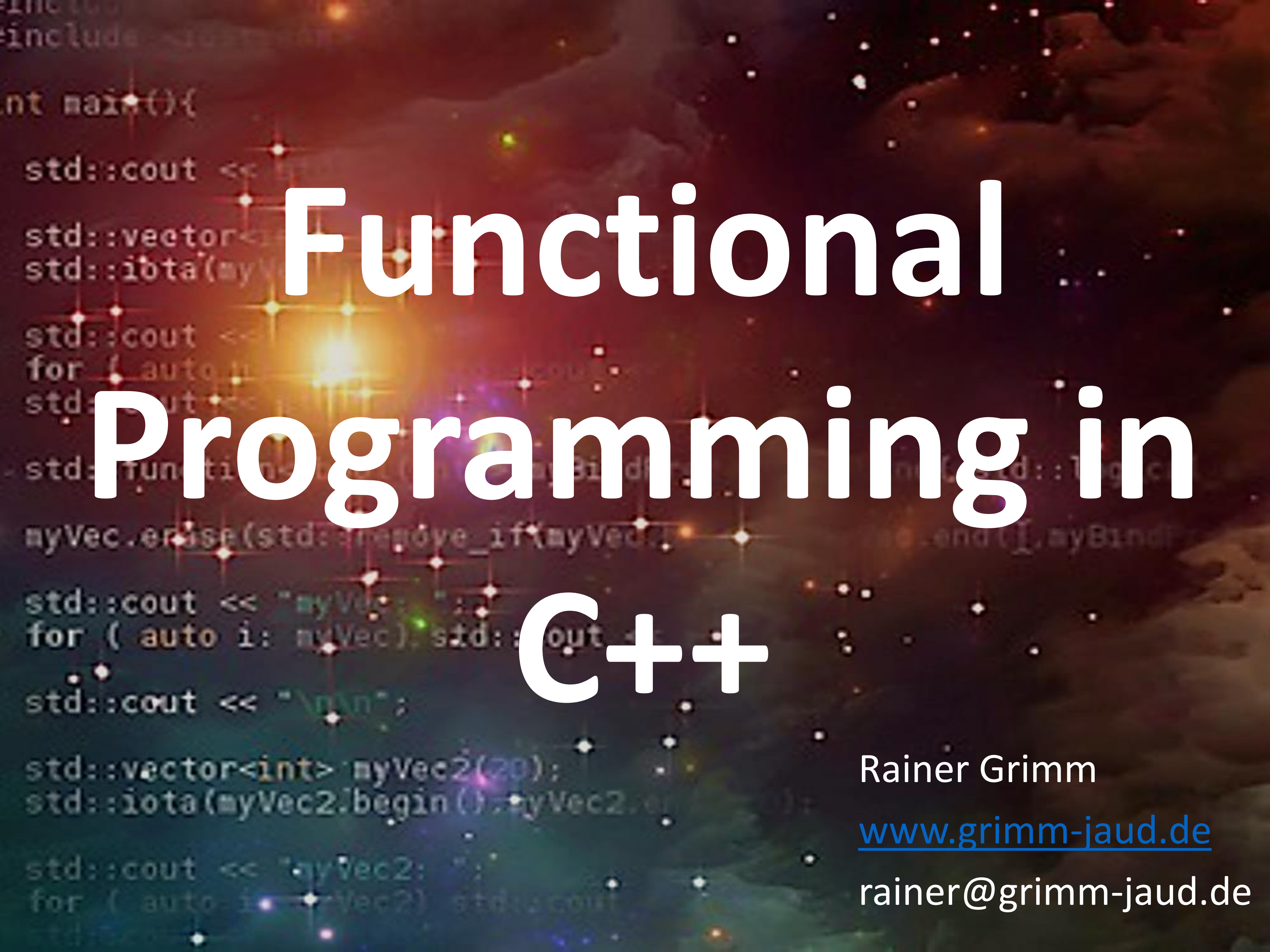


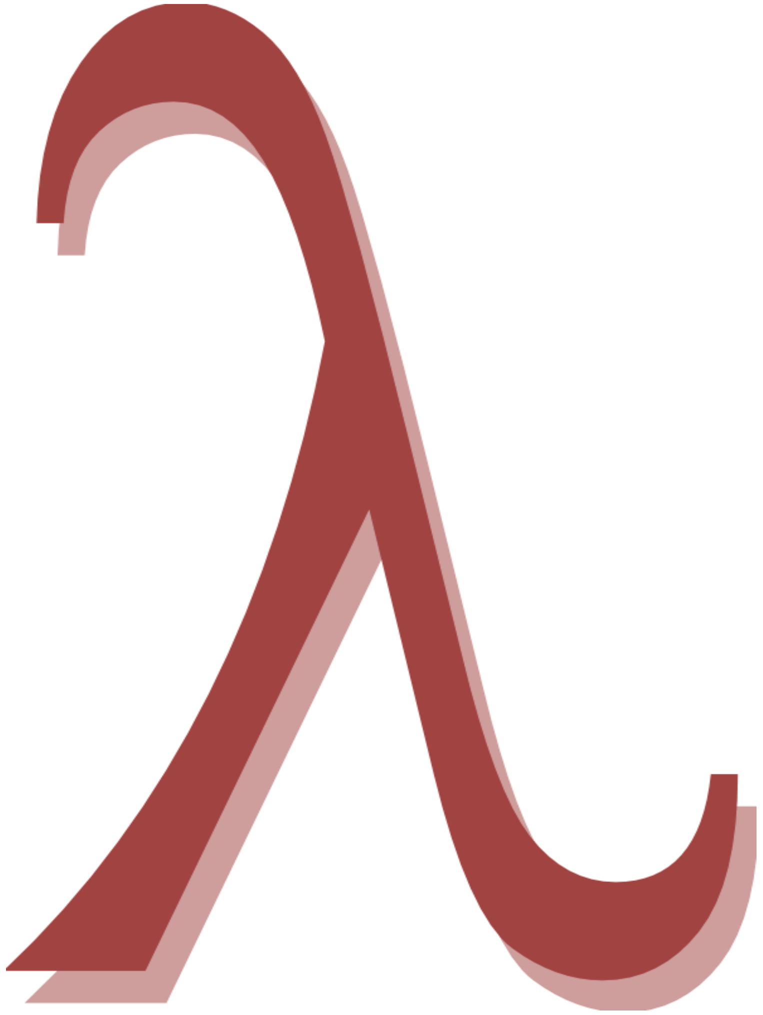
Functional Programming in C++

Rainer Grimm

www.grimm-jaud.de

rainer@grimm-jaud.de

An Overview



- Programming in a functional style
- Why functional programming?
- What is functional programming?
- Characteristics of functional programming
- What's missing?

Functional in C++

- Automatic type deduction

```
std::vector<int> myVec;
auto itVec= myVec.begin();
for ( auto v: myVec ) std::cout << v << " ";
```

- Lambda-functions

```
int a= 2000;
int b= 11;
auto sum= std::async( [=]{return a+b;} );
std::cout << sum.get() << std::endl;
```

➡ **2011**

Rainer Grimm

Functional in C++

Partial function application (Currying)

Haskell Curry

Moses Schönfinkel

1. `std::function` and `std::bind`
2. `lambda-functions` and `auto`

```
int addMe(int a, int b){ return a+b; }

std::function<int(int)> add1= std::bind(addMe,2000,_1);
auto add2= [](int b){ return addMe(2000,b); };
auto add3= [](int a){ return addMe(a,11); };

addMe(2000,11) == add1(11) == add2(11) == add3(2000);
```

➡ **2011**

Functional in C++

Higher-order functions

```
std::vector<int> vec{1,2,3,4,5,6,7,8,9};
std::for_each(vec.begin(),vec.end(),
         [](int v){ cout << " " << v;});
```

➡ **1 2 3 4 5 6 7 8 9**

```
std::for_each(vec.begin(),vec.end(),[](int& v){ v+= 10;});
std::for_each(vec.begin(),vec.end(),
         [](int v){ cout << " " << v;});
```

➡ **11 12 13 14 15 16 17 18 19**

Rainer Grimm

Functional in C++

Generic Programming

- ML introduced generic programming.
- Alexander Stepanov (Father of the Standard Template Library) was influenced by Lisp.
- Template Metaprogramming ist a pure functional programming language in the imperative language C++.

Why functional?

- More effective use of the Standard Template Library

```
std::accumulate(vec.begin(),vec.end(),
                [](int a, int b){return a+b;});
```

- Recognizing functional patterns

```
template <int N>
struct Fac{ static int const val= N * Fac<N-1>::val; };
template <>
struct Fac<0>{ static int const val= 1; };
```

- Better programming style
  - Reasoning about side effects (multithreading)
  - More concise

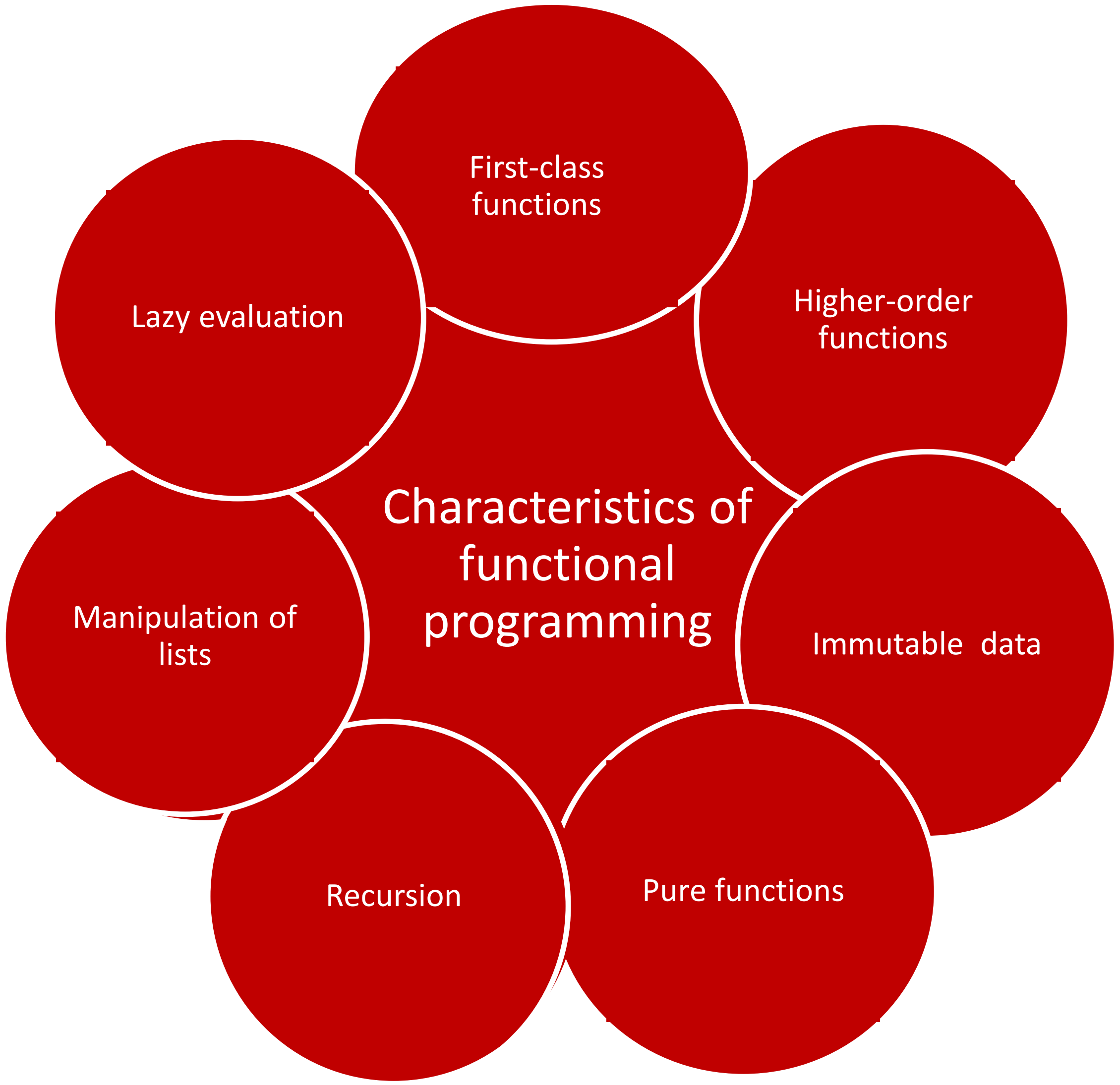
Functional programming?

- **Functional programming** is programming with mathematical functions.

- **Mathematical functions** are functions that each time return the same value when given the same arguments (referential transparency).

- **Consequences:**
  - Functions are not allowed to have side effects.
  - The function invocation can be replaced by the result, rearranged or given to an other thread.
  - The program flow will be driven by the data dependencies.

Characteristics



First-class functions

Functions are first-class objects. They behave like data.

- Functions can be
  - used as arguments to other functions.

```
std::accumulate(vec.begin(),vec.end(),
                []{ int a, int b}{ return a+b; });
```

  - given back from functions.

```
std::function<int(int,int)> makeAdd(){
  return [](int a, int b){ return a + b; };
}
std::function<int(int,int)> myAdd= makeAdd();
myAdd(2000,11);
```

➡ **2011**

  - assigned to functions or stored in variables.

First-class functions

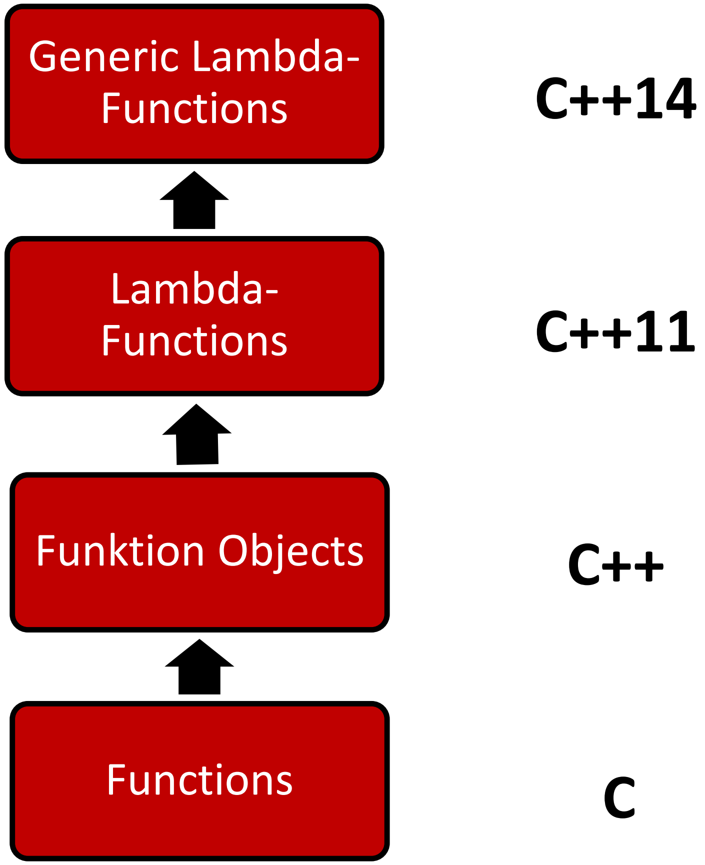
```
std::map<const char,function< double(double,double)> >  tab;

tab.insert( {'+',[](double a,double b){return a + b;} } );
tab.insert( {'-',[](double a,double b){return a - b;} } );
tab.insert( {'*',[](double a,double b){return a * b;} } );
tab.insert( {'/',[](double a,double b){return a / b;} } );

cout << "3.5+4.5= " << tab['+'](3.5,4.5) << endl;
cout << "3.5*4.5= " << tab['*'](3.5,4.5) << endl;

tab.insert( {'^',[](double a,double b){return std::pow(a,b);} } );
cout << "3.5^4.5= " << tab['^'](3.5,4.5) << endl;
```

$\Rightarrow$ **8**

$\Rightarrow$ **15.75**

$\Rightarrow$ **280.741**

First-class functions



Higher-order functions

Higher-order functions are functions that accept other functions as argument or return them as result.

The three classics:

- **map**:
  - Apply a function to each element of a list.
- **filter**:
  - Remove elements from a list.
- **fold**:
  - Reduce a list to a single value by successively applying a binary operation to a list.

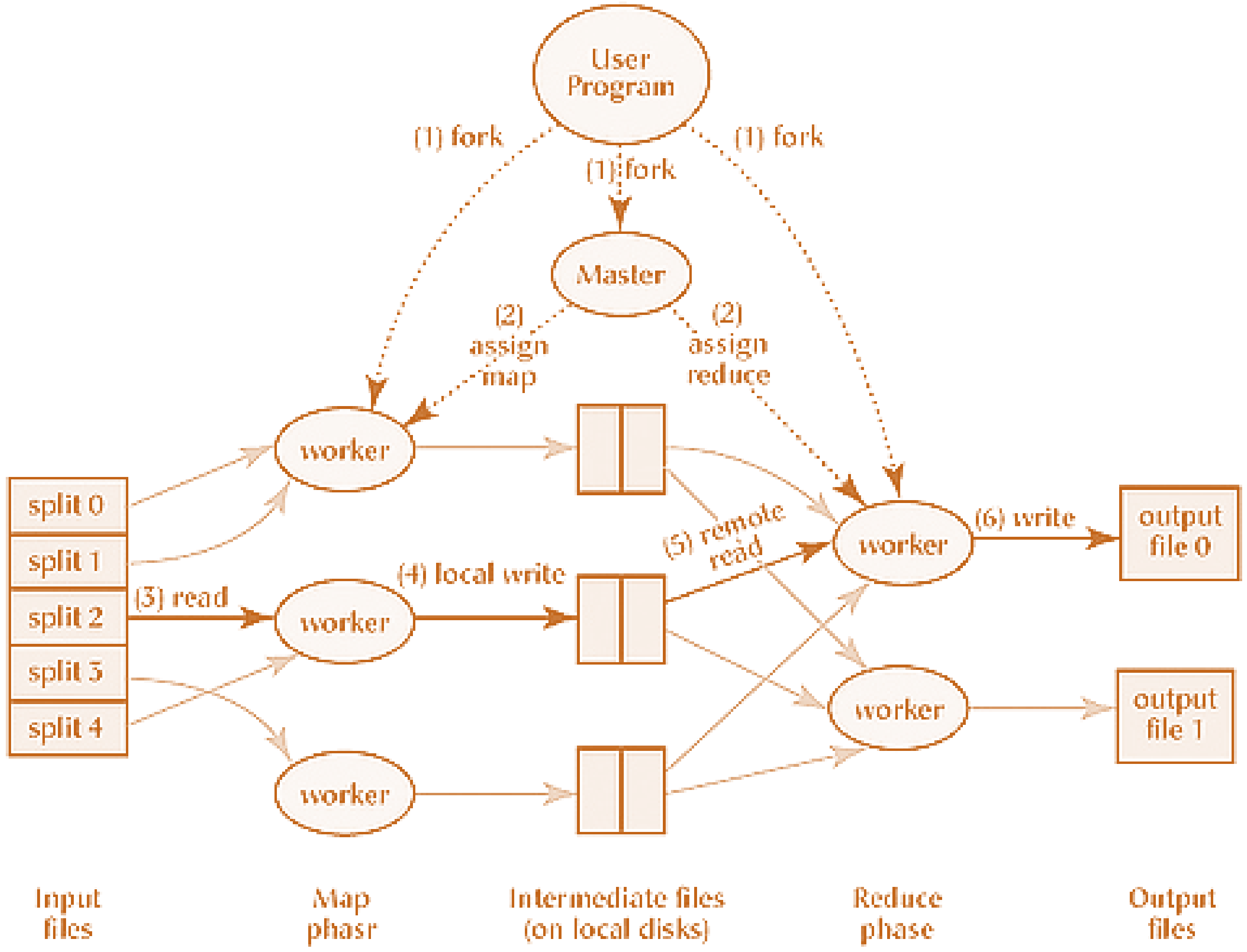
(source: http://musicantic.blogspot.de, 2012-10-16)

Higher-order functions

Each programming language supporting programming in a functional style offers **map**, **filter** and **fold.**

Haskell	C++	Python
map	std::transform	map
filter	std::remove_if	filter
fold*	std::accumulate	reduce

Higher-order functions



Rainer Grimm

Higher-order functions

- Lists and vectors:
  - Haskell

```
vec= [1 . . 9]

str= ["Programming","in","a","functional","style."]
```

  - C++

```
std::vector<int> vec{1,2,3,4,5,6,7,8,9}

std::vector<string>str{"Programming","in","a","functional",
"style."}
```

➡ The results will be displayed in Haskell notation.

Higher-order functions: map

- Haskell

```
map(\a → a*a) vec

map(\a -> length a) str
```

- C++

```
std::transform(vec.begin(),vec.end(),vec.begin(),
               [](int i){ return i*i; });

std::transform(str.begin(),str.end(),std::back_inserter(vec2),
               [](std::string s){ return s.length(); });
```

➡ **[1,4,9,16,25,36,49,64,81]**

[11,2,1,10,6]

Higher-order functions: filter

- Haskell

```
filter(\x-> x<3 || x>8) vec

filter(\x → isUpper(head x)) str
```

- C++

```
auto it= std::remove_if(vec.begin(),vec.end(),
                   [](int i){ return !((i < 3) or (i > 8)) });

auto it2= std::remove_if(str.begin(),str.end(),
             [](std::string s){ return !(std::isupper(s[0])); });
```

➡ **[1,2,9]**

[“Programming”]

Higher-order functions: fold

- Haskell

```
foldl (\a b → a * b) 1 vec

foldl (\a b → a ++ ":" ++ b ) "" str
```

- C++

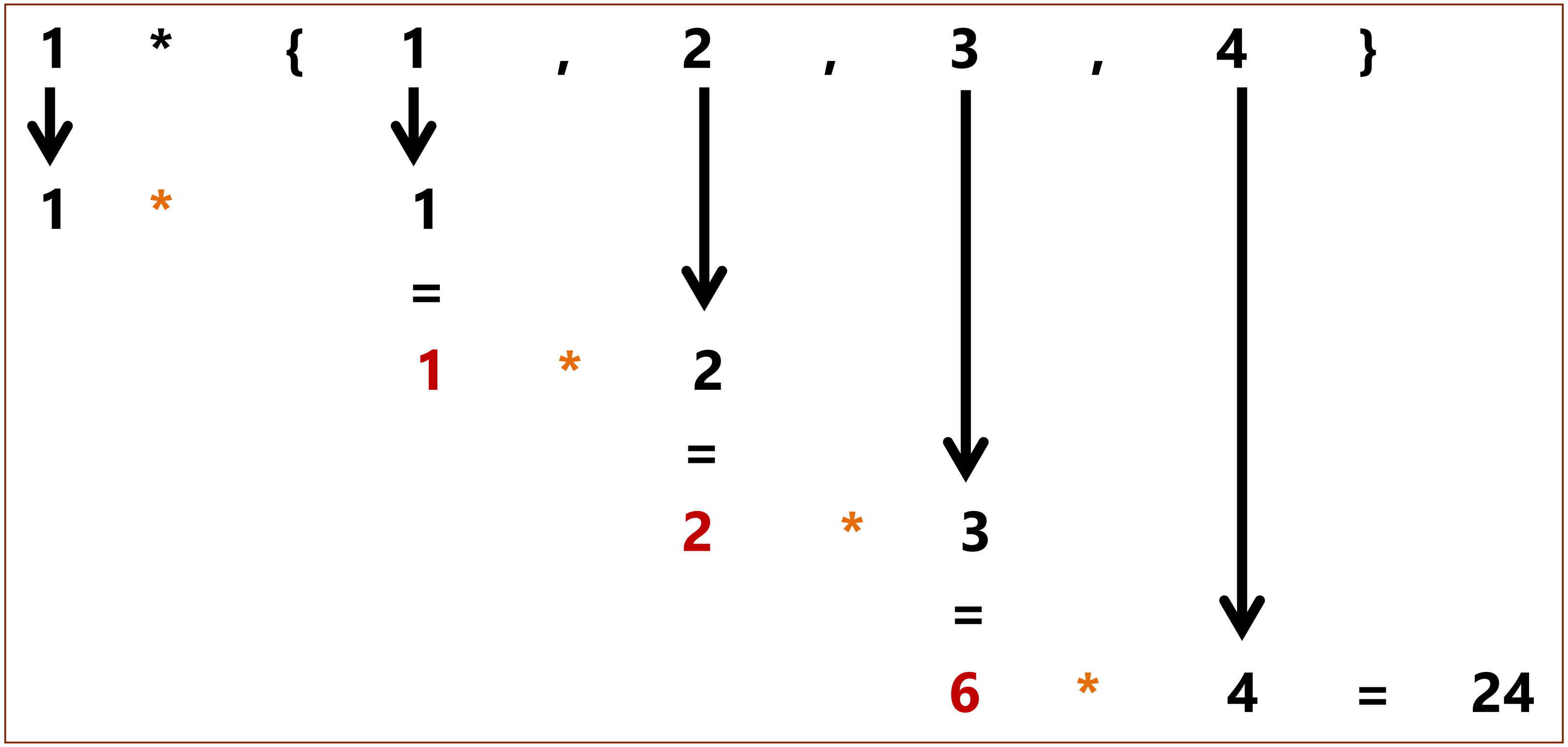
```
std::accumulate(vec.begin(),vec.end(),1,
                [](int a, int b){ return a*b; });

std::accumulate(str.begin(),str.end(),string(""),
    [](std::string a,std::string b){ return a+":"+b; });
```

➡ **362800**

“:Programming:in:a:functional:style.”

Higher-order functions: fold

```
std::vector<int> v{1,2,3,4};
std::accumulate(v.begin(),v.end(),1,[](int a, int b){return a*b;});
```



Higher-order: fold expression

Reduce a parameter pack over a binary operator.

- Part of C++17
- A parameter pack is a parameter, that accepts zero or more arguments
- Simulates left and right fold with and without init value.

➡ `foldl`, `foldl1`, `foldr` and `foldr1` in Haskell

```
template<typename ... Args>
bool all(Args ... args){
  return ( ... && args);
}

bool b= all(true, true, true, false); // ((true && true)&& true)&& false;
```

Rainer Grimm

Immutable data

Data is immutable in pure functional languages.

Consequences

- There is no
  - Assignment: x= x + 1, ++x
  - Loops: for, while , until
- In case of data modification
  - Changed copies of the data will be generated.
  - The original data will be shared.

➡ Immutable data is thread safe.

Immutable data

- Haskell

```haskell
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y < x] ++ [x] ++ qsort [y | y <- xs, y >= x]
```

- C++

```cpp
void quickSort(int arr[], int left, int right) {
  int i = left, j = right;
  int tmp;
  int pivot = arr[abs((left + right) / 2)];
  while (i <= j) {
    while (arr[i] < pivot) i++;
    while (arr[j] > pivot) j--;
    if (i <= j) {
      tmp = arr[i];
      arr[i] = arr[j];
      arr[j] = tmp;
      i++; j--;
    }
  }
  if (left < j) quickSort(arr,left,j);
  if (i < right) quickSort(arr,i,right);
}
```

Immutable data

Working with immutable data is based on discipline.

➡ Use `const` data, Template Metaprogramming or constant expressions (`constexpr`).

- `const` data

```
const int value= 1;
```

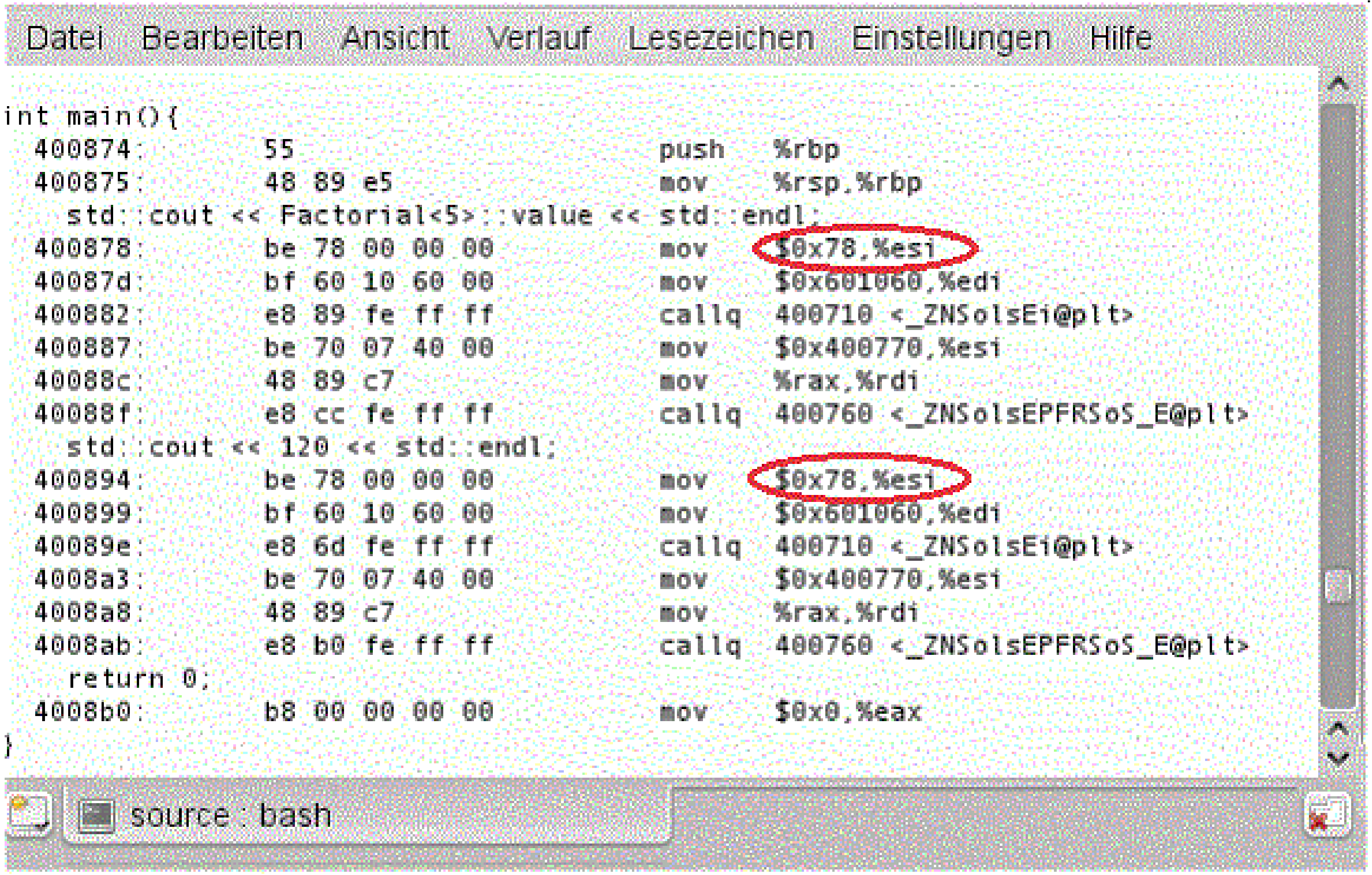
- Template Metaprogramming
  - Is a pure functional language, embedded in the imperative language C++
  - Will be executed at compile time
  - There is no mutation at compile time

Immutable data

```
template <int N>
struct Factorial{
    static int const value= N * Factorial<N-1>::value;
};
template <>
struct Factorial<1>{
    static int const value = 1;
};

std::cout << Factorial<5>::value << std::endl;
std::cout << 120 << std::endl;
```

```
Factorial<5>::value   ➡   5*Factorial<4>::value
                          5*4*Factorial<3>::value
                          5*4*3*Factorial<2>::value
                          5*4*3*2*Factorial<1>::value  = 5*4*3*2*1= 120
```

Immutable data


Datei Bearbeiten Ansicht Verlauf Lesezeichen Einstellungen Hilfe
source : bash

```
int main(){
  400874:       55                      push   %rbp
  400875:       48 89 e5                mov    %rsp,%rbp
    std::cout << Factorial<5>::value << std::endl;
  400878:       be 78 00 00 00          mov    $0x78,%esi
  40087d:       bf 60 10 60 00          mov    $0x601060,%edi
  400882:       e8 89 fe ff ff          callq  400710 <_ZNSolsEi@plt>
  400887:       be 70 07 40 00          mov    $0x400770,%esi
  40088c:       48 89 c7                mov    %rax,%rdi
  40088f:       e8 cc fe ff ff          callq  400760 <_ZNSolsEPFRSoS_E@plt>
    std::cout << 120 << std::endl;
  400894:       be 78 00 00 00          mov    $0x78,%esi
  400899:       bf 60 10 60 00          mov    $0x601060,%edi
  40089e:       e8 6d fe ff ff          callq  400710 <_ZNSolsEi@plt>
  4008a3:       be 70 07 40 00          mov    $0x400770,%esi
  4008a8:       48 89 c7                mov    %rax,%rdi
  4008ab:       e8 b0 fe ff ff          callq  400760 <_ZNSolsEPFRSoS_E@plt>
    return 0;
  4008b0:       b8 00 00 00 00          mov    $0x0,%eax
}
```

Rainer Grimm

Immutable data

- Constant expressions
  - are available as variables, user defined types and functions.
  - can be evaluated at compile time.

  - Variables
    - are implicit const.
    - must be initialized by a const expression.

```
constexpr double pi= 3.14;
```

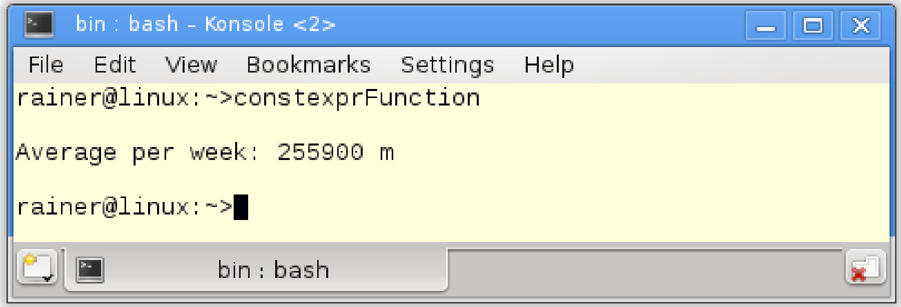
  - User defined type
    - The constructor must be empty and a constant expression.
    - The methods must be constant expression and must not be `virtual`.

⇒ Objects can be created at compile time.

Rainer Grimm

Immutable data

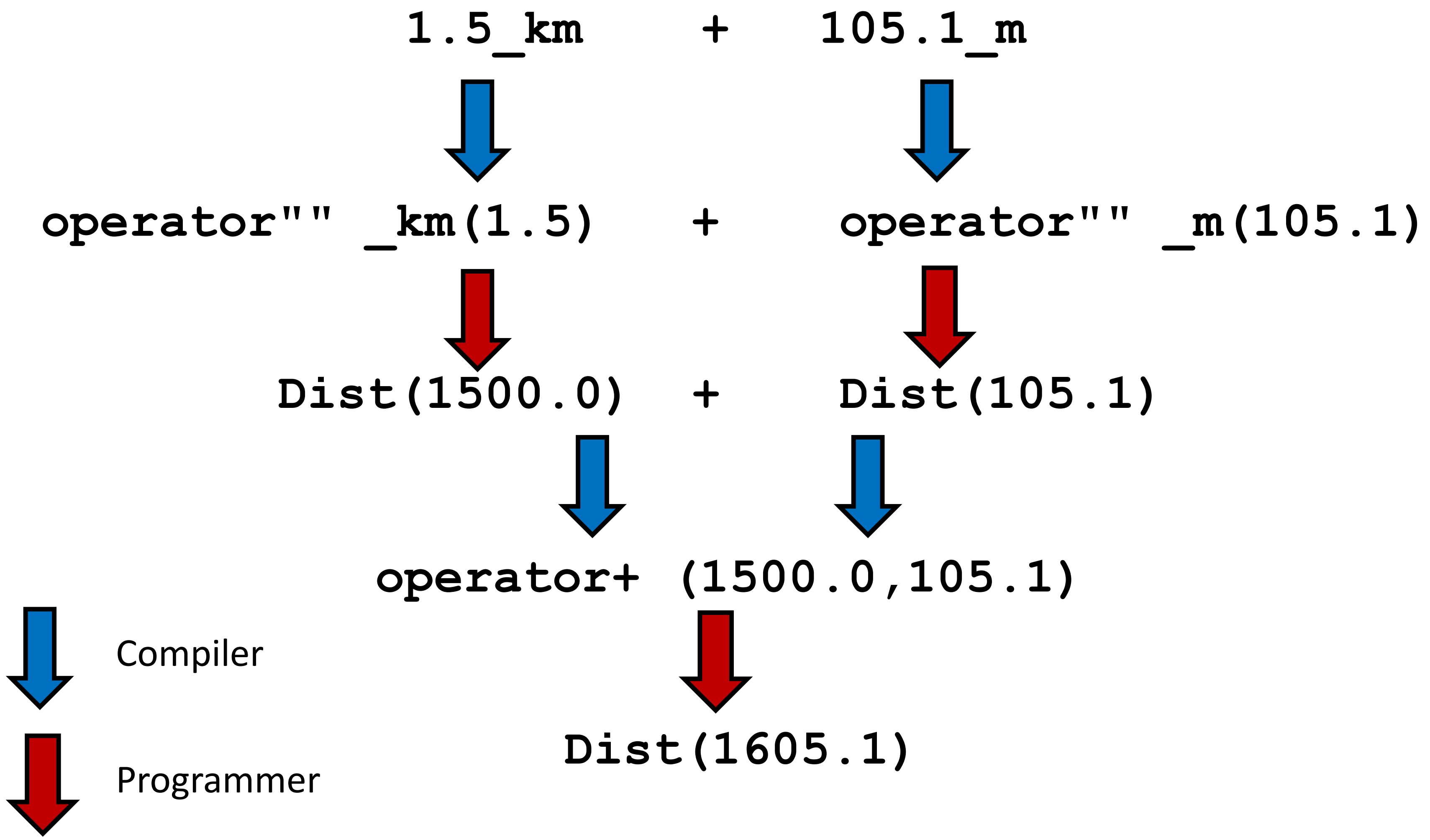
```
int main(){

  constexpr Dist work= 63.0_km;
  constexpr Dist workPerDay= 2 * work;
  constexpr Dist abbreToWork= 5400.0_m;            // abbrevation to work
  constexpr Dist workout= 2 * 1600.0_m;
  constexpr Dist shop= 2 * 1200.0_m;               // shopping

  constexpr Dist distPerWeek1= 4*workPerDay - 3*abbreToWork + workout + shop;
  constexpr Dist distPerWeek2= 4*workPerDay - 3*abbreToWork + 2*workout;
  constexpr Dist distPerWeek3= 4*workout + 2*shop;
  constexpr Dist distPerWeek4= 5*workout + shop;

  constexpr Dist perMonth=  getAverageDistance({distPerWeek1,
                            distPerWeek2,distPerWeek3,distPerWeek4});

  std::cout << "Average per week: " << averagePerWeek << std::endl;

}
```



```
rainer@linux:~>constexprFunction

Average per week: 255900 m

rainer@linux:~>
```

Rainer Grimm

Immutable data



```
1.5_km   +   105.1_m
operator"" _km(1.5)   +   operator"" _m(105.1)
Dist(1500.0)   +   Dist(105.1)
operator+ (1500.0,105.1)
Dist(1605.1)
```

Immutable data

```
namespace Unit{
  Dist constexpr operator "" _km(long double d){
    return Dist(1000*d);
  }
  Dist constexpr operator "" _m(long double m){
    return Dist(m);
  }
  Dist constexpr operator "" _dm(long double d){
    return Dist(d/10);
  }
  Dist constexpr operator "" _cm(long double c){
    return Dist(c/100);
  }
}

constexpr Dist getAverageDistance(std::initializer_list<Dist> inList){
  auto sum= Dist(0.0);
  for ( auto i: inList) sum += i;
  return sum/inList.size();
}
```

Immutable data

```
class Dist{
public:
  constexpr Dist(long double i):m(i){}

  friend constexpr Dist operator +(const Dist& a, const Dist& b){
    return Dist(a.m + b.m);
  }
  friend constexpr Dist operator -(const Dist& a,const Dist& b){
    return Dist(a.m - b.m);
  }
  friend constexpr Dist operator*(double m, const Dist& a){
    return Dist(m*a.m);
  }
  friend constexpr Dist operator/(const Dist& a, int n){
    return Dist(a.m/n);
  }
  friend std::ostream& operator<< (std::ostream &out, const Dist& myDist){
    out << myDist.m << " m";
    return out;
  }
private:
  long double m;
};
```

Pure functions

Pure functions	Impure functions
Always produce the same result when given the same arguments.	May produce different results for the same arguments.
Never have side effects.	May have side effects.
Never alter state.	May alter the global state of the program, system, or the world.

- Advantages
  - Correctness of the code is easier to verify.
  - Simplifies the refactoring and testing of the code.
  - It is possible to save results of pure function invocations.
  - Pure function invocations can be reordered or performed on other threads.

Pure functions

Working with pure functions is based on discipline.

➡ Use ordinary functions, metafunctions or constant expression functions.

- Function

```
int powFunc(int m, int n){
  if (n == 0) return 1;
  return m * powFunc(m, n-1);
}
```

- Metafunction

```
template<int m, int n>
struct PowMeta{
  static int const value = m * PowMeta<m,n-1>::value;
};

template<int m>
struct PowMeta<m,0>{
  static int const value = 1;
};
```

Pure functions

- Constant expression functions

```
constexpr int powConst(int m, int n){
  int r = 1;
  for(int k=1; k<=n; ++k) r*= m;
  return r;
}
```

```
int main(){
  std::cout << powFunc(2,10) << std::endl;                    // 1024
  std::cout << PowMeta<2,10>::value << std::endl;             // 1024
  std::cout << powConst(2,10) << std::endl;                   // 1024
}
```

Pure functions

Monads are the Haskell solution to deal with the impure world.

- Encapsulates the impure world
- Is a imperative subsystem
- Represents a computational structure
- Define the composition of computations

➡ Functional patterns for generic types.

Pure functions

A Monad is a abstract data type, that transforms simple data types in higher (enriched) data types.

A Monad consists of a

1. Type constructor
   - Declares for the underlying type, how to become the monadic type.
2. Functions
   - Unit function: Inject the underlying type to a value in the corresponding monadic type. (`return`)
   - Function composition: Composition of monadic types. (`bind`)
3. The functions have to obey a few axioms
   - The unit function must be the left and right neutral element.
   - Die composition of operations must be associative.

Pure functions

Reader Monad

I/O Monad

List Monad

STM Monad

Error Monad

Parsec

State Monad

Maybe Monad

Coroutine Monads

Exception Monad

optional and ranges

- `std::experimental::optional`
  - Is a value, the may or my not be present ➡ Maybe Monad
  - Part of the namespace `std::experimental`
    - Should be part of C++14
    - May become with high probability part of the next C++-Standard

```
std::optional<int> getFirst(const std::vector<int>& vec){
  if ( !vec.empty() ) return std::optional<int>(vec[0]);
  else return std::optional<int>();
}
```

- Ranges for the Standard Library

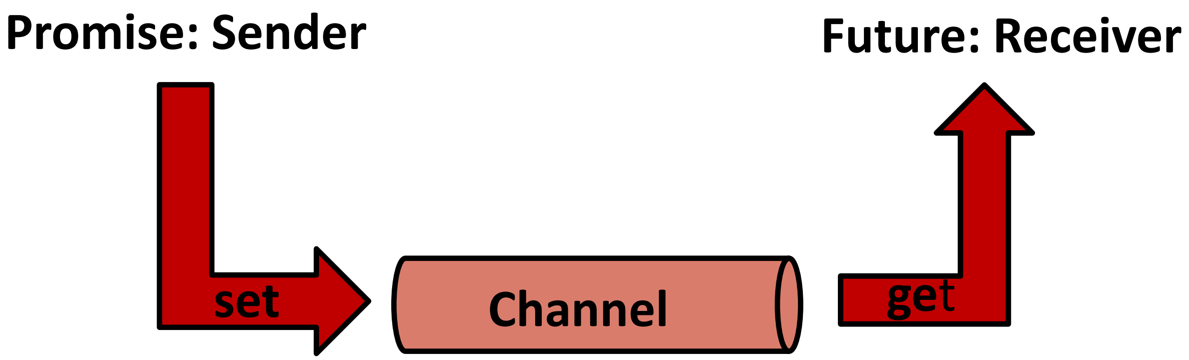
➡ C++ Ranges are Pure Monadic Goodness (Bartosz Milewski)

Pure functions

`std::promise` and `std::future`

- Are channels between a Sender and a Receiver.
- The Producer puts a value in the channel, the Consumer is waiting for.
- The Sender is called Promise, the Receiver Future.

```
int a= 2000, b= 11;
std::future<int> sum= std::async( [=]{return a+b;} );
std::cout << sum.get() << std::endl;
```



std::future improvements

`std::promise` and `std::future`

- Has a few serious short comings $\Rightarrow$ Futures are not composable

- Improvements for composability (TS 19571)
  - `then`: attach a continuation to a future $\Rightarrow$ `fmap` (Functor)
  - `future<future<T>>`: unboxing constructor $\Rightarrow$ `join` (Monad)
  - `make_ready_future`: produce a future that is ready immediately and holds a given value $\Rightarrow$ `return` (Monad)
  - `when_any`: produces a new future, when at least one of the futures is ready $\Rightarrow$ `mplus` (Monad Plus)
  - `when_all`: produces a new future, when all futures are ready

C++17: I See a Monad in Your Future! (Bartosz Milewski)

Recursion

Recursion is the control structure in functional programming.

- A loop needs a running variable `(for int i=0; i <= 0; ++i)`

  ➡ There are no variables in pure functional languages.

➡ Recursion combined with list processing is a powerful pattern in functional languages.

Recursion

- Haskell

```
fac 0= 1
fac n= n * fac (n-1)
```

- C++

```
template<int N>
struct Fac{
  static int const value= N * Fac<N-1>::value;
};
template <>
struct Fac<0>{
  static int const value = 1;
};
```

➡ **fac(5) == Fac<5>::value == 120**

Recursion

```
Fac<5>::value =
          = 5 * Fac<4>::value
          = 5 * 4 * Fac<3>::value
          = 5 * 4 * 3 * Fac<2>::value
          = 5 * 4 * 3 * 2 * Fac<1>::value
          = 5 * 4 * 3 * 2 * 1 * Fac<0>::value
          = 5 * 4 * 3 * 2 * 1 * 1
          = 120
```

List processing

- LISt Processing is the characteristic for functional programming:
  - Transforming a list into another list.
  - Reducing a list to a value.

- The functional pattern for list processing:
  1. Processing the head of the list.
  2. Recursively processing the tail of the list.

```
mySum []     = 0
mySum (x:xs) = x + mySum xs
mySum [1,2,3,4,5]
```

$\Rightarrow$ **15**

List processing

```
template<int ...>
struct mySum;

template<>struct
mySum<>{
  static const int value= 0;
};

template<int i, int ... tail> struct
mySum<i,tail...>{
  static const int value= i + mySum<tail...>::value;
};

int sum= mySum<1,2,3,4,5>::value;
```

→ **sum == 15**

List processing

The key idea: Pattern matching

- First match in Haskell

```
mult n 0 = 0
mult n 1 = n
mult n m = (mult n (m - 1)) + n
```

```
      mult 3 2 = (mult 3 (2 - 1)) + 3
               = (mult 3 1 ) + 3
               = 3 + 3
               = 6
```

List processing

- Best match in C++

```
template < int N, int M >
struct Mult{
  static const int value= Mult<N,M-1>::value + N;
};
template < int N >
struct Mult<N,1> {
  static const int value= N;
};
template < int N >
struct Mult<N,0> {
  static const int value= 0;
};
std::cout << Mult<3,2>::value << std::endl;
```

➡ **6**

Lazy Evaluation

- Evaluate only, if necessary.

  - Haskell is lazy

    ```
    length [2+1, 3*2, 1/0, 5-4]
    ```

  - C++ is eager

    ```
    int onlyFirst(int a, int){ return a; }
    onlyFirst(1,1/0);
    ```

- Advantages:
  - Saving time and memory usage.
  - Working with infinite data structures.

Lazy Evaluation

- Haskell

```
successor i= i: (successor (i+1))

take 5 ( successor 1 )
```

$\Rightarrow$ **[1,2,3,4,5]**

```
odds= takeWhile (< 1000) . filter odd . map (^2)

[1..]= [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 ... Control-C

odds [1..]
```

$\Rightarrow$ **[1,9,25, … , 841,961]**

Rainer Grimm

Lazy Evaluation

C++

- Short circuit evaluation in logical expressions

```
if ( true or (1/0) ) std::cout << "short circuit evaluation"
```

```
short circuit evaluation
```

- Expression Templates
  - Use Operator Overloading and Template Metaprogramming
  - A templated expression store the structure of some arbitrary sub-expression. The recursive evaluation of the expression-tree takes places at the assignment of the result.

```
Vec<int> a(LEN), Vec<int> b(LEN), Vec<int> c(LEN);
Vec<int> res= a*b + b*d;
```

Ranges

Ranges for the Standard Library (N4128) by Eric Niebler

- Algorithm for `std::vector<int> v{1,2,3,4,5,6,7,8,9};`
  - Classical with Iterators

```
std::sort( v.begin(), v.end() );
```

  - New with Ranges

```
std::sort( v );
```

- Range Adaptors support pipelines and lazy evaluation

```
int total= std::accumulate(view::ints(1) |
                view::transform([ ](int x){return x*x;}) |
                view::take(10), 0);
```

⇔ `total= sum $ take 10 $ map (\x -> x*x) [1..]`

⇒ **385**

What's missing?

- List comprehension: Syntactic sugar for map and filter

- Like mathematic

  { ν∗ν | ν ε N , ν μοδ 2 = 0 } : Mathematik

  `[n*n | n <- [1..], n `mod` 2 == 0 ]` : Haskell

- Example

```
[ n | n <- [1..8] ]
```
$\Rightarrow$ **[1,2,3,4,5,6,7]**

```
[ n*n | n <- [1..8] ]
```
$\Rightarrow$ **[1,4,9,16,25,36,49]**

```
[ n*n | n <- [1..8], n `mod`2 == 0]
```
$\Rightarrow$ **[4,16,36]**

Range Comprehension

- Pythagorean triple in Haskell with List Comprehension

```
triples =[(x, y, z)|z <-[1..], x <-[1..z],y <-[x..z] ,x^2 + y^2 == z^2]
triples =
  (>>=) [1..] $ \z ->
    (>>=) [1..z] $ \x ->
      (>>=) [x..z] $ \y ->
         guard (x^2 + y^2 == z^2) >> return (x, y, z)
take 5 triples
```

- Pythagorean triple in C++ with Range Comprehension

```
auto triples =
  view::for_each(view::ints(1), [](int z){
    return view::for_each(view::ints(1, z), [=](int x){
       return view::for_each(view::ints(x, z), [=](int y){
         return yield_if(x*x + y*y == z*z, std::make_tuple(x, y, z));
       });
    });
  });
  for (auto triple: triples | view::take(5)){ ...
```

➡ **[(3,4,5),(6,8,10),(5,12,13),(9,12,15),(8,15,17)]**

Rainer Grimm

What's missing?

Syntactic sugar for Monads: do-Notation

```
triples = do
   z <-[1..]
   x <-[1..z]
   y <-[x..z]
   guard(x^2 + y^2 == z^2)
   return (x,y,z)

 take 5 triples
```

⇒ **[(3,4,5),(6,8,10),(5,12,13),(9,12,15),(8,15,17)]**

What's missing?

Function composition: fluent interface

```
(reverse . sort)[10,2,8,1,9,5,3,6,4,7]
```

➡ **`[10,9,8,7,6,5,4,3,2,1]`**

```
isTit (x:xs)= isUpper x && all isLower xs

sorTitLen= sortBy(comparing length) . filter isTit . words
sorTitLen "A Sentence full of Titles ."
```

➡ **`["A","Titles","Sentence"]`**

What's missing?

Typeclasses are interfaces for similar types.

- Typeclass `Eq`

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    a == b = not (a /= b)
    a /= b = not (a == b)
```

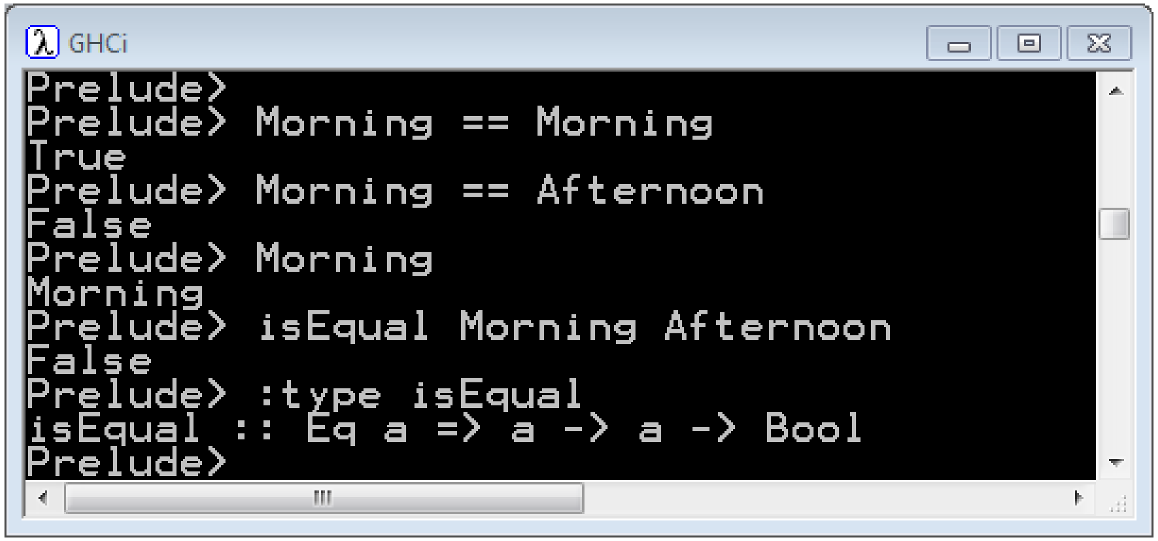
- Type `Bool` as instance of the typeclass `Eq`

```
instance Eq Bool where
True == True = True
False == False = True
_ == _ = False
```

What's missing?

- Datentyp Day as member of Eq and Show

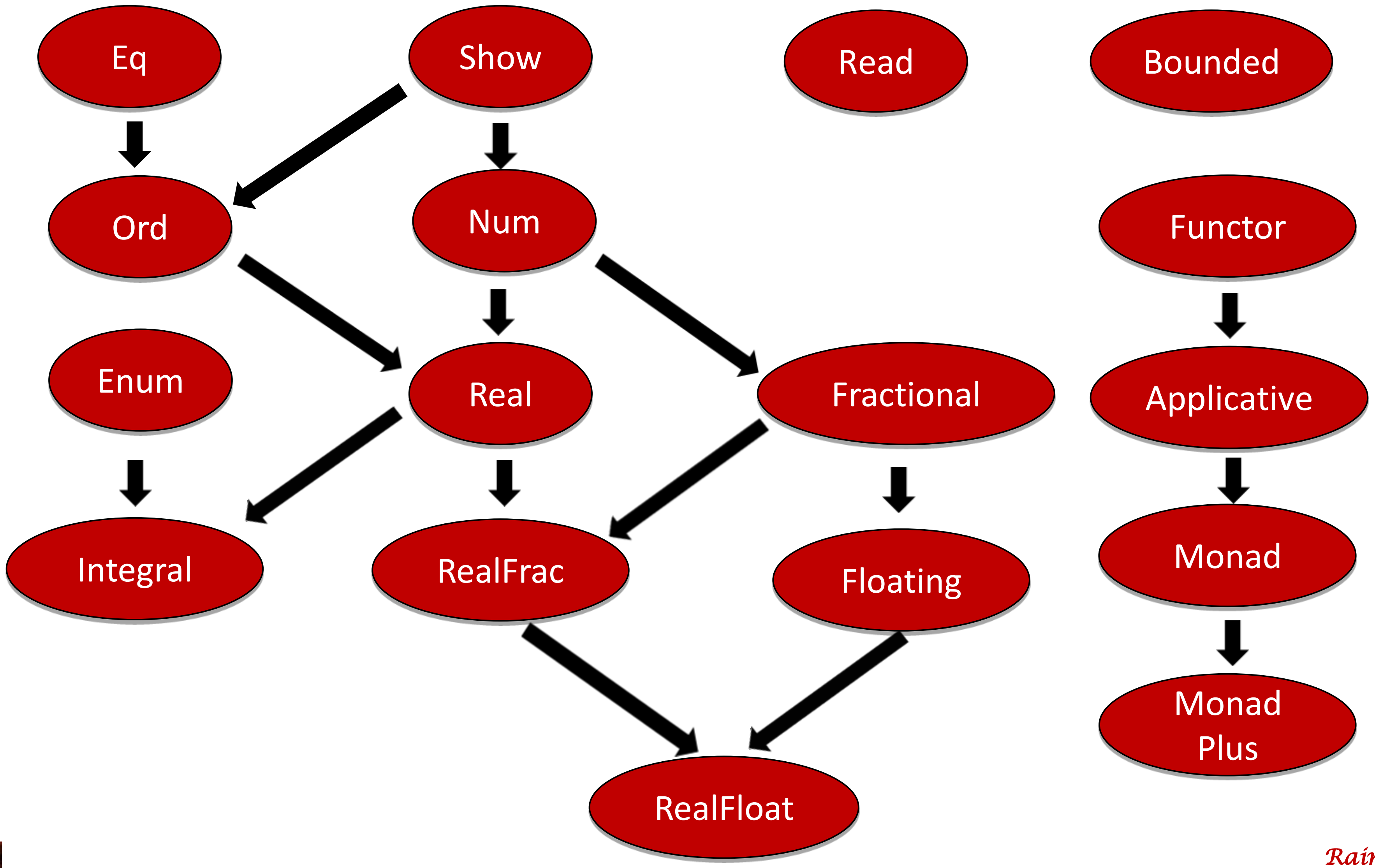
```
data Day= Morning | Afternoon deriving (Eq,Show)
isEqual a b = a == b
```



```
Prelude>
Prelude> Morning == Morning
True
Prelude> Morning == Afternoon
False
Prelude> Morning
Morning
Prelude> isEqual Morning Afternoon
False
Prelude> :type isEqual
isEqual :: Eq a => a -> a -> Bool
Prelude>
```

- Userdefined typeclasses are supported.

What's missing?



Concepts Lite

- Concepts Lite are constraints for templates.

  - Models semantic categories rather then syntactic restrictions.
  - State requirements of templates at there declaration.
  - Support function overloading and class templates specialization based on constraints.
  - Integrates with automatic type deduction.
  - Improves error messages by checking template arguments at the point of template instanziation.

⇒ Additional benefit with no cost for the run time, compile time or code size.

Concepts Lite

- Template declaration

```
template<Sortable Cont>
void sort(Cont& container);
```

equivalent

```
template<typename Cont>
  requires Sortable<Cont>()
void sort(Cont& container);
```

- Sortable must be a predicate (constexpr)

```
template <typename T>
constexpr bool Sortable(){  . . .
```

```
std::list<int> lst = {1998,2014,2003,2011};
sort(lst); // ERROR: lst is no random-access container with <
```

Concepts Lite

- Concepts Lite works for any template

  - Function templates

```
template<LessThanComparable T>
const T& min(const T &x, const T &y) {
  return (y < x) ? y : x;
}
```

  - Class templates

```
template<Object T>
class vector;

vector<int> v1; // OK
vector<int&> v2 // ERROR: int& does not satisfy the
                          constraint Object
```

Concepts Lite

- Member functions of class templates

```
template <Object T>
class vector{

  void push_back(const T& x)
    requires Copyable<T>();

};
```

- Automatic type deduction

```
auto func(Container) -> Sortable;
Sortable x= func(y);
```

Concepts Lite

- Concepts Lite supports
  - Multiple requirements for the template arguments.

```
template <SequenceContainer S,
          EqualityComparable<value_type<S>> T>
Iterator_type<S> find(S&& seq, const T& val);
```

  - Overloading of function and class templates specialization.

```
template<InputIterator I>
void advance(I& iter, int n);
template<BidirectionalIterator I>
void advance(I& iter, int n);
template<RandomAccessIterator I>
void advance(I& iter, int n);
```

```
std::list<int> lst{1,2,3,4,5,6,7,8,9};
std::list<int>:: iterator i= lst.begin();
std::advance(i,2);
```
$\Rightarrow$ **BidirectionalIterator**

Concepts Lite

Basic

- DefaultConstructible
- MoveConstructible
- CopyConstructible
- MoveAssigable
- CopyAssignable
- Destructible

Container

- Container
- ReversibleContainer
- AllocatorAwareContainer
- SequenceContainer
- ContinguousContainer
- AssociativeContainer
- UnorderedAssociativeContainer

Layout

- TriviallyCopyable
- TrivialType
- StandardLayoutType
- PODType

Iterator

- Iterator
- InputIterator
- OutputIterator
- ForwardIterator
- BidirectionalIterator
- RandomAccessIterator
- ContinguousIterator

Container element

Stream I/O functions

Random Number Generation

Library-wide

- EqualityComparable
- LessThenComparable
- Swappable
- ValueSwappable
- NullablePointer
- Hash
- Allocator
- FunctionObject
- Callable
- Predicate
- BinaryPredicate
- Compare

Concurrency

Other

Rainer Grimm

www.grimm-jaud.de

rainer@grimm-jaud.de