Functional Programming in C++11
An Overview

- Programming in a functional style
- Why functional programming?
- What is functional programming?
- Characteristics of functional programming
  - first-class functions
  - higher-order functions
  - pure functions
  - recursion
  - list processing
  - lazy evaluation
- What's missing?
Programming in a functional style

- Automatic type deduction with
  - `auto` and `decltype`
- Support for `lambda-functions`
  - closures
  - functions as data
- Partial function application
  - `std::function` and `std::bind`
  - `lambda-functions` and `auto`
- Higher-order functions in the algorithms of the STL
- *List manipulation* with `variadic templates`
- Pattern matching with full and partial template specialisation
- Lazy evaluation with `std::async`
- Constrained templates (concepts) will be part of C++1y.
Why functional programming?

- **Standard Template Library (STL)**
  - more effective use with *lambda-functions*
    ```cpp
    accumulate(vec.begin(), vec.end(),
                [](int a, int b) { return a + b; });
    ```

- **Template Programming**
  - recognizing functional patterns
    ```cpp
    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
    ```cpp
    for (auto v : vec) cout << v << " " << endl;
    ```
What is 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).
  - Functions are not allowed to have side effects.
  - The function invocation can be replaced by the result.
  - The optimizer is allowed to rearrange the function invocations or he can perform the function invocation on a different thread.
  - The program flow will be driven by the data dependencies and not by the sequence of instructions.
Characteristics of functional programming

- first-class functions
- lazy evaluation
- higher-order functions
- manipulation of lists
- pure functions
- recursion
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: dispatch table

```cpp
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;  // 8

cout << "3.5*4.5= " << tab['*'](3.5, 4.5) << endl;  // 15.75

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

cout << "3.5^4.5= " << tab['^'](3.5, 4.5) << endl;  // 280.741
```
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**.

<table>
<thead>
<tr>
<th></th>
<th>Python</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>map</td>
<td>std::transform</td>
</tr>
<tr>
<td>filter</td>
<td>filter</td>
<td>std::remove_if</td>
</tr>
<tr>
<td>fold*</td>
<td>reduce</td>
<td>std::accumulate</td>
</tr>
</tbody>
</table>

- **map**, **filter** and **fold** are 3 powerful functions which are applicable in many cases.
  - $\text{map} + \text{reduce} = \text{MapReduce}$
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++11:
    - vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9}
    - 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 \to \ a^2) \ vec
    map(\a \to \ \text{length} \ a) \ str
  
  - Python:
    
    map(lambda x : \ x^2 , \ vec)
    map(lambda x : \ \text{len} \ (x) , \ str)
  
  - C++11:
    
    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(); });

- Results: [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
    ```
  - **Python:**
    
    ```
    filter(lambda x: x<3 or x>8 , vec)
    filter(lambda x: x[0].isupper(),str)
    ```
  - **C++11:**
    
    ```
    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])); });
    ```

- Results: [1,2,9] and [“Programming”]
Higher-order functions

- **fold**
  - **Haskell:**
    
    $\text{foldl } (\lambda a \ b \rightarrow a \times b) \ 1 \ \text{vec}$
    
    $\text{foldl } (\lambda a \ b \rightarrow a \ +\ + \ "\::\" \ +\ + \ b) \ "\ " \ \text{str}$
  
  - **Python:**
    
    $\text{reduce}(\lambda a, b: a \times b, \ \text{vec}, \ 1)$
    
    $\text{reduce}(\lambda a, b: a + b, \ \text{str},"\")$
  
  - **C++11:**
    
    $\text{accumulate}(\text{vec.begin()},\text{vec.end()},1,$
    
    $[\lambda \text{a, b}{ \text{return a*b; } }]);$
    
    $\text{accumulate}(\text{str.begin()},\text{str.end()},\text{string}(""),$
    
    $[\lambda \text{a, string b}{ \text{return a+":"+b; } });$

  - **Results:** 362800 and “:Programming:in:a:functional:style.”
Pure functions

- Pure versus impure functions (from the book Real World Haskell)

<table>
<thead>
<tr>
<th>pure functions</th>
<th>impure functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Always produces the same result when given the same</td>
<td>May produce different results for the same parameters.</td>
</tr>
<tr>
<td>parameters.</td>
<td></td>
</tr>
<tr>
<td>Never have side effects.</td>
<td>May have side effects.</td>
</tr>
<tr>
<td>Never alter state.</td>
<td>May alter the global state of the program, system, or world.</td>
</tr>
</tbody>
</table>

- Pure functions are isolated. The program is easier to
  - reason about.
  - refactor and test.

- Great opportunity for optimization
  - 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 in pure Haskell.
  - is a imperative subsystem in Haskell.
  - is a structure which represents computation.
  - has to define the composition of computations.
- Examples:
  - I/O monad for dealing with input and output
  - Maybe monad for computations that can fail
  - List monad for computations with zero or more valid answers
  - State monad for representing stateful computation
  - STM monad for software transactional memory
Recursion

\[
\text{Fac<5>::value = } \\
= 5 \times \text{Fac<4>::value} \\
= 5 \times 4 \times \text{Fac<3>::value} \\
= 5 \times 4 \times 3 \times \text{Fac<2>::value} \\
= 5 \times 4 \times 3 \times 2 \times \text{Fac<1>::value} \\
= 5 \times 4 \times 3 \times 2 \times 1 \times \text{Fac<0>::value} \\
= 120
\]

- Loops:
  - Recursion is the control structure.
  - A loop (for int i=0; i <= 0; ++i) needs a variable i.
    - Mutable variables are not known in functional languages like Haskell.
  - 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;
  };
  ```

  Result: fac(5) == Fac<5>::value == 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 \((x)\) of the list
  2) Recursively processing the tail \((xs)\) of the list => Go to step 1.
- Examples:

  ```
  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

```cpp
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
```

- You do not really want to implement `myMap` with variadic templates.
  
  (http://www.linux-magazin.de/Heft-Abo/Ausgaben/2011/01/C/%28offset%29/2)
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
  - Example:
    mult 3 2 = (mult 3 (2 - 1)) + 3
    = (mult 3 1) + 3
    = 3 + 3
    = 6
  - 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

- Lazy evaluation (non-strict evaluation) evaluates the expression only if needed.
  - Haskell is lazy, as the following works
    ```
    length [2+1, 3*2, 1/0, 5-4]
    ```
  - C++ is eager, but the following works
    ```
    template <typename... Args>
    void mySize(Args... args) {
      cout << sizeof...(args) << endl;
    }
    mySize("Rainer",1/0);
    ```

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

▪ **Examples:**

    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]

▪ **Special case: short circuit evaluation**

    if ( true or (1/0) )cout << "short circuit evaluation in C++\n";
What's missing?

- **List comprehension:**
  - Syntactic sugar of the sweetest kind with map and filter
  - Examples:
    
    \[
    \begin{align*}
    &[(s, \text{len(s)}) \text{ for } s \text{ in } ["Only","for"]] \quad \# \quad [(\text{"Only"}, 4), (\text{"for"}, 3)] \\
    &[i*i \text{ for } i \text{ in } \text{range}(11) \text{ if } i \% 2 == 0] \quad \# \quad [0, 4, 16, 36, 64, 100]
    \end{align*}
    \]

- **Function composition:**
  - Programming with LEGO bricks
  - Examples:
    
    \[
    \begin{align*}
    &\text{(reverse . sort)[10,2,8,1,9,5,3,6,4,7]} - - [10,9,8,7,6,5,4,3,2,1]
    \end{align*}
    \]
    
    \[
    \text{theLongestTitle= head . reverse . sortBy(comparing length) . filter isTitle}
    \]
    
    \[
    \text{theLongestTitle words("A Sentence Full Of Titles.""
    }
    \]
    
    **Result:** "Sentence"
Vielen Dank für Ihre Aufmerksamkeit.

Rainer Grimm
science + computing ag
www.science-computing.de
phone  +49 7071 9457-253
r.grimm@science-computing.de