



PRIMEDIC™

Saves Life. Everywhere.

Funktionale Programmierung mit C++



Überblick



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



Funktional in C++

- Automatische Typableitung

```
for ( auto v: myVec ) std::cout << v << " " ;
```

- Lambda-Funktionen

```
int a= 2000, b= 11;
```

```
auto sum= std::async( [=]{return a+b;} );
```

- Partielle Funktionsanwendung

`std::function` **und** `std::bind`

Lambda-Funktionen **und** `auto`



Haskell Curry



Moses Schönfinkel



Funktional in C++

- Funktionen höherer Ordnung

```
std::vec<int> vec{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

- Generische Programmierung (Templates)
- Standard Template Library
- Template Metaprogrammierung



Alexander Stepanov



Warum Funktional?

- Effizientere Nutzung der Standard Template Library

```
std::accumulate(v.begin(), v.end(), [] (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
 - Kurze 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 ihre Ergebnis ersetzt, umsortiert oder auf einen anderen Thread verteilt werden.
 - Der Programmfluß wird durch den Datenfluß vorgegeben.



Charakteristiken



Funktionen erster Ordnung

- Funktionen sind Funktionen erster Ordnung (first class objects)
➔ Funktionen sind wie Daten.
- Funktionen können
 - als Argument an Funktionen übergeben werden.
 - von Funktionen zurückgegeben werden.
 - an Variablen zugewiesen oder gespeichert werden.



Funktionen erster Ordnung

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

```
tab.insert(std::make_pair('+', [] (double a,double b) {return a + b;}));
```

```
tab.insert(std::make_pair('-', [] (double a,double b) {return a - b;}));
```

```
tab.insert(std::make_pair('*', [] (double a,double b) {return a * b;}));
```

```
tab.insert(std::make_pair('/', [] (double a,double b) {return a / b;}));
```

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

 **8**

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

 **15.75**

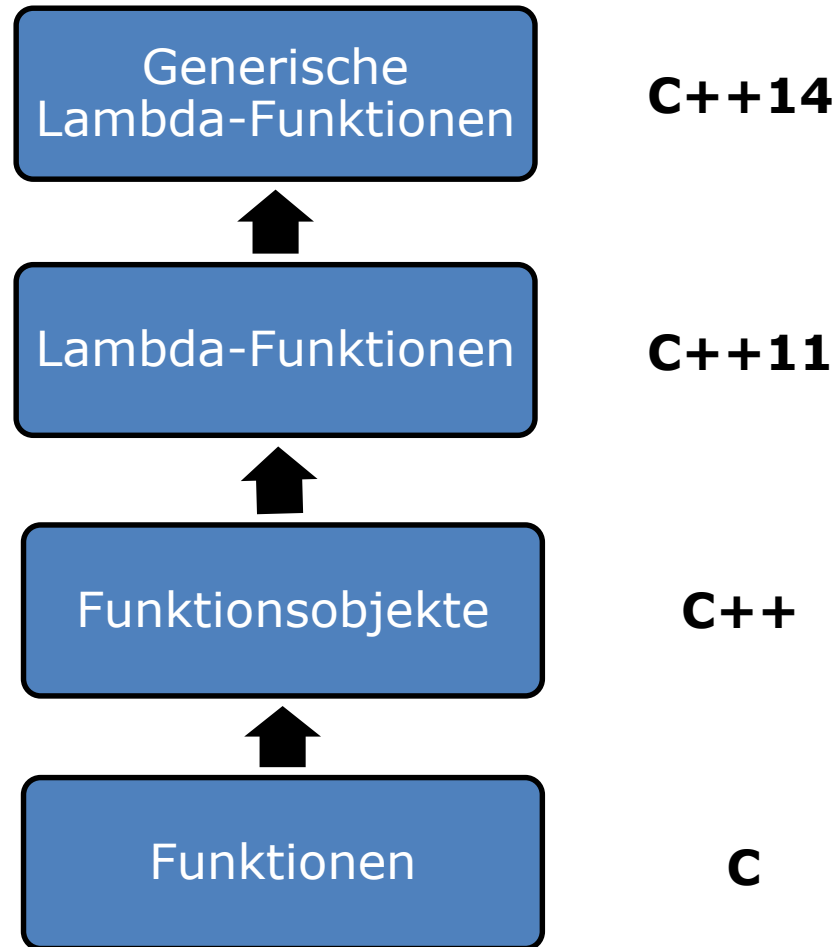
```
tab.insert(std::make_pair('^',  
    [] (double a,double b) {return std::pow(a,b);}));
```

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

 **280.741**



Funktionen erster Ordnung



Funktionen höherer Ordnung

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

Die Klassiker:

- **map:**
 - Wende eine Funktion auf jedes Element einer Liste an.
- **filter:**
 - Entferne Elemente einer Liste.
- **fold:**
 - Wende eine binäre Operation sukzessive auf die Elemente einer Liste an um diese auf einen Wert zu reduzieren.



(Quelle: <http://musicantic.blogspot.de>, 2012-10-16)



PRIMEDICTM
Saves Life. Everywhere.

Funktionen höherer Ordnung

Jede Programmiersprache, die Programmierung in funktionale 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."]
```

- Python

```
vec=range(1,10)
```

```
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."}
```

 Die Ergebnisse werden in Haskell oder Python Notation dargestellt.



Funktionen höherer Ordnung

- Haskell

```
map(\a → a*a) vec
```

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

- Python

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

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

- C++

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

```
std::transform(str.begin(), str.end(), back_inserter(vec2),  
               [](std::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= std::remove_if(vec.begin(),vec.end(),  
                        [](int i){ return !((i < 3) or (i > 8)) });
```

```
auto it2= std::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++

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



362800

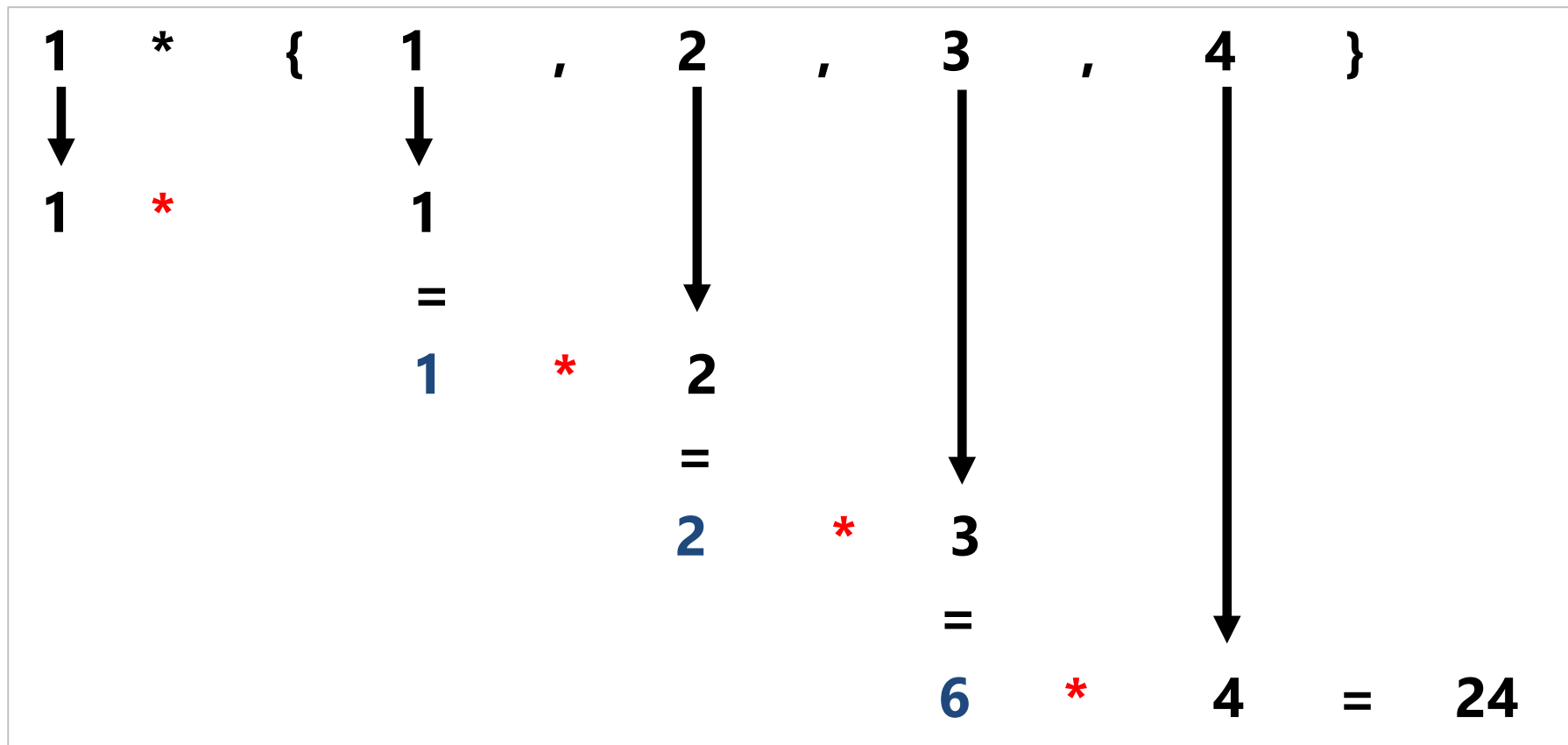
“:Programming:in:a:functional:style.”



Funktionen höherer Ordnung

```
std::vector<int> v{1,2,3,4};
```

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



Unveränderliche Daten

Daten sind in rein funktionalen Programmiersprachen unveränderlich.

➔ Unterscheidung von Variablen und Werten

- Konsequenzen
 - Es gibt keine
 - Zuweisung: `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

- 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[(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.	Können verschiedene Ergebnisse für dieselben Argumente erzeugen.
Besitzen keine Seiteneffekte.	Können Seiteneffekte besitzen.
Verändern nie den Zustand.	Können den globalen Zustand des Programmes ändern.

- Vorteile
 - Korrektheitsbeweise sind einfacher durchzuführen.
 - Refactoring und Test ist einfacher möglich.
 - Ergebnisse von Funktionsaufrufen können gespeichert werden.
 - Die Ausführungsreihenfolge von Funktionen kann umgeordnet oder parallelisiert werden.



Reine Funktionen

Monaden sind Haskells Antwort auf die unreine Welt.

- Eine Monade
 - kapselt die unreine Welt.
 - sind imperative Subsysteme.
 - repräsentieren Rechenstrukturen.
 - definieren die Komposition von Berechnungen.



➔ Funktionale Design Pattern für die generischen Typen.



Reine Funktionen

List Monade

Reader Monade

I/O Monade

STM Monade

Error Monade

State Monade

Continuation Monade

Maybe Monade

**P
a
r
s
e
c**



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

$$\begin{aligned}\text{Fac}<5>::\text{value} &= \\ &= 5 * \text{Fac}<4>::\text{value} \\ &= 5 * 4 * \text{Fac}<3>::\text{value} \\ &= 5 * 4 * 3 * \text{Fac}<2>::\text{value} \\ &= 5 * 4 * 3 * 2 * \text{Fac}<1>::\text{value} \\ &= 5 * 4 * 3 * 2 * 1 * \text{Fac}<0>::\text{value} \\ &= 5 * 4 * 3 * 2 * 1 * 1 \\ &= 120\end{aligned}$$



Verarbeitung von Listen

LIST Processing ist typisch für funktionale Sprachen.

- Transformiere eine Liste in ein andere Liste.
 - Reduziere eine Liste auf einen Wert.
-
- Das funktionale Pattern für die Listenverarbeitung:
 1. Prozessiere den Kopf der Liste.
 2. Prozessiere rekursiv 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
```

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

- 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]**

```
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..]
```

 **[1,9,25, ... , 841,961]**

- Spezialfall in C++: Kurzschlussauswertung

```
if ( true or (1/0) ) std::cout << "short circuit evaluation in C++\n";
```



Was fehlt?

- List comprehension: Syntactic sugar für `map` und `filter`


- An die mathematische Notation angelehnt

$\{ v*v \mid v \in \mathbb{N}, v \bmod 2 = 0 \}$: 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
```

```
sortTitLen= sortBy(comparing length) . filter isTit . words
```

```
sortTitLen "A Sentence full of Titles ."
```

 **["A", "Titles", "Sentence"]**





PRIMEDIC™

Saves Life. Everywhere.

Rainer Grimm

www.primedic.com

phone +49 (0)741 257-245

rainer.grimm@spacelabs.com

