Advanced Developers Conference
Development for Professionals!

Funktionale Programmierung mit C++

Rainer Grimm
Softwarearchitekt

Partner: intel Software embarcadero IncrediBuild BEYOND ACCELERATION ppedv

Überblick

- Programmierung in funktionaler Art
- Warum funktionale Programmierung?
- Was ist funktionale Programmierung?
- Charakteristiken funktionaler Programmierung
- Was fehlt in C++?

Funktional in C++

- Automatische Typableitung

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

- Lambda-Funktionen

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

➡ **2011**

Funktional in C++

Teilweise Funktionsanwendung (Currying) mit

1. `function` und `bind`
2. Lambda-Funktionen und `auto`

Haskell Curry

Moses Schönfinkel

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

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

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

➡ **2011**

Funktional in C++

Funktionen höherer Ordnung

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

➡ **1 2 3 4 5 6 7 8 9**

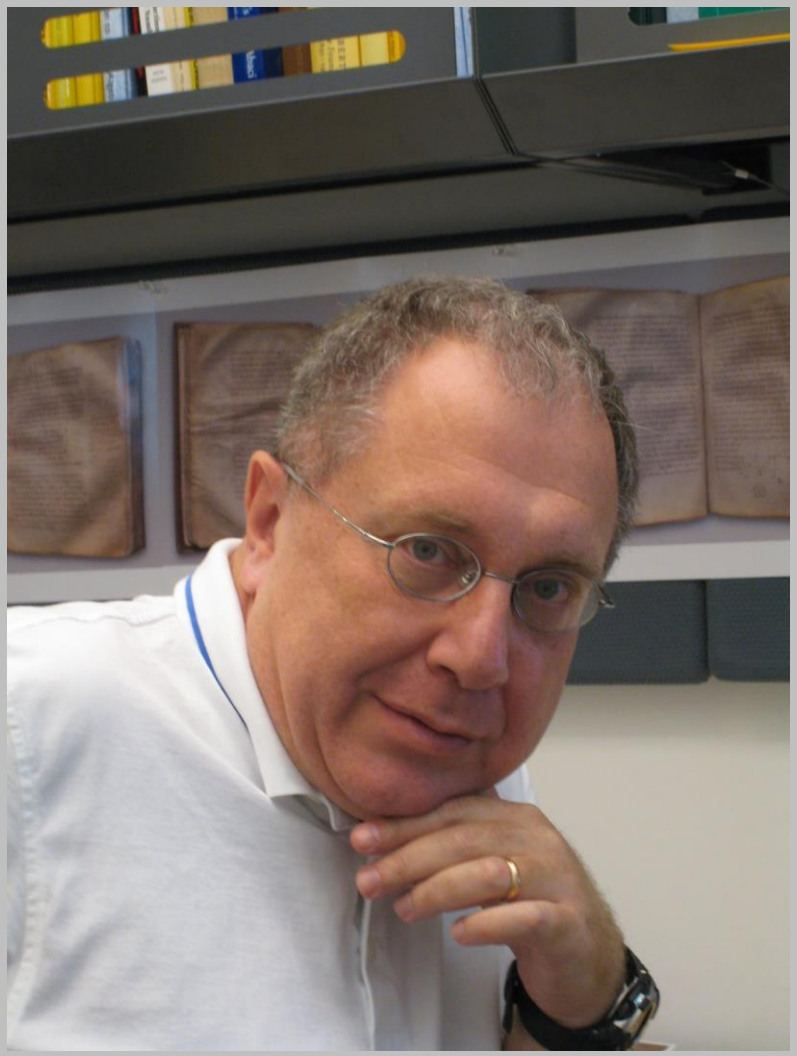
```
for_each(vec.begin(),vec.end(),[ ](int& v){ v+= 10;});
for_each(vec.begin(),vec.end(),
         [ ]{int v){ cout << " " << v;});
```

➡ **11 12 13 14 15 16 17 18 19**

Funktional in C++

Generische Programmierung

- ML führt generische Programmierung ein.
- Alexander Stepanov (Vater der STL) wurde von Lisp beeinflusst.
- Template Metaprogramming ist eine rein funktionale Subsprache in der imperativen Sprache C++.



Warum Funktional?

- Effizientere Nutzung der Standard Template Library

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

- Funktionale Muster erkennen

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

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

- Besserer Programmierstil
  - Bewusster Umgang mit Seiteneffekten
  - Kurz und prägnante Programmierung
  - Verständlichere Programmierung

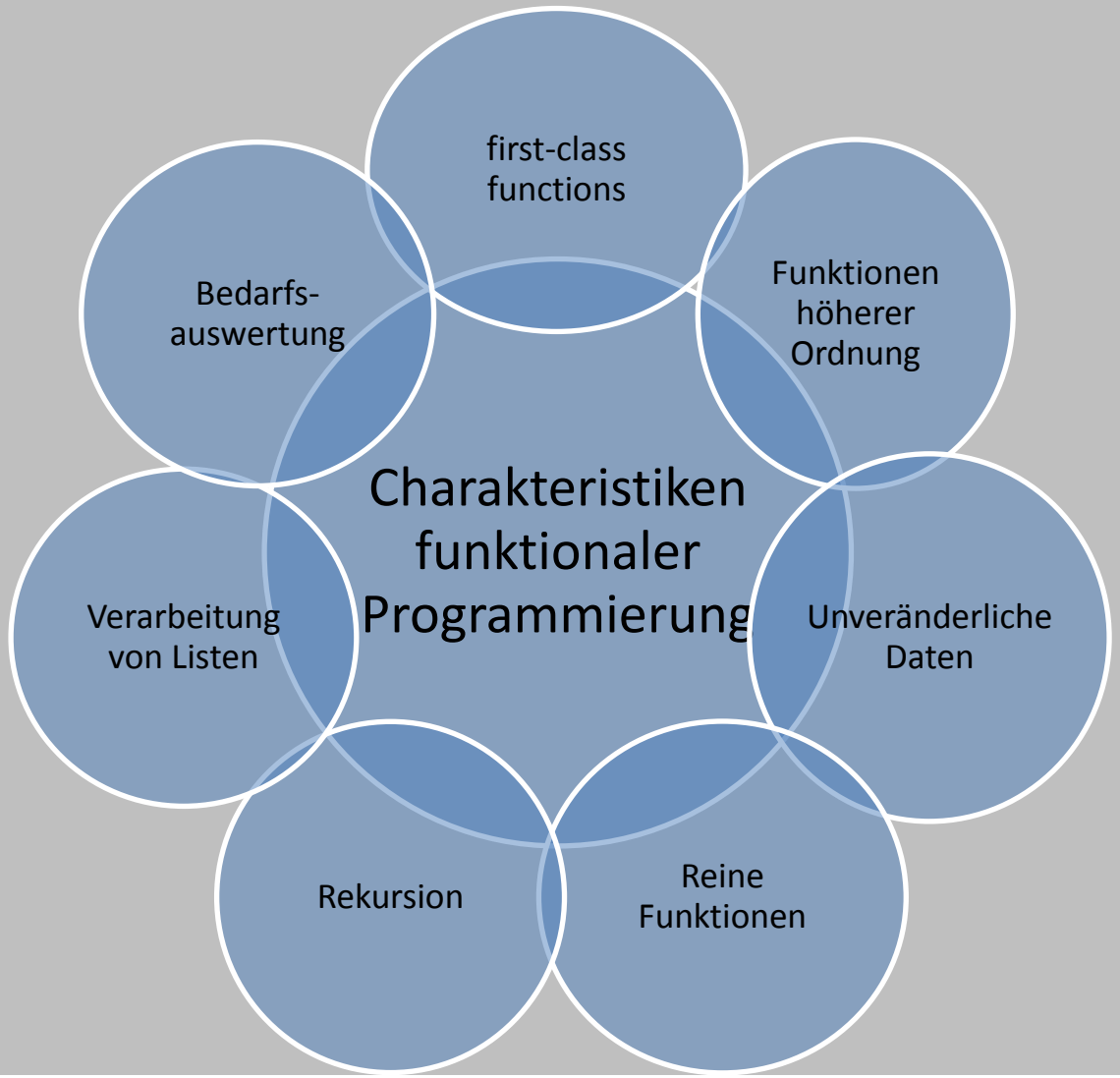
Funktionale Programmierung?

Funktionale Programmierung ist Programmierung mit mathematischen Funktionen.

- Mathematische Funktionen sind Funktionen, die jedes Mal den selben Wert zurückgeben, falls sie die gleichen Argumente erhalten. (Referenzielle Transparenz)

- Konsequenzen
  - Funktionen können keine Seiteneffekte besitzen.
  - Die Funktionsaufrufe können durch ihr Ergebnis ersetzt, umsortiert oder auf einen andere Threads verteilt werden.
  - Der Programmfluß wird durch den Datenfluß vorgegeben.

Charakteristiken



First-class functions

Funktionen sind first-class Objekte. Sie verhalten sich wie Daten.

- Funktionen können
  - als Argumente an Funktionen übergeben werden.

```
accumulate(vec.begin(),vec.end(),0,
                [ ]{ int a, int b}{ return a + b; });
```

  - von Funktionen zurückgegeben werden.

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

2011

  - an Variablen zugewiesen oder gespeichert werden.

First-class functions

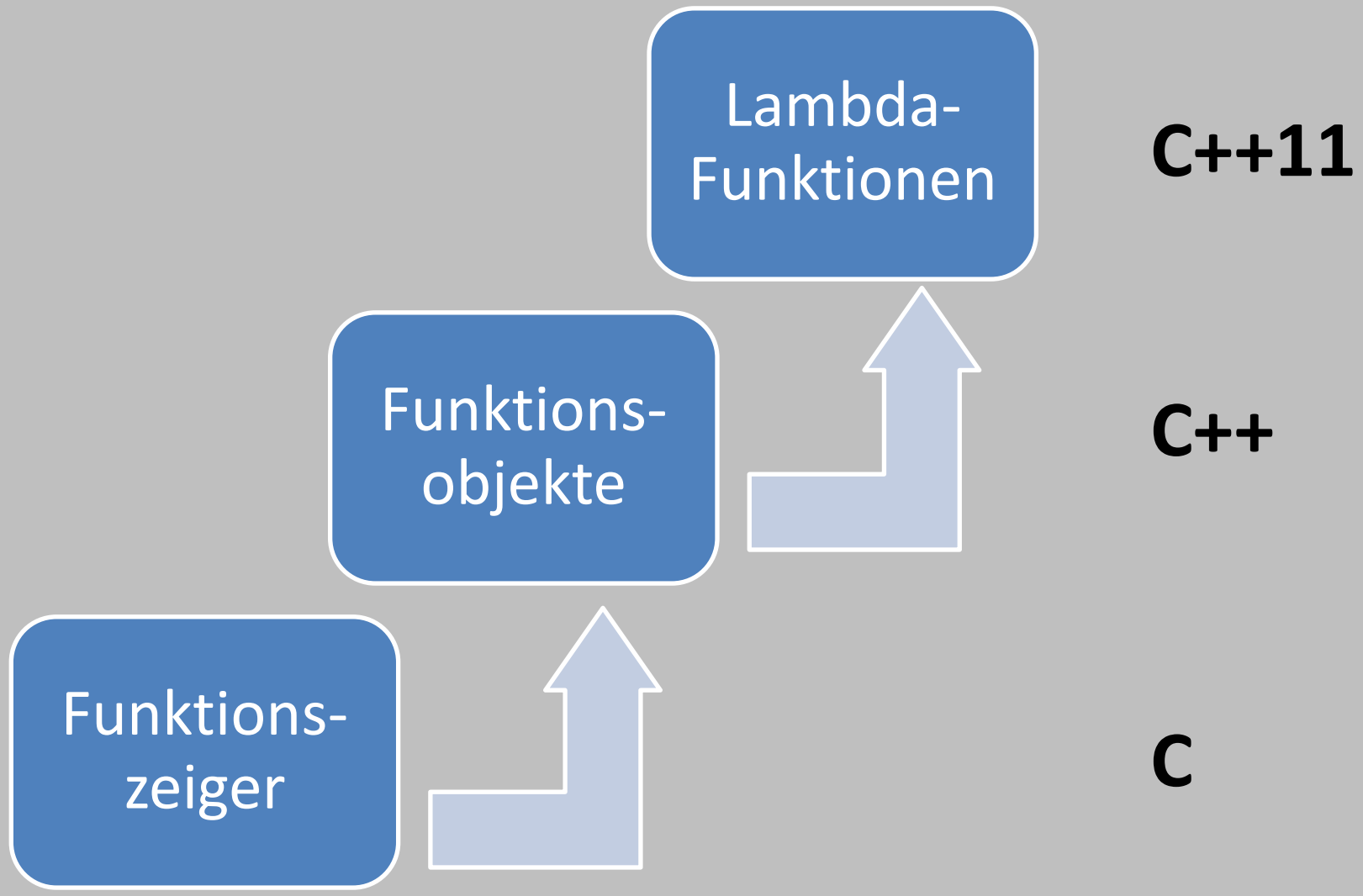
```
map<const char,function<double(double,double)> > tab;
tab.insert(make_pair('+',[](double a,double b){return a + b;}));
tab.insert(make_pair('-',[](double a,double b){return a - b;}));
tab.insert(make_pair('*',[](double a,double b){return a * b;}));
tab.insert(make_pair('/',[](double a,double b){return 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(make_pair('^',
                     [](double a,double b){return pow(a,b);}));

cout << "3.5^4.5= " << tab['^'](3.5,4.5) << endl;
```

→ **8**

→ **15.75**

→ **280.741**

First-class functions



Funktionen höherer Ordnung

Funktionen höherer Ordnung sind Funktionen, die Funktionen als Argumente annehmen oder als Ergebnis zurückgeben können

(Quelle: http.//musicantic.blogspot.de)

- Die drei Klassiker
  - map:

    Wende eine Funktion auf jedes Element einer Liste an.
  - filter:

    Entferne Elemente einer Liste.
  - reduce:

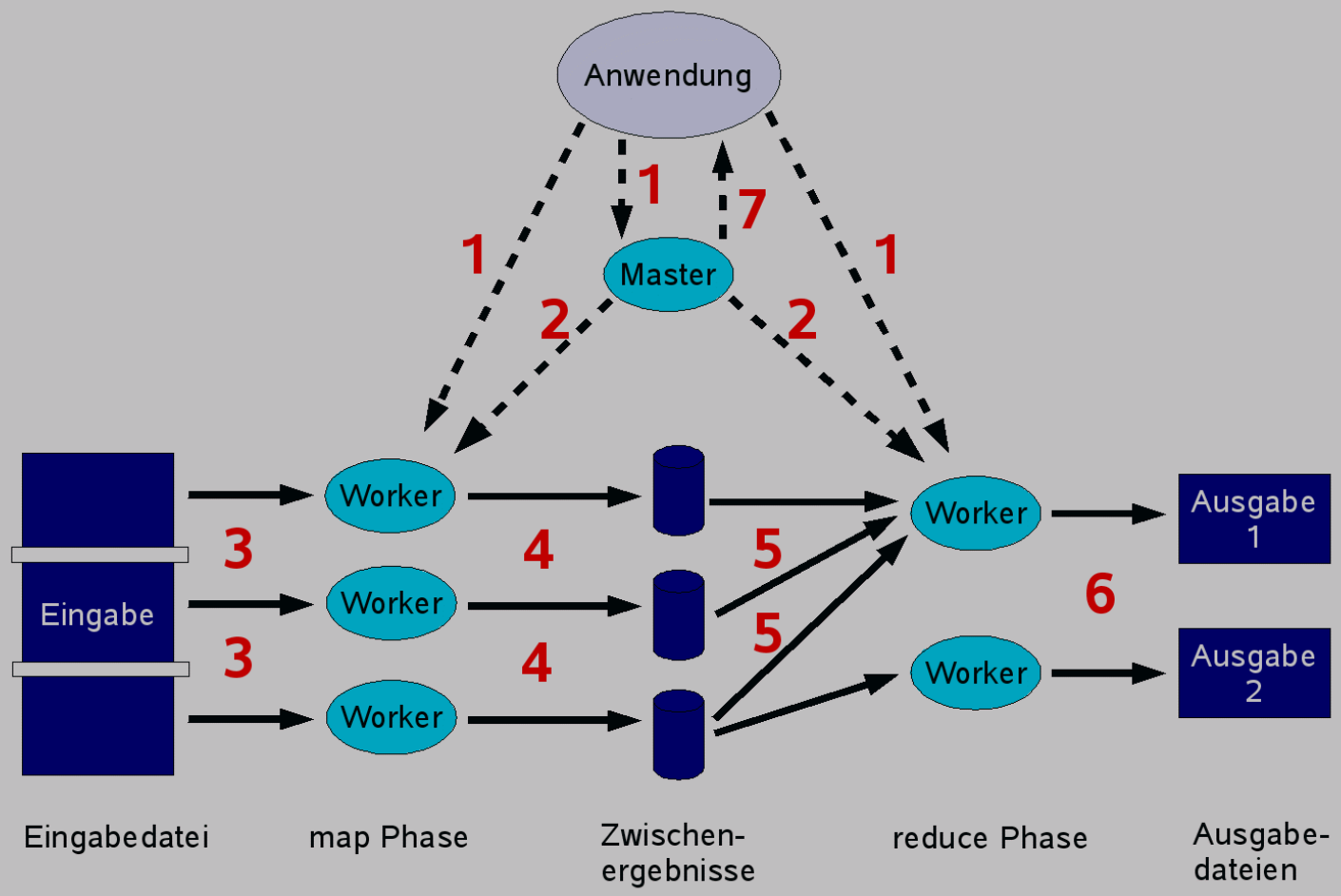
    Wende ein binäre Operation sukzessive auf die Elemente einer Liste an um diese auf einen Wert zu reduzieren.

Funktionen höherer Ordnung

Jede Programmiersprache, die Programmierung in funktionaler Art unterstützt, bietet die Funktionen map, filter und fold an.

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

Funktionen höherer Ordnung



Funktionen höherer Ordnung

Listen und Vektoren

- Haskell

```
vec= [1 . . 9]
str= ["Programming","in","a","functional","style."]
```

- Python

```
vec=range(1,10)
str=["Programming","in","a","functional","style."]
```

- C++

```
vector<int> vec{1,2,3,4,5,6,7,8,9}
vector<string>str{"Programming","in","a","functional","st
yle."}
```

➡ **Die Ergebnisse werden in Haskell oder Python Notation dargestellt.**

Funktionen höherer Ordnung

- Haskell

```
map(\a → a^2) vec
map(\a -> length a) str
```

- Python

```
map(lambda x :x*x , vec)
map(lambda x : len(x),str)
```

- C++

```
transform(vec.begin(),vec.end(),vec.begin(),
          [](int i){ return i*i; });
transform(str.begin(),str.end(),back_inserter(vec2),
          [](string s){return s.length();});
```

➡ **[1,4,9,16,25,36,49,64,81]**

[11,2,1,10,6]

Funktionen höherer Ordnung

- Haskell

```
filter(\x-> x<3 || x>8) vec
filter(\x → isUpper(head x)) str
```

- Python

```
filter(lambda x: x<3 or x>8 , vec)
filter(lambda x: x[0].isupper(),str)
```

- C++

```
auto it= remove_if(vec.begin(),vec.end(),
                   [](int i){ return !((i < 3) or (i > 8))});
auto it2= remove_if(str.begin(),str.end(),
                    [](string s){ return !(isupper(s[0])); });
```

➡ **[1,2,9]**
[“Programming”]

Funktionen höherer Ordnung

- Haskell

```
foldl (\a b → a * b) 1 vec
foldl (\a b → a ++ ":" ++ b ) "" str
```

- Python

```
reduce(lambda a , b: a * b, vec, 1)
reduce(lambda a, b: a + ":" + b, str,"")
```

- C++

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

➡ **362800**

“:Programming:in:a:functional:style.”

Funktionen höherer Ordnung

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

1 * { 1 , 2 , 3 , 4 }

1 * 1

=

1 * 2

=

2 * 3

=

6 * 4 = 24

Unveränderliche Daten

Daten sind in rein funktionalen Programmiersprachen unveränderlich.

➡ Unterscheidung von Variablen und Werten

- Konsequenzen
  - Es gibt keine
    - Zuweisungen: `x= x + 1, ++x`
    - Schleifen: `for, while , until`
  - Beim Daten modifizieren werden
    - veränderte Kopien der Daten erzeugt.
    - die ursprünglichen unveränderlichen Daten geteilt.

➡ Unveränderliche Daten sind Thread-Safe.

Unveränderliche Daten

- Umgang mit Veränderung?
  - Haskell
    - Monaden für das explizite Verändern von Daten
      - State Monade für zustandsbehaftete Berechnung
      - Mvar Monade für Synchronisation von Daten zwischen Threads
  - Scala
    - Variablen können als var (variable) oder val (value) deklariert werden.
    - Per Default sind Daten unveränderlich (val).

```
class Person( val name: String, var age: Int )
def person= new Person("Huber",33)
person.age= 34
```

  - D
    - Variablen können als immutable oder mutable (Default) deklariert werden.

Unveränderliche Daten

- Haskell

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

- C++

```
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);
}
```

Reine Funktionen

Reine Funktionen	Unreine Funktionen
Erzeugen immer dasselbe Ergebnis, wenn sie die gleichen Argumente erhalten.	Kann verschiedene Ergebnisse für dieselben Argumente erzeugen.
Besitzen keine Seiteneffekte.	Kann Seiteneffekte besitzen.
Verändern nie den Zustand.	Kann den globalen Zustand des Programmes ändern.

(Quelle: Real World Haskell)

Vorteile

- Korrektheitsbeweise sind einfacher durchzuführen.
- Refaktoring und Test ist einfacher möglich.
- Zwischenergebnisse von Funktionsaufrufe können gespeichert werden.
- Die Ausführungsreihenfolge der Funktion kann umgeordnet oder parallelisiert werden.

Reine Funktionen

Monaden sind Haskells Antwort auf die unreine Welt.

- kapseln die unreine Welt
- sind ein imperatives Subsystem
- repräsentieren eine Rechenstruktur
- definieren die Komposition von Berechnungen

➡ Funktionale Design Pattern für generische Typen

Reine Funktionen

Monaden sind abstrakte Datentypen, die einfache Typen in einen höheren Typ transformieren.

Eine Monade besteht aus

1. Typkonstruktor (*int -> [int]; int -> std::vector<int>*)
   - Erklärt den zugrundeliegeden Typ der Monade.
2. Funktionen.
   - Einheitsfunktion: Bildet den zugrunde liegenden Typ auf den Monadentyp ab. (`5 -> [5]`)
   - Funktionskompostion: Verknüpfung monadischer Werte.
3. Verknüpfung von Operationen auf Monaden mit den Regeln
   - Die Einheitsfunktion muss links- und rechtsneutral sein.
   - Die Verknüpfung von Operation muss assoziativ sein.

Reine Funktionen

Reader Monade

List Monade

I/O Monade

STM Monade

Error Monade

parsec

State Monade

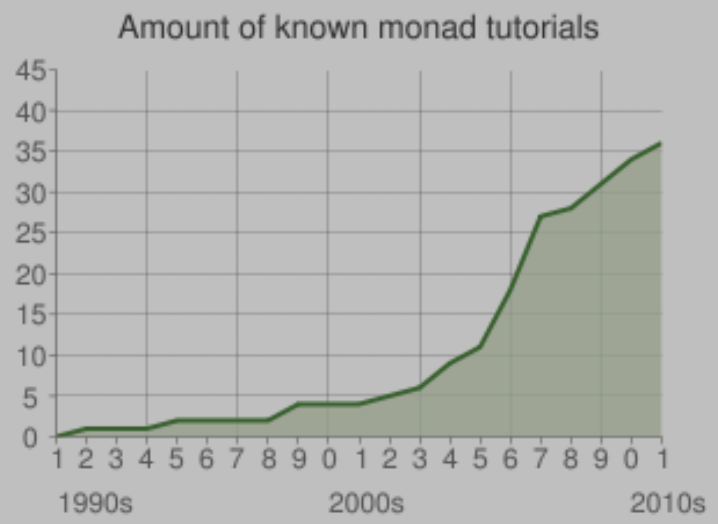
Continuation Monade

Maybe Monade

Reine Funktionen

Monaden werden wichtiger.

➡ Kontrolle von Seiteneffekten.



- Haskell
- Clojure, Scala, C#(LINQ)
- C++
  - `optional` war für C++14 geplant

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

  - `future` als Monade
    - C++17: I See a Monad in Your Future! (http://bartoszmilewski.com/)

Rekursion

Ist die Kontrollstruktur in der funktionalen Programmierung.

- Schleifen
  - benötigen eine Laufvariable: `(for int i=0; i <= 0; ++i)`

  ➡ Variablen existieren nicht in rein funktionalen Sprachen.

- Rekursion in Kombination mit dem Verarbeiten von Listen ist ein mächtiges Pattern in funktionalen Sprachen.

Rekursion

- 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;
};
```

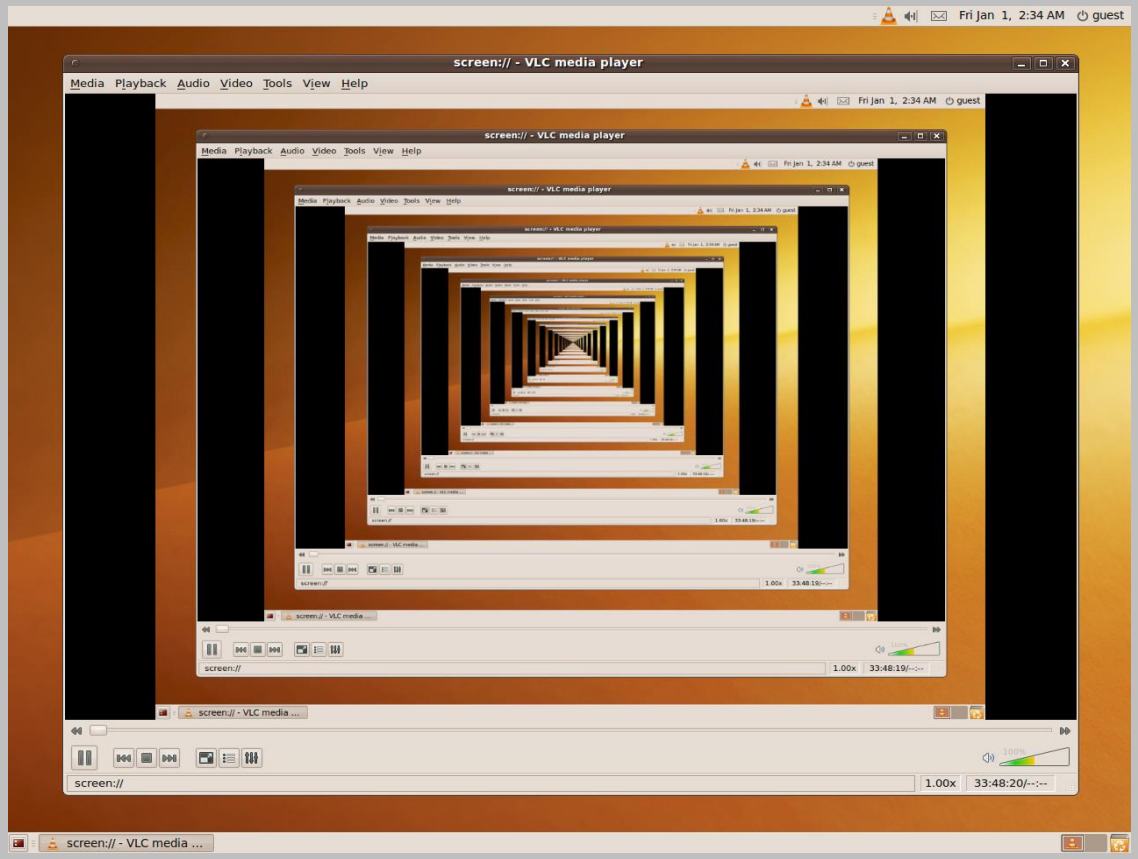
➡ **fac(5) == Fac<5>::value == 120**

Rekursion

```
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
```

Rekursion

Eine Funktion f ist endrekursiv, wenn der rekursive Aufruf die letzte Berechnung der Funktion f darstellt.



- Funktionale Sprachen unterstützen *tail call optimisation*
- rekursive Funktionsaufrufe benötigen nur einen Stackframe

Verarbeitung von Listen

List Processing (Lisp) ist typisch für funktionale Sprachen.

- Transformiere die Liste in ein andere Liste.
- Reduziere die Liste auf einen Wert.

- Funktionale Pattern für die Listenverarbeitung
  1. Prozessiere den Kopf der Liste.
  2. Prozessiere rekursive den Rest der Liste .

```
mySum []= 0
mySum (x:xs) = x + mySum xs
mySum [1,2,3,4,5] ➡ 15

myMap f [] = []
myMap f (x:xs)= f x: myMap f xs
myMap (\x → x*x)[1,2,3] ➡ [1,4,9]
```

Verarbeitung von Listen

```
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**

Verarbeitung von Listen

Die zentrale Idee: Pattern Matching

- First Match in Haskell

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

- Best Match in C++

```
template < int N1, int N2 > class Mult { … };
template < int N1 > class Mult <N1,1> { … };
template < int N1 > class Mult <N1,0> { … };
```

Bedarfsauswertung

Werte nur aus, was benötigt wird.

- Haskell ist faul (lazy evaluation)

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

- C++ ist gierig (greedy evaluation)

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

- Vorteile
  - Sparen von Zeit und Speicher
  - Verarbeitung von unendlichen Datenstrukturen

Bedarfsauswertung

- Haskell

```
successor i= i: (successor (i+1))
take 5 ( successor 10 )
```

➡ [10,11,12,13,14]

```
[1..]= [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 ...
Control-C
odds= takeWhile (< 1000) . filter odd . map (^2)
odds [1..]
```

➡ [1,9,25, … , 841,961]

- C++ besitzt den Spezialfall Kurzschlussauswertung

```
if ( true or (1/0) ) cout << "works in C++";
```

Was fehlt?

List comprehension: Syntactic Sugar für map und filter

- An mathematische Notation angelehnt

$\{ \nu * \nu \mid \nu \ \varepsilon \ N \ , \nu \ \mu o \delta \ 2 = 0 \}$: Mathematik

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

- Python

```
[n for n in range(8)]
```
→ **[0,1,2,3,4,5,6,7]**

```
[n*n for n in range(8)]
```
→ **[0,1,4,9,16,25,36,49]**

```
[n*n for n in range(8) if n%2 == 0]
```
→ **[0,4,16,36]**

Was fehlt?

Funktionskomposition: fluent interface

- Haskell

```
(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"]**

Was fehlt?

Typklassen:

- Schnittstellen für Instanzen einer Familie von Typen

- Typklasse `Eq`

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

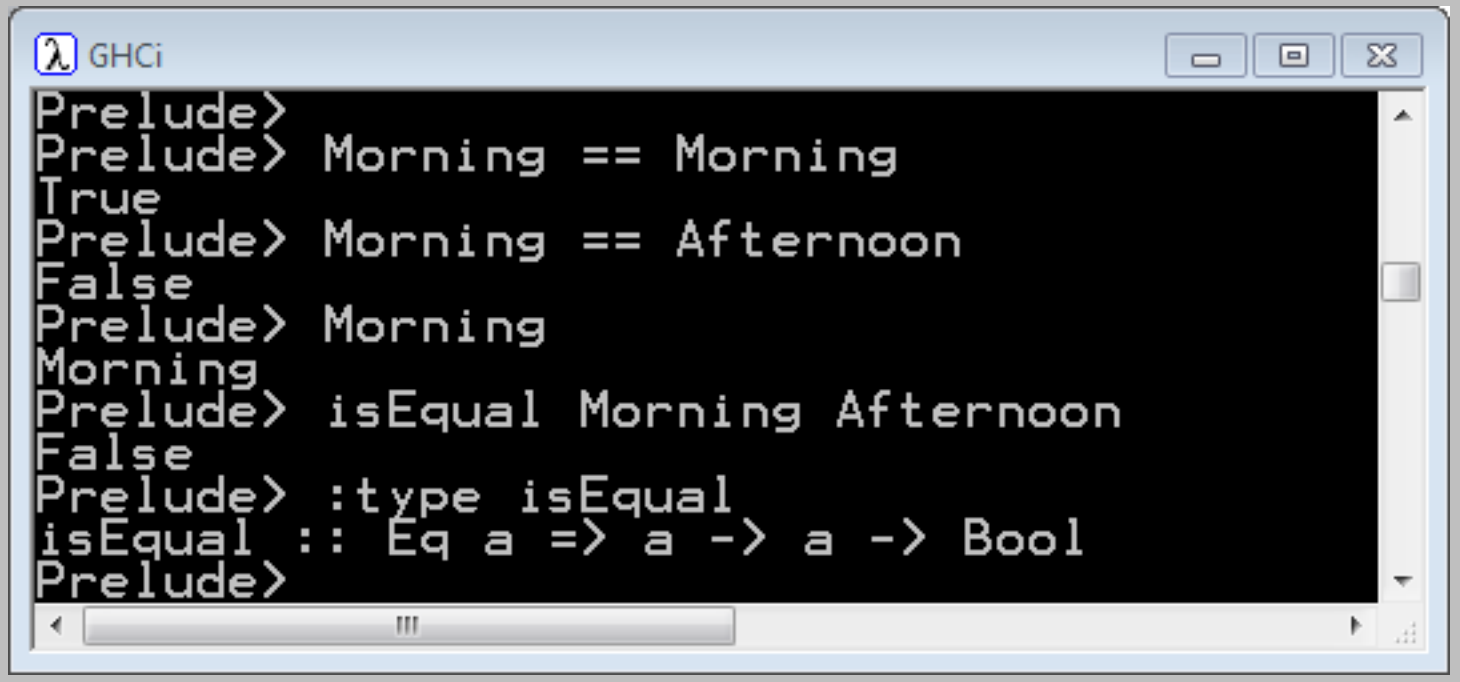
- Typ `Bool` als Instanz der Typklasse `Eq`

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

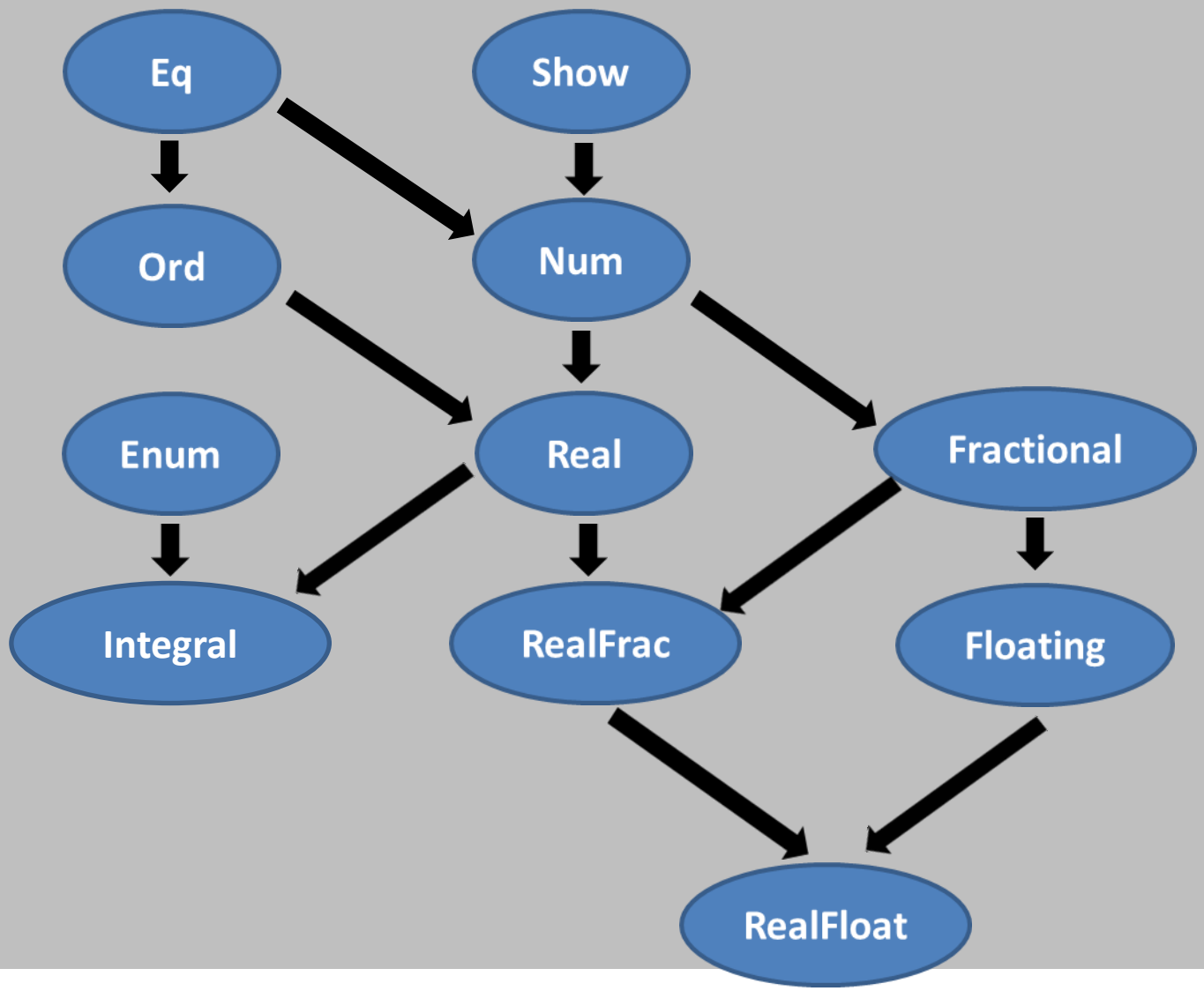
Was fehlt?

- Datentyp Day

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



Was fehlt?


Eq
Show
Ord
Num
Enum
Real
Fractional
Integral
RealFrac
Floating
RealFloat

Was fehlt?

Concepts Lite: Zusicherungen an Templates mit C++17

- Anwendung

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

Äquivalent

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

- Prädikat

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

Was fehlt?

Zusicherungen an Templates erlauben

- Mehrere Bedingungen an Templateargumente zu stellen
- Überladen von Funktions-Templates und Klassen-Template Spezialisierung

```
template<Input_iterator I>
void advance(I& iter, int n);

template<Random_access_iterator I>
void advance(I& iter, int n);
```

- Verbesserte Fehlermeldungen bei der Templateinstanziierung
- Mehrwert ohne zusätzliche Kosten für die Lauf- oder Übersetzungszeit

Was fehlt?

Höhere Threadabstraktion mit C++17

- Transaktional Memory
  - Datenbankidee der Transaktion auf Software angewandt
  - Codeblöcke
    - genügen der ACI(D)-Forderung
    - werden optimistisch ausgeführt und veröffentlich, falls die Ausgangsbedingungen noch gelten
  - Clojure, Scala und Haskell bieten Software Transaktional Memory an
- Erweiterte Unterstützung von Tasks (`async`)
  - Proposal N3721, N3865 und N3896

FRAGEN?

Ich freue mich auf Ihr Feedback!

Vielen Dank!

Rainer Grimm

Partner: intel Software embarcadero IncrediBuild BEYOND ACCELERATION ppedv