (Ring and Vector Space over any Ring)



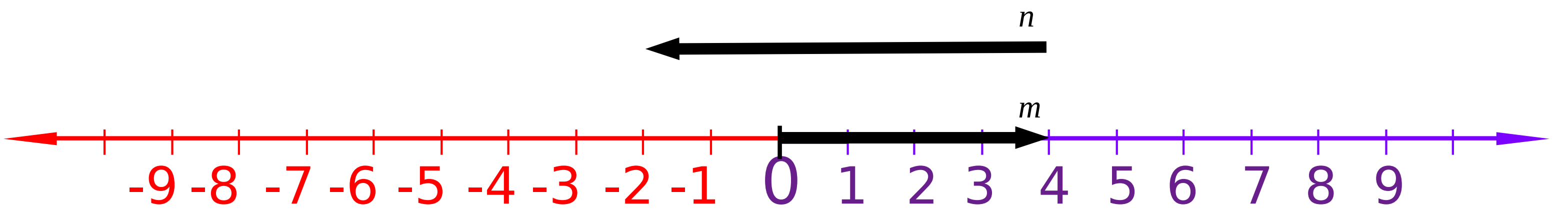
$$y = \frac{1}{4}x^3 + \frac{3}{4}x^2 - \frac{3}{2}x - 2$$

Integers

Defined as: $$(m,n),\{m,n\in\mathbb{N}\}$$

Written as: $m \backslash n$ $\mathbb{Z}$

Geometrical interpretation:



Integers

```
template <Semiring S>
struct Integer
{
    S m = Zero<S>;
    S n = Zero<S>;

    Integer() = default;

    Integer(S const& x)
        : m{x}, n{Zero<S>}
    {}

    Integer(S const& x0, S const& x1)
        : m{x0}, n{x1}
    {}
};
```

Integers, total ordering

$$m_0 \setminus n_0 = m_1 \setminus n_1 \Leftrightarrow m_0 + n_1 = m_1 + n_0$$

$$m_0 \setminus n_0 < m_1 \setminus n_1 \Leftrightarrow m_0 + n_1 < m_1 + n_0$$

Integers, total ordering

```
template <Semiring S>
constexpr auto
operator==(Integer<S> const& x0, Integer<S> const& x1) -> bool
{
    return x0.m + x1.n == x1.m + x0.n;
}

template <Semiring S>
constexpr auto
operator<(Integer<S> const& x0, Integer<S> const& x1) -> bool
{
    return x0.m + x1.n < x1.m + x0.n;
}
```

Integers, semiring arithmetic

$$m_0 \backslash n_0 + m_1 \backslash n_1 = m_0 + m_1 \backslash n_0 + n_1$$

$$m_0 \backslash n_0 \cdot m_1 \backslash n_1 = m_0 m_1 + n_0 n_1 \backslash m_0 n_1 + n_0 m_1$$

$$-(m \backslash n) = n \backslash m$$

Integers, semiring arithmetic

```
template <Semiring S>
constexpr auto
operator+(Integer<S> const& x0, Integer<S> const& x1) -> Integer<S> {
    return {x0.m + x1.m, x0.n + x1.n};
}

template <Semiring S>
constexpr auto
operator*(Integer<S> const& x0, Integer<S> const& x1) -> Integer<S> {
    return {x0.m * x1.m + x0.n * x1.n, x0.m * x1.n + x0.n * x1.m};
}

template <Semiring S>
constexpr auto
operator-(Integer<S> const& x) -> Integer<S> {
    return {x.n, x.m};
}
```

Integers, type functions

```
template <Semiring S>
struct zero_tf<Integer<S>> { static Integer<S> const value; };
template <Semiring S>
Integer<S> const zero_tf<Integer<S>>::value = Integer<S>{};



template <Semiring S>
struct one_tf<Integer<S>> { static Integer<S> const value; };
template <Semiring S>
Integer<S> const one_tf<Integer<S>>::value = Integer<S>{One<S>};
```

Natural numbers as a string of dots

```
#include <string>

struct PaleolithicNatural
{
    std::string k;

    PaleolithicNatural() = default;

    // Constructor for the Stone Age person
    PaleolithicNatural(std::string const& x)
       : k(x.length(), '.')
    {}

    // Constructor for the Renissance person
    PaleolithicNatural(unsigned x)
       : k(x, '.')
    {}
};
```

Natural numbers, total ordering

```
auto
operator==(PaleolithicNatural const& x0, PaleolithicNatural const& x1) ->
bool
{
    return x0.k.length() == x1.k.length();
}

auto
operator<(PaleolithicNatural const& x0, PaleolithicNatural const& x1) ->
bool
{
    return x0.k.length() < x1.k.length();
}
```

Natural numbers, addition

$$" . . . " \quad " . . . . . " \ = \ " . . . . "$$

Natural numbers, addition

```
auto
operator+(PaleolithicNatural const& x0, PaleolithicNatural const& x1) ->
PaleolithicNatural
{
    return x0.k + x1.k; // Commutative!
}
```

Commutativity

$$"\ldots" \; "\ldots\ldots" = "\ldots\ldots\ldots"$$

$$"\ldots\ldots" \; "\ldots" = "\ldots\ldots\ldots"$$

Natural numbers, multiplication

” . . .”   “ . . .”   “ . . .”

“ . . .”   “ . . .”   “ . . .”

“ . . .”   “ . . .”   “ . . .”

“ . . .”   “ . . .”   “ . . .”

Natural numbers, naïve multiplication

```
auto
operator*(PaleolithicNatural const& x0, PaleolithicNatural const& x1) ->
PaleolithicNatural
{
    PaleolithicNatural y;
    for (size_t i = 0; i != x0.k.length(); ++i)
    {
        y = y + x1;
    }
    return y;
}
```

Natural numbers, type functions

```
template <>
struct zero_tf<PaelolithicNatural>
{
    static PaleolithicNatural const value;
};
PaleolithicNatural const zero_tf<PaleolithicNatural>::value{""};

template <>
struct one_tf<PaelolithicNatural>
{
    static PaleolithicNatural const value;
};
PaleolithicNatural const one_tf<PaleolithicNatural>::value{"."};
```

Is this good enough?

```
auto
operator*(PaleolithicNatural const& x0, PaleolithicNatural const& x1) ->
PaleolithicNatural
{
    PaleolithicNatural y;
    for (size_t i = 0; i != x0.k.length(); ++i)
    {
        y = y + x1;
    }
    return y;
}
```

$$12 \cdot 3$$

" . . . " " . . . " " . . . "

" . . . " " . . . " " . . . "

" . . . " " . . . " " . . . "

" . . . " " . . . " " . . . "

$$12 \cdot 3 = (6 \cdot 3) + (6 \cdot 3)$$

(" . . . "   " . . . "   " . . . "
" . . . "   " . . . "   " . . . ")
(" . . . "   " . . . "   " . . . "
" . . . "   " . . . "   " . . . ")

$$12 \cdot 3 = (6 \cdot 3) \cdot 2$$

```
("..." "..." "..."
 "..." "..." "...") * 2
```

$$12 \cdot 3 = (6 \cdot 3) \cdot 2$$

("...""..." "..."
"..." "..." "...") ".................."

$13 \cdot 3$

" . . . "   " . . . "   " . . . "

" . . . "   " . . . "   " . . . "

" . . . "   " . . . "   " . . . "

" . . . "   " . . . "   " . . . "   " . . . "

$$13 \cdot 3 = (6 \cdot 3) + (6 \cdot 3) + 3$$

(" . . . "   " . . . "   " . . . "

" . . . "   " . . . "   " . . . ")

(" . . . "   " . . . "   " . . . "

" . . . "   " . . . "   " . . . ")   " . . . "

$$13 \cdot 3 = (6 \cdot 3) \cdot 2 + 3$$

(″ . . . ″   “ . . . ″   “ . . . ″
  “ . . . ″   “ . . . ″   “ . . . ″)
(“ . . . . ″)   “ . . . ″

$$13 \cdot 3 = (((1 \cdot 3) \cdot 2 + 3) \cdot 2) \cdot 2 + 3$$

(" . . . ")

(" . . . ")   " . . . "

(" . . . . ")

(" . . . . ")   " . . . "

Egyptian multiplication

Rhind mathematical papyrus (Thebes, c. 1550 BC)

Egyptian multiplication

Pyramid of Khafre and Sphinx (Giza, c. 2532 - 2570 BC)

Natural numbers, egyptian multiplication

```
auto
multiply(PaleolithicNatural x0, PaleolithicNatural x1) ->
PaleolithicNatural
{
    if (x0 == Zero<PaleolithicNatural>) return Zero<PaleolithicNatural>;
    if (x0 == One<PaleolithicNatural>) return x1;
    auto y = multiply(half(x0), x1 + x1); // Done if x0 is even
    if (is_odd(x0)) y = y + x1;
    return y;
}
```

Natural numbers, egyptian multiplication

```
auto
multiply(PaleolithicNatural x0, PaleolithicNatural x1) ->
PaleolithicNatural
{
    if (x0 == Zero<PaleolithicNatural>) return Zero<PaleolithicNatural>;
    if (x0 == One<PaleolithicNatural>) return x1;
    auto y = multiply(half(x0), x1 + x1); // Done if x0 is even
    if (is_odd(x0)) y = y + x1;
    return y;
}
```

Natural numbers, half, is_odd

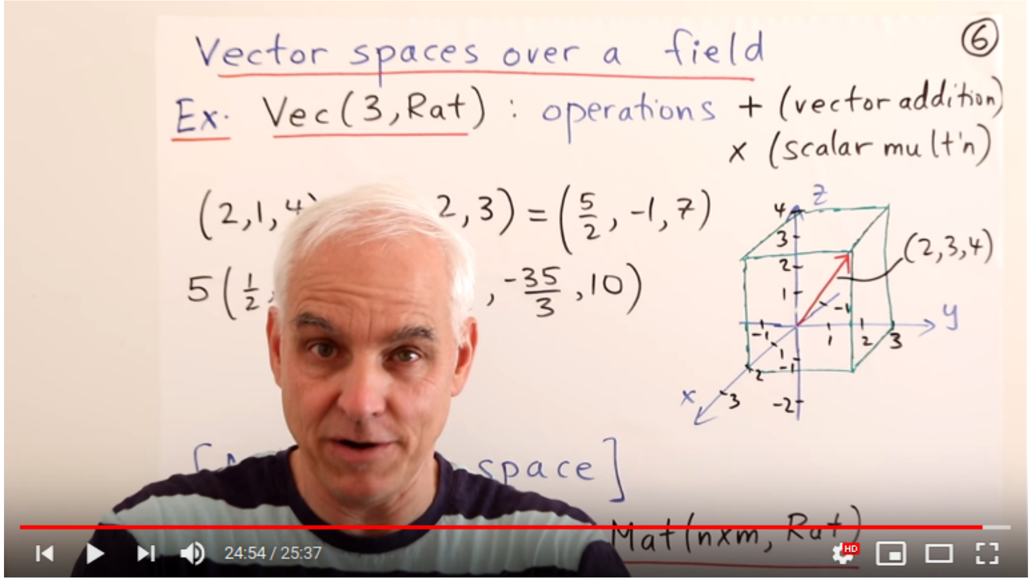
```
auto half(PaleolithicNatural const& x) -> PaleolithicNatural
{
    return PaleolithicNatural(x.k.length() >> 1);
}



auto is_odd(PaleolithicNatural const& x) -> bool
{
    return x.k.length() & 1;
}
```

Challenge: Optimize this algorithm!

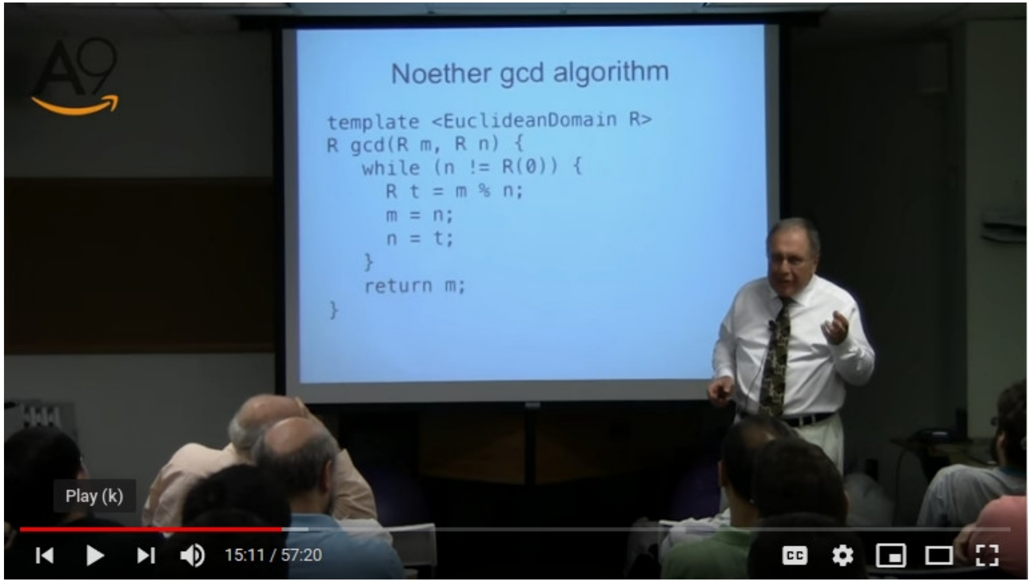
```
auto
multiply(PaleolithicNatural x0, PaleolithicNatural x1) ->
PaleolithicNatural
{
    if (x0 == Zero<PaleolithicNatural>) return Zero<NeolithicNatural>;
    if (x0 == One<PaleolithicNatural>) return x1;
    auto y = multiply(half(x0), x1 + x1); // Done if x0 is even
    if (is_odd(x0)) y = y + x1;
    return y;
}
```

- Convert from recursive to iterative
- Optimize the loop
- Think about how to generalize it
- What are the algebraic concept requirements?

Inspiration for this talk

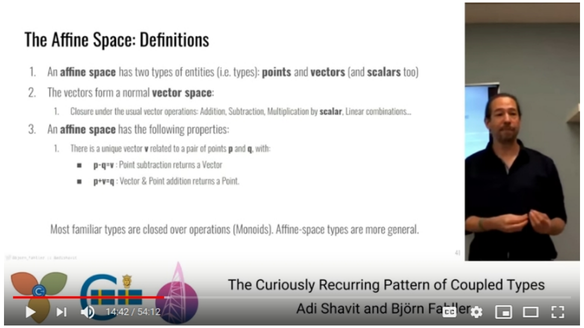


Norman J. Wildberger – Math Foundations

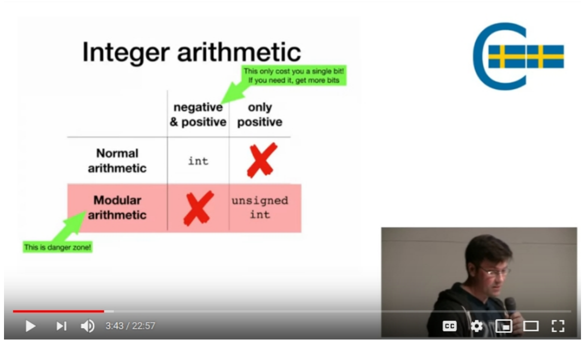


Alexander A. Stepanov – Four Mathematical Journeys

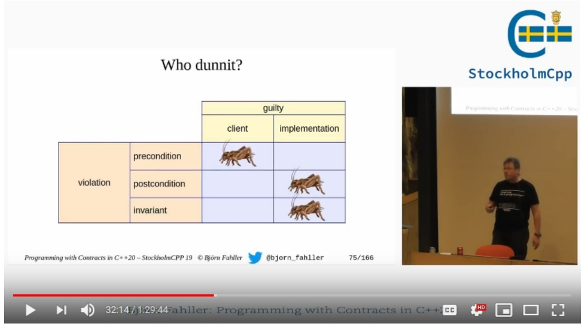
More inspiration for this talk!



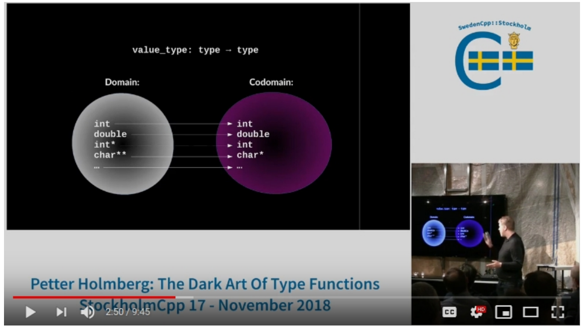
Adi Shavit & Björn Fahller – The Curiously Recurring Pattern of Coupled Types



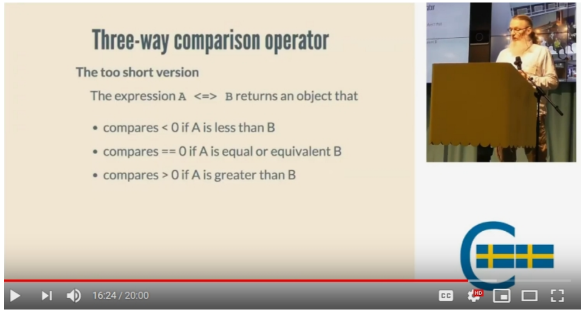
Arvid Norberg – Integers in C++



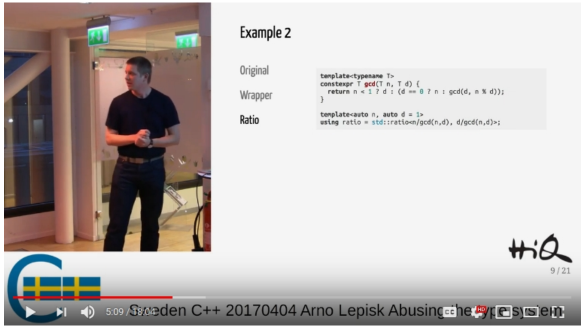
Björn Fahller – Programming with Contracts in C++20



Petter Holmberg – The Dark Art of Type Functions



Harald Achitz – Less is more, let’s build a spaceship!



Arno Lepisk – Abusing the type system for fun and profit

Takeaways

- Remember equivalence, weak and **total ordering**
- Remember commutativity, **associativity**, distributivity
- When overloading operators, **math** (and C++) decides the semantics
- C++ provides a good framework for exploring **algebraic concepts**
- **Ancient elementary math** is fundamentally important in programming

Slides: https://github.com/petter-holmberg/talks/blob/master/AncientMathModernC++CppSthlm17.pdf
Code: https://github.com/petter-holmberg/talks/blob/master/math.cpp