Funktionale Programmierung mit modernem C++

Rainer Grimm

Schulungen, Coaching und Technologieberatung

Funktionale Programmierung in modernem C++

- Funktional in C++
- Warum Funktional?
- Definition
- Charakteristiken
- Was fehlt (noch) in C++?

Funktionale Programmierung in modernem C++

- **Funktional in C++**
- Warum funktional?
- Definition
- Charakteristiken
- Was fehlt (noch) in C++?

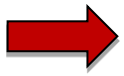
Funktional in C++

- Automatische Typableitung

```
std::vector<int> myVec;
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() << std::endl;
```



2011

Funktional in C++

Partial function application (Currying)

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

Haskell Curry

Moses Schönfinkel

```
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

Funktional in C++

Generische Programmierung

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

Funktionale Programmierung in modernem C++

- Funktional in C++
- **Warum Funktional?**
- Definition
- Charakteristiken
- Was fehlt (noch) in C++?

Warum Funktional?

- Effizientere Nutzung der Standard Template Library

```
std::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 (globalen Variablen)
  - Kurz und prägnante Programmierung
  - Verständlichere Programmierung

Funktionale Programmierung in modernem C++

Funktional in C++

Warum Funktional?

Definition

Charakteristiken

Was fehlt (noch) in C++?

Definition

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).

- Mathematische Funktionen werden auch reine Funktionen genannt.

Funktionale Programmierung in modernem C++

Funktional in C++

Warum Funktional?

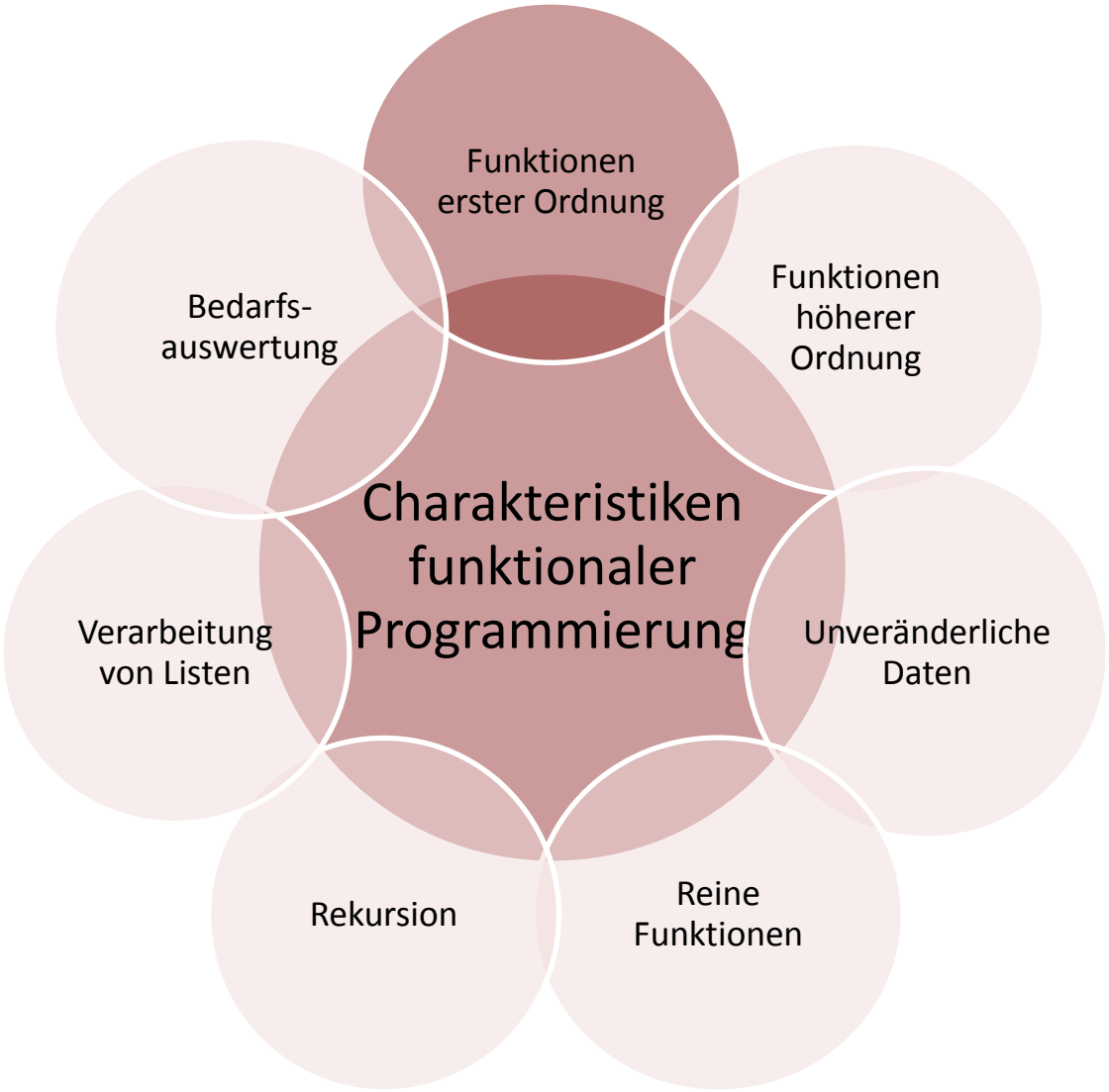
Definition

Charakteristiken

Was fehlt (noch) in C++?

Charakteristiken

Charakteristiken



Funktionen erster Ordnung

Funktionen sind Funktionen erster Ordnung (first class objects). ➡ Sie verhalten sich wie Daten.

- Funktionen können
  - als Argumente an Funktionen übergeben werden.
  - von Funktionen zurückgegeben werden.
  - an Variablen zugewiesen oder gespeichert werden.

Funktionen erster Ordnung

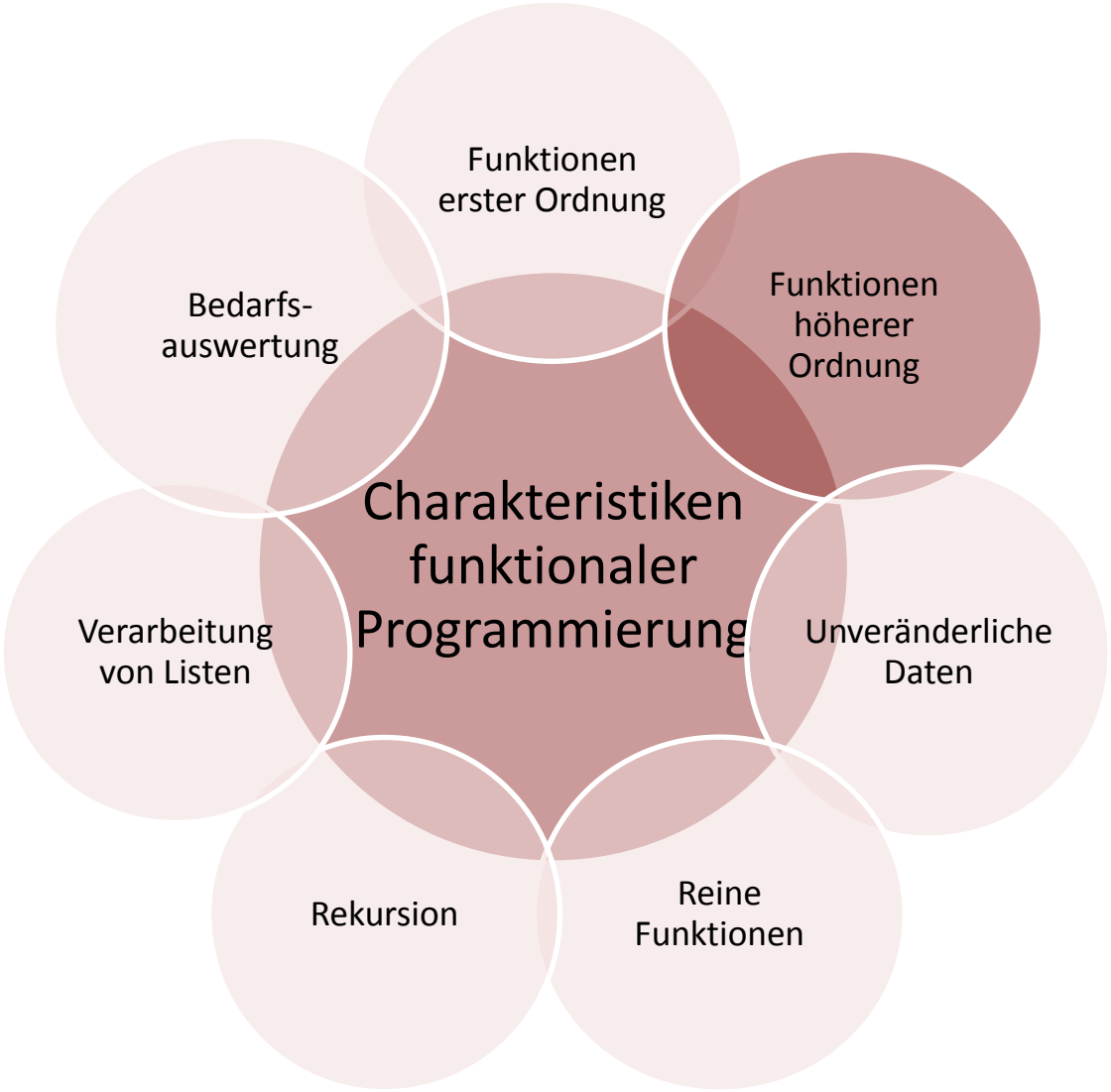
```
map<const char,function<double(double, double)>> tab;
tab.insert(make_pair('+', [](double a, double b){ return a + b;} ));
tab.insert(make_pair('-', [](double a, double b){ return a - b;} ));
tab.insert(make_pair('*', [](double a, double b){ return a * b;} ));
tab.insert(make_pair('/', [](double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

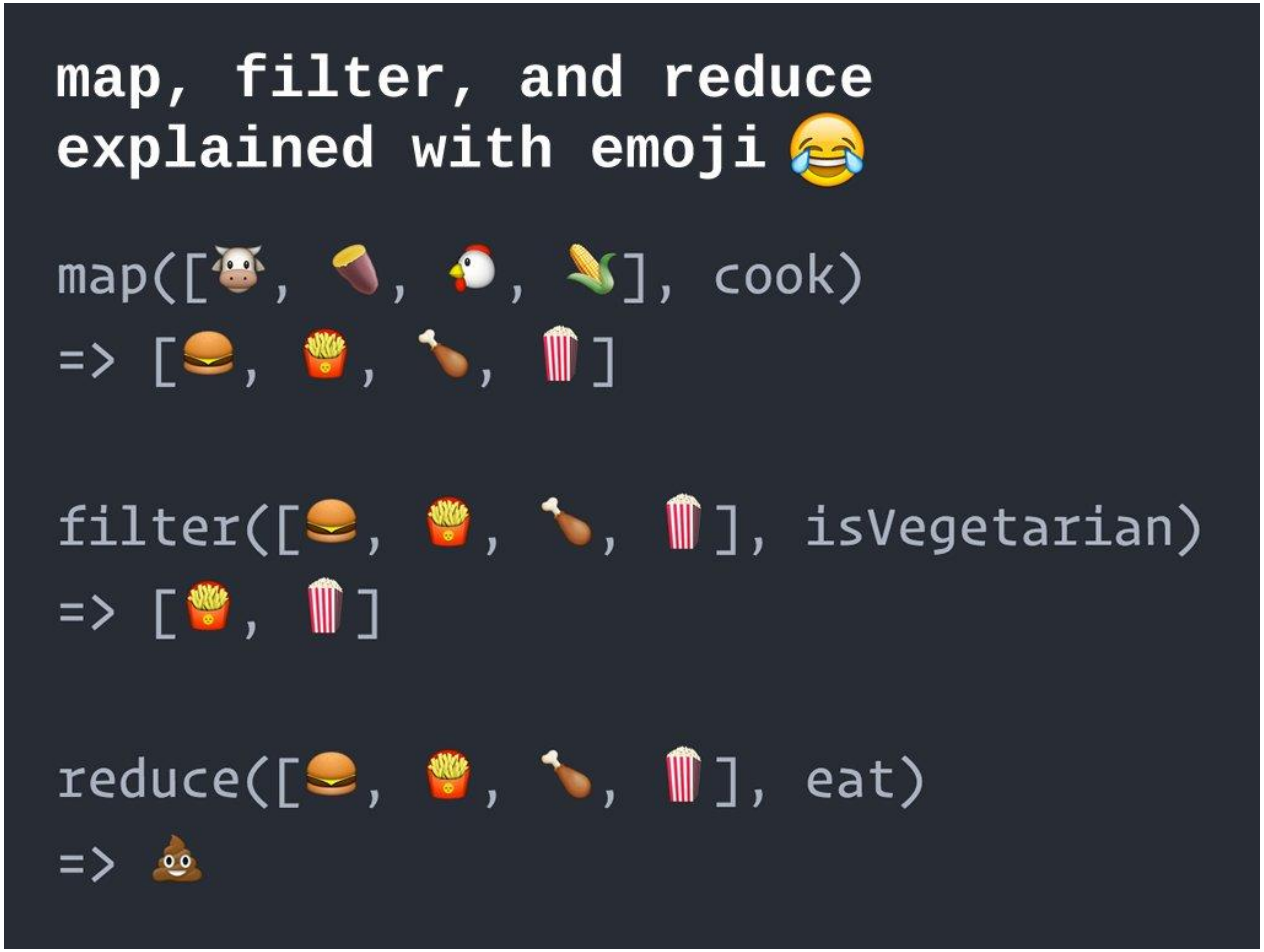
280.741

Charakteristiken



Funktionen höherer Ordnung

Funktionen, die Funktionen als Argumente annehmen oder als Ergebnis zurückgeben können.



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`

- **`map`**, **`filter`** und **`fold`** sind drei mächtige Funktionen, die in vielen Kontexten verwendet werden

  ⇨ map + reduce = MapReduce

Funktionen höherer Ordnung

Listen und Vektoren

- Haskell

```
vec= [1 .. 9]
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",
"style."}
```

Die Ergebnisse werden in Haskell Notation dargestellt.

Funktionen höherer Ordnung: map

- Haskell

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

- Haskell

```
filter(\x-> x<3 || x>8) vec
filter(\x -> isUpper(head x)) 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: reduce

- Haskell

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

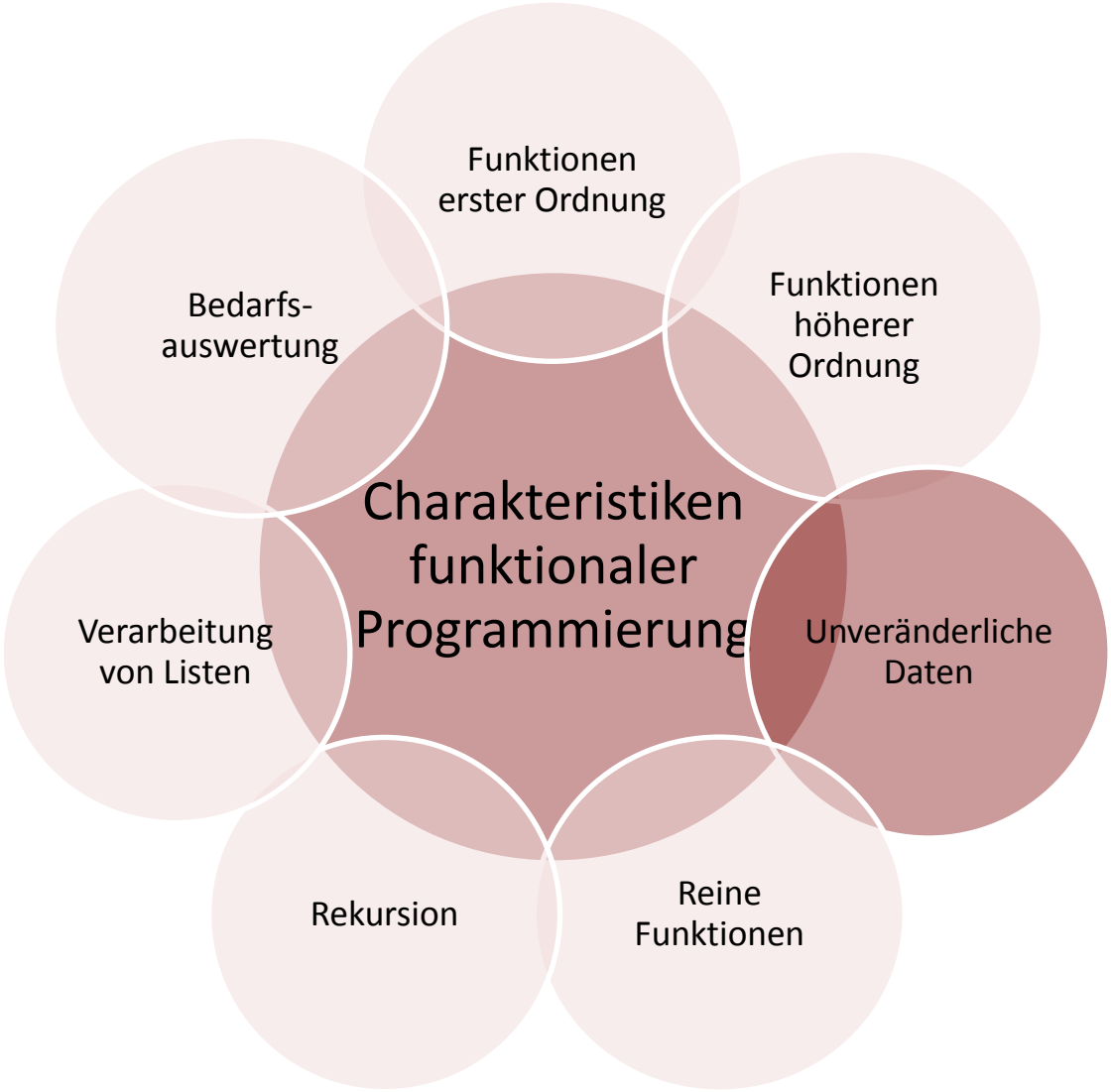
- C++

```
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."

Charakteristiken



Unveränderliche Daten

Daten sind in rein funktionalen Programmiersprachen unveränderlich.

- 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-sicher.

Unveränderliche Daten

- Haskell

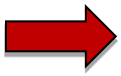
```
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y < x] ++ [x] ++ qsort [y | y <- xs, y >= x]
```

- 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);
}
```

Unveränderlichen Daten

Der Umgang mit unveränderlichen Daten setzt auf Disziplin.



Verwende `const`, Template Metaprogrammierung oder konstante Ausdrücke (`constexpr`).

- `const` Daten

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

- konstante Ausdrücke

```
constexpr int value= 1;
```

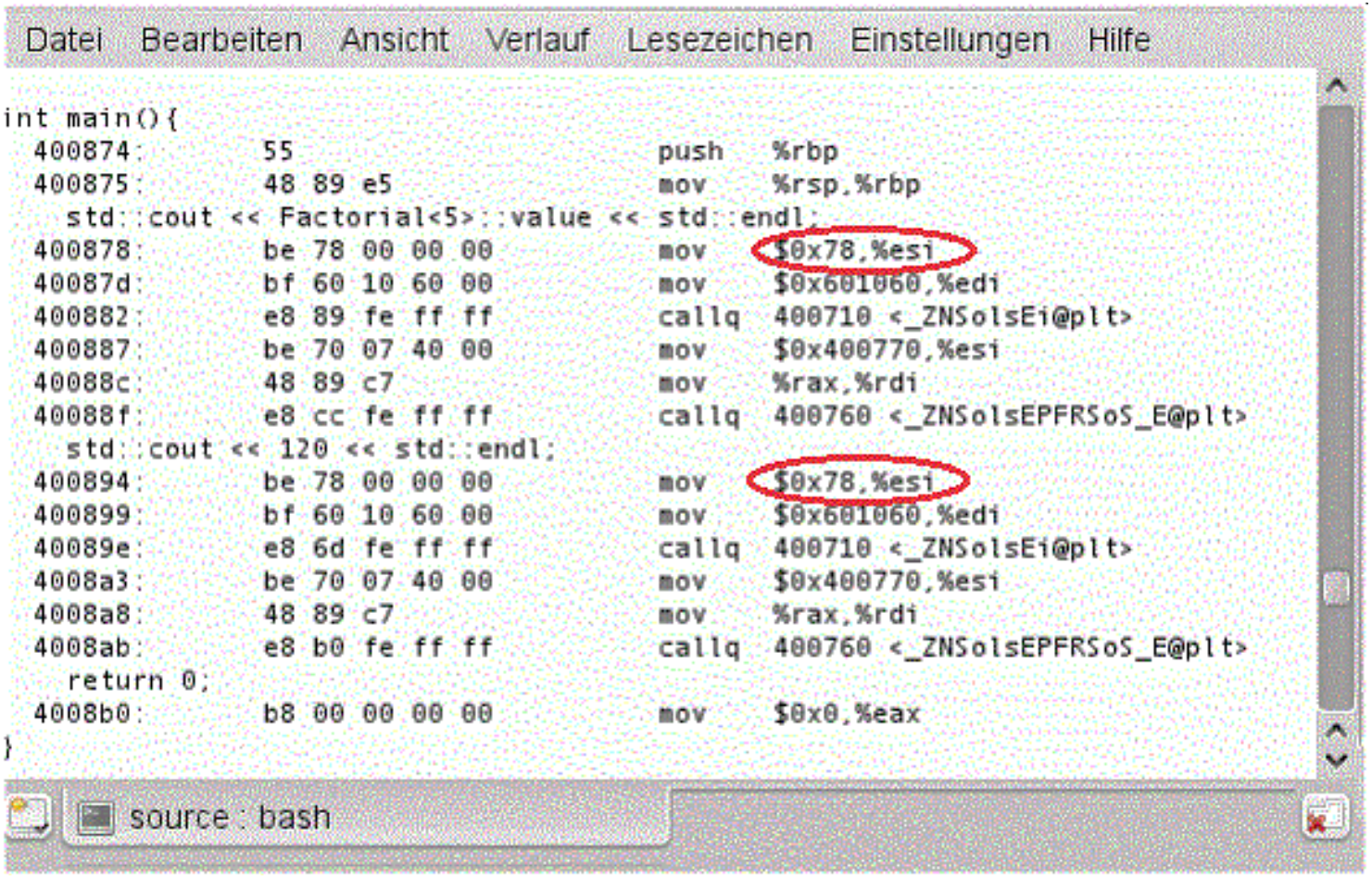
- Template Metaprogrammierung
  - Ist ein rein funktionale Sprache in der imperative Sprache C++.
  - Wird zu Compilezeit ausgeführt.
  - Zur Compilezeit gibt es nur unveränderliche Daten.

Unveränderliche Daten

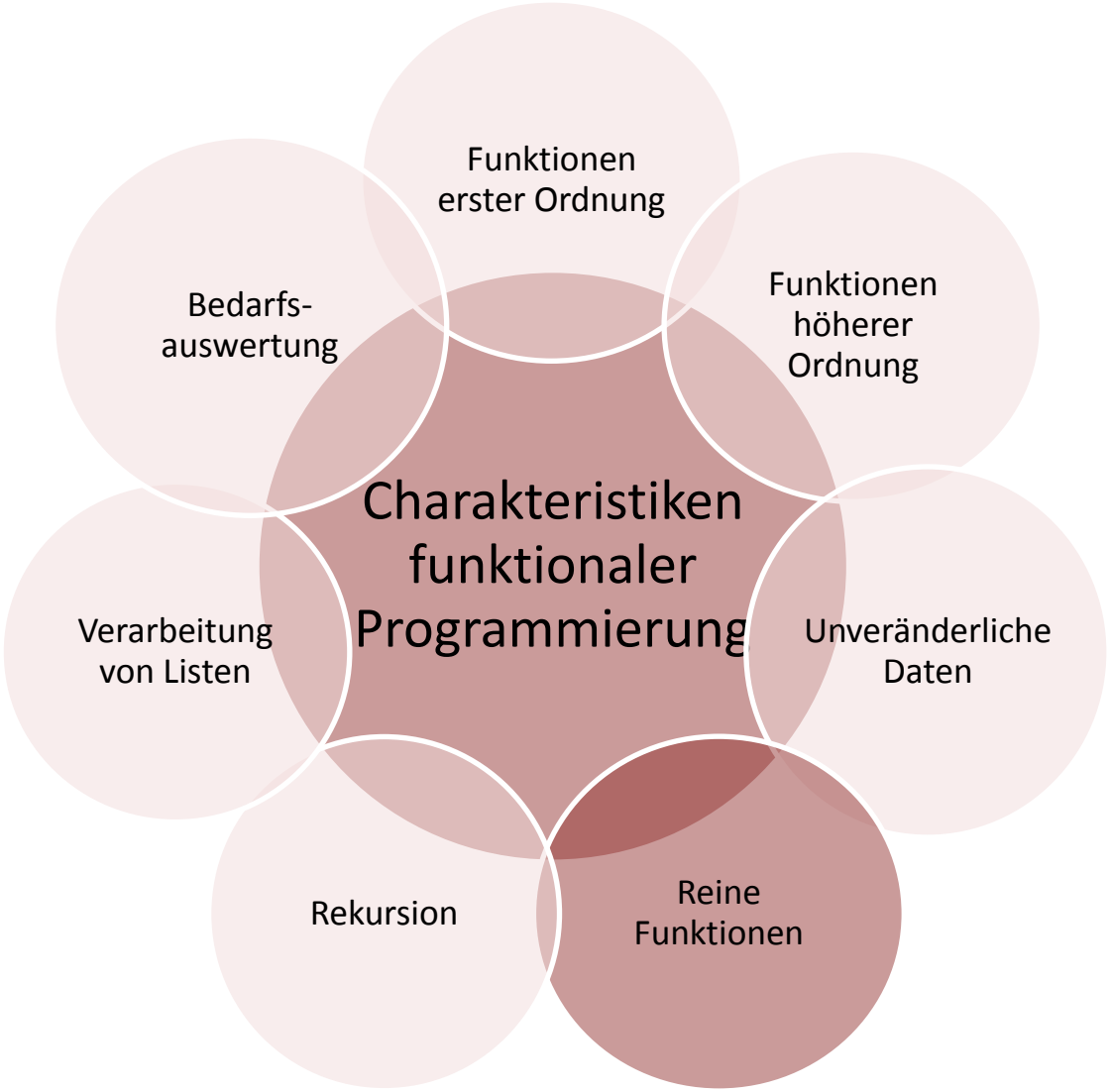
```
template <int N>
struct Factorial{
    static int const value= N * Factorial<N-1>::value;
};
template <>
struct Factorial<1>{
    static int const value = 1;
};

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
```

Unveränderliche Daten



Charakteristiken



Reine Funktionen

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

Vorteile

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

Reine Funktionen

Arbeiten mit reinen Funktionen setzt auf Disziplin

➡ Verwende normale Funktionen, Metafunktionen oder `constexpr` Funktionen.

- Funktionen

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

- Metafunktionen

```
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;
};
```

Reine Funktionen

- `constexpr` Funktionen

```
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
}
```

Reine Funktionen

Monaden sind Haskells Antwort auf die unreine Welt.

- Eine Monade
  - kapselt die unreine Welt.
  - ist ein imperatives Subsystem.
  - repräsentieren eine Rechenstruktur.
  - definieren die Komposition von Berechnungen

➡ Funktionale Design Pattern für generische Typen

Reine Funktionen

Reader Monade

List Monade

I/O Monade

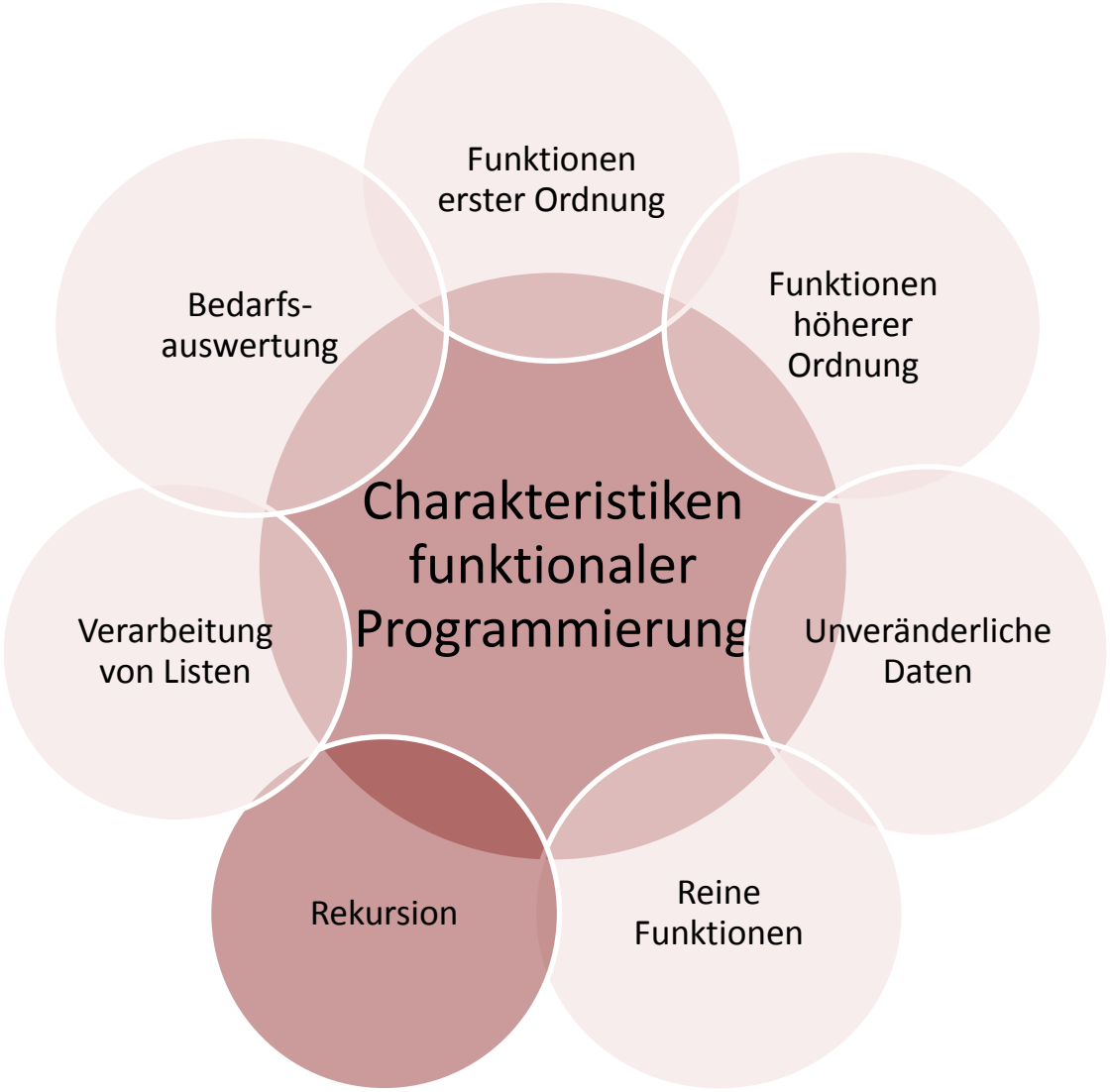
STM Monade

Error Monade

State Monade

Maybe Monade

Charakteristiken



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

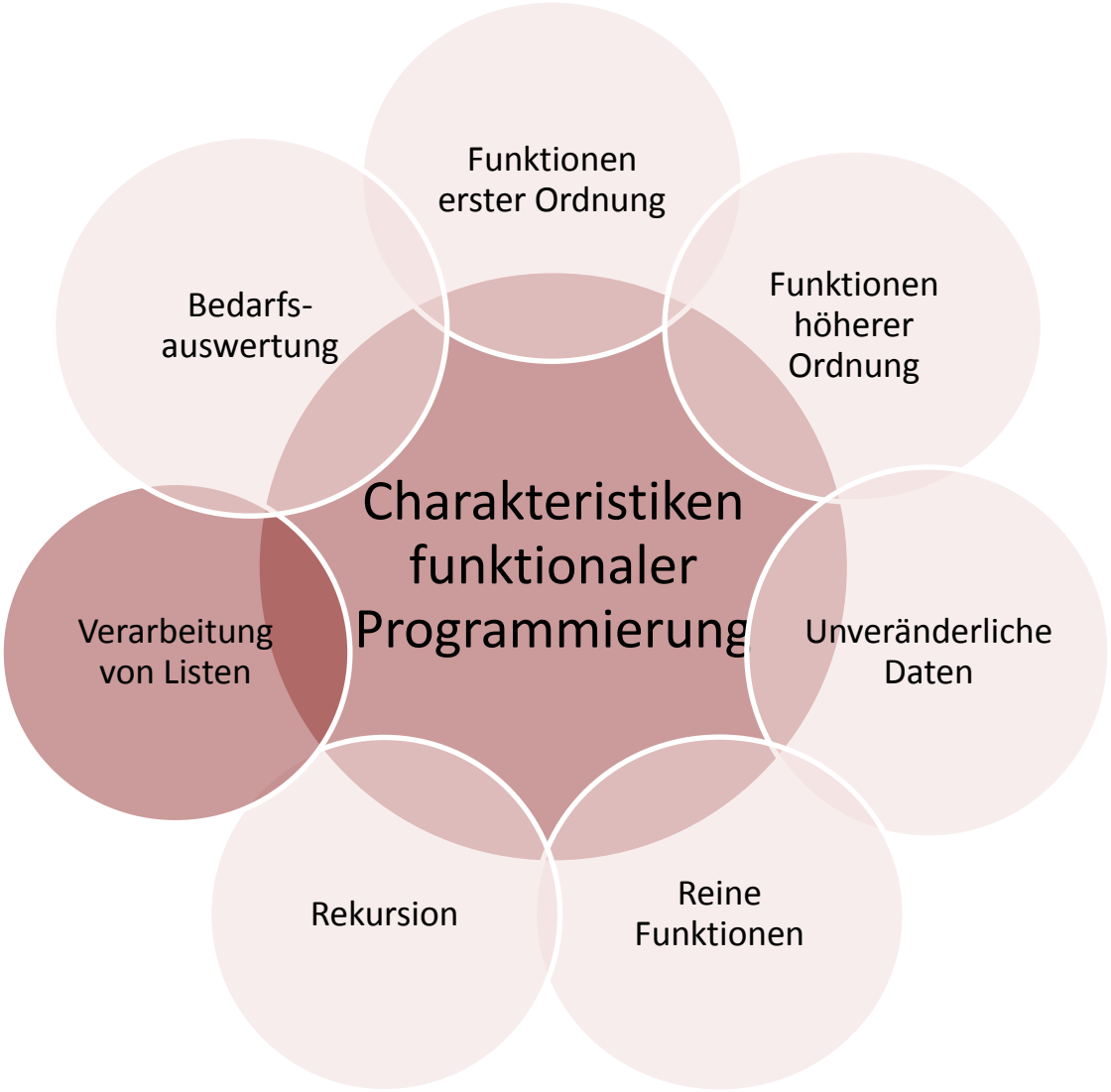
```
fac 0= 1
fac n= n * fac (n-1)
```

- 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**

Charakteristiken



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

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

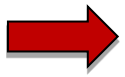
```
mult n 0 = 0
mult n 1 = n
mult n m = (mult n (m - 1)) + n
```

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

Verarbeitung von Listen

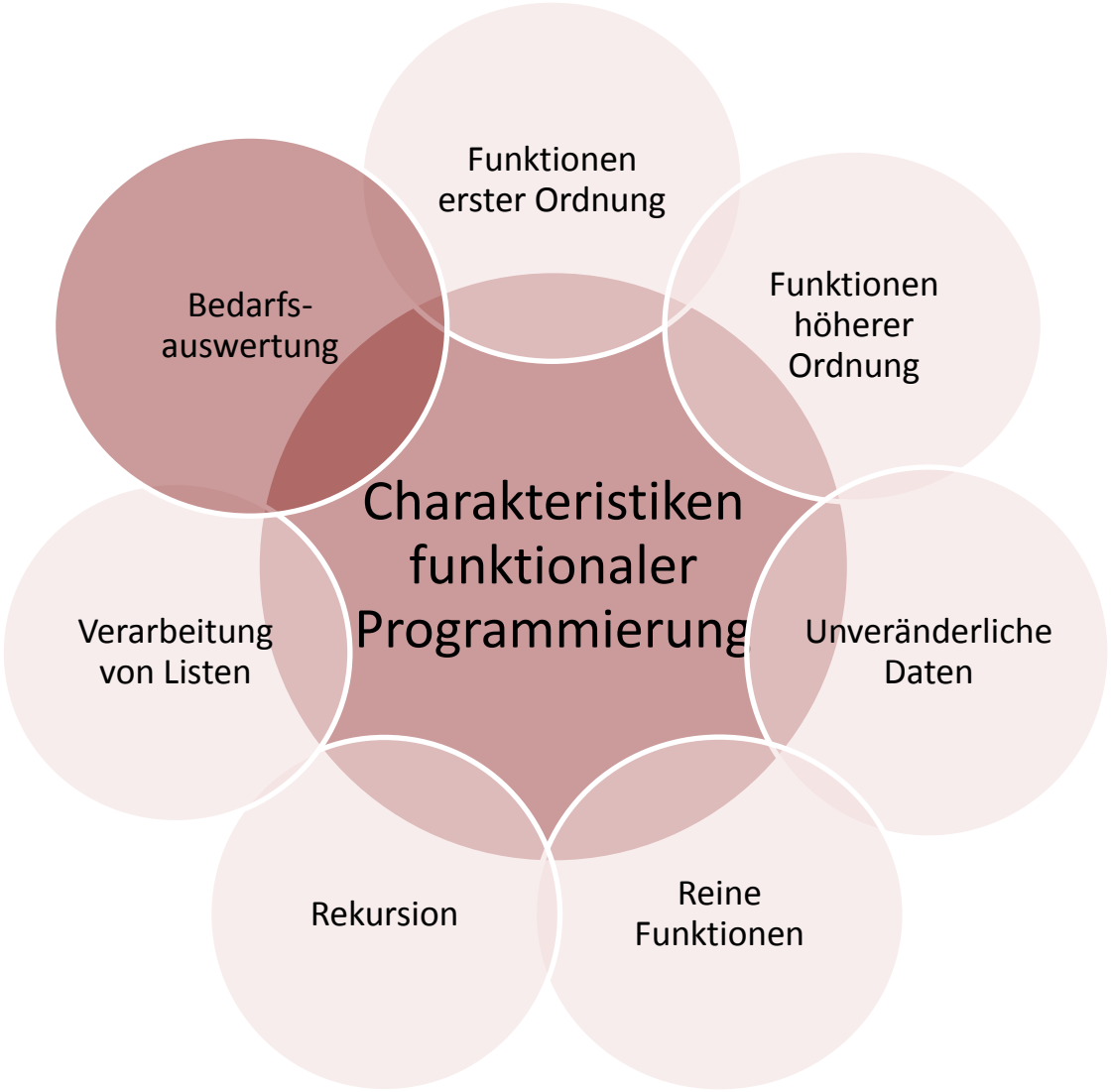
- Best match in C++

```
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

Charakteristiken



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..]
take 5 (successor 10)
```

➡ **[10, 11, 12, 13, 14]**

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

➡ **[1, 9, 25, ... , 841, 961]**

- Spezialfall in C++: Kurzschlussauswertung

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

Funktionale Programmierung in modernem C++

Funktional in C++

Warum Funktional?

Definition

Charakteristiken

Was fehlt (noch) in C++?

Was fehlt (noch) in C++?

List comprehension: Syntactic Sugar für **`map`** und **`filter`**

- An mathematische Notation angelehnt

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

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

- Haskell

  `[n | n <- [1..6] ]` ⇨ **[1, 2, 3, 4, 5, 6]**

  `[n*n | n <- [1..6] ]` ⇨ **[1, 4, 9, 16, 25, 36]**

  `[n*n | n <- [1..6], n `mod` 2 == 0]` ⇨ **[4, 16, 36]**

Was fehlt (noch) in C++?

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 (noch) in C++?

Mit C++20 erhält C++ die Ranges-Library von Eric Niebler

- Algorithmen arbeiten auf dem ganzen Container

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

std::sort(v.begin(), v.end());        // classic

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

- Ranges unterstützen Bedarfsauswertung, Funktionskomposition und Range Comprehension

```
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..]
```

Was fehlt (noch) in C++?

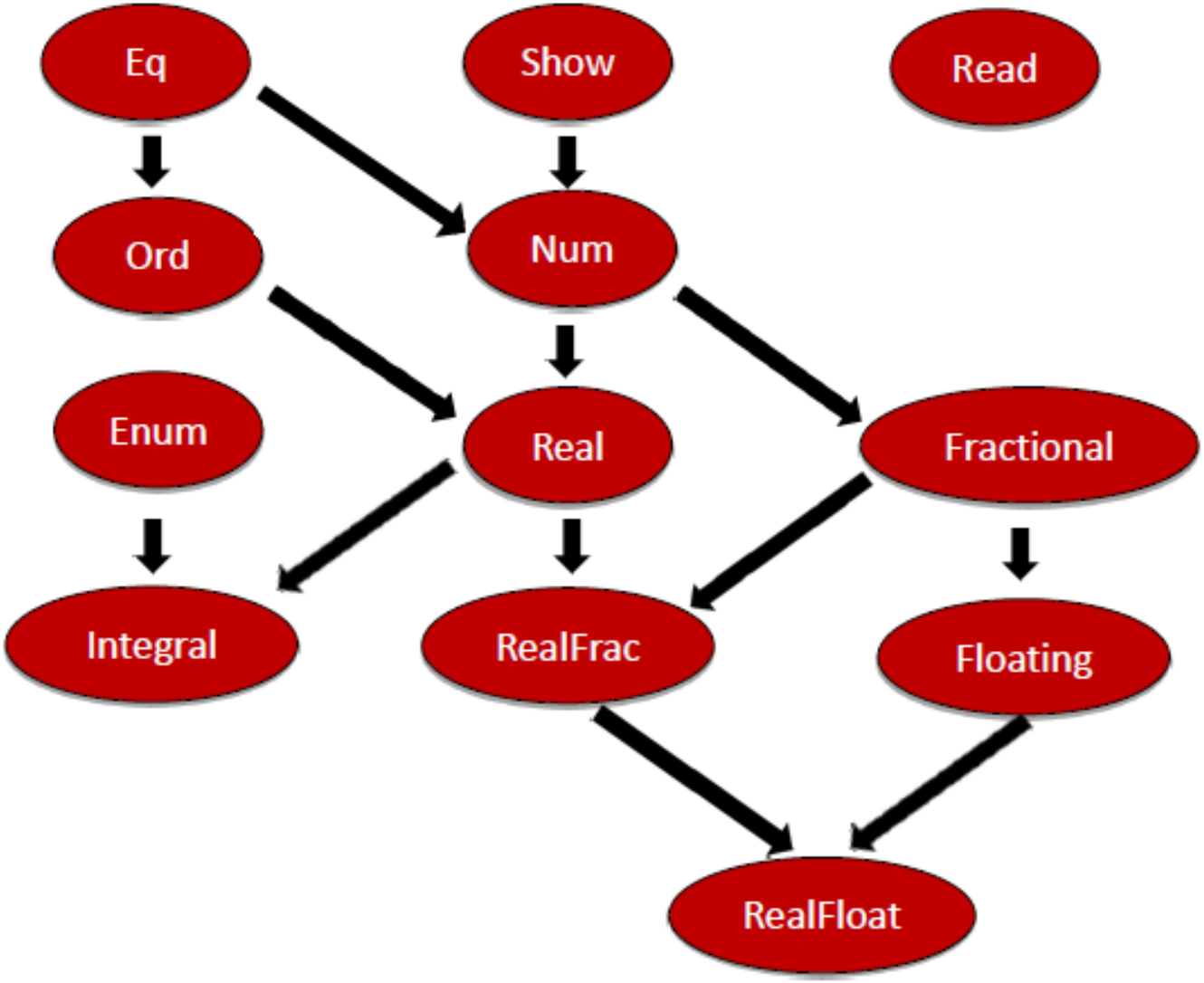
- Typklassen in Haskell für ähnliche Typen.
  - Typeklasse `Eq`

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

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

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

Was fehlt (noch) in C++?



Was fehlt (noch) in C++?

- Concepts Lite können auf alle Templates angewandt werden.
  - Funktions-Templates

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

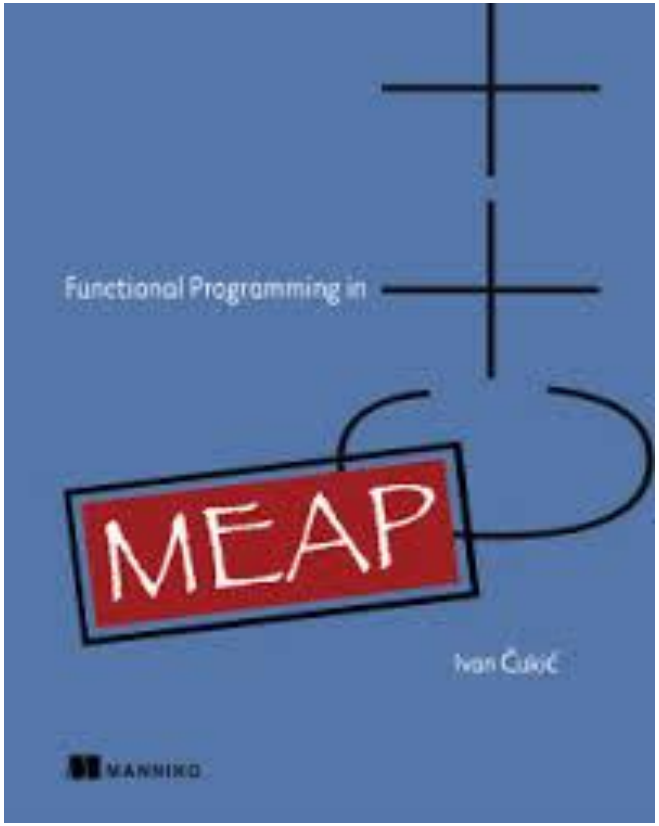
  - Klassen-Templates

```
template<Object T>
class vector;

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

Funktionale Programmierung in modernem C++

- Funktional in C++
- Warum Funktional?
- Definition
- Charakteristiken
- Was fehlt (noch) in C++?

Modernes C++



[Functional Programming in C++]

How to improve your C++ programs using functional techniques

- Von Ivan Čukić
- Werde sechs Gutscheine auf www.ModernesCpp.com verlosen
- Erscheinungsdatum März 2017

Modernes C++

- Schulungen, Coaching und Technologieberatung durch Rainer Grimm
  - www.ModernesCpp.de
- Blogs zu modernem C++
  - www.grimm-jaud.de (Deutsch)
  - www.ModernesCpp.com (Englisch)
- Kontakt
  - @rainer_grimm (Twitter)
  - schulung@grimm-jaud.de