PRIMEDIC™

Saves Life. Everywhere.

Functional Programming in C++

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

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

- Lambda-functions

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

- Partial function application

  `std::function` and `std::bind`

  `lambda-functions` and `auto`

Haskell Curry

Moses Schönfinkel

Functional in C++

- Higher-order functions

```
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**

- Generic Programming (Templates)
- Standard Template Library
- Template Metaprogramming

Alexander Stepanov

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
  - 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

Characteristics of functional programing

First-class functions

Higher-order functions

Immutable data

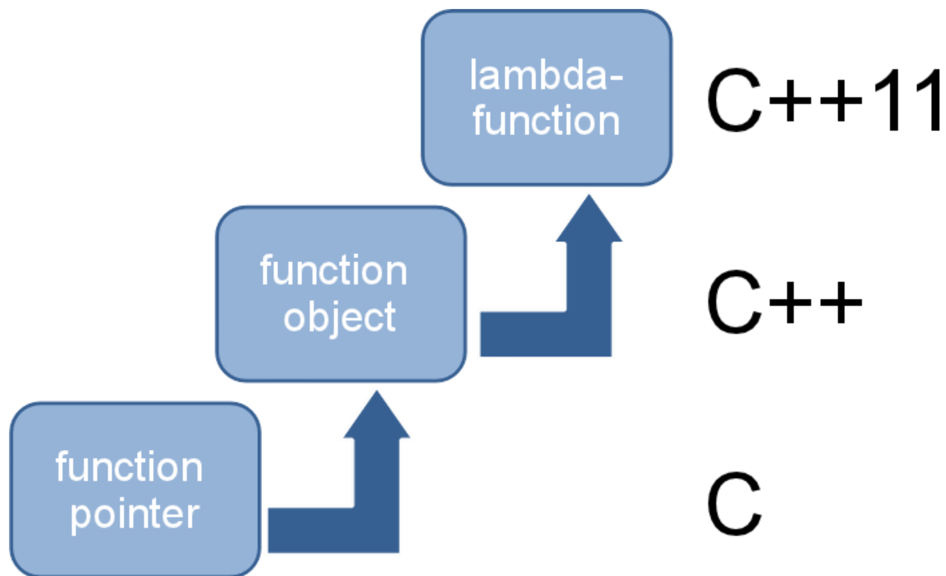
Pure functions

Recursion

Manipulation of lists

Lazy evaluation

First-class functions



- First-class functions are first-class citizens.
  - ➡ Functions are like data.
- Functions
  - can be passed as arguments to other functions.
  - can be returned from other functions.
  - can be assigned to variables or stored in a data structure.

First-class functions

```
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;
```

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

```
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;
```

PRIMEDIC™
Saves Life. Everywhere.

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.

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

- **map**, **filter** and **fold** are 3 powerful functions which are applicable in many cases.

Higher-order functions

- Lists and vectors:
  - 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."}
```

The results will be displayed in Haskell or Python notation.

Higher-order functions: map

- 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++

```
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(); });
```

Higher-order functions: filter

- 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”]

Higher-order functions: fold

- 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.”

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;});
```

1 * { 1 , 2 , 3 , 4 }

1 * 1

=

1 * 2

=

2 * 3

=

6 * 4 = 24

Immutable data

Data are immutable in pure functional languages.

→ Distinction between variables and values

- 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 are 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);
}
```

Pure functions

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

- Advantages
  - Correctness of the code is easier to verify.
  - Refactor and test is possible
  - Saving results of pure function invocations.
  - Reordering pure function invocations or performing them on other threads.

Pure functions

- Monads are the Haskell solution to deal with the impure world.
- A Monad
  - encapsulates the impure world.
  - is a imperative subsystem in.
  - represents a computation structure.
  - define the composition of computations.
- Examples:
  - I/O monad for input and output
  - Maybe monad for computations that can fail
  - List monad for computations with zero or more valid answers
  - State monad for stateful computation
  - STM monad for software transactional memory

Recursion

- Recursion is the control structure in functional programming.
- A loop (for int i=0; i <= 0; ++i) needs a variable i.

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
= 120

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;
};
```

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 (x) of the list
  2. Recursively processing the tail (xs) of the list => Go to step 1).

```
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]

List processing

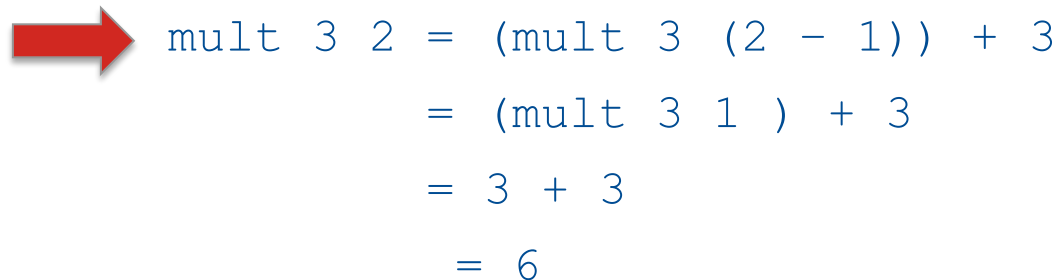
```
template<int ...> struct mySum;

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 behind list processing is 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++11

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

Lazy Evaluation

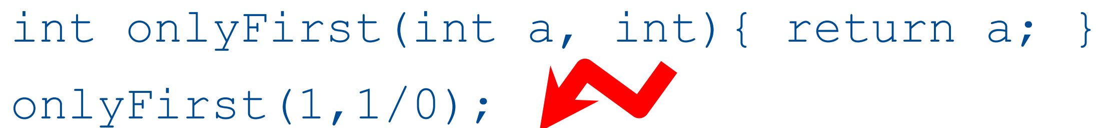
- Evaluate only, if necessary.

  - 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))
```

```
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
```

- Special case in C++: short circuit evaluation

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

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

- Python

```
[n for n in range(8)]
```

```
[n*n for n in range(8)]
```

```
[n*n for n in range(8) if n%2 == 0]
```

What's missing?

Function composition: 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"]

PRIMEDIC™

Saves Life. Everywhere.

Rainer Grimm

www.primedic.com

phone +49 (0)741 257-245

rainer.grimm@primedic.com