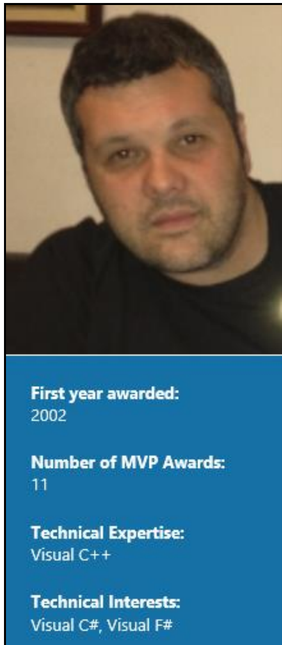
Programação Funcional
e
C++ Moderno

Fabio Galuppo, M.Sc.

http://fabiogaluppo.com

fabiogaluppo@acm.org

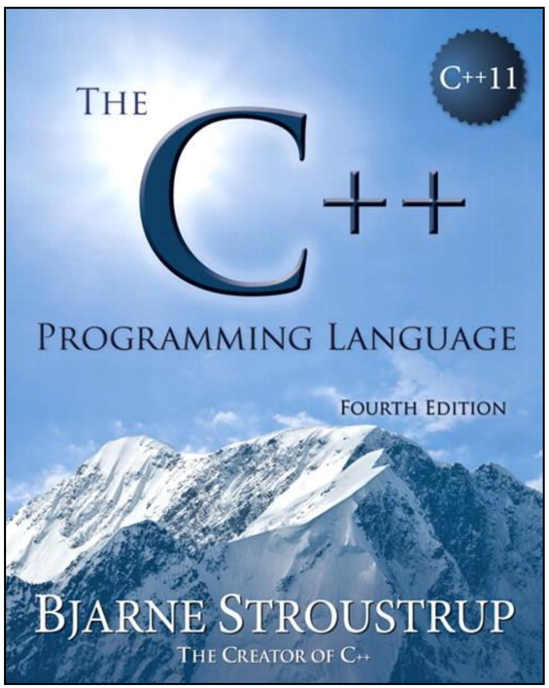


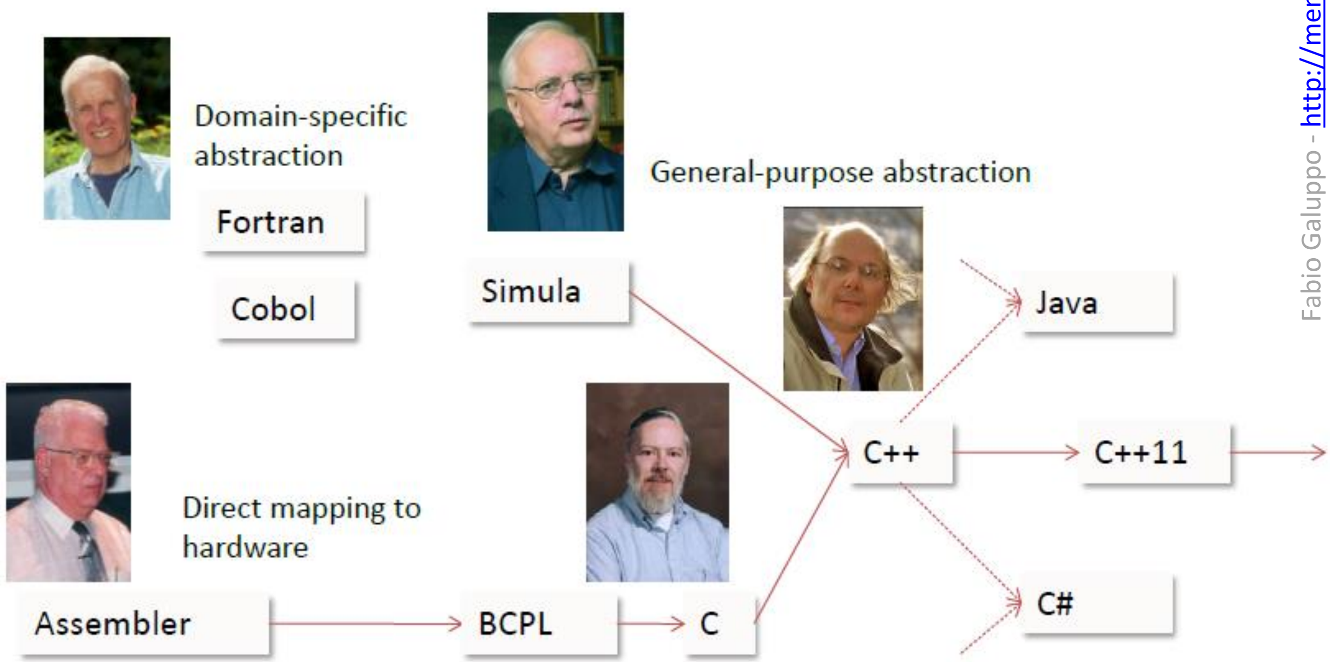
The C++ Programming Language

C++ is a general purpose programming language with a bias towards systems programming that

- is a better C
- supports data abstraction
- supports object-oriented programming
- supports generic programming.

C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer.





Fonte: http://www.stroustrup.com/

Fabio Galuppo - http://member.acm.org/~fabiogaluppo - fabiogaluppo@acm.org

It’s all about Polyglot Programming!

C++ supports systems programming. This implies that C++ code is able to effectively interoperate with software written in other languages on a system. The idea of writing all software in a single language is a fantasy. From the beginning, C++ was designed to interoperate simply and efficiently with C, assembler, and Fortran. By that, I meant that a C++, C, assembler, or Fortran function could call functions in the other languages without extra overhead or conversion of data structures passed among them.

http://www.youtube.com/watch?v=NvWTnIoQZj4

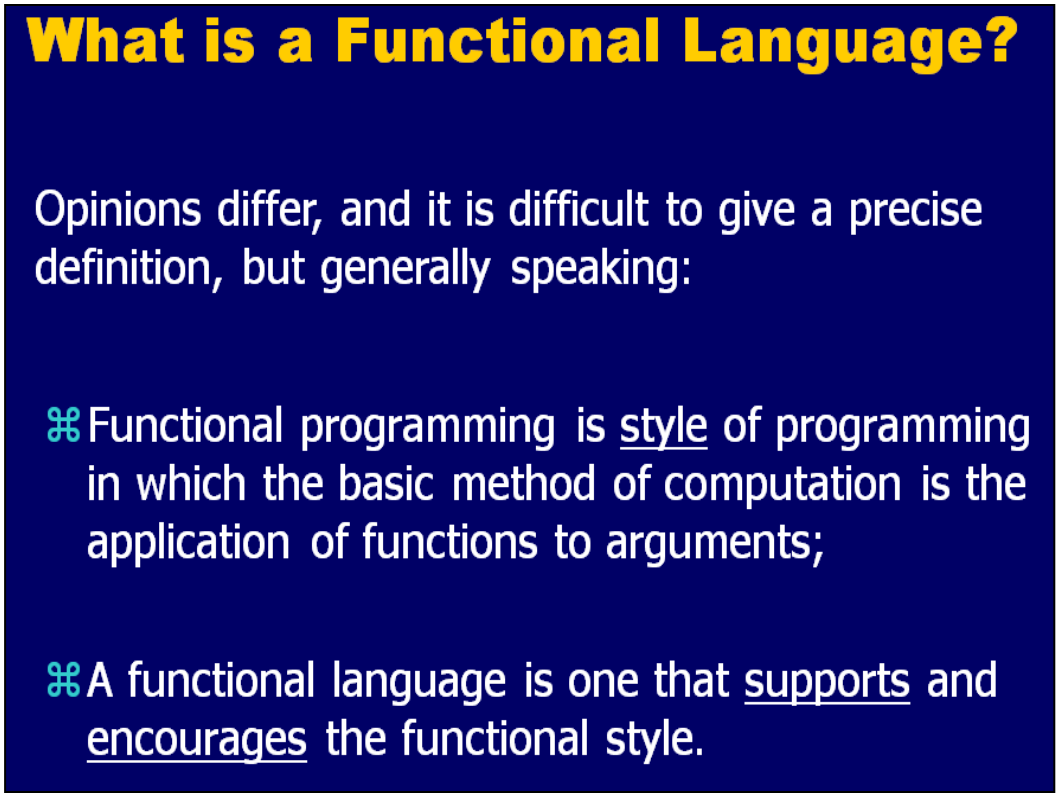
Bjarne Stroustrup: The 5 Programming Languages You Need to Know

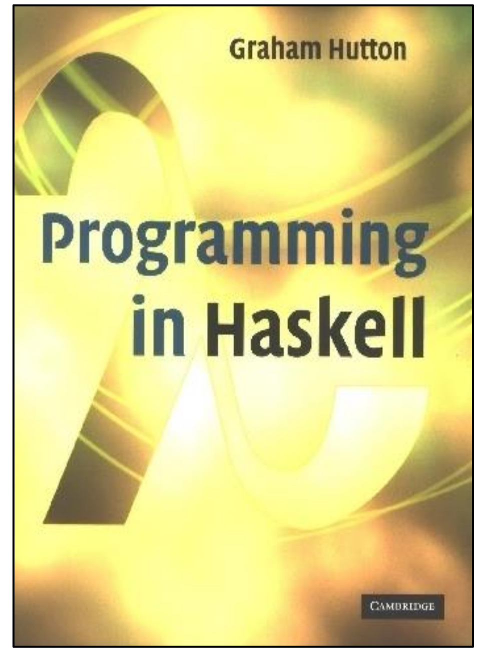
“Nobody should call themselves a professional if they only knew one language.”

...**C++**, of course; **Java**; maybe **Python** for mainline work... And if you know those, you can't help know sort of a little bit about **Ruby** and **JavaScript**, you can't help knowing **C** because that's what fills out the domain and of course **C#**. But again, **these languages create a cluster** so that **if you knew either five of the ones that I said, you would actually know the others**...

“Inclua a esta lista F#, Scala, Haskell, Erlang, Clojure, Lua e/ou Racket” – Fabio Galuppo

Programação Funcional



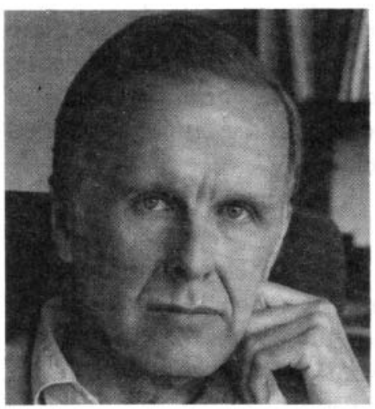


Fonte: http://www.cs.nott.ac.uk/~gmh/book.html

Programação Funcional

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose



Author's address: 91 Saint Germain Ave., San Francisco, CA 94114.

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

An alternative functional style of programming is founded on the use of combining forms for creating programs. Functional programs deal with structured data, are often nonrepetitive and nonrecursive, are hierarchically constructed, do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages.

Fonte: http://dl.acm.org/citation.cfm?id=359579

Functional Programming Basics

fabiogaluppo — ghc — 80×37

```
Fabios-MacBook-Pro:~ fabiogaluppo$ ghci
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> let a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Prelude> let xs = take 4 a
Prelude> let ys = drop 3 a
Prelude> xs
[1,2,3,4]
Prelude> ys
[4,5,6,7,8,9,10]
Prelude> head a
1
Prelude> head xs
1
Prelude> tail xs
[2,3,4]
Prelude> head (drop 2 xs)
3
Prelude> init a
[1,2,3,4,5,6,7,8,9]
Prelude> last a
10
Prelude> let b = []
Prelude> head b
*** Exception: Prelude.head: empty list
Prelude> let b = [1, 2, 3, 4]
Prelude> head b
1
Prelude> init b
[1,2,3]
Prelude> tail b
[2,3,4]
Prelude> last b
4
Prelude>
```

F# Interactive

```
Microsoft (R) F# Interactive version 12.0.30110.0
Copyright (c) Microsoft Corporation. All Rights Reserved.

For help type #help;;

> let a = [1 .. 10];;

val a : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]

> let b = [1; 2; 3; 4];;

val b : int list = [1; 2; 3; 4]

> let xs = Seq.take 4 a;;

val xs : seq<int>

> Seq.iter (fun x -> printf "%d " x) b; printfn "";;
1 2 3 4

val it : unit = ()

> Seq.iteri (fun idx x -> printfn "%d : %d" idx x) b;;
0 : 1
1 : 2
2 : 3
3 : 4
val it : unit = ()
> printfn "%d" (Seq.fold (fun acc x -> acc + x) 0 b);;
10
val it : unit = ()
>
```

F# Interactive | Package Manager Console | Output | Find Symbol Results | Err

Fabio Galuppo - http://member.acm.org/~fabiogaluppo - fabiogaluppo@acm.org

Standard Template Library Basics

```cpp
std::array<int, 26> xs;
std::iota(xs.begin(), xs.end(), 1);

display(xs.begin(), xs.end());

auto is_even = [](int x) { return (x & 0x1) == 0x0; };

auto bound_iterator = std::partition(xs.begin(), xs.end(), is_even);

display(xs.begin(), bound_iterator);
display(bound_iterator, xs.end());
display(xs.begin(), xs.end()); //after partition

std::random_shuffle(xs.begin(), xs.end());

std::array<int, 13> evens, odds;
std::partition_copy(xs.begin(), xs.end(), evens.begin(), odds.begin(), is_even);

display(evens.begin(), evens.end());
display(odds.begin(), odds.end());
display(xs.begin(), xs.end()); //after random_shuffle

std::array<int, 13> ys;
auto sum_func = [](int lhs, int rhs) { return lhs + rhs; };
std::transform(evens.begin(), evens.end(), odds.begin(), ys.begin(), std::bind(sum_func, std::placeholders::_1, std::
    placeholders::_2));

display(ys.begin(), ys.end()); //after transform

std::sort(ys.begin(), ys.end());

display(ys.begin(), ys.end()); //after sort

auto total = std::accumulate(ys.begin(), ys.end(), 0, std::plus<int>());
std::cout << total << "\n";

auto it = xs.begin();
while ((it = std::find_if(it, xs.end(), is_even)) != xs.end())
{
    std::cout << std::setw(2) << *it << " is even!" << "\n";
    ++it;
}
```

FunctionalCpp 〉 Thread 1 〉 0 main

```
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
26  2 24  4 22  6 20  8 18 10 16 12 14
13 15 11 17  9 19  7 21  5 23  3 25  1
26  2 24  4 22  6 20  8 18 10 16 12 14 13 15 11 17  9 19  7 21  5 23  3 25  1
18  6 10 14  8  2 12 22  4 26 24 20 16
23 11 15 17  1  9  5  3  7 25 21 19 13
```

Fabio Galuppo - http://fabiogaluppo.com - fabiogaluppo@acm.org

Standard Template Library Basics

```cpp
std::array<int, 26> xs;
std::iota(xs.begin(), xs.end(), 1);
```

```cpp
template<class _FwdIt,
    class _Ty> inline
    void _Iota(_FwdIt _First, _FwdIt _Last, _Ty _Val)
    {   // compute increasing sequence into [_First, _Last)
    for (; _First != _Last; ++_First, ++_Val)
        *_First = _Val;
    }

template<class _FwdIt,
    class _Ty> inline
    void iota(_FwdIt _First, _FwdIt _Last, _Ty _Val)
    {   // compute increasing sequence into [_First, _Last)
    _DEBUG_RANGE(_First, _Last);
    _Iota(_Unchecked(_First), _Unchecked(_Last), _Val);
    }
```

function template

std::iota C++11 <numeric>

```cpp
template <class ForwardIterator, class T>
  void iota (ForwardIterator first, ForwardIterator last, T val);
```

Store increasing sequence

Assigns to every element in the range `[first,last)` successive values of *val*, as if incremented with `++val` after each element is written.

The behavior of this function template is equivalent to:

```cpp
template <class ForwardIterator, class T>
  void iota (ForwardIterator first, ForwardIterator last, T val)
{
  while (first!=last) {
    *first = val;
    ++first;
    ++val;
  }
}
```

http://www.cplusplus.com/reference/numeric/iota/

Higher-order Function

In mathematics and computer science, a **higher-order function** (also **functional form**, **functional** or **functor**) is a function that does at least one of the following:

- takes one or more functions as an input
- outputs a function

All other functions are *first-order functions*. In mathematics higher-order functions are also known as *operators* or *functionals*. The derivative in calculus is a common example, since it maps a function to another function.

Fonte: http://en.wikipedia.org/wiki/Higher-order_function

WIKIPEDIA
The Free Encyclopedia

Higher-order Function: Exemplos

```cpp
#include <functional>
#include <iostream>

float plus(float a, float b) { return a + b; }

int main() {
  auto f1 = std::bind(plus, 100.f, 200.f);
  auto f2 = std::bind(std::plus<float>(), 100.f, 200.f);
  std::cout << f1() << '\n' << f2() << '\n';
}
```

```cpp
template <class T> struct negate {
  T operator() (const T& x) const {return -x;}
  typedef T argument_type;
  typedef T result_type;
};
```

```cpp
// negate example
#include <iostream>     // std::cout
#include <functional>   // std::negate
#include <algorithm>    // std::transform

int main () {
  int numbers[]={10,-20,30,-40,50};
  std::transform (numbers, numbers+5, numbers, std::negate<int>());
  for (int i=0; i<5; i++)
    std::cout << numbers[i] << ' ';
  std::cout << '\n';
  return 0;
}
```

Fonte: http://www.cplusplus.com/reference/functional/negate/

Fabio Galuppo - http://member.acm.org/~fabiogaluppo - fabiogaluppo@acm.org

C++ Lambda Expressions

5.1.2 Lambda expressions [expr.prim.lambda]

Lambda expressions provide a concise way to create simple function objects. [*Example:*

```
#include <algorithm>
#include <cmath>
void abssort(float* x, unsigned N) {
  std::sort(x, x + N,
    [](float a, float b) {
      return std::abs(a) < std::abs(b);
    });
}
```

— *end example*]

lambda-expression:
  lambda-introducer lambda-declarator$_{opt}$ *compound-statement*

lambda-introducer:
  [*lambda-capture*$_{opt}$]

lambda-capture:
  capture-default
  capture-list
  capture-default , *capture-list*

capture-default:
  &
  =

capture-list:
  capture ...$_{opt}$
  capture-list , *capture* ...$_{opt}$

capture:
  simple-capture
  init-capture

simple-capture:
  identifier
  & *identifier*
  `this`

init-capture:
  identifier initializer
  & *identifier initializer*

lambda-declarator:
  (*parameter-declaration-clause*) `mutable`$_{opt}$
    exception-specification$_{opt}$ *attribute-specifier-seq*$_{opt}$ *trailing-return-type*$_{opt}$

Working Draft, Standard for Programming Language C++

Fonte: https://isocpp.org/files/papers/N3797.pdf

C++ Lambda Expressions

[ ]   ( )$_{opt}$ -> $_{opt}$ { }

[captures]   (params) -> ret   { statements; }

May 18th, 2011 — C++0x Lambda Functions — Herb Sutter: http://nwcpp.org/may-2011.html

```
fold([](int acc, int x) { return acc + x; }, 0, xs);
```

```
int i = 10;
fold([i](int acc, int x) { return acc + x + i; }, 0, xs);
```

```
int i = 10;
std::bind([&i](int a, int b) { i = a + b;  return i; }, 10, 20)();
//i == 30
```

Morfismo ou "Notação de" Flecha

$$\frac{f\colon a \to b \qquad g\colon b \to c}{g \circ f\colon a \to c}$$

In many fields of mathematics, **morphism** refers to a *structure-preserving* *mapping* from one mathematical structure to another. The notion of morphism recurs in much of contemporary mathematics. In set theory, morphisms are functions; in linear algebra, linear transformations; in group theory, group homomorphisms; in topology, continuous functions, and so on.

In category theory, *morphism* is a broadly similar idea, but somewhat more abstract: the mathematical objects involved need not be sets, and the relationship between them may be something more general than a map.

The study of morphisms and of the structures (called objects) over which they are defined, is central to category theory. Much of the terminology of morphisms, as well as the intuition underlying them, comes from concrete categories, where the *objects* are simply *sets with some additional structure*, and *morphisms* are *structure-preserving functions*. In category theory, morphisms are sometimes also called **arrows**.

Fonte: http://en.wikipedia.org/wiki/Morphism

WIKIPEDIA
The Free Encyclopedia

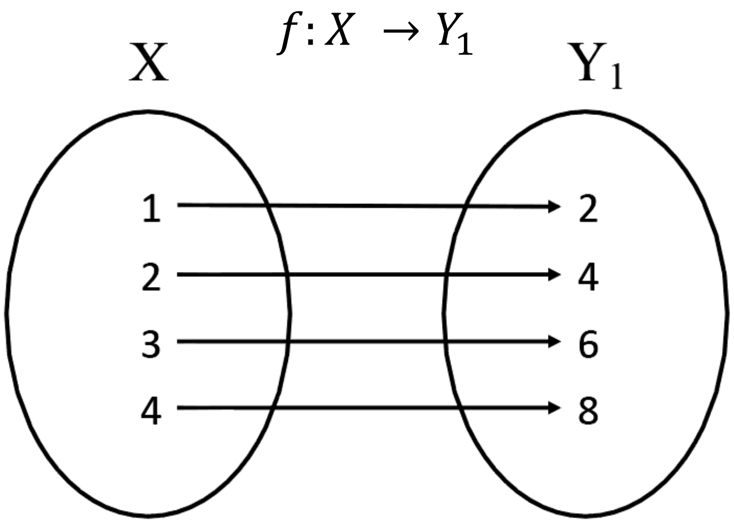
```
auto f(int a)   -> float  { return 2.f * a; }
auto g(float b) -> double { return 3.  * b; }
```

Morfismo & Composição em C++

```
#include <functional>

auto f(int a)   -> float  { return 2.f * a; }
auto g(float b) -> double { return 3.  * b; }

auto compose(std::function<float(int)> f, std::function<double(float)> g) ->
     std::function<double(int)>
{
    return [=](int x) -> double { return g(f(x)); };
}

int main()
{
   double c = compose(f, g)(10);
   //c = 60.000000000000000
}
```

Fabio Galuppo - http://member.acm.org/~fabiogaluppo - fabiogaluppo@acm.org

Exemplo de *Mapping* – *função injetora (isomorfismo)*



Visual C++ 2013 (VC++ 12.0) - fsi

```
C:\Users\Fabio Galuppo>fsi

Microsoft (R) F# Interactive version 12.0.30110.0
Copyright (c) Microsoft Corporation. All Rights Reserved.

For help type #help;;

> [1; 2; 3; 4] |> Seq.map (fun x -> 2 * x);;
val it : seq<int> = seq [2; 4; 6; 8]
```

GHCi

```
GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> [i * 2 | i <- [1..4]]
[2,4,6,8]
```

```cpp
Z&& f(const Z& input, const Z& expected_output)
{
    ZxZ result; //mapping f: input -> output

    //Injective (one-to-one) and surjective (onto), also bijective = X -> Y1
    //X.map(x -> 2 * x)
    std::transform(input.begin(), input.end(), std::back_inserter(result), [](int x){ return std::make_tuple(x, 2 * x); });
    print("f", input, result, expected_output);

    Z output;
    std::transform(result.begin(), result.end(), std::back_inserter(output), [](const ZxZ_Item& x){ return std::get<1>(x); });
    return std::move(output);
}
```

```cpp
Z X{ 1, 2, 3, 4 }; //X = {1, 2, 3, 4}
Z Y1{ 2, 4, 6, 8 }; //Y1 = {2, 4, 6, 8}
```

```
f(X, Y1);
```

```
f: {1, 2, 3, 4} -- {(1, 2), (2, 4), (3, 6), (4, 8)} --> {2, 4, 6, 8}
```

Fabio Galuppo - http://member.acm.org/~fabiogaluppo - fabiogaluppo@acm.org

Monad

In functional programming, a **monad** is a structure that represents computations defined as sequences of steps. A type with a monad structure defines what it means to chain operations, or nest functions of that type together. This allows the programmer to build pipelines that process data in steps, in which each action is decorated with additional processing rules provided by the monad.[1] As such, monads have been described as "programmable semicolons"; a semicolon is the operator used to chain together individual statements in many imperative programming languages,[1] thus the expression implies that extra code will be executed between the statements in the pipeline. Monads have also been explained with a physical metaphor as assembly lines, where a conveyor belt transports data between functional units that transform it one step at a time.[2] They can also be seen as a functional design pattern to build generic types.[3]

Purely functional programs can use monads to structure procedures that include sequenced operations like those found in structured programming.[4][5] Many common programming concepts can be described in terms of a monad structure, including side effects such as input/output, variable assignment, exception handling, parsing, nondeterminism, concurrency, and continuations. This allows these concepts to be defined in a purely functional manner, without major extensions to the language's semantics. Languages like Haskell provide monads in the standard core, allowing programmers to reuse large parts of their formal definition and apply in many different libraries the same interfaces for combining functions.[6]

Formally, a monad consists of a type constructor *M* and two operations, *bind* and *return* (where *return* is often also called *unit*). The operations must fulfill several

Fonte: http://en.wikipedia.org/wiki/Monad_(functional_programming)

```
class Monad m where
    return :: a → m a
    (>>=)  :: m a → (a → m b) → m b
```

1. *Left unit*:

$$unit\ a \times \lambda b.n = n[a/b]$$

2. *Right unit*:

$$m \times \lambda a.unit\ a = m$$

3. *Associativity*:

$$m \times (\lambda a.n \times \lambda b.o) = (m \times \lambda a.n) \times \lambda b.o$$

Figura 26: As leis da Monad (Adaptação da formalização de Wadler (WADLER, 1995)).

Fonte: http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.100.9674&rank=1

WIKIPEDIA
The Free Encyclopedia

Haskell List Monad

```
fabiogaluppo — ghc — 80×24
Last login: Wed Aug  6 21:36:19 on ttys002
Fabios-MacBook-Pro:~ fabiogaluppo$ ghci
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> let a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Prelude> let b = a
Prelude> b >>= \i -> [i * 2]
[2,4,6,8,10,12,14,16,18,20]
Prelude> b >>= \i -> [(fromIntegral i :: Float) * 1.5]
[1.5,3.0,4.5,6.0,7.5,9.0,10.5,12.0,13.5,15.0]
Prelude> let xs = [ "hello", "world" ]
Prelude> xs >>= \s -> concatMap (++"") [s]
"helloworld"
Prelude> (xs >>= \s -> concatMap (++"") [s]) == ['h','e','l','l','o','w','o','r'
,'l','d']
True
Prelude>
```

$$\gg= \quad \equiv \quad bind \quad \equiv \quad map$$

Fabio Galuppo - http://member.acm.org/~fabiogaluppo - fabiogaluppo@acm.org

C++ Container Monad

```cpp
template<typename T, template <typename, typename> class ContainerT>
struct container_monad final
{
    template<typename U = T, template <typename, typename> class ContainerU = ContainerT, class Function>
    auto bind(Function f) const -> container_monad<U, ContainerU>
    {
        ContainerU<U, std::allocator<U>> result;
        for (const auto& x : C)
        {
            const auto& fs = f(x);
            std::copy(std::begin(fs), std::end(fs), std::back_inserter(result));
        }
        return unit(std::move(result));
        //return unit(result);
    }

    template<typename U, template <typename, typename> class ContainerU>
    friend auto unit(const ContainerU<U, std::allocator<U>>& xs) -> container_monad<U, ContainerU>;

    template<typename U, template <typename, typename> class ContainerU>
    friend auto unit(ContainerU<U, std::allocator<U>>&& xs) -> container_monad<U, ContainerU>;

    container_monad(const container_monad&) = default;
    ~container_monad() = default;

private:
    container_monad& operator=(const container_monad&) = delete;
    container_monad() = default;
    container_monad(ContainerT<T, std::allocator<T>>&& xs)
        : C(std::move(xs)) {}

    ContainerT<T, std::allocator<T>> C;
};
```

Template template parameters
http://www.informit.com/articles/article.aspx?p=376878

```cpp
std::vector<int> a = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

auto b = unit(a);

auto c = b.bind([](int i){
    std::vector<int> result = { i * 2 };
    return result;
});
```

Fabio Galuppo - http://member.acm.org/~fabiogaluppo - fabiogaluppo@acm.org

Estendendo o Container Monad

(e renomeando alguns membros do tipo função)

```
template<typename U = T, template <typename, typename> class ContainerU = ContainerT, class Function>
auto flat_map(Function f) const -> container_monad_ex<U, ContainerU>
{
    ContainerU<U, std::allocator<U>> result;
    for (const auto& x : C)
    {
        const auto& fs = f(x);
        std::copy(std::begin(fs), std::end(fs), std::back_inserter(result));
    }
    return make_monad(std::move(result));
}

template<typename U = T, template <typename, typename> class ContainerU = ContainerT, class Function>
auto map(Function f) const -> container_monad_ex<U, ContainerU>
{
    ContainerU<U, std::allocator<U>> result;
    std::transform(std::begin(C), std::end(C), std::back_inserter(result), f);
    return make_monad(std::move(result));
}

template<class Function>
auto for_each(Function f) const -> void
{
    for (const auto& x : C) f(x);
}

template<typename U, template <typename, typename> class ContainerU>
friend auto make_monad(const ContainerU<U, std::allocator<U>>& xs) -> container_monad_ex<U, ContainerU>;

template<typename U, template <typename, typename> class ContainerU>
friend auto make_monad(ContainerU<U, std::allocator<U>>&& xs) -> container_monad_ex<U, ContainerU>;
```

```
std::vector<int> a = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

auto b = make_monad(a);

auto c = b.map([](int i){ return 10 * i; });

path p("..");
std::vector<path> dir;
std::copy(directory_iterator(p), directory_iterator(), std::back_inserter(dir));

auto m = make_monad(dir);
auto files = m.flat_map<std::string>([](const path& p){ return get_files_as_string(p); });
files.for_each([](const std::string& file){ std::cout << file << "\n"; });
```

Fabio Galuppo - http://member.acm.org/~fabiogaluppo - fabiogaluppo@acm.org

Java 8 Stream

Fonte: http://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

java.util.stream

Interface Stream<T>

Type Parameters:

T - the type of the stream elements

All Superinterfaces:

AutoCloseable, BaseStream<T,Stream<T>>

---

public interface **Stream<T>**
extends BaseStream<T,Stream<T>>

A sequence of elements supporting sequential and parallel aggregate operations. The following example illustrates an aggregate operation using Stream and IntStream:

```
int sum = widgets.stream()
                 .filter(w -> w.getColor() == RED)
                 .mapToInt(w -> w.getWeight())
                 .sum();
```

In this example, widgets is a Collection<Widget>. We create a stream of Widget objects via Collection.stream(), filter it to produce a stream containing only the red widgets, and then transform it into a stream of int values representing the weight of each red widget. Then this stream is summed to produce a total weight.

In addition to Stream, which is a stream of object references, there are primitive specializations for IntStream, LongStream, and DoubleStream, all of which are referred to as "streams" and conform to the characteristics and restrictions described here.

To perform a computation, stream operations are composed into a *stream pipeline*. A stream pipeline consists of a source (which might be an array, a collection, a generator function, an I/O channel, etc), zero or more *intermediate operations* (which transform a stream into another stream, such as filter(Predicate)), and a *terminal operation* (which produces a result or side-effect, such as count() or forEach(Consumer)). Streams are lazy; computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed.

```
std::vector<int> a = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

auto m = make_stream(a);
float total = m.reduce(0.f, [](float acc, const int x) {
    return acc + x;
});

std::cout << total << "\n";

auto filtered = m.filter([](int i) { return i >= 5; });
filtered.for_each([](int i) { std::cout << i << " "; });
std::cout << "\n";

std::vector<char> b = { 'a', 'b', 'c', 'd', 'e', 'f' };
auto result1 = m.zip(b);
auto result2 = make_stream(b).zip(a);

auto result3 = m.to_container();
auto result4 = m.to_container<std::list>();

auto result5 = make_stream(b)
               .sorted([](int i, int j) { return j < i; })
               .to_container();
```

C# IEnumerable<T> Monad

```csharp
static class Program
{
    static IEnumerable<int> GetEnumerableFiltered(this IEnumerable<int> xs)
    {
        return xs.Where(i => i >= 3)
                 .Where(i => i <= 8)
                 .Where(i => i >= 5);
    }

    static void Main(string[] args)
    {
        var xs = Enumerable.Range(1, 10);
        var ys = xs.GetEnumerableFiltered();

        var a = xs;
        var b = ys.ToList();

        var c = a.Where(i => i >= 3)
                 .Where(i => i <= 8)
                 .SelectMany(i =>
                 {
                     return new float[] { i * 10.0f };
                 })
                 .ToList();

        var d = a.Where(i => i >= 3).ToList();

        var e = ys.Aggregate(0, (acc, x) => acc + x);

        var f = ys.Where(i => i <= 6).Aggregate(1.0f, (acc, x) => acc * x);

        var g = ys.Sum();

        ys.ToList().ForEach(i => Console.Write(i + " "));
    }
}
```

```cpp
enumerable_monad<int, std::vector> get_enumerable_filtered(const std::vector<int>& xs)
{
    auto a = make_enumerable(xs);
    return a.where([](int i) { return i >= 3; })
            .where([](int i) { return i <= 8; })
            .where([](int i) { return i >= 5; });
}

void run_enumerable_monad()
{
    std::vector<int> xs = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    auto ys = get_enumerable_filtered(xs);

    auto a = make_enumerable(xs);

    auto b = ys.to_container();

    auto c = a.where([](int i) { return i >= 3; })
              .where([](int i) { return i <= 8; })
              .select_many<float>([](int i) {
                std::vector<float> v;
                v.push_back(i * 10.f);
                return v; })
              .to_container();

    auto d = a.where([](int i) { return i >= 3; })
              .to_container();

    auto e = ys.aggregate(0, [](int acc, int x) { return acc + x; });

    auto f = ys.where([](int i) { return i <= 6; })
               .aggregate(1.f, [](float acc, int x) { return acc * x; });

    auto g = ys.sum();

    ys.for_each([](int i) { std::cout << i << " "; });
}
```

Scala Try[T] Monad

```
The Scala Programming Language
Welcome to Scala version 2.10.3 (Java HotSpot(TM) Client VM, Java 1.6.0_26).
Type in expressions to have them evaluated.
Type :help for more information.

scala> :paste
// Entering paste mode (ctrl-D to finish)

val x : scala.util.Try[Int] = scala.util.Success(1)
x match {
    case scala.util.Success(i) => println(i)
    case scala.util.Failure(e) => println(e)
}

val y : scala.util.Try[Int] = scala.util.Failure(new Exception)
y match {
    case scala.util.Success(i) => println(i)
    case scala.util.Failure(e) => println(e)
}

val z : scala.util.Try[Int] = scala.util.Success(1)
z.map( i => 10.toFloat * i ) match {
    case scala.util.Success(i) => println(i)
    case scala.util.Failure(e) => println(e)
}

val w : scala.util.Try[Int] = scala.util.Failure(new NullPointerException)
w.map( i => 10.toFloat * i ) match {
    case scala.util.Success(i) => println(i)
    case scala.util.Failure(e) => println(e)
}

// Exiting paste mode, now interpreting.

1
java.lang.Exception
10.0
java.lang.NullPointerException
x: scala.util.Try[Int] = Success(1)
y: scala.util.Try[Int] = Failure(java.lang.Exception)
z: scala.util.Try[Int] = Success(1)
w: scala.util.Try[Int] = Failure(java.lang.NullPointerException)

scala>
```

```cpp
void test(const try_monad_ptr<int>& x)
{
    auto result =
    //match<int, void>(x,
    match<int, int>(x,
    [](int i) { //success
        return 1;
        //std::cout << i << "\n";
    },
    [](const std::exception& e) { //failure
        return 0;
        //std::cout << e.what() << "\n";
    });
    std::cout << result << "\n";

    x->match<void>(
    [](int i) { //success
        std::cout << i << "\n";
    }, [](const std::exception& e) { //failure
        std::cout << e.what() << "\n";
    });

    x->map<float>([](int i){ return 10.f * i; })
     ->match<void>([](float i) { //success
        std::cout << i << "\n";
    }, [](const std::exception& e) { //failure
        std::cout << e.what() << "\n";
    });
}
```

```cpp
void run_try_monad()
{
    test(make_try(1));
    test(make_try<int>(std::runtime_error("error...")));
}
```

std::future & .then()

```cpp
//future and async
std::future<int> f =
    std::async([]() -> int {
    std::this_thread::sleep_for(std::chrono::seconds(2));
    return 100;
});

std::cout << f.get() << std::endl;
```

```cpp
#include <future>
using namespace std;

int main() {

    future<int> f1 = async([]() { return possibly_long_computation(); });

    if(!f1.ready()) {
        //if not ready, attach a continuation and avoid a blocking wait
        f1.then([] (future<int> f2) {
                int v = f2.get();
                process_value(v);
        });
    }
    //if ready, then no need to add continuation, process value right away
    else {
        int v = f1.get();
        process_value(v);
    }
}
```

A Standardized Representation of Asynchronous Operations: www.open-std.org/JTC1/SC22/WG21/docs/papers/2013/n3558.pdf

C++17: I See a Monad in Your Future!

Home | About

February 26, 2014

C++17: I See a Monad in Your Future!

Posted by Bartosz Milewski under C++, Concurrency, Functional Programming, Monads, Multicore, Multithreading, Programming

[21] Comments

29 Votes

[If you prefer, you may watch the video of my talk on this topic (here are the slides).]

If you thought you were safe from functional programming in your cozy C++ niche, think again! First the lambdas and function objects and now the monad camouflaged as `std::future`. But do not despair, it's all just patterns. You won't find them in the Gang of Four book, but once you see them, they will become obvious.

Let me give you some background: I was very disappointed with the design of C++11 `std::future`. I described my misgivings in: Broken Promises — C++0x futures. I also made a few suggestions as how to fix it: Futures Done Right. Five years went by and, lo and behold, a proposal to improve `std::future` and related API, N3721, was presented to the Standards Committee for discussion. I thought it would be a no brainer, since the proposal was fixing obvious holes in the original design. A week ago I attended the meetings of the C++ Standards Committee in Issaquah — since it was within driving distance from me — and was I in for a surprise! Apparently some design patterns that form the foundation of functional programming are not obvious to everybody. So now I find myself on the other side of the discussion and will try to explain why the improved design of `std::future` is right.

Design arguments are not easy. You can't mathematically prove that one design is better than another, or a certain set of abstractions is better than another — unless you discover some obvious design flaws in one of them. You might have a gut feeling that a particular solution is elegant, but how do you argue about elegance?

Thankfully, when designing a library, there are some well known and accepted criteria. The most important ones, in my mind, are orthogonality, a.k.a., separation of concerns, and composability. It also helps if the solution has been previously implemented and tested, especially in more than one language. I will argue that this is indeed the case with the extended `std::future` design. In the process, I will describe some programming patterns that might be new to C++ programmers but have been tried and tested in functional languages. They tend to pop up more and more in imperative languages, especially in connection with concurrency and parallelism.

Archived Entry

Post Date :
February 26, 2014 at 11:59 am

Category :
C++, Concurrency, Functional Programming, Monads, Multicore, Multithreading, Programming

Do More :
You can leave a response, or trackback from your own site.

http://bartoszmilewski.com/2014/02/26/c17-i-see-a-monad-in-your-future/

Fabio Galuppo - http://member.acm.org/~fabiogaluppo - fabiogaluppo@acm.org

Programação Funcional
e
C++ Moderno

Fabio Galuppo, M.Sc.

http://fabiogaluppo.com

fabiogaluppo@acm.org