Functional Programming in C++

C++ Support for Functional Paradigm

Bartosz Kwaśniewski

Hi did it for all of us ;)

Bjarne Stroustrup

Hi did it for all of us ;)

Bjarne Stroustrup

"I wonder what would happen, if there were a language so complicated, so difficult to learn, that nobody would ever be able to swamp the market with programmers"

Claims and Questions

- Claims:

Claims and Questions

- Claims:
  - C++ supports or has just recently started to support functional programming

Claims and Questions

- Claims:
  - C++ supports or has just recently started to support functional programming
- Questions:

Claims and Questions

- Claims:
  - C++ supports or has just recently started to support functional programming
- Questions:
  - Is it true, is it really true?

Claims and Questions

- Claims:
  - C++ supports or has just recently started to support functional programming
- Questions:
  - Is it true, is it really true?
  - What does it mean?

Key Messages

1. Paradigm shift towards functional programming **is necessary** and has already started

Key Messages

1. Paradigm shift towards functional programming **is necessary** and has already started
2. You could use **many styles** including object oriented and functional to make **better software**

Key Messages

1. Paradigm shift towards functional programming **is necessary** and has already started
2. You could use **many styles** including object oriented and functional to make **better software**
3. There is no other option, **no turning back:** you **must learn** the functional paradigm

But, Why?

- Why?

But, Why?

- Why?
  - Moore's law (free lunch is over)

But, Why?

- Why?
  - Moore's law (free lunch is over)
  - Heat wall, no more clock race (max 3.4GHz)

But, Why?

- Why?
  - Moore's law (free lunch is over)
  - Heat wall, no more clock race (max 3.4GHz)
  - Multi-core arrival (solution)

But, Why?

- Why?
  - Moore's law (free lunch is over)
  - Heat wall, no more clock race (max 3.4GHz)
  - Multi-core arrival (solution)
  - Concurrent software (require redesign)

But, Why?

- Why?
  - Moore's law (free lunch is over)
  - Heat wall, no more clock race (max 3.4GHz)
  - Multi-core arrival (solution)
  - Concurrent software (require redesign)
  - Object oriented pitfalls

But, Why?

- Why?
  - Moore's law (free lunch is over)
  - Heat wall, no more clock race (max 3.4GHz)
  - Multi-core arrival (solution)
  - Concurrent software (require redesign)
  - Object oriented pitfalls
  - Composability (concurrency not scale well)

But, Why?

- Why?
  - Moore's law (free lunch is over)
  - Heat wall, no more clock race (max 3.4GHz)
  - Multi-core arrival (solution)
  - Concurrent software (require redesign)
  - Object oriented pitfalls
  - Composability (concurrency not scale well)
  - Different approach: functional programming (purity, immutability)

Really smart guy's says

"The problem with object-oriented languages is that if you wanted a banana but what you got was a gorilla holding the banana and the entire jungle"

Joe Armstrong, Erlang Creator

Really smart guy's says

"The problem with object-oriented languages is that if you wanted a banana but what you got was a gorilla holding the banana and the entire jungle"

Joe Armstrong, Erlang Creator

"It was only supposed to be a joke. Anyone with half a brain can see that object-oriented programming is counter-intuitive, illogical and inefficient"

Bjarne Stroustrup

Really smart guy's says

"The problem with object-oriented languages is that if you wanted a banana but what you got was a gorilla holding the banana and the entire jungle"

Joe Armstrong, Erlang Creator

"It was only supposed to be a joke. Anyone with half a brain can see that object-oriented programming is counter-intuitive, illogical and inefficient"

Bjarne Stroustrup

"When the limestone of imperative programming is worn away, the granite of functional programming will be observed."

Simon Peyton Jones (GHC)

Table of Contents

About Programming Paradigms

About Programming Paradigms

- What does it mean that a language supports a paradigm well?

Paradigm Definition

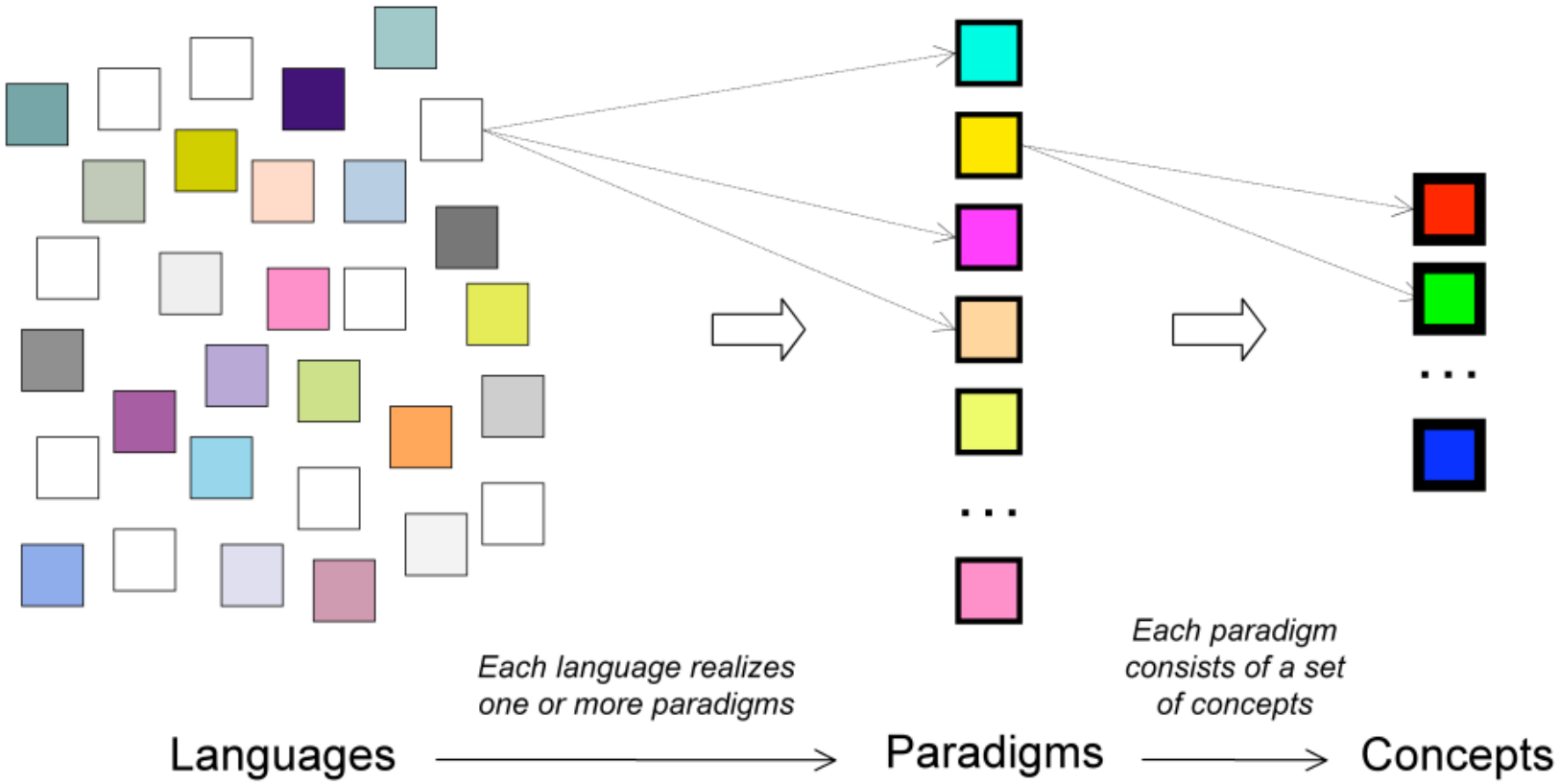
- What is a paradigm?

Paradigm Definition

- What is a paradigm?

Programming style or approach based on a **coherent set** of programming concepts aimed to solve particular kind of problem.

Programming Languages, Paradigms and Concepts



Multiparadigm Programming

- What is **Multiparadigm** programming?

Multiparadigm Programming

- What is **Multiparadigm** programming?
  1. Programming using **more than one** programming style, each to its **best effect**

Multiparadigm Programming

- What is **Multiparadigm** programming?
  1. Programming using **more than one** programming style, each to its **best effect**
  2. The same as **just "programming"** but using different features **in combination** as needed

Multiparadigm Programming

- What is **Multiparadigm** programming?
  1. Programming using **more than one** programming style, each to its **best effect**
  2. The same as **just "programming"** but using different features **in combination** as needed
- Why do we need multiparagim languages?

Multiparadigm Programming

- What is **Multiparadigm** programming?
  1. Programming using **more than one** programming style, each to its **best effect**
  2. The same as **just "programming"** but using different features **in combination** as needed
- Why do we need multiparagim languages?
  1. Real and different problems need **different programming concepts** to solve them cleanly

Multiparadigm Programming

- What is **Multiparadigm** programming?
  1. Programming using **more than one** programming style, each to its **best effect**
  2. The same as **just "programming"** but using different features **in combination** as needed
- Why do we need multiparagim languages?
  1. Real and different problems need **different programming concepts** to solve them cleanly
  2. However: this requires the **expetise level**

Paradigm Support: Conclusion

- Good (just) support:

Paradigm Support: Conclusion

- Good (just) support:
  1. Convenient, reasonably easy, safe, and efficient

Paradigm Support: Conclusion

- Good (just) support:
  1. Convenient, reasonably easy, safe, and efficient
  2. Built-in facilities, standard libraries

Paradigm Support: Conclusion

- Good (just) support:
  1. Convenient, reasonably easy, safe, and efficient
  2. Built-in facilities, standard libraries
- No (merely) support:

Paradigm Support: Conclusion

- Good (just) support:
  1. Convenient, reasonably easy, safe, and efficient
  2. Built-in facilities, standard libraries
- No (merely) support:
  1. Exceptional effort, skill, struggle

Paradigm Support: Conclusion

- Good (just) support:
  1. Convenient, reasonably easy, safe, and efficient
  2. Built-in facilities, standard libraries
- No (merely) support:
  1. Exceptional effort, skill, struggle
  2. Work arounds, no libraries, custom solutions, external tools and libraries

Paradigm Support: Conclusion

- Good (just) support:
  1. Convenient, reasonably easy, safe, and efficient
  2. Built-in facilities, standard libraries
- No (merely) support:
  1. Exceptional effort, skill, struggle
  2. Work arounds, no libraries, custom solutions, external tools and libraries
  3. Only self-discipline

Functional Programming

Functional Programming

Functional Programming

Functional Programming

- How does C++ implement functional concepts?

Functional Programming Language

- What is a functional language?

Functional Programming Language

- What is a functional language?

> It is a language that emphasises programming with **pure functions** and immutable data with the **control of side effects** and mutability.

Functional Programming Language

- What is a functional language?

  It is a language that emphasises programming with **pure functions** and immutable data with the **control of side effects** and mutability.

- How the solution is expressed in functional program?

Functional Programming Language

- What is a functional language?

  It is a language that emphasises programming with **pure functions** and immutable data with the **control of side effects** and mutability.

- How the solution is expressed in functional program?

  The program **is defined** by declarations or expression, not described by the series of steps.

Functional Programming Characteristics

1. Focusing on what rather on how

Functional Programming Characteristics

1. Focusing on what rather on how
2. The program is defined in more mathematical sense

Functional Programming Characteristics

1. Focusing on what rather on how
2. The program is defined in more mathematical sense
3. The functions are the primary constructs

Functional Programming Characteristics

1. Focusing on what rather on how
2. The program is defined in more mathematical sense
3. The functions are the primary constructs
4. The mutability and side effects are controlled

Functional Programming Characteristics

1. Focusing on what rather on how
2. The program is defined in more mathematical sense
3. The functions are the primary constructs
4. The mutability and side effects are controlled
5. The operator = is an equivalence operator

Functional Programming Characteristics

1. Focusing on what rather on how
2. The program is defined in more mathematical sense
3. The functions are the primary constructs
4. The mutability and side effects are controlled
5. The operator $=$ is an equivalence operator
6. There is substitution model

Functional Programming Foundations

- What are the math foundation?

Functional Programming Foundations

- What are the math foundation?
  1. λ-Calculus

Functional Programming Foundations

- What are the math foundation?
  1. λ-Calculus
  2. Combinatory logic

Functional Programming Foundations

- What are the math foundation?
  1. λ-Calculus
  2. Combinatory logic
  3. Category theory

Functional Programming Foundations

- What are the math foundation?
  1. λ-Calculus
  2. Combinatory logic
  3. Category theory
- No, we don't need to know that to write functional programs

Functional Programming Features

- Advantages:

Functional Programming Features

- Advantages:
  1. Modularity and composability

Functional Programming Features

- Advantages:
  1. Modularity and composability
  2. Equational reasoning and proofability of correctness

Functional Programming Features

- Advantages:
  1. Modularity and composability
  2. Equational reasoning and proofability of correctness
  3. Inherent parallelism and concurrency

Functional Programming Features

- Advantages:
  1. Modularity and composability
  2. Equational reasoning and proofability of correctness
  3. Inherent parallelism and concurrency
  4. More consist and shorter code

Functional Programming Features

- Advantages:
  1. Modularity and composability
  2. Equational reasoning and proofability of correctness
  3. Inherent parallelism and concurrency
  4. More consist and shorter code
  5. Automatic memory management

Functional Programming Features

- Advantages:
  1. Modularity and composability
  2. Equational reasoning and proofability of correctness
  3. Inherent parallelism and concurrency
  4. More consist and shorter code
  5. Automatic memory management
- Disadvantages:

Functional Programming Features

- Advantages:
  1. Modularity and composability
  2. Equational reasoning and proofability of correctness
  3. Inherent parallelism and concurrency
  4. More consist and shorter code
  5. Automatic memory management
- Disadvantages:
  1. Hard to learn for developers used to imperative languages

Functional Programming Features

- Advantages:
  1. Modularity and composability
  2. Equational reasoning and proofability of correctness
  3. Inherent parallelism and concurrency
  4. More consist and shorter code
  5. Automatic memory management
- Disadvantages:
  1. Hard to learn for developers used to imperative languages
  2. Not mainstream paradigm and language (TIOBE Index)

Functional Programming Features

- Advantages:
  1. Modularity and composability
  2. Equational reasoning and proofability of correctness
  3. Inherent parallelism and concurrency
  4. More consist and shorter code
  5. Automatic memory management
- Disadvantages:
  1. Hard to learn for developers used to imperative languages
  2. Not mainstream paradigm and language (TIOBE Index)
     - 2.1 Java - 18.7%, C - 9.6%, C++ - 5.8%

Functional Programming Features

- Advantages:
  1. Modularity and composability
  2. Equational reasoning and proofability of correctness
  3. Inherent parallelism and concurrency
  4. More consist and shorter code
  5. Automatic memory management
- Disadvantages:
  1. Hard to learn for developers used to imperative languages
  2. Not mainstream paradigm and language (TIOBE Index)
     - 2.1 Java - 18.7%, C - 9.6%, C++ - 5.8%
     - 2.2 Scala - 0.6%, F# - 0.6%, Haskell - 0.5%

C++ Language Features

C++ Language Features

C++ Language

- C++ language is:

C++ Language

- C++ language is:
  - Imperative, eager, impure

C++ Language

- C++ language is:
  - Imperative, eager, impure
  - Huge:

C++ Language

- C++ language is:
  - Imperative, eager, impure
  - Huge:
    - C++98 - 879p

C++ Language

- C++ language is:
  - Imperative, eager, impure
  - Huge:
    - C++98 - 879p
    - C++11 - 1324p

C++ Language

- C++ language is:
  - Imperative, eager, impure
  - Huge:
    - C++98 - 879p
    - C++11 - 1324p
    - C++14 - 1366p

C++ Language

- C++ language is:
  - Imperative, eager, impure
  - Huge:
    - C++98 - 879p
    - C++11 - 1324p
    - C++14 - 1366p
    - C++17 - 1586p

C++ Language

- C++ language is:
  - Imperative, eager, impure
  - Huge:
    - C++98 - 879p
    - C++11 - 1324p
    - C++14 - 1366p
    - C++17 - 1586p
  - Multiparadigm:

C++ Language

- C++ language is:
  - Imperative, eager, impure
  - Huge:
    - C++98 - 879p
    - C++11 - 1324p
    - C++14 - 1366p
    - C++17 - 1586p
  - Multiparadigm:
    - procedural

C++ Language

- C++ language is:
  - Imperative, eager, impure
  - Huge:
    - C++98 - 879p
    - C++11 - 1324p
    - C++14 - 1366p
    - C++17 - 1586p
  - Multiparadigm:
    - procedural
    - data abstraction

C++ Language

- C++ language is:
  - Imperative, eager, impure
  - Huge:
    - C++98 - 879p
    - C++11 - 1324p
    - C++14 - 1366p
    - C++17 - 1586p
  - Multiparadigm:
    - procedural
    - data abstraction
    - object oriented

C++ Language

- C++ language is:
  - Imperative, eager, impure
  - Huge:
    - C++98 - 879p
    - C++11 - 1324p
    - C++14 - 1366p
    - C++17 - 1586p
  - Multiparadigm:
    - procedural
    - data abstraction
    - object oriented
    - generic

C++ Language

- C++ language is:
  - Imperative, eager, impure
  - Huge:
    - C++98 - 879p
    - C++11 - 1324p
    - C++14 - 1366p
    - C++17 - 1586p
  - Multiparadigm:
    - procedural
    - data abstraction
    - object oriented
    - generic
    - functional

C++ Language

- C++ language is:
  - Imperative, eager, impure
  - Huge:
    - C++98 - 879p
    - C++11 - 1324p
    - C++14 - 1366p
    - C++17 - 1586p
  - Multiparadigm:
    - procedural
    - data abstraction
    - object oriented
    - generic
    - functional
  - General purpose

C++ Computation Types I

- runtime computations

```
//1. runtime computations
vector<int> v1{1,2,3};
transform(v1.begin(), v1.end(), v1.begin(), [](int i){return i + 1;});
```

C++ Computation Types I

- runtime computations

```
//1. runtime computations
vector<int> v1{1,2,3};
transform(v1.begin(), v1.end(), v1.begin(), [](int i){return i + 1;});
```

- constexpr computations

```
//2. constexpr computations
constexpr int factorial(int n) {
    return n <= 1? 1 : (n * factorial(n - 1));
}
static_assert(factorial(4) == 24);
```

C++ Computation Types II

- types computations

```
//3. types computations
using types = mpl::vector<int, char, float, void>;
using pointers = mpl::transform<types, add_const_pointer<mpl::_1>>::type;
```

C++ Computation Types II

- types computations

```
//3. types computations
using types = mpl::vector<int, char, float, void>;
using pointers = mpl::transform<types, add_const_pointer<mpl::_1>>::type;
```

- heterogeneous computations

```
//4. heterogeneous computations
fusion::vector<int, std::string, float> seq{1, "abc", 3.4f};
fusion::vector<std::string, std::string, std::string> strings
    = fusion::transform(seq, [](auto t) {std::stringstream ss;
       ss << t;  return ss.str();});
```

C++ Language Constructs

- C++98

C++ Language Constructs

- C++98
  - function pointers, custom functors, autoptr, const, templates, partial_sum, accumulate, **transform**

C++ Language Constructs

- C++98
  - function pointers, custom functors, autoptr, const, templates, partial_sum, accumulate, **transform**
- C++11

C++ Language Constructs

- C++98
  - function pointers, custom functors, autoptr, const, templates, partial_sum, accumulate, **transform**
- C++11
  - **auto**, **decltype**, type aliases, move sematics, smart pointers, **lambda expression**, variadic templates, **constant expressions**, tuple

C++ Language Constructs

- C++98
  - function pointers, custom functors, autoptr, const, templates, partial_sum, accumulate, **transform**
- C++11
  - **auto**, **decltype**, type aliases, move sematics, smart pointers, **lambda expression**, variadic templates, **constant expressions**, tuple
- C++14

C++ Language Constructs

- C++98
  - function pointers, custom functors, autoptr, const, templates, partial_sum, accumulate, **transform**
- C++11
  - **auto**, **decltype**, type aliases, move sematics, smart pointers, **lambda expression**, variadic templates, **constant expressions**, tuple
- C++14
  - **generic lambdas**, generalized return type deduction, decltype(auto), generalized lambda captures, extended constexpr, type _t aliases, variable template

C++ Language Constructs

- C++98
  - function pointers, custom functors, autoptr, const, templates, partial_sum, accumulate, **transform**
- C++11
  - **auto**, **decltype**, type aliases, move sematics, smart pointers, **lambda expression**, variadic templates, **constant expressions**, tuple
- C++14
  - **generic lambdas**, generalized return type deduction, decltype(auto), generalized lambda captures, extended constexpr, type _t aliases, variable template
- C++17

C++ Language Constructs

- C++98
  - function pointers, custom functors, autoptr, const, templates, partial_sum, accumulate, **transform**
- C++11
  - **auto**, **decltype**, type aliases, move sematics, smart pointers, **lambda expression**, variadic templates, **constant expressions**, tuple
- C++14
  - **generic lambdas**, generalized return type deduction, decltype(auto), generalized lambda captures, extended constexpr, type _t aliases, variable template
- C++17
  - **fold expressions**, constexpr if, constexpr lambda expressions, class template deduction, optional, any, variant, reduce

Functional Concepts

Functional Concepts

Functional Concepts

Functional Concepts

- What programming concepts define functional paradigm?

Functional Concepts

Functional Concepts

- What programming concepts define functional paradigm?
- How does C++ implement those concepts?

FP Concepts and C++

concept	built-in	substitution
first-class function	-	function pointer, functors, std::function
higher-order function	-	function pointer, functors, std::function
pure function	-	const, move semantics, pass by value
anonymous function	-	lambda expression
nested function	-	nested classes, lambda expressions
type inference	auto, decltype	-
closures	-	lambda expressions
parial application	-	std::bind
currying	-	custom solutions with lambdas and templates
recursion with tco	sometimes	-
algebraic data types	-	compositions
pattern matching	-	mach7
lazy valuation	-	custom solution, suspension idiom
immutable data	-	custom solutions
polymorphic types	-	generics
list comprehension	-	custom solutions, range-v3
type classes and overloading	-	polymorhpism
automatic memory management	semi, smart pointers	-
monads	opional, variant	custom, libraries
combinators	-	custom solutions, libraries

Functional Programming Concepts

Deterministic function: Haskell

- What is a deterministic function?

Deterministic function: Haskell

- What is a deterministic function?
  - It is a pure function (transformation)

Deterministic function: Haskell

- What is a deterministic function?
  - It is a pure function (transformation)
  - A function without side effects

Deterministic function: Haskell

- What is a deterministic function?
  - It is a pure function (transformation)
  - A function without side effects
  - Result depends only on its arguments

Deterministic function: Haskell

- What is a deterministic function?
  - It is a pure function (transformation)
  - A function without side effects
  - Result depends only on its arguments
  - Without involvement of any external or global states

Deterministic function: Haskell

- What is a deterministic function?
  - It is a pure function (transformation)
  - A function without side effects
  - Result depends only on its arguments
  - Without involvement of any external or global states

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

Pure Function: C++

- How are pure functions represented in C++?

Pure Function: C++

- How are pure functions represented in C++?
  - They aren't, but...

Pure Function: C++

- How are pure functions represented in C++?
  - They aren't, but...
- There are extensions:

Pure Function: C++

- How are pure functions represented in C++?
  - They aren't, but...
- There are extensions:
  - const (read and write not allowed from globals)
  - pure (read is allowed)

Pure Function: C++

- How are pure functions represented in C++?
  - They aren't, but...
- There are extensions:
  - const (read and write not allowed from globals)
  - pure (read is allowed)

```
int sum1(int a, int b) __attribute__((const));
int sum2(int a, int b) __attribute__((pure));
```

Pure Function: C++

- How are pure functions represented in C++?
  - They aren't, but...
- There are extensions:
  - const (read and write not allowed from globals)
  - pure (read is allowed)

```
int sum1(int a, int b) __attribute__((const));
int sum2(int a, int b) __attribute__((pure));
```

- Unfortunately ...

Pure Function: C++

- How are pure functions represented in C++?
  - They aren't, but...
- There are extensions:
  - const (read and write not allowed from globals)
  - pure (read is allowed)

```
int sum1(int a, int b) __attribute__((const));
int sum2(int a, int b) __attribute__((pure));
```

- Unfortunately ...
  - Those are just hints for optimization

Pure Function: C++

- How are pure functions represented in C++?
  - They aren't, but...
- There are extensions:
  - const (read and write not allowed from globals)
  - pure (read is allowed)

```
int sum1(int a, int b) __attribute__((const));
int sum2(int a, int b) __attribute__((pure));
```

- Unfortunately ...
  - Those are just hints for optimization
  - Purity is NOT checked or enforced

First-Class and Higher-Order function: Haskell

- What is a first-class function?

First-Class and Higher-Order function: Haskell

- What is a first-class function?
  - Functions that could be used in all contexts as all other entities:

First-Class and Higher-Order function: Haskell

- What is a first-class function?
  - Functions that could be used in all contexts as all other entities:
    - Create new functions from preexisting functions at run-time

First-Class and Higher-Order function: Haskell

- What is a first-class function?
  - Functions that could be used in all contexts as all other entities:
    - Create new functions from preexisting functions at run-time
    - Store functions in collections

First-Class and Higher-Order function: Haskell

- What is a first-class function?
  - Functions that could be used in all contexts as all other entities:
    - Create new functions from preexisting functions at run-time
    - Store functions in collections
    - Use functions as arguments to other functions

First-Class and Higher-Order function: Haskell

- What is a first-class function?
  - Functions that could be used in all contexts as all other entities:
    - Create new functions from preexisting functions at run-time
    - Store functions in collections
    - Use functions as arguments to other functions
    - Use functions as return values of other functions

First-Class and Higher-Order function: Haskell

- What is a first-class function?
  - Functions that could be used in all contexts as all other entities:
    - Create new functions from preexisting functions at run-time
    - Store functions in collections
    - Use functions as arguments to other functions
    - Use functions as return values of other functions
- What is a higher-order function?

First-Class and Higher-Order function: Haskell

- What is a first-class function?
  - Functions that could be used in all contexts as all other entities:
    - Create new functions from preexisting functions at run-time
    - Store functions in collections
    - Use functions as arguments to other functions
    - Use functions as return values of other functions
- What is a higher-order function?
  - A function that takes another function as an argument or returns a function

First-Class and Higher-Order function: Haskell

- What is a first-class function?
  - Functions that could be used in all contexts as all other entities:
    - Create new functions from preexisting functions at run-time
    - Store functions in collections
    - Use functions as arguments to other functions
    - Use functions as return values of other functions
- What is a higher-order function?
  - A function that takes another function as an argument or returns a function

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
print(twice (+1) 2) -- prints 4
```

First Class and Higher Order Functions: C++

- Does C++ functions are first-class citizens and higher-order?

First Class and Higher Order Functions: C++

- Does C++ functions are first-class citizens and higher-order?
  - No, but... What are first-class entities in C++?

First Class and Higher Order Functions: C++

- Does C++ functions are first-class citizens and higher-order?
  - No, but... What are first-class entities in C++?
  - Yes, classes are.

First Class and Higher Order Functions: C++

- Does C++ functions are first-class citizens and higher-order?
  - No, but... What are first-class entities in C++?
  - Yes, classes are.
- So there are:

First Class and Higher Order Functions: C++

- Does C++ functions are first-class citizens and higher-order?
  - No, but... What are first-class entities in C++?
  - Yes, classes are.
- So there are:
  - callable objects (std::function, custom)

First Class and Higher Order Functions: C++

- Does C++ functions are first-class citizens and higher-order?
  - No, but... What are first-class entities in C++?
  - Yes, classes are.
- So there are:
  - callable objects (std::function, custom)
  - lambda expressions

First Class and Higher Order Functions: C++

- Does C++ functions are first-class citizens and higher-order?
  - No, but... What are first-class entities in C++?
  - Yes, classes are.
- So there are:
  - callable objects (std::function, custom)
  - lambda expressions

```
auto inc = [](int a){return a + 1;};
auto twice = [](auto f, int a){return f(f(a));};
cout << twice(inc, 2); //prints 4

auto hanatwice = hana::compose(inc,inc);
cout << hanatwice(2);
```

First Class and Higher Order Functions: C++

- Does C++ functions are first-class citizens and higher-order?
  - No, but... What are first-class entities in C++?
  - Yes, classes are.
- So there are:
  - callable objects (std::function, custom)
  - lambda expressions

```
auto inc = [](int a){return a + 1;};
auto twice = [](auto f, int a){return f(f(a));};
cout << twice(inc, 2); //prints 4

auto hanatwice = hana::compose(inc,inc);
cout << hanatwice(2);
```

- But what about function pointers?

First Class and Higher Order Functions: C++

- Does C++ functions are first-class citizens and higher-order?
  - No, but... What are first-class entities in C++?
  - Yes, classes are.
- So there are:
  - callable objects (std::function, custom)
  - lambda expressions

```
auto inc = [](int a){return a + 1;};
auto twice = [](auto f, int a){return f(f(a));};
cout << twice(inc, 2); //prints 4

auto hanatwice = hana::compose(inc,inc);
cout << hanatwice(2);
```

- But what about function pointers?
  - It is only the address without bounded variables

Nested function: Haskell

- What is a nested function?

Nested function: Haskell

- What is a nested function?
  - A function defined within another function

Nested function: Haskell

- What is a nested function?
  - A function defined within another function

```
nested :: Integer -> Integer
nested x = f 1 + f 2 where f y = x + y
print (nested 1)  -- prints 5
```

Nested Function: C++

- Does C++ support nested functions?

Nested Function: C++

- Does C++ support nested functions?
  - No, but we have...

Nested Function: C++

- Does C++ support nested functions?
  - No, but we have...
  - lambda expressions

Nested Function: C++

- Does C++ support nested functions?
  - No, but we have...
  - lambda expressions

```
auto nested = [](int x){
  auto f = [x](int y){return x + y;};
  return f(1) + f(2);
};
```

Anonymous Function: Haskell

- What is an anonymous functions?

Anonymous Function: Haskell

- What is an anonymous functions?
  - It is a function definition that is not bound to an identifier

Anonymous Function: Haskell

- What is an anonymous functions?
  - It is a function definition that is not bound to an identifier
  - used in higher-order functions, as arguments or a result that needs to be returned

Anonymous Function: Haskell

- What is an anonymous functions?
  - It is a function definition that is not bound to an identifier
  - used in higher-order functions, as arguments or a result that needs to be returned

```
print ((\x y -> x + y) 1 2)
```

Anonymous Function: C++

- Does C++ support anonymous functions?

Anonymous Function: C++

- Does C++ support anonymous functions?
  - No, but we have ...

Anonymous Function: C++

- Does C++ support anonymous functions?
  - No, but we have ...
  - lambda expressions.

Anonymous Function: C++

- Does C++ support anonymous functions?
  - No, but we have ...
  - lambda expressions.

```
cout << [](auto f){
        return [f](int a){
            return f(a + 1);
        };
    }([](int c){return c + 1;})(2)
```

Anonymous Function: C++

- Does C++ support anonymous functions?
  - No, but we have ...
  - lambda expressions.

```
cout << [](auto f){
        return [f](int a){
            return f(a + 1);
        };
    }([](int c){return c + 1;})(2)
```

- How many lambda expressions do we have here?

Anonymous Function: C++

- Does C++ support anonymous functions?
  - No, but we have ...
  - lambda expressions.

```
cout << [](auto f){
        return [f](int a){
            return f(a + 1);
        };
    }([](int c){return c + 1;})(2)
```

- How many lambda expressions do we have here?
  - Three anonymous lambda expressions

Closure: Haskell

- What is a closure?

Closure: Haskell

- What is a closure?
  - Instance of a function together with bindings to variables used locally but defined in enclosing scope

Closure: Haskell

- What is a closure?
  - Instance of a function together with bindings to variables used locally but defined in enclosing scope
  - Callable object with all the captured variables

Closure: Haskell

- What is a closure?
  - Instance of a function together with bindings to variables used locally but defined in enclosing scope
  - Callable object with all the captured variables

```
sum :: Num a => a -> (a -> a)
sum y = \x -> x + y
```

Closure: C++

- Does C++ support closures?

Closure: C++

- Does C++ support closures?
  - No, but we have...

Closure: C++

- Does C++ support closures?
  - No, but we have...
  - lambda expressions.

Closure: C++

- Does C++ support closures?
  - No, but we have...
  - lambda expressions.
- Closure terms related:

Closure: C++

- Does C++ support closures?
  - No, but we have...
  - lambda expressions.
- Closure terms related:
  - Lambda expression: expression, part of a source code

Closure: C++

- Does C++ support closures?
  - No, but we have...
  - lambda expressions.
- Closure terms related:
  - Lambda expression: expression, part of a source code
  - Closure class: a functor class generated for each lambda expression

Closure: C++

- Does C++ support closures?
  - No, but we have...
  - lambda expressions.
- Closure terms related:
  - Lambda expression: expression, part of a source code
  - Closure class: a functor class generated for each lambda expression
  - Closure: object, instance of closure class

```
function<function<int(int)>(int, int)> closure = [](int a, int b) {
  return [a,b](int c){return a + b + c;};
};
cout << closure(1,2)(3) << endl; //prints 6
```

Partial application: Haskell

- What is a partial application?

Partial application: Haskell

- What is a partial application?
  - Supplying fewer than the total number of arguments

Partial application: Haskell

- What is a partial application?
  - Supplying fewer than the total number of arguments
  - Process of creating a new function with lower arity by fixing a number of arguments to a function

Partial application: Haskell

- What is a partial application?
  - Supplying fewer than the total number of arguments
  - Process of creating a new function with lower arity by fixing a number of arguments to a function

```
partialSum :: Num a => a -> a -> a -> a
partialSum a b c = a + b + c
partialSum1 = partialSum 1
partialSum2 = partialSum1 2
print (partialSum2 3) -- prints 6
```

Partial Application: C++

- Does C++ support partial application?

Partial Application: C++

- Does C++ support partial application?
  - No, but we have...

Partial Application: C++

- Does C++ support partial application?
  - No, but we have...
  - bind and boost::hana library.

Partial Application: C++

- Does C++ support partial application?
  - No, but we have...
  - bind and boost::hana library.

```
int addabc(int a, int b, int c) {
    return a + b + c;
}
//cout << addabc(1,2)(3);
//error: no matching function for call to addabc
//note: candidate function not viable: requires 3 args, but 2 were provided

cout << bind(addabc, 1, 2, _1)(3);
cout << hana::partial(addabc, 1, 2)(3);

cout << bind(addabc, 1, _1, _2)(2,3);
cout << hana::partial(addabc, 1)(2,3);

cout << bind(bind(addabc, 1, _1, _2), 2, _1)(3);
cout << hana::partial(hana::partial(addabc, 1),2)(3);
```

Currying: Haskell

- What is a currying application?

Currying: Haskell

- What is a currying application?
  - It is a translation of a one multi-argument function into a chain of multi one argument functions

Currying: Haskell

- What is a currying application?
  - It is a translation of a one multi-argument function into a chain of multi one argument functions
  - In haskell all functions are one argument functions

Currying: Haskell

- What is a currying application?
  - It is a translation of a one multi-argument function into a chain of multi one argument functions
  - In haskell all functions are one argument functions

```
--int to function
--f :: Int -> (Int -> Int)
f :: Int -> Int -> Int
f a b = a + b

--function to int
g :: (Int, Int) -> Int
g (a, b) = a + b

--Bidirectional transformation:
print (f 1 2 + g(1,2) + (curry g) 1 2 + (uncurry f)(1,2))
```

Currying: C++

- Does C++ support currying?

Currying: C++

- Does C++ support currying?
  - No, but we have...

Currying: C++

- Does C++ support currying?
  - No, but we have...
  - templates and boost::hana library

Currying: C++

- Does C++ support currying?
  - No, but we have...
  - templates and boost::hana library

```
int addabc(int a, int b, int c) {
    return a + b + c;
}

cout << hana::curry<3>(addabc)(1)(2)(3) << endl;
cout << hana::curry<3>(addabc)(1)(2,3) << endl;
cout << hana::curry<3>(addabc)(1,2,3) << endl;
```

Recursive Function and TCO: Haskell

- Are there loops in functional languages?

Recursive Function and TCO: Haskell

- Are there loops in functional languages?
  - No, the recursive functions act like loops

Recursive Function and TCO: Haskell

- Are there loops in functional languages?
  - No, the recursive functions act like loops
- What is a recursive function

Recursive Function and TCO: Haskell

- Are there loops in functional languages?
  - No, the recursive functions act like loops
- What is a recursive function
  - Ok, that was a rhetoric question

Recursive Function and TCO: Haskell

- Are there loops in functional languages?
  - No, the recursive functions act like loops
- What is a recursive function
  - Ok, that was a rhetoric question
- When Tail Call Optimization happened?

Recursive Function and TCO: Haskell

- Are there loops in functional languages?
  - No, the recursive functions act like loops
- What is a recursive function
  - Ok, that was a rhetoric question
- When Tail Call Optimization happened?
  - When there is only a single recursive call and this call is last statement in the function

Recursive Function and TCO: Haskell

- Are there loops in functional languages?
  - No, the recursive functions act like loops
- What is a recursive function
  - Ok, that was a rhetoric question
- When Tail Call Optimization happened?
  - When there is only a single recursive call and this call is last statement in the function
  - TCO reuses the same call stack, and TCO is critical in functional languages

Recursive Function and TCO: Haskell

- Are there loops in functional languages?
  - No, the recursive functions act like loops
- What is a recursive function
  - Ok, that was a rhetoric question
- When Tail Call Optimization happened?
  - When there is only a single recursive call and this call is last statement in the function
  - TCO reuses the same call stack, and TCO is critical in functional languages
  - No jump-and-add-stack-frame immediately followed by a pop-stack-frame-and-return-to-caller

Recursive Function and TCO: Haskell

- Are there loops in functional languages?
  - No, the recursive functions act like loops
- What is a recursive function
  - Ok, that was a rhetoric question
- When Tail Call Optimization happened?
  - When there is only a single recursive call and this call is last statement in the function
  - TCO reuses the same call stack, and TCO is critical in functional languages
  - No jump-and-add-stack-frame immediately followed by a pop-stack-frame-and-return-to-caller
  - But simply jump to the start of the second routine (goto)

Recursion: Factorial Implementations

```
-- Junior Haskell programmer:
factorial 0 = 1
factorial n = n * factorial (n - 1)

-- Another junior haskell developer (tail call optimization)
factorial x = fac' x 1 where
    factorial' 1 y = y
    factorial' x y = factorial' (x-1) (x*y)

-- Senior haskell developer (TCO with strictness)
factorial x = fac' x 1 where
    factorial' 1 y = y
    factorial' x y = factorial' (x-1) $!  (x*y)

-- Tenured professor (teaching Haskell to freshmen)
factorial n = product [1..n]
```

Tail Call Optimization: C++

- Does C++ support TCO?

Tail Call Optimization: C++

- Does C++ support TCO?
  - No, Tail Call Optimisation isn't in the C++ standard

Tail Call Optimization: C++

- Does C++ support TCO?
  - No, Tail Call Optimisation isn't in the C++ standard
  - All modern compilers support it (O2)

Tail Call Optimization: C++

- Does C++ support TCO?
  - No, Tail Call Optimisation isn't in the C++ standard
  - All modern compilers support it (O2)
  - Difficult to implement

Tail Call Optimization: C++

- Does C++ support TCO?
  - No, Tail Call Optimisation isn't in the C++ standard
  - All modern compilers support it (O2)
  - Difficult to implement

```
int factco_times (int n, int acc) {
    if (n == 0) return acc;
    else return factco_times(n - 1, acc * n);
}

int factco (int n) {
    return factco_times (n, 1);
}

cout << factco(5) << endl; //prints 120
```

Types System: Haskell and C++

- What type system is important in FP?

Types System: Haskell and C++

- What type system is important in FP?
  - Only static and strong typy system?

Types System: Haskell and C++

- What type system is important in FP?
  - Only static and strong typy system?
- Why? Why not weak and dynamic?

Types System: Haskell and C++

- What type system is important in FP?
  - Only static and strong typy system?
- Why? Why not weak and dynamic?
  - Do we want to make monkeys happy, or do we want to produce correct programs?

Types System: Haskell and C++

- What type system is important in FP?
  - Only static and strong typy system?
- Why? Why not weak and dynamic?
  - Do we want to make monkeys happy, or do we want to produce correct programs?
  - Types Are About Composability

Types System: Haskell and C++

- What type system is important in FP?
  - Only static and strong typy system?
- Why? Why not weak and dynamic?
  - Do we want to make monkeys happy, or do we want to produce correct programs?
  - Types Are About Composability
  - Type checking provides barrier against nonsensical programs

Types System: Haskell and C++

- What type system is important in FP?
  - Only static and strong typy system?
- Why? Why not weak and dynamic?
  - Do we want to make monkeys happy, or do we want to produce correct programs?
  - Types Are About Composability
  - Type checking provides barrier against nonsensical programs
  - This is what Haskell and C++ have

Types System: Haskell and C++

- What type system is important in FP?
  - Only static and strong typy system?
- Why? Why not weak and dynamic?
  - Do we want to make monkeys happy, or do we want to produce correct programs?
  - Types Are About Composability
  - Type checking provides barrier against nonsensical programs
  - This is what Haskell and C++ have
  - Strong and static type system with type inference

Type Inference: Haskell

- What is a type inference?

Type Inference: Haskell

- What is a type inference?
  - Automatic deduction of the data type of an expression

Type Inference: Haskell

- What is a type inference?
  - Automatic deduction of the data type of an expression
  - Based on value of expression

Type Inference: Haskell

- What is a type inference?
  - Automatic deduction of the data type of an expression
  - Based on value of expression

```
Prelude> let sum a b = a + b
Prelude> :t sum
sum :: Num a => a -> a -> a
```

Type Inference: C++

- Does C++ support type inference?

Type Inference: C++

- Does C++ support type inference?
  - And...

Type Inference: C++

- Does C++ support type inference?
  - And...
  - Yes

Type Inference: C++

- Does C++ support type inference?
  - And...
  - Yes
- C++ Features related to type inference

Type Inference: C++

- Does C++ support type inference?
  - And...
  - Yes
- C++ Features related to type inference
  - auto keyword, deducing the type of variable from its initializer expression

Type Inference: C++

- Does C++ support type inference?
  - And...
  - Yes
- C++ Features related to type inference
  - auto keyword, deducing the type of variable from its initializer expression
  - decltype(auto) and return type deduction for normal functions

Type Inference: C++

- Does C++ support type inference?
  - And...
  - Yes
- C++ Features related to type inference
  - auto keyword, deducing the type of variable from its initializer expression
  - decltype(auto) and return type deduction for normal functions
  - Alternate type deduction when declaring a variable

Type Inference: C++

- Does C++ support type inference?
  - And...
  - Yes
- C++ Features related to type inference
  - auto keyword, deducing the type of variable from its initializer expression
  - decltype(auto) and return type deduction for normal functions
  - Alternate type deduction when declaring a variable
  - Type deduction for lambda arguments

Type Inference: C++

```
//decltype(auto) and return type deduction for normal functions
auto f() { return 42; } // return type is int
template<typename T, typename U>
auto add(T t, U u) -> decltype(t + u); //return type is operator+(T,U)

//Alternate type deduction when declaring a variable
int i;
auto x4a = (i);                // decltype(x4a) is int
decltype(auto) x4d = (i);      // decltype(x4d) is int&

//Type deduction for lambda arguments
[](auto x, auto y) {return x + y;};
```

Lazy Evaluation: Haskell

- What is a lazy evaluation?

Lazy Evaluation: Haskell

- What is a lazy evaluation?
  - Evaluation at a time of need not at a time of a definition

Lazy Evaluation: Haskell

- What is a lazy evaluation?
  - Evaluation at a time of need not at a time of a definition
  - Only when value is demanded

Lazy Evaluation: Haskell

- What is a lazy evaluation?
  - Evaluation at a time of need not at a time of a definition
  - Only when value is demanded

```
lazy n = take n (filter odd [1..])
print (lazy 5) -- prints 1,3,5,7,9
```

Lazy Evaluation

- Does C++ support lazy evaluation?

Lazy Evaluation

- Does C++ support lazy evaluation?
  - No, C++ is inherently eager.

Lazy Evaluation

- Does C++ support lazy evaluation?
  - No, C++ is inherently eager.
- But... there are:

Lazy Evaluation

- Does C++ support lazy evaluation?
  - No, C++ is inherently eager.
- But... there are:
  - Custom solutions based on: suspensions, memoization with thunks, infinite lists, streams with cels combinators.

Lazy Evaluation

- Does C++ support lazy evaluation?
  - No, C++ is inherently eager.
- But... there are:
  - Custom solutions based on: suspensions, memoization with thunks, infinite lists, streams with cels combinators.
  - Hana and Range v3 library

Lazy Evaluation

- Does C++ support lazy evaluation?
  - No, C++ is inherently eager.
- But... there are:
  - Custom solutions based on: suspensions, memoization with thunks, infinite lists, streams with cels combinators.
  - Hana and Range v3 library
- Why laziness is important?

Lazy Evaluation

- Does C++ support lazy evaluation?
  - No, C++ is inherently eager.
- But... there are:
  - Custom solutions based on: suspensions, memoization with thunks, infinite lists, streams with cels combinators.
  - Hana and Range v3 library
- Why laziness is important?
  - Inverts the flow of control

Lazy Evaluation

- Does C++ support lazy evaluation?
  - No, C++ is inherently eager.
- But... there are:
  - Custom solutions based on: suspensions, memoization with thunks, infinite lists, streams with cels combinators.
  - Hana and Range v3 library
- Why laziness is important?
  - Inverts the flow of control
  - Makes the code more reusable

Lazy Evaluation

- Does C++ support lazy evaluation?
  - No, C++ is inherently eager.
- But... there are:
  - Custom solutions based on: suspensions, memoization with thunks, infinite lists, streams with cels combinators.
  - Hana and Range v3 library
- Why laziness is important?
  - Inverts the flow of control
  - Makes the code more reusable
- But, laziness is tricky

Lazy Evaluation

- Does C++ support lazy evaluation?
  - No, C++ is inherently eager.
- But... there are:
  - Custom solutions based on: suspensions, memoization with thunks, infinite lists, streams with cels combinators.
  - Hana and Range v3 library
- Why laziness is important?
  - Inverts the flow of control
  - Makes the code more reusable
- But, laziness is tricky
  - Harder to reason about

Lazy Evaluation

- Does C++ support lazy evaluation?
  - No, C++ is inherently eager.
- But... there are:
  - Custom solutions based on: suspensions, memoization with thunks, infinite lists, streams with cels combinators.
  - Hana and Range v3 library
- Why laziness is important?
  - Inverts the flow of control
  - Makes the code more reusable
- But, laziness is tricky
  - Harder to reason about
  - Performance bottleneck of exhaustive memory deallocations

Lazy Evaluation: Haskell and C++

- Haskell (All)

```
triples = [(x, y, z) | z<-[1..], x<-[1..z], y<-[x..z], x^2 + y^2 == z^2]
main = print (take 10 triples)
```

- C++ (printing part)

```
auto strm = triples().take(10);
forEach(move(strm), [](tuple<int, int, int> const & t) {
    cout << get<0>(t) << ", " << get<1>(t) << ", "<< get<2>(t) << endl;
});
```

Lazy Evaluation: More C++ (Triple)

```
Stream<std::tuple<int, int, int>> triples()
{
    return mbind(intsFrom(1), [](int z)
    {
        return mbind(ints(1, z), [z](int x)
        {
            return mbind(ints(x, z), [x, z](int y)
            {
                return mthen(guard(x*x + y*y == z*z), [x, y, z]()
                {
                    return mreturn(std::make_tuple(x, y, z));
                });
            });
        });
    });
}
```

Lazy Evaluation: More C++ (Functions)

```
Stream<void*> guard(bool b)
{
    if (b) return Stream<void*>(nullptr);
    else return Stream<void*>();
}

template<class T>
Stream<T> mreturn(T v)
{
    return Stream<T>([v]() {
        return Cell<T>(v);
    });
}

template<class T, class F>
auto mthen(Stream<T> stm, F f) -> decltype(f())
{
    return mjoin(fmapv(stm, f));
}
```

Lazy Evaluation: More C++ (Functions)

```
template<class T, class F>
auto fmapv(Stream<T> stm, F f)->Stream<decltype(f())>
{
    using U = decltype(f());
    static_assert(std::is_convertible<F, std::function<U()>>::value,
        "fmapv requires a function type U()");

    if (stm.isEmpty()) return Stream<U>();
    return Stream<U>([stm, f]()
    {
        return Cell<U>(f(), fmapv(stm.pop_front(), f));
    });
}
```

Lazy Evaluation: More C++ (Functions)

```
template<class T, class F>
auto mbind(Stream<T> stm, F f) -> decltype(f(stm.get()))
{
    return mjoin(fmap(stm, f));
}
```

Lazy Evaluation: More C++ (Functions)

```
template<class T>
Stream<T> mjoin(Stream<Stream<T>> stm)
{
    while (!stm.isEmpty() && stm.get().isEmpty())
    {
        stm = stm.pop_front();
    }
    if (stm.isEmpty()) return Stream<T>();
    return Stream<T>([stm]()
    {
        Stream<T> hd = stm.get();
        return Cell<T>( hd.get()
                      , concat(hd.pop_front(), mjoin(stm.pop_front())));
    });
}
```

Lazy Evaluation: More C++ (Functions)

```
template<class T, class F>
void forEach(Stream<T> strm, F f)
{
    while (!strm.isEmpty())
    {
        f(strm.get());
        strm = strm.pop_front();
    }
}
```

Lazy Evaluation: More C++ (Functions)

```
Stream take(int n) const {
    if (n == 0 || isEmpty())
        return Stream();
    auto cell = _lazyCell;
    return Stream([cell, n]()
    {
        auto v = cell->get().val();
        auto t = cell->get().pop_front();
        return Cell<T>(v, t.take(n - 1));
    });
}
```

Lazy Evaluation: More C++ (Functions)

```
Stream<int> ints(int n, int m)
{
    if (n > m)
        return Stream<int>();
    return Stream<int>([n, m]()
    {
        return Cell<int>(n, ints(n + 1, m));
    });
}

Stream<int> intsFrom(int n)
{
    return Stream<int>([n]()
    {
        return Cell<int>(n, intsFrom(n + 1));
    });
}
```

Lazy Evaluation: More C++ (Stream)

```
template<class T>
class Stream
{
    std::shared_ptr <Susp<Cell<T>>> _lazyCell;
public:
    Stream() {}
    Stream(std::function<Cell<T>()> f)
        : _lazyCell(std::make_shared<Susp<Cell<T>>>(f))
    {}
    Stream(Stream && stm)
        : _lazyCell(std::move(stm._lazyCell))
    {}
    Stream & operator=(Stream && stm)
    {
        _lazyCell = std::move(stm._lazyCell);
        return *this;
    }
    bool isEmpty() const   { return !_lazyCell;  }
    T get() const  {return _lazyCell->get().val(); }
    Stream<T> pop_front() const  { return _lazyCell->get().pop_front(); }
};
```

Lazy Evaluation: More C++ (Cell)

```
template<class T>
class Cell
{
public:
    Cell() {} // need default constructor for memoization
    Cell(T v, Stream<T> const & tail)
        : _v(v), _tail(tail)
    {}
    explicit Cell(T v) : _v(v) {}
    T val() const {
        return _v;
    }
    Stream<T> pop_front() const {
        return _tail;
    }
private:
    T _v;
    Stream<T> _tail;
};
```

Lazy Evaluation: More C++ (Susp)

```
template<class T> class Susp
{
    static T const & thunkForce(Susp * susp) {
        return susp->setMemo();
    }
    static T const & thunkGet(Susp * susp) {return susp->getMemo();}
    T const & getMemo() {return _memo;}
    T const & setMemo() {
        _memo = _f();
        _thunk = &thunkGet;
        return getMemo();
    }
public:
    explicit Susp(std::function<T()> f)
        : _f(f), _thunk(&thunkForce), _memo(T()) {}
    T const & get() {return _thunk(this); }
private:
    T const & (*_thunk)(Susp *);
    mutable T   _memo;
    std::function<T()> _f;
};
```

C++ Lazy Evaluation Summary

- No standard support for Laziness in C++

C++ Lazy Evaluation Summary

- No standard support for Laziness in C++
- Hundred of lines vs 2 lines

C++ Lazy Evaluation Summary

- No standard support for Laziness in C++
- Hundred of lines vs 2 lines
- Fortunately there are Range V3 library, but about it later...

Immutable Data: Haskell

- What is an immutable object?

Immutable Data: Haskell

- What is an immutable object?
  - It is an object whose state cannot be altered after its creation

Immutable Data: Haskell

- What is an immutable object?
  - It is an object whose state cannot be altered after its creation
- What are the features of immutable objects?

Immutable Data: Haskell

- What is an immutable object?
  - It is an object whose state cannot be altered after its creation
- What are the features of immutable objects?
  - Simple and efficient threading, correctness and speed

Immutable Data: Haskell

- What is an immutable object?
  - It is an object whose state cannot be altered after its creation
- What are the features of immutable objects?
  - Simple and efficient threading, correctness and speed
  - Object creation on modification

Immutable and Persistent Data: C++

- Does C++ support persistent data structures?

Immutable and Persistent Data: C++

- Does C++ support persistent data structures?
  - No, but you could write one.

Immutable and Persistent Data: C++

- Does C++ support persistent data structures?
  - No, but you could write one.
- What are the features of persistent structures?

Immutable and Persistent Data: C++

- Does C++ support persistent data structures?
  - No, but you could write one.
- What are the features of persistent structures?
  - Immutability: avoiding implicit long-distance couplings (easier maintenance)

Immutable and Persistent Data: C++

- Does C++ support persistent data structures?
  - No, but you could write one.
- What are the features of persistent structures?
  - Immutability: avoiding implicit long-distance couplings (easier maintenance)
  - Multithreading: thread-safe, read only data structures, no data race

Immutable and Persistent Data: C++

- Does C++ support persistent data structures?
  - No, but you could write one.
- What are the features of persistent structures?
  - Immutability: avoiding implicit long-distance couplings (easier maintenance)
  - Multithreading: thread-safe, read only data structures, no data race
  - Performance: drawback, naive requires a lot of copying (deep), storing only differences and shallow copying

Immutable and Persistent Data: C++

- Does C++ support persistent data structures?
  - No, but you could write one.
- What are the features of persistent structures?
  - Immutability: avoiding implicit long-distance couplings (easier maintenance)
  - Multithreading: thread-safe, read only data structures, no data race
  - Performance: drawback, naive requires a lot of copying (deep), storing only differences and shallow copying
- What are the requirements and techniques?

Immutable and Persistent Data: C++

- Does C++ support persistent data structures?
  - No, but you could write one.
- What are the features of persistent structures?
  - Immutability: avoiding implicit long-distance couplings (easier maintenance)
  - Multithreading: thread-safe, read only data structures, no data race
  - Performance: drawback, naive requires a lot of copying (deep), storing only differences and shallow copying
- What are the requirements and techniques?
  - Copy new on modification

Immutable and Persistent Data: C++

- Does C++ support persistent data structures?
  - No, but you could write one.
- What are the features of persistent structures?
  - Immutability: avoiding implicit long-distance couplings (easier maintenance)
  - Multithreading: thread-safe, read only data structures, no data race
  - Performance: drawback, naive requires a lot of copying (deep), storing only differences and shallow copying
- What are the requirements and techniques?
  - Copy new on modification
  - Based on shallow copy, use of shared pointers

Immutable and Persistent Data: C++

- Does C++ support persistent data structures?
  - No, but you could write one.
- What are the features of persistent structures?
  - Immutability: avoiding implicit long-distance couplings (easier maintenance)
  - Multithreading: thread-safe, read only data structures, no data race
  - Performance: drawback, naive requires a lot of copying (deep), storing only differences and shallow copying
- What are the requirements and techniques?
  - Copy new on modification
  - Based on shallow copy, use of shared pointers
  - Enforce constness

Immutable and Persistent Data: C++ Example

- Persistent data structures should be the part of STL or at least of Boost.

Immutable and Persistent Data: C++ Example

- Persistent data structures should be the part of STL or at least of Boost.
- Example: Single Linked List

Immutable and Persistent Data: C++ Example

- Persistent data structures should be the part of STL or at least of Boost.
- Example: Single Linked List
  - A list of T is either empty or consists of an element of type T followed by a list of T.

Immutable and Persistent Data: C++

```
template<class T> class List {
struct Item {
    Item(T v, std::shared_ptr<const Item> const & tail)
        : _val(v), _next(tail) {}
    T _val;
    std::shared_ptr<const Item> _next;
}
public:
    List() : _head(nullptr) {}
    List(T v, List const & tail)  :
        _head(std::make_shared<Item>(v, tail._head)) {}
    bool isEmpty() const { return !_head; }
    //and many many other functions: pop, push, front map, filter, size,
    //fold, concat, reverse, foreach,
private:
    Item const * _head;
};
```

Algebraic Data Types: Haskell

- What is an Algebraic Data Type?

Algebraic Data Types: Haskell

- What is an Algebraic Data Type?
  - **Composed** structure by combination of other types with **algebraic** operations

Algebraic Data Types: Haskell

- What is an Algebraic Data Type?
  - **Composed** structure by combination of other types with **algebraic** operations
    - **sum**: is alternation (A | B, meaning A or B but not both)

Algebraic Data Types: Haskell

- What is an Algebraic Data Type?
  - **Composed** structure by combination of other types with **algebraic** operations
    - **sum**: is alternation (A | B, meaning A or B but not both)
    - **product**: is combination (A B, meaning A and B together)

Algebraic Data Types: Haskell

- What is an Algebraic Data Type?
  - **Composed** structure by combination of other types with **algebraic** operations
    - **sum**: is alternation (A | B, meaning A or B but not both)
    - **product**: is combination (A B, meaning A and B together)
- What are the types of Algebraic Data Type?

Algebraic Data Types: Haskell

- What is an Algebraic Data Type?
  - **Composed** structure by combination of other types with **algebraic** operations
    - **sum**: is alternation (A | B, meaning A or B but not both)
    - **product**: is combination (A B, meaning A and B together)
- What are the types of Algebraic Data Type?
  - parametric: support full parametric polymorphism

Algebraic Data Types: Haskell

- What is an Algebraic Data Type?
  - **Composed** structure by combination of other types with **algebraic** operations
    - **sum**: is alternation (A | B, meaning A or B but not both)
    - **product**: is combination (A B, meaning A and B together)
- What are the types of Algebraic Data Type?
  - parametric: support full parametric polymorphism
  - recursive

Algebraic Data Types: Haskell

- What is an Algebraic Data Type?
  - **Composed** structure by combination of other types with **algebraic** operations
    - **sum**: is alternation (A | B, meaning A or B but not both)
    - **product**: is combination (A B, meaning A and B together)
- What are the types of Algebraic Data Type?
  - parametric: support full parametric polymorphism
  - recursive

```
data Bool = False | True deriving (Ord)
data Day = Monday | ... | Sunday  deriving (Eq, Ord, Show, Bounded, Enum)
data Point = Point Float Float
data Shape = Circle Point Float | Rectangle Point Point
data Maybe a = Nothing | Just a  --Parametric
data List a = Nil | Cons a (List a) --Recursive
data Vector a = Vector a a a deriving (Show)
data Car a b = Car {model::a, year::b} deriving (Show)
```

Algebraic Data Types: C++

- Does C++ support Algebraic Data Types?

Algebraic Data Types: C++

- Does C++ support Algebraic Data Types?
  - No, but there are ...

Algebraic Data Types: C++

- Does C++ support Algebraic Data Types?
  - No, but there are ...
  - classes (RTTI, virtuality), unions, and std::variant, so you can have its equivalents

Algebraic Data Types: C++

- Does C++ support Algebraic Data Types?
  - No, but there are ...
  - classes (RTTI, virtuality), unions, and std::variant, so you can have its equivalents

```
struct nothing {};
template<class T>
using maybe<T> = boost::variant<T, nothing>;
```

Pattern Matching: Haskell

- What is a pattern matching?

Pattern Matching: Haskell

- What is a pattern matching?
  - Creating a function definition based on pattern

Pattern Matching: Haskell

- What is a pattern matching?
  - Creating a function definition based on pattern
  - Is the elimination construct for algebraic data types

Pattern Matching: Haskell

- What is a pattern matching?
  - Creating a function definition based on pattern
  - Is the elimination construct for algebraic data types

```
lucky :: Int -> String
lucky 7 = "LUCKY NUMBER SEVEN!"
lucky x = "Sorry, you're out of luck, pal!"

addVectors :: Double a => (a, a) -> (a, a) -> (a, a)
addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

Pattern Matching: C++

- Does C++ support pattern matching?

Pattern Matching: C++

- Does C++ support pattern matching?
  - No, but you have ...

Pattern Matching: C++

- Does C++ support pattern matching?
  - No, but you have ...
  - Mach7 library (by Yuriy Solodkyy, Gabriel Dos Reis and Bjarne Stroustrup)

Pattern Matching: C++ Mach7 Example

- The use of patterns with built-in types

```
int fib(int n)
{
    var<int> m;

    Match(n)
    {
      Case(1)      return 1;
      Case(2)      return 1;
      Case(2*m)    return sqr(fib(m+1)) - sqr(fib(m-1));
      Case(2*m+1)  return sqr(fib(m+1)) + sqr(fib(m));
    }
    EndMatch
}
```

Substitution Model: Haskell

- What is a substitution model?

Substitution Model: Haskell

- What is a substitution model?
  - The most important concept defining functional paradigm

Substitution Model: Haskell

- What is a substitution model?
  - The most important concept defining functional paradigm
  - Process of reduction expressions on equals-for-equals basis

Substitution Model: Haskell

- What is a substitution model?
  - The most important concept defining functional paradigm
  - Process of reduction expressions on equals-for-equals basis
- Prerequisites?

Substitution Model: Haskell

- What is a substitution model?
  - The most important concept defining functional paradigm
  - Process of reduction expressions on equals-for-equals basis
- Prerequisites?
  - No assignment operator, equal sign is a equivalence/defining operator

Substitution Model: Haskell

- What is a substitution model?
  - The most important concept defining functional paradigm
  - Process of reduction expressions on equals-for-equals basis
- Prerequisites?
  - No assignment operator, equal sign is a equivalence/defining operator
  - Purity

Substitution Model: Haskell

- What is a substitution model?
  - The most important concept defining functional paradigm
  - Process of reduction expressions on equals-for-equals basis
- Prerequisites?
  - No assignment operator, equal sign is a equivalence/defining operator
  - Purity
- Example, what does it mean?

Substitution Model: Haskell

- What is a substitution model?
  - The most important concept defining functional paradigm
  - Process of reduction expressions on equals-for-equals basis
- Prerequisites?
  - No assignment operator, equal sign is a equivalence/defining operator
  - Purity
- Example, what does it mean?

```
x = 5
```

Substitution Model: C++

- Does C++ support substitution model?

Substitution Model: C++

- Does C++ support substitution model?
  - No, It can not.

Substitution Model: C++

- Does C++ support substitution model?
  - No, It can not.
- Why:

Substitution Model: C++

- Does C++ support substitution model?
  - No, It can not.
- Why:
  - The C++ is not pure (programming with self discipline)

Substitution Model: C++

- Does C++ support substitution model?
  - No, It can not.
- Why:
  - The C++ is not pure (programming with self discipline)
  - Functions are subroutines

Substitution Model: C++

- Does C++ support substitution model?
  - No, It can not.
- Why:
  - The C++ is not pure (programming with self discipline)
  - Functions are subroutines
  - There is an assignment operator

Substitution Model: C++

- Does C++ support substitution model?
  - No, It can not.
- Why:
  - The C++ is not pure (programming with self discipline)
  - Functions are subroutines
  - There is an assignment operator
  - It is imperative language

Substitution Model: C++

- Does C++ support substitution model?
  - No, It can not.
- Why:
  - The C++ is not pure (programming with self discipline)
  - Functions are subroutines
  - There is an assignment operator
  - It is imperative language
  - It is impossible to substitute equal-for-equal

Polymorphic Types: Haskell

- What are the polymorphic types?

Polymorphic Types: Haskell

- What are the polymorphic types?
  - Are types that are universally quantified in some way over all types

Polymorphic Types: Haskell

- What are the polymorphic types?
  - Are types that are universally quantified in some way over all types
  - Support parametric polymorphism (generics)

Polymorphic Types: Haskell

- What are the polymorphic types?
  - Are types that are universally quantified in some way over all types
  - Support parametric polymorphism (generics)
  - Describe families of types

Polymorphic Types: Haskell

- What are the polymorphic types?
  - Are types that are universally quantified in some way over all types
  - Support parametric polymorphism (generics)
  - Describe families of types

```
id :: a -> a
id a = a

map :: (a->b) -> [a] -> [b]
map f  []  =  []
map f (x:xs)  =  f x : map f xs
```

Type Classes and Overloading: Haskell

- What are type classes and overloading?

Type Classes and Overloading: Haskell

- What are type classes and overloading?
  - It is an interface that defines some behaviour

Type Classes and Overloading: Haskell

- What are type classes and overloading?
  - It is an interface that defines some behaviour
  - It is a type system construct that supports ad-hoc polymorphism (overloading)

Type Classes and Overloading: Haskell

- What are type classes and overloading?
  - It is an interface that defines some behaviour
  - It is a type system construct that supports ad-hoc polymorphism (overloading)

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

data Foo = Foo {x :: Integer, str :: String}

instance Eq Foo where
   (Foo x1 str1) == (Foo x2 str2) = (x1 == x2) && (str1 == str2)
```

Polymorphism: C++

- Does C++ support polymorphism?

Polymorphism: C++

- Does C++ support polymorphism?
  - No

Polymorphism: C++

- Does C++ support polymorphism?
  - No
- Just Kidding... How many polymorphism are in C++?

Polymorphism: C++

- Does C++ support polymorphism?
  - No
- Just Kidding... How many polymorphism are in C++?
- There are four kind of polymorphism in C++

Polymorphism: C++

- Does C++ support polymorphism?
  - No
- Just Kidding... How many polymorphism are in C++?
- There are four kind of polymorphism in C++
  - Subtype polymorphism (runtime)

Polymorphism: C++

- Does C++ support polymorphism?
  - No
- Just Kidding... How many polymorphism are in C++?
- There are four kind of polymorphism in C++
  - Subtype polymorphism (runtime)
  - Parametric polymorphism (compile-time)

Polymorphism: C++

- Does C++ support polymorphism?
  - No
- Just Kidding... How many polymorphism are in C++?
- There are four kind of polymorphism in C++
  - Subtype polymorphism (runtime)
  - Parametric polymorphism (compile-time)
  - Ad-hoc polymorphism (overloading)

Polymorphism: C++

- Does C++ support polymorphism?
  - No
- Just Kidding... How many polymorphism are in C++?
- There are four kind of polymorphism in C++
  - Subtype polymorphism (runtime)
  - Parametric polymorphism (compile-time)
  - Ad-hoc polymorphism (overloading)
  - Coercion polymorphism (implicit or explicit casting)

List Comprehension: Haskell

- What is a list comprehension?

List Comprehension: Haskell

- What is a list comprehension?
  - Returns a list of elements created by evaluation of the generators

List Comprehension: Haskell

- What is a list comprehension?
  - Returns a list of elements created by evaluation of the generators
  - Similar to the mathematical concept of set comprehensions

List Comprehension: Haskell

- What is a list comprehension?
  - Returns a list of elements created by evaluation of the generators
  - Similar to the mathematical concept of set comprehensions

```
print ([x*2 | x <- [1..10], x*2 >= 12])
-- prints [12,14,16,18,20]
```

List Comprehension: C++

- Does C++ support list comprehension?

List Comprehension: C++

- Does C++ support list comprehension?
  - No, not directly, but there is ...

List Comprehension: C++

- Does C++ support list comprehension?
  - No, not directly, but there is ...
- Range comprehension, by Range v3 library

List Comprehension: C++

- Does C++ support list comprehension?
  - No, not directly, but there is ...
- Range comprehension, by Range v3 library
  - Range library for C++11/14/17

List Comprehension: C++

- Does C++ support list comprehension?
  - No, not directly, but there is ...
- Range comprehension, by Range v3 library
  - Range library for C++11/14/17
  - Basis of a formal proposal to add range support to the C++ standard library

List Comprehension: C++

- Does C++ support list comprehension?
  - No, not directly, but there is ...
- Range comprehension, by Range v3 library
  - Range library for C++11/14/17
  - Basis of a formal proposal to add range support to the C++ standard library
  - Is lazy

List Comprehension: C++

```
// Lazy ranges for generating integer sequences
auto const intsFrom = view::iota;
auto const ints = [=](int i, int j) {
    return view::take(intsFrom(i), j-i+1);
};
//Define an infinite range of all the Pythagorean triples:
auto triples =
  view::for_each(intsFrom(1), [](int z) {
    return view::for_each(ints(1, z), [=](int x) {
      return view::for_each(ints(x, z), [=](int y) {
        return yield_if(x*x + y*y == z*z, std::make_tuple(x, y, z));
        });
     });
 });
```

Automatic Memory Management: Haskell

- Why purity, laziness and immutability requires automatic dynamic memory management?

Automatic Memory Management: Haskell

- Why purity, laziness and immutability requires automatic dynamic memory management?
  - Immutability forces to produce a lot of temporary data

Automatic Memory Management: Haskell

- Why purity, laziness and immutability requires automatic dynamic memory management?
  - Immutability forces to produce a lot of temporary data
  - The lifetime of an object can only be determined at runtime

Automatic Memory Management: C++

- Does C++ support automatic memory management?

Automatic Memory Management: C++

- Does C++ support automatic memory management?
  - No built-in garbage collector

Automatic Memory Management: C++

- Does C++ support automatic memory management?
  - No built-in garbage collector
  - RAII technique realized by smart pointers

Automatic Memory Management: C++

- Does C++ support automatic memory management?
  - No built-in garbage collector
  - RAII technique realized by smart pointers
  - Semi-automatic memory management

Category Concepts: Functor, Applicative, Monad

- What are the category concepts?

Category Concepts: Functor, Applicative, Monad

- What are the category concepts?
  - **Functor**: a container that can be mapped over (map)

Category Concepts: Functor, Applicative, Monad

- What are the category concepts?
  - **Functor**: a container that can be mapped over (map)
  - **Applicative**: a Functor plus lifting values and combining computations (lift, ap)

Category Concepts: Functor, Applicative, Monad

- What are the category concepts?
  - **Functor**: a container that can be mapped over (map)
  - **Applicative**: a Functor plus lifting values and combining computations (lift, ap)
  - **Monad**: an Applicative plus flattening nested levels of structure (flatten,chain;bind,return;join,unit)

Category Concepts: Functor, Applicative, Monad

- What are the category concepts?
  - **Functor**: a container that can be mapped over (map)
  - **Applicative**: a Functor plus lifting values and combining computations (lift, ap)
  - **Monad**: an Applicative plus flattening nested levels of structure (flatten,chain;bind,return;join,unit)
- Why are those concepts important?

Category Concepts: Functor, Applicative, Monad

- What are the category concepts?
  - **Functor**: a container that can be mapped over (map)
  - **Applicative**: a Functor plus lifting values and combining computations (lift, ap)
  - **Monad**: an Applicative plus flattening nested levels of structure (flatten,chain;bind,return;join,unit)
- Why are those concepts important?
  - Formalize many programming structures and techniques

Category Concepts: Functor, Applicative, Monad

- What are the category concepts?
  - **Functor**: a container that can be mapped over (map)
  - **Applicative**: a Functor plus lifting values and combining computations (lift, ap)
  - **Monad**: an Applicative plus flattening nested levels of structure (flatten,chain;bind,return;join,unit)
- Why are those concepts important?
  - Formalize many programming structures and techniques
  - In Haskell very important

Category Concepts: Functor, Applicative, Monad

- What are the category concepts?
  - **Functor**: a container that can be mapped over (map)
  - **Applicative**: a Functor plus lifting values and combining computations (lift, ap)
  - **Monad**: an Applicative plus flattening nested levels of structure (flatten,chain;bind,return;join,unit)
- Why are those concepts important?
  - Formalize many programming structures and techniques
  - In Haskell very important
  - In C++ getting more and more popular and important

Category Concepts: Functor, Applicative, Monad

- What are the category concepts?
  - **Functor**: a container that can be mapped over (map)
  - **Applicative**: a Functor plus lifting values and combining computations (lift, ap)
  - **Monad**: an Applicative plus flattening nested levels of structure (flatten,chain;bind,return;join,unit)
- Why are those concepts important?
  - Formalize many programming structures and techniques
  - In Haskell very important
  - In C++ getting more and more popular and important
- C++ Implementations?

Category Concepts: Functor, Applicative, Monad

- What are the category concepts?
  - **Functor**: a container that can be mapped over (map)
  - **Applicative**: a Functor plus lifting values and combining computations (lift, ap)
  - **Monad**: an Applicative plus flattening nested levels of structure (flatten,chain;bind,return;join,unit)
- Why are those concepts important?
  - Formalize many programming structures and techniques
  - In Haskell very important
  - In C++ getting more and more popular and important
- C++ Implementations?
  - Boost.Hana

Category Concepts: Functor, Applicative, Monad

- What are the category concepts?
  - **Functor**: a container that can be mapped over (map)
  - **Applicative**: a Functor plus lifting values and combining computations (lift, ap)
  - **Monad**: an Applicative plus flattening nested levels of structure (flatten,chain;bind,return;join,unit)
- Why are those concepts important?
  - Formalize many programming structures and techniques
  - In Haskell very important
  - In C++ getting more and more popular and important
- C++ Implementations?
  - Boost.Hana
    - Applicative, Comonad, Functor, Group, Monad, MonadPlus, Monoid, Product, Ring

Category Concepts: Functor, Applicative, Monad

- What are the category concepts?
  - **Functor**: a container that can be mapped over (map)
  - **Applicative**: a Functor plus lifting values and combining computations (lift, ap)
  - **Monad**: an Applicative plus flattening nested levels of structure (flatten,chain;bind,return;join,unit)
- Why are those concepts important?
  - Formalize many programming structures and techniques
  - In Haskell very important
  - In C++ getting more and more popular and important
- C++ Implementations?
  - Boost.Hana
    - Applicative, Comonad, Functor, Group, Monad, MonadPlus, Monoid, Product, Ring
  - Many C++ constructs turns out those concepts

Category Concepts: Functor in Haskell

- What is a functor?

Category Concepts: Functor in Haskell

- What is a functor?
  - A type (container) that can be mapped (transform) over

Category Concepts: Functor in Haskell

- What is a functor?
  - A type (container) that can be mapped (transform) over
  - transforms a container into a new container

Category Concepts: Functor in Haskell

- What is a functor?
  - A type (container) that can be mapped (transform) over
  - transforms a container into a new container
  - or lifts a "normal" function into a "higher" function

Category Concepts: Functor in Haskell

- What is a functor?
  - A type (container) that can be mapped (transform) over
  - transforms a container into a new container
  - or lifts a "normal" function into a "higher" function

```
-- typeclass
class Functor f where
  fmap :: (a -> b) -> f a -> f b  --fmap :: (a -> b) -> (f a -> f b)

-- list functor
instance Functor [] where
  fmap _ []     = []
  fmap g (x:xs) = g x : fmap g xs  -- or we could just say fmap = map

--laws
fmap id = id
fmap (g . h) = (fmap g) . (fmap h)
```

Category Concepts: Functor in C++

- It is a concept that represents types that can be mapped over (by boost.hana))

Category Concepts: Functor in C++

- It is a concept that represents types that can be mapped over (by boost.hana))
- What are the concrete functors in C++?

Category Concepts: Functor in C++

- It is a concept that represents types that can be mapped over (by boost.hana))
- What are the concrete functors in C++?
  - STL Containers with std::transform and iterators

Category Concepts: Functor in C++

- It is a concept that represents types that can be mapped over (by boost.hana))
- What are the concrete functors in C++?
  - STL Containers with std::transform and iterators
  - Boost.Hana: hana::lazy, hana::optional, hana::tuple

Category Concepts: Functor in C++

- It is a concept that represents types that can be mapped over (by boost.hana))
- What are the concrete functors in C++?
  - STL Containers with std::transform and iterators
  - Boost.Hana: hana::lazy, hana::optional, hana::tuple
  - Range v3: views and ranges

Category Concepts: Functor in C++

- It is a concept that represents types that can be mapped over (by boost.hana))
- What are the concrete functors in C++?
  - STL Containers with std::transform and iterators
  - Boost.Hana: hana::lazy, hana::optional, hana::tuple
  - Range v3: views and ranges

```
//laws (boost.hana)
transform(xs, id) == xs
transform(xs, compose(g, f)) == transform(transform(xs, f), g)
//examples
auto to_string = [](auto x) { ostringstream ss; ss << x; return ss.str(); };
BOOST_HANA_RUNTIME_CHECK(
hana::transform(hana::make_tuple(1, '2', "345", string{"67"}), to_string)
== hana::make_tuple("1", "2", "345", "67"));

std::vector<int> vi{1,2,3,4,5,6,7,8,9,10};
auto rng = vi | view::transform([](int i){return i * 2;});
```

Category Concepts: Applicative in Haskell

- Apply container of functions to container of values

Category Concepts: Applicative in Haskell

- Apply container of functions to container of values

```
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b --apply

pure id <*> v = v
pure f <*> pure x = pure (f x)
u <*> pure y = pure ($ y) <*> u
u <*> (v <*> w) = pure (.) <*> u <*> v <*> w

instance Applicative [] where
  pure x = [x]
  fs <*> xs = [f x | f <- fs, x <- xs]

print (pure 123 :: [Int]) --[123]
print([(*0),(+100),(^2)] <*> [1,2,3]) --[0,0,0,101,102,103,1,4,9]
print([(+),(*)] <*> [1,2] <*> [3,4] )
--[(+1),(+2),(*1),(*2)] <*> [3,4], [4,5,5,6,3,4,6,8]
```

Category Concepts: Applicative in C++

- What are applicative in C++?

Category Concepts: Applicative in C++

- What are applicative in C++?
  - Represents Functors with the ability to lift values and combine computations (by boost.hana)

Category Concepts: Applicative in C++

- What are applicative in C++?
  - Represents Functors with the ability to lift values and combine computations (by boost.hana)
  - hana::ap, hana::lift

Category Concepts: Applicative in C++

- What are applicative in C++?
  - Represents Functors with the ability to lift values and combine computations (by boost.hana)
  - hana::ap, hana::lift

```
//laws
ap(lift<F>(id), xs) == xs
ap(ap(lift<F>(compose), fs, gs), xs) == ap(fs, ap(gs, xs))
ap(lift<F>(f), lift<F>(x)) == lift<F>(f(x))
ap(fs, lift<F>(x)) == ap(lift<F>(apply(-, x)), fs)

//example
hana::ap(hana::just([](auto a, auto b, auto c) {return a * b * c;}),
  hana::just(1), hana::just(2), hana::just(3)
);
```

Category Concepts: Monads in General

1. What are monads?

Category Concepts: Monads in General

1. What are monads?
   - mathematical constructs from category theory

Category Concepts: Monads in General

1. What are monads?
   - mathematical constructs from category theory
   - programming concepts or design patterns

Category Concepts: Monads in General

1. What are monads?
   - mathematical constructs from category theory
   - programming concepts or design patterns
   - typeclass (or just types)

Category Concepts: Monads in General

1. What are monads?
   - mathematical constructs from category theory
   - programming concepts or design patterns
   - typeclass (or just types)
   - Functor $\rightarrow$Applicative $\rightarrow$Monad

Category Concepts: Monads in General

1. What are monads?
   - mathematical constructs from category theory
   - programming concepts or design patterns
   - typeclass (or just types)
   - Functor $\rightarrow$Applicative $\rightarrow$Monad
2. Why monads are important in fp?

Category Concepts: Monads in General

1. What are monads?
   - mathematical constructs from category theory
   - programming concepts or design patterns
   - typeclass (or just types)
   - Functor $\rightarrow$Applicative $\rightarrow$Monad
2. Why monads are important in fp?
   - formalize some useful things (side effects, io)

Category Concepts: Monads in General

1. What are monads?
   - mathematical constructs from category theory
   - programming concepts or design patterns
   - typeclass (or just types)
   - Functor $\rightarrow$Applicative $\rightarrow$Monad
2. Why monads are important in fp?
   - formalize some useful things (side effects, io)
3. Do we have monads in C++?

Category Concepts: Monads in General

1. What are monads?
   - mathematical constructs from category theory
   - programming concepts or design patterns
   - typeclass (or just types)
   - Functor $\rightarrow$Applicative $\rightarrow$Monad
2. Why monads are important in fp?
   - formalize some useful things (side effects, io)
3. Do we have monads in C++?
   - Yes, monads are in many places (STL, boost:hana, range v3)

Category Concepts: Monads in General

1. What are monads?
   - mathematical constructs from category theory
   - programming concepts or design patterns
   - typeclass (or just types)
   - Functor →Applicative →Monad
2. Why monads are important in fp?
   - formalize some useful things (side effects, io)
3. Do we have monads in C++?
   - Yes, monads are in many places (STL, boost:hana, range v3)
4. Are monads important in C++?

Category Concepts: Monads in General

1. What are monads?
   - mathematical constructs from category theory
   - programming concepts or design patterns
   - typeclass (or just types)
   - Functor $\rightarrow$Applicative $\rightarrow$Monad
2. Why monads are important in fp?
   - formalize some useful things (side effects, io)
3. Do we have monads in C++?
   - Yes, monads are in many places (STL, boost:hana, range v3)
4. Are monads important in C++?
   - no, not really, but ...

Category Concepts: Monads in General

1. What are monads?
   - mathematical constructs from category theory
   - programming concepts or design patterns
   - typeclass (or just types)
   - Functor →Applicative →Monad
2. Why monads are important in fp?
   - formalize some useful things (side effects, io)
3. Do we have monads in C++?
   - Yes, monads are in many places (STL, boost:hana, range v3)
4. Are monads important in C++?
   - no, not really, but ...
   - are getting more and more important

Category Concepts: Monad in Math

- A monad over a category $\mathcal{C}$ is a triple (T, η, μ) of:

Category Concepts: Monad in Math

- A monad over a category $\mathcal{C}$ is a triple (T, η, μ) of:
  - An endofunctor $T : \mathcal{C} \to \mathcal{C}$

Category Concepts: Monad in Math

- A monad over a category $\mathcal{C}$ is a triple (T, η, μ) of:
  - An endofunctor $T : \mathcal{C} \to \mathcal{C}$
  - A natural unit transformation $\eta : 1_{\mathcal{C}} \to T$

Category Concepts: Monad in Math

- A monad over a category $\mathcal{C}$ is a triple (T, η, μ) of:
  - An endofunctor $T : \mathcal{C} \to \mathcal{C}$
  - A natural unit transformation $\eta : 1_{\mathcal{C}} \to T$
  - A natural join transformation $\mu : T^2 \to T$

Category Concepts: Monad in Math

- A monad over a category $\mathcal{C}$ is a triple (T, η, μ) of:
  - An endofunctor $T : \mathcal{C} \to \mathcal{C}$
  - A natural unit transformation $\eta : 1_{\mathcal{C}} \to T$
  - A natural join transformation $\mu : T^2 \to T$
- with rules:

Category Concepts: Monad in Math

- A monad over a category $\mathcal{C}$ is a triple (T, η, μ) of:
  - An endofunctor $T : \mathcal{C} \to \mathcal{C}$
  - A natural unit transformation $\eta : 1_{\mathcal{C}} \to T$
  - A natural join transformation $\mu : T^2 \to T$
- with rules:
  - An associative square: $\mu \circ T\mu = \mu \circ \mu T$

Category Concepts: Monad in Math

- A monad over a category $\mathcal{C}$ is a triple (T, η, μ) of:
  - An endofunctor $T : \mathcal{C} \to \mathcal{C}$
  - A natural unit transformation $\eta : 1_{\mathcal{C}} \to T$
  - A natural join transformation $\mu : T^2 \to T$
- with rules:
  - An associative square: $\mu \circ T\mu = \mu \circ \mu T$
  - a triangle equality: $\mu \circ T\eta = \mu \circ \eta T = 1_{\mathcal{C}}$

Category Concepts: Monad in Math

- A monad over a category $\mathcal{C}$ is a triple (T, η, μ) of:
  - An endofunctor $T : \mathcal{C} \to \mathcal{C}$
  - A natural unit transformation $\eta : 1_{\mathcal{C}} \to T$
  - A natural join transformation $\mu : T^2 \to T$
- with rules:
  - An associative square: $\mu \circ T\mu = \mu \circ \mu T$
  - a triangle equality: $\mu \circ T\eta = \mu \circ \eta T = 1_{\mathcal{C}}$

```
class Endofunctor c t => Monad c t where
  eta :: c a (t a)
  mu  :: c (t (t a)) (t a)
  (>>=) :: (Monad c t) => c a (t b) -> c (t a) (t b)
  (>>=) f = mu . fmap f
  return :: (Monad c t) => c a (t a)
  return = eta
```

Category Concepts: Monad in Haskell

- What are the Monad from programming pov?

Category Concepts: Monad in Haskell

- What are the Monad from programming pov?
  - bind function: the structure of a monadic computation can change based on intermediate results

Category Concepts: Monad in Haskell

- What are the Monad from programming pov?
  - bind function: the structure of a monadic computation can change based on intermediate results

```
class Applicative m => Monad m where
  return :: a -> m a  --pure
  (>>=)  :: m a -> (a -> m b) -> m b  --bind

data  Maybe a  =  Nothing | Just a
instance  Monad Maybe  where
    (Just x) >>= k      = k x
    Nothing  >>= _      = Nothing

maybeHalf :: Int -> Maybe Int
maybeHalf a
         | even a = Just (div a 2)
         | otherwise = Nothing
```

Category Concepts: Monad in C++

- What is the Monad from C++ pov?

Category Concepts: Monad in C++

- What is the Monad from C++ pov?
  - Represents Applicatives with the ability to flatten nested levels of structure (by boost.hana)

Category Concepts: Monad in C++

- What is the Monad from C++ pov?
  - Represents Applicatives with the ability to flatten nested levels of structure (by boost.hana)

```
hana::flatten(hana::make_tuple(hana::make_tuple(1, 2, 3), hana::make_tuple(4, 5
    ==
    hana::make_tuple(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Combinators: Haskell

- What are the combinators?

Combinators: Haskell

- What are the combinators?
  - A function which builds program fragments from program fragments: map, filter, fold/reduce

Combinators: Haskell

- What are the combinators?
  - A function which builds program fragments from program fragments: map, filter, fold/reduce
  - Operating on streams

```
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)

foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

Combinators: C++

- Does C++ have combinators?

Combinators: C++

- Does C++ have combinators?
  - Yes, those are functions from STL and from other libraries

Combinators: C++

- Does C++ have combinators?
  - Yes, those are functions from STL and from other libraries
- STL

Combinators: C++

- Does C++ have combinators?
  - Yes, those are functions from STL and from other libraries
- STL
  - transform, reduce, transform_reduce, accumulate, min, max, partition, for_each, for_each_n, all_of, any_of, none_of

Combinators: C++

- Does C++ have combinators?
  - Yes, those are functions from STL and from other libraries
- STL
  - transform, reduce, transform_reduce, accumulate, min, max, partition, for_each, for_each_n, all_of, any_of, none_of
- Boost.Hana

Combinators: C++

- Does C++ have combinators?
  - Yes, those are functions from STL and from other libraries
- STL
  - transform, reduce, transform_reduce, accumulate, min, max, partition, for_each, for_each_n, all_of, any_of, none_of
- Boost.Hana
  - fold, fold_right, fold_left, min, max, for_each, sum, count, fill, replace, transform, chain, flatten, tap, append, filter, suffix, all, in, find, reverse, scan, zip, zip_with, unfold, partition, insert,

Combinators: C++ Examples

- hana::fold combinator

Combinators: C++ Examples

- hana::fold combinator

```
auto to_string = [](auto x) {
    std::ostringstream ss;
    ss << x;
    return ss.str();
};
int main() {
    auto f = [=](std::string s, auto element) {
        return "f(" + s + ", " + to_string(element) + ")";
    };
    BOOST_HANA_RUNTIME_CHECK(
        hana::fold(hana::make_tuple(2, '3', 4, 5.0), "1", f)
            ==
        "f(f(f(f(1, 2), 3), 4), 5)"
    );
}
```

Concepts Support Summary: Good

- Good support, by built-in features for:

Concepts Support Summary: Good

- Good support, by built-in features for:
  - Type inference: auto and decltype

Concepts Support Summary: Good

- Good support, by built-in features for:
  - Type inference: auto and decltype
  - First class functions: lambda expressions and callable objects

Concepts Support Summary: Good

- Good support, by built-in features for:
  - Type inference: auto and decltype
  - First class functions: lambda expressions and callable objects
  - Combinators: STL

Concepts Support Summary: Good

- Good support, by built-in features for:
  - Type inference: auto and decltype
  - First class functions: lambda expressions and callable objects
  - Combinators: STL
  - Memory management: RAII including smart pointers

Concepts Support Summary: Good

- Good support, by built-in features for:
  - Type inference: auto and decltype
  - First class functions: lambda expressions and callable objects
  - Combinators: STL
  - Memory management: RAII including smart pointers
  - Closures: lambda expressions

Concepts Support Summary: Good

- Good support, by built-in features for:
  - Type inference: auto and decltype
  - First class functions: lambda expressions and callable objects
  - Combinators: STL
  - Memory management: RAII including smart pointers
  - Closures: lambda expressions
  - Static typing

Concepts Support Summary: Good

- Good support, by built-in features for:
  - Type inference: auto and decltype
  - First class functions: lambda expressions and callable objects
  - Combinators: STL
  - Memory management: RAII including smart pointers
  - Closures: lambda expressions
  - Static typing
  - Polymorphism

Concepts Support Summary: Good

- Good support, by built-in features for:
  - Type inference: auto and decltype
  - First class functions: lambda expressions and callable objects
  - Combinators: STL
  - Memory management: RAII including smart pointers
  - Closures: lambda expressions
  - Static typing
  - Polymorphism
  - Algebraic Data Types: std::variant, composition

Concepts Support Summary: Some

- Some support, by libraries for:

Concepts Support Summary: Some

- Some support, by libraries for:
  - Pattern matching: mach7

Concepts Support Summary: Some

- Some support, by libraries for:
  - Pattern matching: mach7
  - Range comprehension: Range v3

Concepts Support Summary: Some

- Some support, by libraries for:
  - Pattern matching: mach7
  - Range comprehension: Range v3
  - Category concepts: Boost.Hana

Concepts Support Summary: Some

- Some support, by libraries for:
  - Pattern matching: mach7
  - Range comprehension: Range v3
  - Category concepts: Boost.Hana
  - Laziness: Range v3, Boost.Hana

Concepts Support Summary: Some

- Some support, by libraries for:
  - Pattern matching: mach7
  - Range comprehension: Range v3
  - Category concepts: Boost.Hana
  - Laziness: Range v3, Boost.Hana
  - Combinators: Boost.Hana, Range v3

Concepts Support Summary: Some

- Some support, by libraries for:
  - Pattern matching: mach7
  - Range comprehension: Range v3
  - Category concepts: Boost.Hana
  - Laziness: Range v3, Boost.Hana
  - Combinators: Boost.Hana, Range v3
  - Partial application and currying: Boost.Hana

Concepts Support Summary: None

- None support:

Concepts Support Summary: None

- None support:
  - Purity: just self discipline

Concepts Support Summary: None

- None support:
  - Purity: just self discipline
  - Immutability: custom solutions, self discipline

Concepts Support Summary: None

- None support:
  - Purity: just self discipline
  - Immutability: custom solutions, self discipline
  - Substitution model: impossible at all

Concepts Support Summary

C++ is in a half way towards full functional paradigm support

C++ Metaprogramming

Metaprogramming as pure functional language

C++ Metaprogramming

Metaprogramming as pure functional language

- Does metaprogramming support functional paradigm?

C++ Metaprogramming

"Within C++, there is a much smaller and cleaner language struggling to get out"

Bjarne Stroustrup

C++ Metaprogramming

"Within C++, there is a much smaller and cleaner language struggling to get out"

Bjarne Stroustrup

- C++ metaprogramming is a pure functional compile-time programming

C++ Metaprogramming

"Within C++, there is a much smaller and cleaner language struggling to get out"

Bjarne Stroustrup

- C++ metaprogramming is a pure functional compile-time programming
- Haskell could be used as a pseudocode for it

C++ Metaprogramming: Characteristics

- Why MP is pure functional language

C++ Metaprogramming: Characteristics

- Why MP is pure functional language
- no side effects by design

C++ Metaprogramming: Characteristics

- Why MP is pure functional language
- no side effects by design
  - no mutable variables

C++ Metaprogramming: Characteristics

- Why MP is pure functional language
- no side effects by design
  - no mutable variables
  - no assignment operator*

C++ Metaprogramming: Characteristics

- Why MP is pure functional language
- no side effects by design
  - no mutable variables
  - no assignment operator*
  - constant expressions

C++ Metaprogramming: Characteristics

- Why MP is pure functional language
- no side effects by design
  - no mutable variables
  - no assignment operator*
  - constant expressions
  - no loops, only recursions*

C++ Metaprogramming: Haskell Factorial

- Factorial in Haskell

C++ Metaprogramming: Haskell Factorial

- Factorial in Haskell

```
fact 0 = 1
fact n = n * fact(n - 1)
main = print(fact 5) --prints 120
```

C++ Metaprogramming: Haskell Factorial

- Factorial in Haskell

```
fact 0 = 1
fact n = n * fact(n - 1)
main = print(fact 5) --prints 120
```

  - two liner

C++ Metaprogramming: Haskell Factorial

- Factorial in Haskell

```
fact 0 = 1
fact n = n * fact(n - 1)
main = print(fact 5) --prints 120
```

  - two liner
  - pattern matching with two definitions

C++ Metaprogramming: Haskell Factorial

- Factorial in Haskell

```
fact 0 = 1
fact n = n * fact(n - 1)
main = print(fact 5) --prints 120
```

  - two liner
  - pattern matching with two definitions
  - recursion

TMP Factorial

- Factorial in C++98 Template Meta Programming

TMP Factorial

- Factorial in C++98 Template Meta Programming

```
template <int n> struct fact98 {
  static const int value = n * fact98<n-1>::value;
};

template <> struct fact98<0> {
  static const int value = 1;
};

std::cout << fact98<5>::value << std::endl; //prints 120
```

TMP Factorial

- Factorial in C++98 Template Meta Programming

```
template <int n> struct fact98 {
  static const int value = n * fact98<n-1>::value;
};

template <> struct fact98<0> {
  static const int value = 1;
};

std::cout << fact98<5>::value << std::endl; //prints 120
```

  - many liner

TMP Factorial

- Factorial in C++98 Template Meta Programming

```
template <int n> struct fact98 {
  static const int value = n * fact98<n-1>::value;
};

template <> struct fact98<0> {
  static const int value = 1;
};

std::cout << fact98<5>::value << std::endl; //prints 120
```

  - many liner
  - template instantiation acts as function calls

TMP Factorial

- Factorial in C++98 Template Meta Programming

```
template <int n> struct fact98 {
  static const int value = n * fact98<n-1>::value;
};

template <> struct fact98<0> {
  static const int value = 1;
};

std::cout << fact98<5>::value << std::endl; //prints 120
```

  - many liner
  - template instantiation acts as function calls
  - pattern matching implemented as specialization

Constexpr C++11 factorial

- Factorial in C++11 Constexpr Meta Programming

Constexpr C++11 factorial

- Factorial in C++11 Constexpr Meta Programming

```
constexpr int fact11(int n) {
    return n <= 1? 1 : (n * fact11(n - 1));
}
std::cout << Int<fact11(5)>::value << std::endl; //prints 120
```

Constexpr C++11 factorial

- Factorial in C++11 Constexpr Meta Programming

```
constexpr int fact11(int n) {
    return n <= 1? 1 : (n * fact11(n - 1));
}
std::cout << Int<fact11(5)>::value << std::endl; //prints 120
```

  - looks better

Constexpr C++14 factorial

- Factorial in C++14 Constexpr Meta Programming

Constexpr C++14 factorial

- Factorial in C++14 Constexpr Meta Programming

```
constexpr int fact14(int n) {
    switch(n) {
      case 0: return 1;
      default: return n * fact14(n-1);
    }
}
constexpr int fact14loop(int n) {
    int s = 1;
    for (int i=1; i<=n; i++) {s = s * i;}
    return s;
}
```

Constexpr C++14 factorial

- Factorial in C++14 Constexpr Meta Programming

```
constexpr int fact14(int n) {
    switch(n) {
      case 0: return 1;
      default: return n * fact14(n-1);
    }
}
constexpr int fact14loop(int n) {
    int s = 1;
    for (int i=1; i<=n; i++) {s = s * i;}
    return s;
}
```

  - pure functional compile-time meta programming in imperative way

- Within imperative language there is pure functional one with an imperative looking syntax

Predicates: Haskell

- Predicate in Haskell:

Predicates: Haskell

- Predicate in Haskell:

```
isZero 0 = True
isZero n = False
```

Predicates: Haskell

- Predicate in Haskell:

```
isZero 0 = True
isZero n = False
```

- Predicate in C+98 TMP:

Predicates: Haskell

- Predicate in Haskell:

```
isZero 0 = True
isZero n = False
```

- Predicate in C+98 TMP:

```
template <int> struct isZero {
  static const bool value = false;
};
template <> struct isZero<0> {
  static const bool value = true;
};
```

Predicates: Haskell

- Predicate in Haskell:

```
isZero 0 = True
isZero n = False
```

- Predicate in C+98 TMP:

```
template <int> struct isZero {
  static const bool value = false;
};
template <> struct isZero<0> {
  static const bool value = true;
};
```

- Almost the same

OR combinator and Closure: Haskell

- Or combinator:

OR combinator and Closure: Haskell

- Or combinator:

```
ori :: (Int -> Bool) -> (Int -> Bool) -> Int -> Bool
ori f g = \x -> (f x) || (g x)
main = print(ori isZero isOne 0) --prints True
```

OR combinator and Closure: Haskell

- Or combinator:

```
ori :: (Int -> Bool) -> (Int -> Bool) -> Int -> Bool
ori f g = \x -> (f x) || (g x)
main = print(ori isZero isOne 0) --prints True
```

  - Higher order function, lambda with closure, predicate composition

OR combinator and Closure: TMP C++

- Or with closure in C++98 TMP:

OR combinator and Closure: TMP C++

- Or with closure in C++98 TMP:

```
//int
template <template <int> class f, template <int> class g>
struct or {
  template<int n> struct closure {
    static const bool value = f<n>::value || g<n>::value;
   };
};

//any type
template <template <class> class F, template <class> class G>
struct Or {
  template<class T> struct closure {
    static const bool value = F<T>::value || G<T>::value;
  };
};

cout << or<isZero,isOne>::closure<0>::value; //prints 1
cout << Or<IsIntZero, IsIntOne>::closure<Int<0>>::value; //prints 1
```

all: haskell

- Higher order function operating on list

all: haskell

- Higher order function operating on list
- All algorithm in haskell

all: haskell

- Higher order function operating on list
- All algorithm in haskell

```
all pred [] = True
all pred (head:tail) = (pred head) && (all pred tail)

--all application on int list with two predicate and or combinator
main = print(alli (ori isZero isOne) [1,1,0,1,0]) --true
```

all: haskell

- Higher order function operating on list
- All algorithm in haskell

```
all pred [] = True
all pred (head:tail) = (pred head) && (all pred tail)

--all application on int list with two predicate and or combinator
main = print(alli (ori isZero isOne) [1,1,0,1,0]) --true
```

- pattern matching, recursion and composition

all: TMP C++

- The all algorithm equivalence in C++

all: TMP C++

- The all algorithm equivalence in C++

```
template <template <int> class pred, int... list> struct alli;

template <template <int> class pred>
struct alli<pred> { static const bool value = true; };

template <template <int> class pred, int head, int... tail>
struct alli<pred, head, tail...> {
  static const bool value = pred<head>::value && alli<pred, tail...>::value;
};

cout << alli<ori<isZero,isOne>::closure,1,1,0,1,0>::value;
```

all: TMP C++

- The all algorithm equivalence in C++

```
template <template <int> class pred, int... list> struct alli;

template <template <int> class pred>
struct alli<pred> { static const bool value = true; };

template <template <int> class pred, int head, int... tail>
struct alli<pred, head, tail...> {
  static const bool value = pred<head>::value && alli<pred, tail...>::value;
};

cout << alli<ori<isZero,isOne>::closure,1,1,0,1,0>::value;
```

- Forward declaration, template specialization, variadic templates and parameter pack

fold right: haskell

- The foldr algorithm in Haskell

fold right: haskell

- The foldr algorithm in Haskell

```
--definition
foldr f init [] = init
foldr f init (head:tail) = f head (foldr f init tail)

--with anonymous function
print(foldr (\n acc -> n + acc) 1 [1,2,3]) --prints 7

--with auxilary function
add n acc = n + acc
print(foldr add 1 [1,2,3]) --prints 7
```

fold right: haskell

- The foldr algorithm in Haskell

```
--definition
foldr f init [] = init
foldr f init (head:tail) = f head (foldr f init tail)

--with anonymous function
print(foldr (\n acc -> n + acc) 1 [1,2,3]) --prints 7

--with auxilary function
add n acc = n + acc
print(foldr add 1 [1,2,3]) --prints 7
```

- lambda expression, anonymous function, recursion, pattern matching

fold right: TMP C++

- The foldr algorithm equivalence in C++

fold right: TMP C++

- The foldr algorithm equivalence in C++

```
template<template<int, int> class, int, int...> struct foldr;

template<template<int, int> class f, int init> struct
foldr<f, init> {static const int value = init; };

template<template<int, int> class f, int init, int head, int... tail>
struct foldr<f, init, head, tail...> {
    static const int value = f<head, foldr<f, init, tail...>::value>::value;
};

template<int a, int b> struct add {  static const int value = a + b;};
std::cout << foldr<add, 1, 1, 2, 3>::value; //prints 7
```

fold right: TMP C++

- The foldr algorithm equivalence in C++

```
template<template<int, int> class, int, int...> struct foldr;

template<template<int, int> class f, int init> struct
foldr<f, init> {static const int value = init; };

template<template<int, int> class f, int init, int head, int... tail>
struct foldr<f, init, head, tail...> {
    static const int value = f<head, foldr<f, init, tail...>::value>::value;
};

template<int a, int b> struct add {  static const int value = a + b;};
std::cout << foldr<add, 1, 1, 2, 3>::value; //prints 7
```

- Fortunately there is fold C++17 expression

sum: haskell

- Sum algorithm in Haskell

sum: haskell

- Sum algorithm in Haskell
- with pattern matching and recursion:

sum: haskell

- Sum algorithm in Haskell
- with pattern matching and recursion:

```
sum1 [] = 0
sum1 (head:tail) = head + (sum1 tail)
print(sum1 [1,2,3]) --prints 6
```

sum: haskell

- Sum algorithm in Haskell
- with pattern matching and recursion:

```
sum1 [] = 0
sum1 (head:tail) = head + (sum1 tail)
print(sum1 [1,2,3]) --prints 6
```

- with function composition and lambda:

sum: haskell

- Sum algorithm in Haskell
- with pattern matching and recursion:

```
sum1 [] = 0
sum1 (head:tail) = head + (sum1 tail)
print(sum1 [1,2,3]) --prints 6
```

- with function composition and lambda:

```
sum2 lst = foldr2 (\n acc -> n + acc) 0 lst
print(sum2 [1,2,3]) --prints 6
```

sum: TMP C++

- The sum algorithm equivalence in C++

sum: TMP C++

- The sum algorithm equivalence in C++
- TMP C++ directly:

sum: TMP C++

- The sum algorithm equivalence in C++
- TMP C++ directly:

```
template <int...> struct sum;
template <> struct sum<> {
  static const int value = 0;
};
template <int head, int... tail> struct sum<head, tail...> {
  static const int value = head + sum<tail...>::value;
};
std::cout << sum<1,2,3>::value << std::endl; //prints 6
```

sum: TMP C++

- The sum algorithm equivalence in C++
- TMP C++ directly:

```
template <int...> struct sum;
template <> struct sum<> {
  static const int value = 0;
};
template <int head, int... tail> struct sum<head, tail...> {
  static const int value = head + sum<tail...>::value;
};
std::cout << sum<1,2,3>::value << std::endl; //prints 6
```

- TMP C++ with composition:

sum: TMP C++

- The sum algorithm equivalence in C++
- TMP C++ directly:

```
template <int...> struct sum;
template <> struct sum<> {
  static const int value = 0;
};
template <int head, int... tail> struct sum<head, tail...> {
  static const int value = head + sum<tail...>::value;
};
std::cout << sum<1,2,3>::value << std::endl; //prints 6
```

- TMP C++ with composition:

```
template <int... ints> struct sum2 {
  static const int value = foldr<add,0,ints...>::value;
};
std::cout << sum2<1,2,3>::value << std::endl;//prints 6
```

sum: C++17

- By C++17 fold expression

sum: C++17

- By C++17 fold expression

```
template <int... ints> struct sum3 {
  static const int value = (ints + ...);
};
std::cout << sum3<1,2,3>::value << std::endl;//prints 6
```

count: haskell

- Count algorithm in Haskell

count: haskell

- Count algorithm in Haskell
- with recursion:

count: haskell

- Count algorithm in Haskell
- with recursion:

```
count1 [] = 0
count1 (head:tail) = 1 + count1 tail
```

count: haskell

- Count algorithm in Haskell
- with recursion:

```
count1 [] = 0
count1 (head:tail) = 1 + count1 tail
```

- with list comprehension:

count: haskell

- Count algorithm in Haskell
- with recursion:

```
count1 [] = 0
count1 (head:tail) = 1 + count1 tail
```

- with list comprehension:

```
count2 lst = sum2 [1 | x <- lst]
```

count: haskell

- Count algorithm in Haskell
- with recursion:

```
count1 [] = 0
count1 (head:tail) = 1 + count1 tail
```

- with list comprehension:

```
count2 lst = sum2 [1 | x <- lst]
```

  - Could we do "list comprehension" in TMP C++?

count: haskell

- Count algorithm in Haskell
- with recursion:

```
count1 [] = 0
count1 (head:tail) = 1 + count1 tail
```

- with list comprehension:

```
count2 lst = sum2 [1 | x <- lst]
```

  - Could we do "list comprehension" in TMP C++?
  - Yes, of course, but ...

count: TMP C++

- Count algorithm in C++

count: TMP C++

- Count algorithm in C++
- with recursion:

count: TMP C++

- Count algorithm in C++
- with recursion:

```
template <class... list> struct count;
template <> struct count<> {
  static const int value = 0;
};
template <class head, class... tail> struct count<head, tail...> {
  static const int value = 1 + count<tail...>::value;
};
cout << count1<Int<1>, Int<4>, Char<'a'>>::value; //prints 3
```

count: List Comprehension in TMP C++

- Count algorithm in C++ with list comprehension:

count: List Comprehension in TMP C++

- Count algorithm in C++ with list comprehension:

```
template <class> struct one { static const int value = 1; };
template<class... lst> struct count {
  static const int value = sum<one<lst>::value...>::value;
};
cout << count<Int<1>, Int<4>, Char<'a'>>::value
```

count: List Comprehension in TMP C++

- Count algorithm in C++ with list comprehension:

```
template <class> struct one { static const int value = 1; };
template<class... lst> struct count {
  static const int value = sum<one<lst>::value...>::value;
};
cout << count<Int<1>, Int<4>, Char<'a'>>::value
```

- How many expansions are there?

count: List Comprehension in TMP C++

- Count algorithm in C++ with list comprehension:

```
template <class> struct one { static const int value = 1; };
template<class... lst> struct count {
  static const int value = sum<one<lst>::value...>::value;
};
cout << count<Int<1>, Int<4>, Char<'a'>>::value
```

- How many expansions are there?

```
sum<one<Int<1>>::value, one<Int<4>>::value,  one<Char<'a'>>>.
```

count: List Comprehension in TMP C++

- Count algorithm in C++ with list comprehension:

```
template <class> struct one { static const int value = 1; };
template<class... lst> struct count {
  static const int value = sum<one<lst>::value...>::value;
};
cout << count<Int<1>, Int<4>, Char<'a'>>::value
```

- How many expansions are there?

```
sum<one<Int<1>>::value, one<Int<4>>::value,  one<Char<'a'>>>.
```

  - Just one, it is a one-to-one paradigm match

count: List Comprehension in TMP C++

- Count algorithm in C++ with list comprehension:

```
template <class> struct one { static const int value = 1; };
template<class... lst> struct count {
  static const int value = sum<one<lst>::value...>::value;
};
cout << count<Int<1>, Int<4>, Char<'a'>>::value
```

- How many expansions are there?

```
sum<one<Int<1>>::value, one<Int<4>>::value,  one<Char<'a'>>>.
```

  - Just one, it is a one-to-one paradigm match
- There is sizeof... operator, but...

count: List Comprehension in TMP C++

- Count algorithm in C++ with list comprehension:

```
template <class> struct one { static const int value = 1; };
template<class... lst> struct count {
  static const int value = sum<one<lst>::value...>::value;
};
cout << count<Int<1>, Int<4>, Char<'a'>>::value
```

- How many expansions are there?

```
sum<one<Int<1>>::value, one<Int<4>>::value,  one<Char<'a'>>>.
```

  - Just one, it is a one-to-one paradigm match
- There is sizeof... operator, but...
  - it is Just sizeof, and...

count: List Comprehension in TMP C++

- Count algorithm in C++ with list comprehension:

```
template <class> struct one { static const int value = 1; };
template<class... lst> struct count {
  static const int value = sum<one<lst>::value...>::value;
};
cout << count<Int<1>, Int<4>, Char<'a'>>::value
```

- How many expansions are there?

```
sum<one<Int<1>>::value, one<Int<4>>::value,  one<Char<'a'>>>.
```

  - Just one, it is a one-to-one paradigm match
- There is sizeof... operator, but...
  - it is Just sizeof, and...
  - The TMP list comprehension is applicable to any transforming structure

map: haskell

- map and map with continuation

map: haskell

- map and map with continuation

```
--continuation
sum2 [] = 0
sum2 (head:tail) = head + (sum2 tail)

--maps
map f lst = [f x | x <- lst]
mapcont cont f lst = cont [f x | x <- lst]

--transformation
doubleme x = x + x

--application
print(map doubleme [1,2,3]) --prints 2,4,6
print(mapcont sum doubleme [1,2,3]) --prints 12
```

map with continuation: TMP C++

- The map algorithm equivalence in C++

map with continuation: TMP C++

- The map algorithm equivalence in C++

```
//continuation
template <class... Ts> struct Sum;
template <> struct Sum<> {static const int value = 0;};
template <class head, class... tail> struct Sum<head, tail...> {
  static const int value = head::value + Sum<tail...>::value;
};
//transformation
template <class T> struct Twice {
  static const int value = T::value + T::value;
};
//map with continuation
template <template <class...> class Cont,
template <class> class F, class... Ts> struct Map {
  static const int value = Cont<F<Ts>...>::value;
};
//application
cout <<  Map<Sum, Twice, Int<1>, Int<2>, Int<3>>::value; //prints 12
```

C++ Functional Metaprogramming Libraries

- Boost.MPL
  - Template metaprogramming framework of compile-time algorithms, sequences and metafunctions.

C++ Functional Metaprogramming Libraries

- Boost.MPL
  - Template metaprogramming framework of compile-time algorithms, sequences and metafunctions.
- Boost.Fusion
  - Heterogeneous containers with algorithms

C++ Functional Metaprogramming Libraries

- Boost.MPL
  - Template metaprogramming framework of compile-time algorithms, sequences and metafunctions.
- Boost.Fusion
  - Heterogeneous containers with algorithms
- Boost.Hana
  - Header-only library for C++ metaprogramming suited for computations on both types and values

C++ Functional Metaprogramming Libraries

- Boost.MPL
  - Template metaprogramming framework of compile-time algorithms, sequences and metafunctions.
- Boost.Fusion
  - Heterogeneous containers with algorithms
- Boost.Hana
  - Header-only library for C++ metaprogramming suited for computations on both types and values
- Range v3
  - Experimental range library for C++11/14/17 with range comprehension

Hana: Monad:Flatten, MonadPlus:Filter

- Monad::Flatten

Hana: Monad:Flatten, MonadPlus:Filter

- Monad::Flatten

```
//flatten - collapse two levels of monadic structure into a single level
static_assert(
hana::flatten(hana::make_tuple(hana::make_tuple(1, 2, 3),
    hana::make_tuple(4, 5),
    hana::make_tuple(6, 7, 8, 9)))
    ==
    hana::make_tuple(1, 2, 3, 4, 5, 6, 7, 8, 9), "");
```

Hana: Monad:Flatten, MonadPlus:Filter

- Monad::Flatten

```
//flatten - collapse two levels of monadic structure into a single level
static_assert(
hana::flatten(hana::make_tuple(hana::make_tuple(1, 2, 3),
    hana::make_tuple(4, 5),
    hana::make_tuple(6, 7, 8, 9)))
    ==
    hana::make_tuple(1, 2, 3, 4, 5, 6, 7, 8, 9), "");
```

- Monad::Filter

Hana: Monad:Flatten, MonadPlus:Filter

- Monad::Flatten

```
//flatten - collapse two levels of monadic structure into a single level
static_assert(
hana::flatten(hana::make_tuple(hana::make_tuple(1, 2, 3),
    hana::make_tuple(4, 5),
    hana::make_tuple(6, 7, 8, 9)))
    ==
    hana::make_tuple(1, 2, 3, 4, 5, 6, 7, 8, 9), "");
```

- Monad::Filter

```
//filter - filter a monadic structure using a custom predicate
static_assert(
hana::filter(hana::make_tuple(1, 2.0, 3, 4.0), is_integral)
==
hana::make_tuple(1, 3), "");
```

Hana: Sequence: Zip,Partition

- Sequence::Zip

Hana: Sequence: Zip,Partition

- Sequence::Zip

```
//zip_shortest_with - zip one sequence or more with a given function
static_assert(
    hana::zip_shortest_with(hana::mult,
    hana::make_tuple(1, 2, 3, 4), hana::make_tuple(5, 6, 7, 8, "ignored"))
    ==  hana::make_tuple(5, 12, 21, 32), "");
```

Hana: Sequence: Zip,Partition

- Sequence::Zip

```
//zip_shortest_with - zip one sequence or more with a given function
static_assert(
    hana::zip_shortest_with(hana::mult,
    hana::make_tuple(1, 2, 3, 4), hana::make_tuple(5, 6, 7, 8, "ignored"))
    ==  hana::make_tuple(5, 12, 21, 32), "");
```

- Sequence::Partition

Hana: Sequence: Zip,Partition

- Sequence::Zip

```
//zip_shortest_with - zip one sequence or more with a given function
static_assert(
    hana::zip_shortest_with(hana::mult,
    hana::make_tuple(1, 2, 3, 4), hana::make_tuple(5, 6, 7, 8, "ignored"))
    ==  hana::make_tuple(5, 12, 21, 32), "");
```

- Sequence::Partition

```
//partition - Partition a sequence based on a predicate
BOOST_HANA_CONSTANT_CHECK(
    hana::partition(hana::tuple_c<int, 1, 2, 3, 4, 5, 6, 7>, [](auto x) {
        return x % hana::int_c<2> != hana::int_c<0>;
    })  ==
    hana::make_pair(
        hana::tuple_c<int, 1, 3, 5, 7>,
        hana::tuple_c<int, 2, 4, 6>
    ));
```

Hana: Functional: partial, curry, compose

- Functional::partial

Hana: Functional: partial, curry, compose

- Functional::partial

```
//partial - Partially apply a function to some arguments
constexpr auto increment = hana::partial(hana::plus, 1);
static_assert(increment(2) == 3, "");
```

Hana: Functional: partial, curry, compose

- Functional::partial

```
//partial - Partially apply a function to some arguments
constexpr auto increment = hana::partial(hana::plus, 1);
static_assert(increment(2) == 3, "");
```

- Functional::curry

Hana: Functional: partial, curry, compose

- Functional::partial

```
//partial - Partially apply a function to some arguments
constexpr auto increment = hana::partial(hana::plus, 1);
static_assert(increment(2) == 3, "");
```

- Functional::curry

```
//curry - curry a function up to the given number of arguments
hana::curry<3>(add)(1)(2)(3) == 1 + 2 + 3);
hana::curry<3>(add)(1, 2, 3) == hana::curry<3>(add)(1)(2)(3));
```

Hana: Functional: partial, curry, compose

- Functional::partial

```
//partial - Partially apply a function to some arguments
constexpr auto increment = hana::partial(hana::plus, 1);
static_assert(increment(2) == 3, "");
```

- Functional::curry

```
//curry - curry a function up to the given number of arguments
hana::curry<3>(add)(1)(2)(3) == 1 + 2 + 3);
hana::curry<3>(add)(1, 2, 3) == hana::curry<3>(add)(1)(2)(3));
```

- Functional::compose

Hana: Functional: partial, curry, compose

- Functional::partial

```
//partial - Partially apply a function to some arguments
constexpr auto increment = hana::partial(hana::plus, 1);
static_assert(increment(2) == 3, "");
```

- Functional::curry

```
//curry - curry a function up to the given number of arguments
hana::curry<3>(add)(1)(2)(3) == 1 + 2 + 3);
hana::curry<3>(add)(1, 2, 3) == hana::curry<3>(add)(1)(2)(3));
```

- Functional::compose

```
//compose - return the composition of two functions or more
hana::compose(to_char, increment)(3) == '4');
```

Metaprogramming Summary: Key Points

- Some support, but not easy:

Metaprogramming Summary: Key Points

- Some support, but not easy:
  - not tersed: one liner became ten liners

Metaprogramming Summary: Key Points

- Some support, but not easy:
  - not tersed: one liner became ten liners
  - structures acts as functions

Metaprogramming Summary: Key Points

- Some support, but not easy:
  - not tersed: one liner became ten liners
  - structures acts as functions
  - template specializations acts as partial matching and recursions

Metaprogramming Summary: Key Points

- Some support, but not easy:
  - not tersed: one liner became ten liners
  - structures acts as functions
  - template specializations acts as partial matching and recursions
  - operates on template parameters, mostly on types

Metaprogramming Summary: Key Points

- Some support, but not easy:
  - not tersed: one liner became ten liners
  - structures acts as functions
  - template specializations acts as partial matching and recursions
  - operates on template parameters, mostly on types
  - only integers are supported

Metaprogramming Summary: Key Points

- Some support, but not easy:
  - not tersed: one liner became ten liners
  - structures acts as functions
  - template specializations acts as partial matching and recursions
  - operates on template parameters, mostly on types
  - only integers are supported
  - no lambdas and nested functions

Metaprogramming Summary: Key Points

- Some support, but not easy:
  - not tersed: one liner became ten liners
  - structures acts as functions
  - template specializations acts as partial matching and recursions
  - operates on template parameters, mostly on types
  - only integers are supported
  - no lambdas and nested functions
  - compile time only

Metaprogramming Summary: Key Points

- Some support, but not easy:
  - not tersed: one liner became ten liners
  - structures acts as functions
  - template specializations acts as partial matching and recursions
  - operates on template parameters, mostly on types
  - only integers are supported
  - no lambdas and nested functions
  - compile time only
  - lacking of built in combinators

Metaprogramming Summary: Key Points

- Some support, but not easy:
  - not tersed: one liner became ten liners
  - structures acts as functions
  - template specializations acts as partial matching and recursions
  - operates on template parameters, mostly on types
  - only integers are supported
  - no lambdas and nested functions
  - compile time only
  - lacking of built in combinators
- However:

Metaprogramming Summary: Key Points

- Some support, but not easy:
  - not tersed: one liner became ten liners
  - structures acts as functions
  - template specializations acts as partial matching and recursions
  - operates on template parameters, mostly on types
  - only integers are supported
  - no lambdas and nested functions
  - compile time only
  - lacking of built in combinators
- However:
  - There is **Boost.Hana** that greatly simplify the functional metaprogramming

Future Work, Conclusion, Recommendation

Future Work, Conclusion, Recommendation

Future Work: Design Consideration

- What is the influence of functional paradigm shift **on architecture**?

Future Work: Design Consideration

- What is the influence of functional paradigm shift **on architecture**?
  - Where is the **biggest impact**: in the large, medium and the small level?

Future Work: Design Consideration

- What is the influence of functional paradigm shift **on architecture**?
  - Where is the **biggest impact**: in the large, medium and the small level?
  - Does OOD and FP **are orthogonal**?

Future Work: Design Consideration

- What is the influence of functional paradigm shift **on architecture**?
  - Where is the **biggest impact**: in the large, medium and the small level?
  - Does OOD and FP **are orthogonal**?
    - Is one concept or paradigm does not either imply or exclude the other?

Future Work: Design Consideration

- What is the influence of functional paradigm shift **on architecture**?
  - Where is the **biggest impact**: in the large, medium and the small level?
  - Does OOD and FP **are orthogonal**?
    - Is one concept or paradigm does not either imply or exclude the other?
    - Could we use both to make better programs (cleaner, more beauty)?

Future Work: Design Consideration

- What is the influence of functional paradigm shift **on architecture**?
  - Where is the **biggest impact**: in the large, medium and the small level?
  - Does OOD and FP **are orthogonal**?
    - Is one concept or paradigm does not either imply or exclude the other?
    - Could we use both to make better programs (cleaner, more beauty)?
  - Does architecture smells (**rigidity, fragility, immobility**) and its remedies, the **SOLID principles** are affected?

Future Work: Design Consideration

- What is the influence of functional paradigm shift **on architecture**?
  - Where is the **biggest impact**: in the large, medium and the small level?
  - Does OOD and FP **are orthogonal**?
    - Is one concept or paradigm does not either imply or exclude the other?
    - Could we use both to make better programs (cleaner, more beauty)?
  - Does architecture smells (**rigidity, fragility, immobility**) and its remedies, the **SOLID principles** are affected?
  - What about **design patterns**?

Future Work: Design Consideration

- What is the influence of functional paradigm shift **on architecture**?
  - Where is the **biggest impact**: in the large, medium and the small level?
  - Does OOD and FP **are orthogonal**?
    - Is one concept or paradigm does not either imply or exclude the other?
    - Could we use both to make better programs (cleaner, more beauty)?
  - Does architecture smells (**rigidity, fragility, immobility**) and its remedies, the **SOLID principles** are affected?
  - What about **design patterns**?
    - Maybe some design patterns are **NOT needed** any more or could be **simplified**?

Future Work: Design Consideration

- What is the influence of functional paradigm shift **on architecture**?
  - Where is the **biggest impact**: in the large, medium and the small level?
  - Does OOD and FP **are orthogonal**?
    - Is one concept or paradigm does not either imply or exclude the other?
    - Could we use both to make better programs (cleaner, more beauty)?
  - Does architecture smells (**rigidity, fragility, immobility**) and its remedies, the **SOLID principles** are affected?
  - What about **design patterns**?
    - Maybe some design patterns are **NOT needed** any more or could be **simplified**?
  - Does language **composability capabilities** are increased?

Conclusion

- Is a C++ a functional language?

Conclusion

- Is a C++ a functional language?
  - No. It is an **imperative language** with some functional concepts.

Conclusion

- Is a C++ a functional language?
  - No. It is an **imperative language** with some functional concepts.
  - Yes. The **metaprogramming** is the compile-time **pure functional** language.

Conclusion

- Is a C++ a functional language?
  - No. It is an **imperative language** with some functional concepts.
  - Yes. The **metaprogramming** is the compile-time **pure functional** language.
- Does a paradigm shift in C++ in a FP direction is a good thing?

Conclusion

- Is a C++ a functional language?
  - No. It is an **imperative language** with some functional concepts.
  - Yes. The **metaprogramming** is the compile-time **pure functional** language.
- Does a paradigm shift in C++ in a FP direction is a good thing?
  - Yes, open more opportunities for writing better software, but... its harder from developer pov ;)

Recommendation

- Use **whatever concepts** satisfy your need

Recommendation

- Use **whatever concepts** satisfy your need
- Just **have a fun with** programming

Recommendation

- Use **whatever concepts** satisfy your need
- Just **have a fun with** programming
- **Learn and practice** functional programming by doing haskell

Recommendation

- Use **whatever concepts** satisfy your need
- Just **have a fun with** programming
- **Learn and practice** functional programming by doing haskell

- Sorry, I couldn't resists ...

Recommendation

- Use **whatever concepts** satisfy your need
- Just **have a fun with** programming
- **Learn and practice** functional programming by doing haskell

- Sorry, I couldn't resists ...
- C++ is more functional then JavaScript

Recommendation

- Use **whatever concepts** satisfy your need
- Just **have a fun with** programming
- **Learn and practice** functional programming by doing haskell

- Sorry, I couldn't resists ...
- C++ is more functional then JavaScript
- Even there is a book "Functional JavaScript"

Recommendation

- Use **whatever concepts** satisfy your need
- Just **have a fun with** programming
- **Learn and practice** functional programming by doing haskell

- Sorry, I couldn't resists ...
- C++ is more functional then JavaScript
- Even there is a book "Functional JavaScript"
- I'm (We're) **eagerly** awaiting for **"Functional C++"**

Open Questions

- How does **the creators** of C++ understand the functional paradigm?

Open Questions

- How does **the creators** of C++ understand the functional paradigm?
- Where the functional paradigm shift **will take** a C++ language?

Open Questions

- How does **the creators** of C++ understand the functional paradigm?
- Where the functional paradigm shift **will take** a C++ language?
- Is there a **clear vision** behind the evolution of C++?

The End

Thank You Very Much

Bibliography

- Why Functional programming Matters. John Hughes. 1990.
- Bartosz Milewski's Programming Cafe, bartoszmilewski.com
- The C++ Programming Language 4th Edition, B. Stroustrup, 2013
- C++ FAQ, isocpp.org/faq
- A History of Haskell: Being Lazy With Class. Hudak, 2007
- Learn You a Haskell for Great Good, Lipovaca, 2011
- Boost.Hana User Manual
- And many, many others ...