
C++SIBERIA

The Present and The Future of Functional Programming in C++

Alexander Granin
graninas@gmail.com

C++ Siberia 2019

Eric Niebler, C++ Siberia 2015

Ranges for the Standard Library

Eric Niebler, C++ Siberia 2015
Ranges for the Standard Library

Bartosz Milewski, C++ User Group 2014
Re-discovering monads in C++

Eric Niebler, C++ Siberia 2015
Ranges for the Standard Library

Bartosz Milewski, C++ User Group 2014
Re-discovering monads in C++

Ivan Čukić, C++ Siberia 2017
Atom heart monad (FRP in C++)

My talks about Functional Programming in C++

C++ User Group 2014

Functional and Declarative Design in C++

My talks about Functional Programming in C++

C++ User Group 2014
Functional and Declarative Design in C++

C++ Siberia 2015
Functional Microscope: Lenses in C++

My talks about Functional Programming in C++

C++ User Group 2014
Functional and Declarative Design in C++

C++ Siberia 2015
Functional Microscope: Lenses in C++

C++ Russia 2016
Functional "Life": Parallel Cellular Automata and Comonads in C++

My talks about Functional Programming in C++

C++ User Group 2014
Functional and Declarative Design in C++

C++ Siberia 2015
Functional Microscope: Lenses in C++

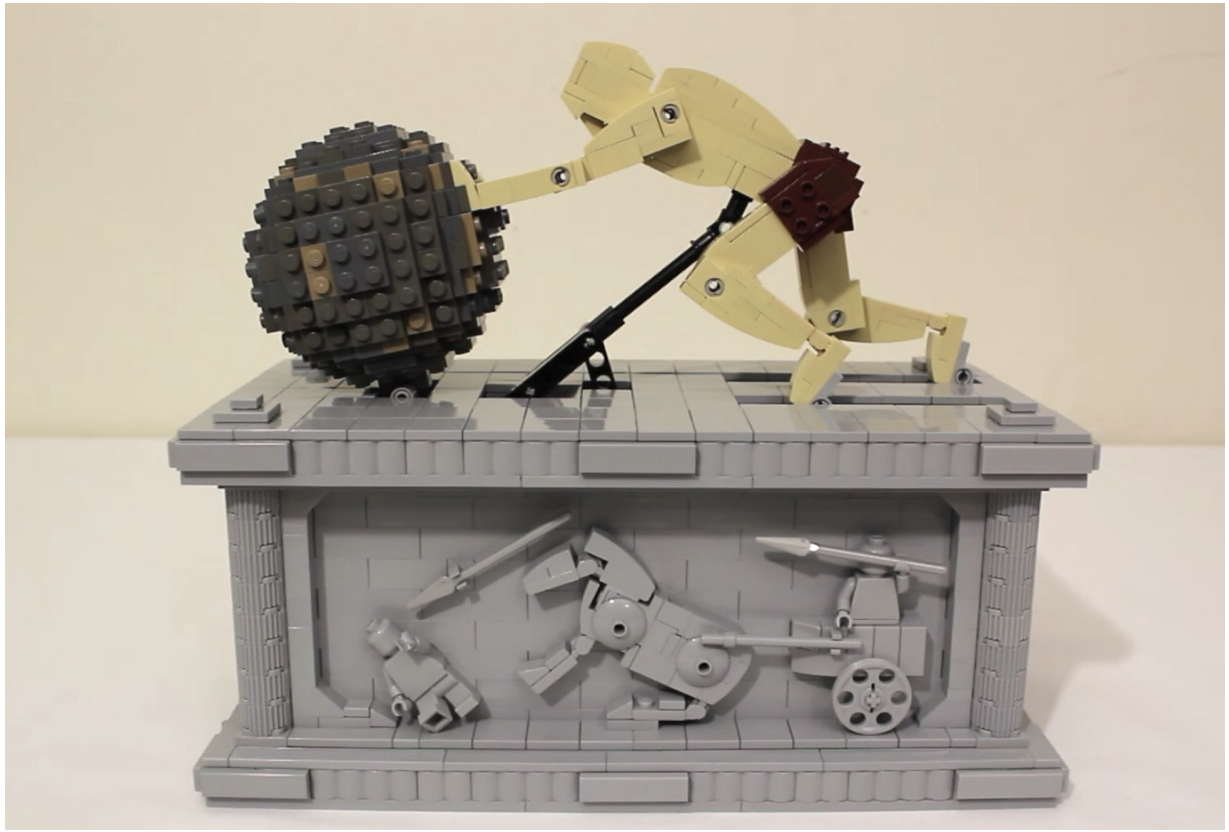
C++ Russia 2016
Functional "Life": Parallel Cellular Automata and Comonads in C++

C++ Russia 2018
Functional Approach to Software Transactional Memory in C++

```
struct Presentation
{
    Philosophy of Functional Programming in C++
    Elements of Functional Programming in C++
    Functional Design, Patterns and Approaches In C++
    Limitations of Functional Programming in C++
};
```

Philosophy
of Functional Programming in C++

What is Programming?



instructions, jumps, loops

-- unstructured programming

`instructions, jumps, loops`

-- unstructured programming

`sequence, selection, iteration`

-- structured programming

`instructions, jumps, loops`

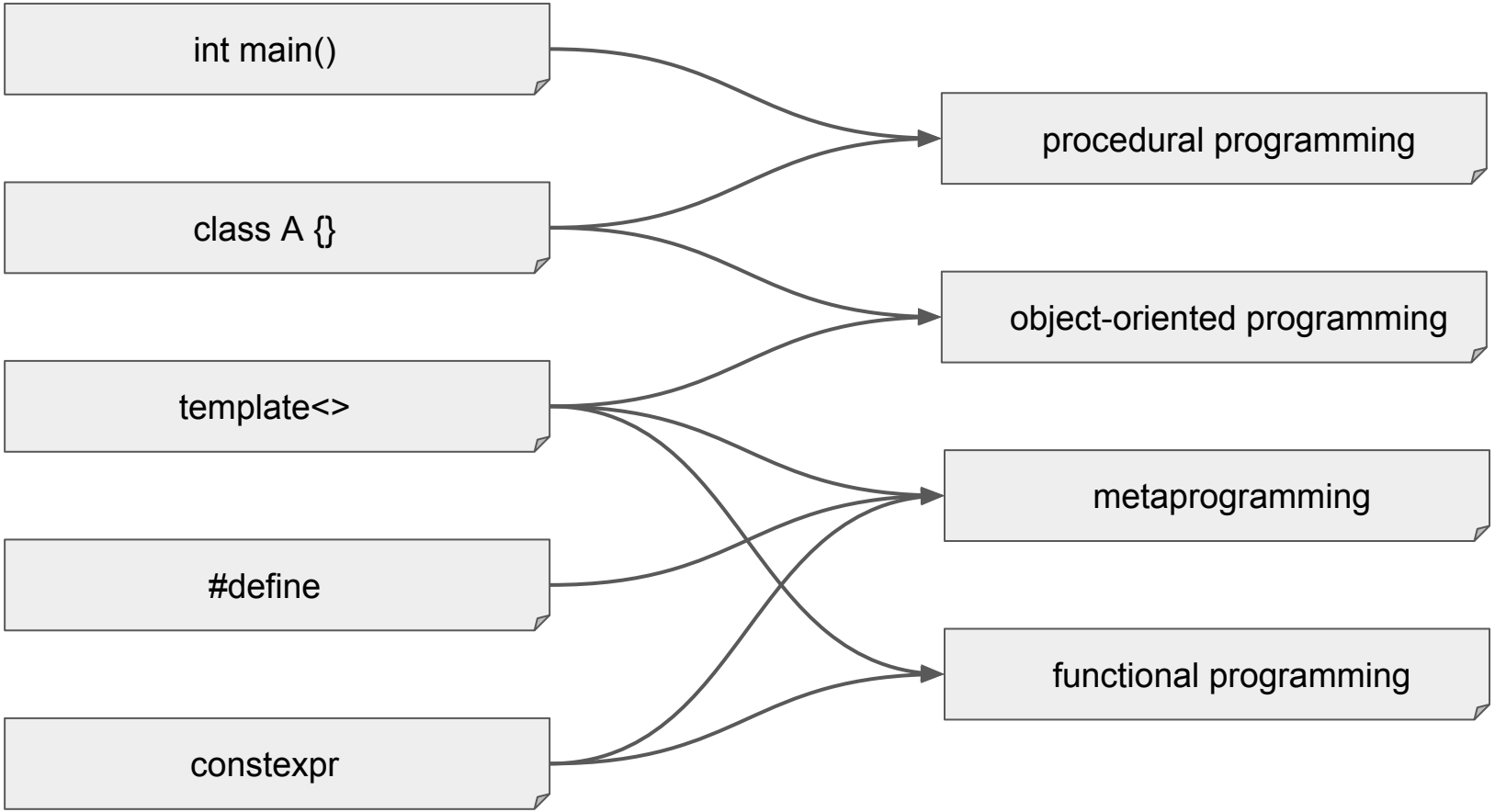
-- unstructured programming

`sequence, selection, iteration`

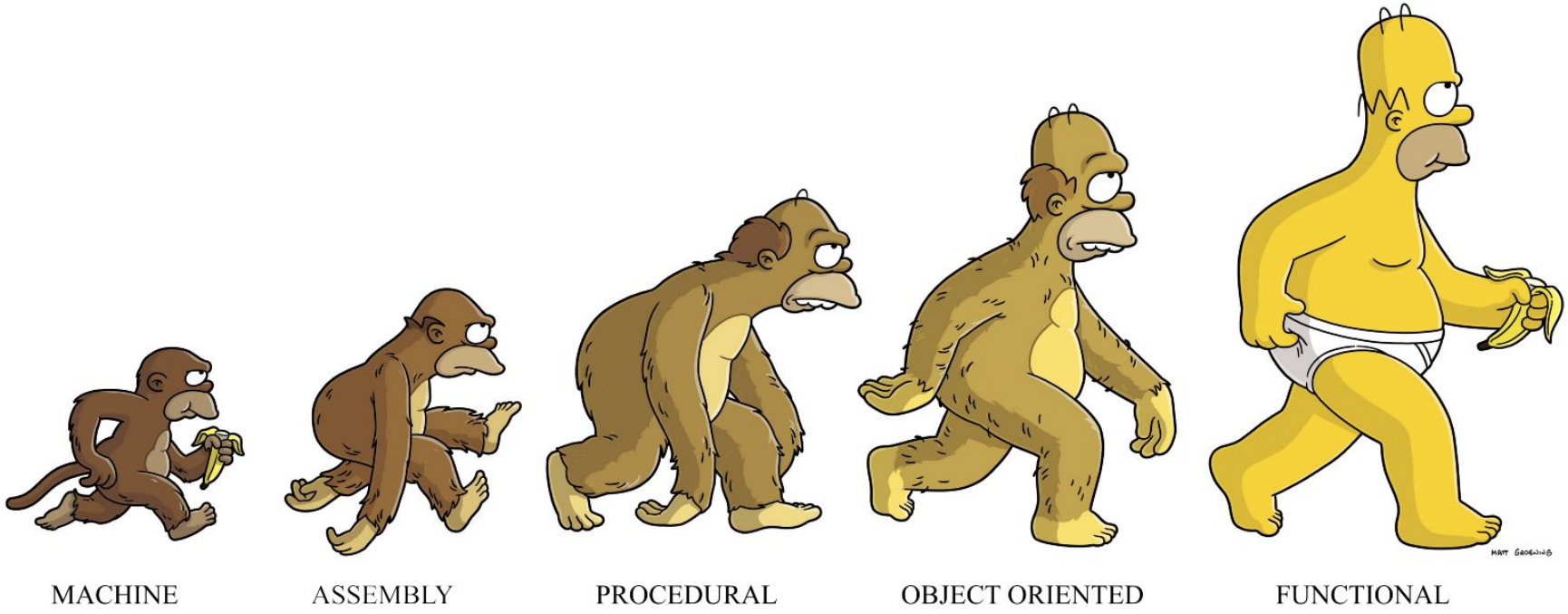
-- structured programming

`declaration, application, recursion`

-- functional programming


int main()
class A {}
template<>
#define
constexpr
procedural programming
object-oriented programming
metaprogramming
functional programming

C++ Developer Evolution



C++ Developer Evolution

Abstract C++ developer: **C++** -> **C++11/14** -> **C++17/20**

C++ Developer Evolution

Abstract C++ developer: **C++** -> **C++11/14** -> **C++17/20**

Real C++ developer: **C++** -> *Pain!* -> **C++11/14** -> *My brain!* -> **C++17/20**

C++ Developer Evolution

Abstract C++ developer: **C++** -> **C++11/14** -> **C++17/20**

Real C++ developer: **C++** -> *Pain!* -> **C++11/14** -> *My brain!* -> **C++17/20**

Beginner C++ developer: **C++** -> *WTF* -> **Go**

C++ Developer Evolution

Abstract C++ developer: **C++** -> **C++11/14** -> **C++17/20**

Real C++ developer: **C++** -> *Pain!* -> **C++11/14** -> *My brain!* -> **C++17/20**

Beginner C++ developer: **C++** -> *WTF* -> **Go**

Modern C++ developer: **C++** -> **C++11/14** -> *Why?!!* -> **Haskell** -> **C++17/20**

C++ Developer Evolution

Abstract C++ developer: **C++** -> **C++11/14** -> **C++17/20**

Real C++ developer: **C++** -> *Pain!* -> **C++11/14** -> *My brain!* -> **C++17/20**

Beginner C++ developer: **C++** -> *WTF* -> **Go**

Modern C++ developer: **C++** -> **C++11/14** -> *Why?!!* -> **Haskell** -> **C++17/20**

Tired C++ developer: **C++** -> **C++11/14** -> *Uh-Oh* -> **Rust**

C++ Developer Evolution

Abstract C++ developer: **C++** -> **C++11/14** -> **C++17/20**

Real C++ developer: **C++** -> *Pain!* -> **C++11/14** -> *My brain!* -> **C++17/20**

Beginner C++ developer: **C++** -> *WTF* -> **Go**

Modern C++ developer: **C++** -> **C++11/14** -> *Why?!!* -> **Haskell** -> **C++17/20**

Tired C++ developer: **C++** -> **C++11/14** -> *Uh-Oh* -> **Rust**

Old School C++ developer: **C++98** -> *Please, leave me alone* -> **C++98**

Functional programming in my C++?!!

Functional programming in my C++?!!

Verbosity of imperative and OO code ⟶ FP data types and constructs

algebraic data types, higher order functions, lambdas, closures
pattern-matching, currying, partial application

Functional programming in my C++?!!

Verbosity of imperative and OO code ⟶ FP data types and constructs

Usafety, bugs and lack or guarantees ⟶ Reliable FP concepts, guarantees

purity, immutability, side effects control, type level programming
recursion, range-based iteration, monoids, functors, monads...

Functional programming in my C++?!!

Verbosity of imperative and OO code ⟶ FP data types and constructs

Usafety, bugs and lack or guarantees ⟶ Reliable FP concepts, guarantees

Leaking and heavy abstractions ⟶ Declarative and functional design

domain specific languages, persistent data types
laziness, composition, combinators, first class functions

Functional programming in my C++?!!

Verbosity of imperative and OO code ⟶ FP data types and constructs

Usafety, bugs and lack or guarantees ⟶ Reliable FP concepts, guarantees

Leaking and heavy abstractions ⟶ Declarative and functional design

Complexity of multithreaded programs ⟶ Functional approaches

functional reactive programming, software transactional memory,
async / par / future monad, automatic parallelism

Elements
of Functional Programming in C++

λambdas

```
[](auto i) { return i + 1; }
```

```
\i -> i + 1
```

```
i => i + 1
```

```
lambda i: i + 1
```

```
| i | -> i + 1
```

λambdas in C++

```
<>(int x, int y) -> int {return x + y;}

[|b|]() { return |b| -> set_text("World"); }

i => i + 1


[](auto i) { return i + 1; }

[]<>(auto i) { return i + 1; }
```

λambdas in C++

```
<>(int x, int y) -> int {return x + y;}

[|b|]() { return |b| -> set_text("World"); }

i => i + 1
```

```
[](auto i) { return i + 1; }

[]<>(auto i) { return i + 1; }
```

Idea of the feature

Standard issuing

Feature adoption

Feature improvements

== ***You are standing here*** ==

Feature improvements

λambdas in C++

```
<>(int x, int y) -> int {return x + y;}

[|b|]() { return |b| -> set_text("World"); }

i => i + 1
```

```
[](auto i) { return i + 1; }

[]<>(auto i) { return i + 1; }
```

...2000... ⟶ Idea of the feature

2011 ⟶ Standard issuing

...2011... ⟶ Feature adoption

2014, 2017 ⟶ Feature improvements

2019 ⟶ == ***You are standing here*** ==

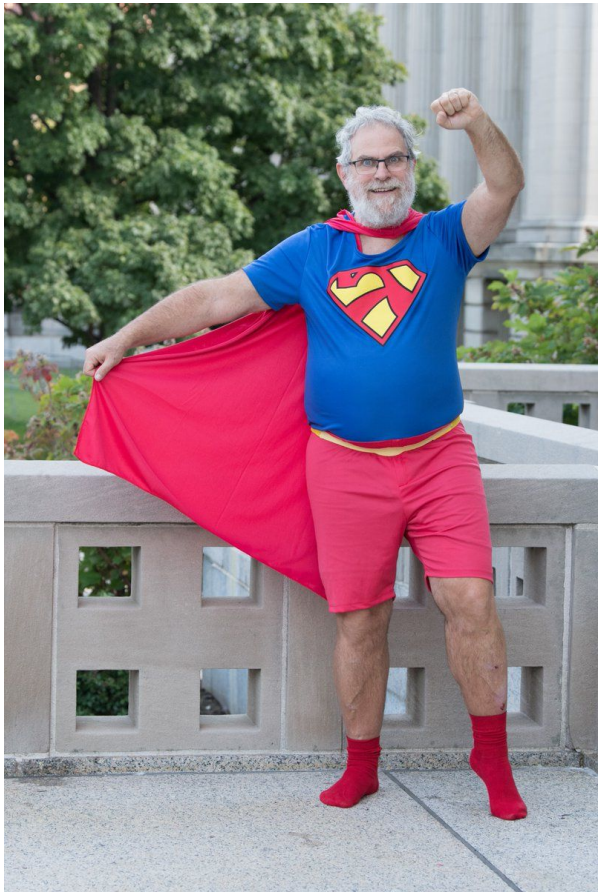
2020 ⟶ Feature improvements

λambdas in C++

```
<>(int x, int y) -> int {return x + y;}

[|b|]() { return |b| -> set_text("World"); }

i => i + 1


[](auto i) { return i + 1; }

[]<>(auto i) { return i + 1; }
```

...2000...	⟶	Idea of the feature
		Pondering and thinking
		Reference implementation
		Discussions in Committee
2006 - 2008	⟶	Drafting in Standard
2008 - 2011	⟶	Implementation in compilers
2011	⟶	Standard issuing
...2011...	⟶	Feature adoption
2014, 2017	⟶	Feature improvements
2019	⟶	== ***You are standing here*** ==
2020	⟶	Feature improvements

What is λambda?



And who is
λambdaman?

“λambda is just an object with operator() overloaded”

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

struct Summator
{
    int sum = 0;

    void operator()(int i) {
        sum += i;
    }
};

Summator summator = std::for_each(v.begin(), v.end(), Summator());

std::cout << summator.sum;    // 10
```

“λambda is an anonymous function”

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

auto sum = 0;
std::for_each(v.begin(), v.end(), [&sum](int i) { sum += i; });

std::cout << sum;    // 10
```

“λambda is an universal combinator”

```
auto I = [](auto x) {
    return x;
};
```

$$\mathbf{I}x = x$$

```
auto K = [](auto x) {
    return [=](auto y) {
        return x;
    };
};
```

$$\mathbf{K}xy = x$$

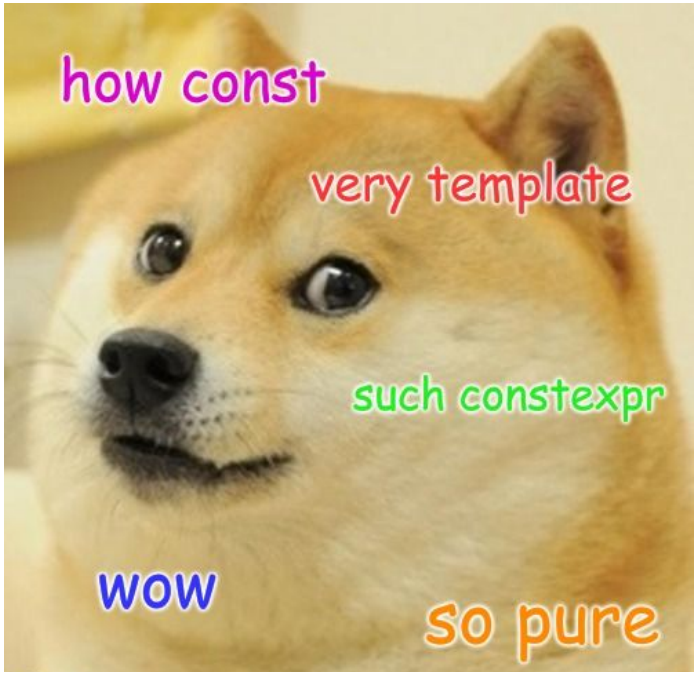
```
auto S = [](auto x) {
    return [=](auto y) {
        return [=](auto z) {
            return (x(z))(y(z));
        };
    };
};
```

$$\mathbf{S}xyz = xz(yz)$$

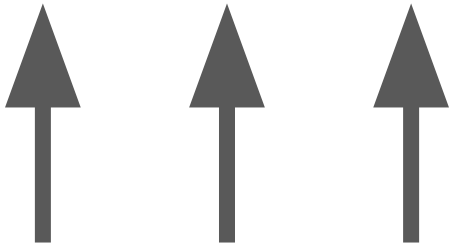
λambdas evolution

C++11	Lambda expressions	`auto mul2 = [](int i) { return i * 2; }`
C++14	Polymorphic lambdas	`auto mul2 = [](auto i) { return i * 2; }`
C++17	Constexpr lambdas	`constexpr auto mul2 = [](auto i) { return i * 2; }`
C++20	“Familiar” syntax	`auto mul2 = []<typename T>(T i) { return i * 2; }`
C++??	*Abbreviated syntax*	*`auto mul2 = i => i * 2;`*

Purity and Immutability



Pure Functional Programming in C++



Knuth’s Up-Arrow Notation

$$a \uparrow\uparrow b = \underbrace{a \uparrow (a \uparrow (\cdots \uparrow a))}_{b} = \underbrace{a^{a^{\cdot^{\cdot^{\cdot^{a}}}}}}_{b}$$

Knuth’s Up-Arrow Notation

$$a \uparrow\uparrow b = \underbrace{a \uparrow (a \uparrow (\cdots \uparrow a))}_{b} = \underbrace{a^{a^{\cdot^{\cdot^{\cdot^{a}}}}}}_{b}$$

$$a \uparrow\uparrow\uparrow b = \underbrace{a \uparrow\uparrow (a \uparrow\uparrow (\cdots \uparrow\uparrow a))}_{b}$$

Knuth's Up-Arrow Notation

$$a \uparrow\uparrow b = \underbrace{a \uparrow (a \uparrow (\cdots \uparrow a))}_{b} = \underbrace{a^{a^{\cdot^{\cdot^{\cdot^{a}}}}}}_{b}$$

$$a \uparrow\uparrow\uparrow b = \underbrace{a \uparrow\uparrow (a \uparrow\uparrow (\cdots \uparrow\uparrow a))}_{b}$$

$$a \underbrace{\uparrow\uparrow \ldots \uparrow}_{n} b = a \underbrace{\uparrow \ldots \uparrow}_{n-1} (a \underbrace{\uparrow \ldots \uparrow}_{n-1} (a \ \ldots (a \underbrace{\uparrow \ldots \uparrow}_{n-1} a)))$$

Knuth's Up-Arrow Notation

$$4 \uparrow 3 = 4^3 = 64$$

$$4 \uparrow\uparrow 3 = 4 \uparrow (4 \uparrow 4) = 4^{4^4} = 4 \uparrow (4 \uparrow 4) = 4^{256} \approx 1.3 \times 10^{154}$$

$$3 \uparrow\uparrow 2 = 3^3 = 27$$

$$3 \uparrow\uparrow 3 = 3^{3^3} = 3^{27} = 7\,625\,597\,484\,987$$

$$3 \uparrow\uparrow 4 = 3^{3^{3^3}} = 3^{7625597484987}$$

$$3 \uparrow\uparrow 5 = 3^{3^{3^{3^3}}} = 3^{3^{7625597484987}}$$

constexpr ALL the Things!

https://www.youtube.com/watch?v=PJwd4JLYJJY

Recursive constexpr pow() function

```
using UInt = uint64_t;

constexpr UInt pow(UInt a, UInt b) {
   return b == 0
            ? 1
            : a * pow(a, b - 1);
}
```

$$a^b$$

Recursive constexpr tower() function

```
using UInt = uint64_t;

constexpr UInt pow(UInt a, UInt b) {
   return b == 0
            ? 1
            : a * pow(a, b - 1);
}


constexpr UInt tower(UInt a, UInt b) {
   return b == 0
            ? throw std::invalid_argument("b == 0")
            : b == 1
                ? a
                : pow(a, tower(a, b - 1));
}
```

$$a^b$$

$$\underbrace{a^{a^{\cdot^{\cdot^{\cdot^{a}}}}}}_{b}$$

Throwing exception at compile time?

```
using UInt = uint64_t;

constexpr UInt pow(UInt a, UInt b) {
   return b == 0
            ? 1
            : a * pow(a, b - 1);
}


constexpr UInt tower(UInt a, UInt b) {
   return b == 0
            ? throw std::invalid_argument("b == 0")
            : b == 1
                ? a
                : pow(a, tower(a, b - 1));
}
```

$$a^b$$

$$\underbrace{a^{a^{\cdot^{\cdot^{\cdot^{a}}}}}}_{b}$$

Recurrent formula

$$a \underbrace{\uparrow\uparrow\ldots\uparrow}_{n} b = a \underbrace{\uparrow\ldots\uparrow}_{n-1}(a \underbrace{\uparrow\ldots\uparrow}_{n-1}(a \ \ldots(a \underbrace{\uparrow\ldots\uparrow}_{n-1} a)))$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{b}$$

Double recursion constexpr arr_impl() function

$$a \underbrace{\uparrow\uparrow\ldots\uparrow}_{n} b = a \underbrace{\uparrow\ldots\uparrow}_{n-1}(a \underbrace{\uparrow\ldots\uparrow}_{n-1}(a \ldots(a \underbrace{\uparrow\ldots\uparrow}_{n-1} a)))\Bigg\}_{b}$$

```
constexpr UInt arr_impl(bool brackets_rec, UInt cnt, UInt a, UInt n, UInt b)
{
  return brackets_rec
      ? (cnt == 0 ? ... ... ...)                          // recursion by brackets
      : (n == 1
          ? pow(a, b)
          : n == 2
              ? tower(a, b)
              : arr_impl(true, b - 1, a, n - 1, a));    // recursion by arrows
}
```

Evaluation

```
constexpr UInt arr(UInt a, UInt n, UInt b)
{
   return arr_impl(false, 0, a, n, b);
}


constexpr UInt x1 = arr(3, 2, 3);      // Compile-time evaluated

UInt x2 = arr(3, 2, 3);                // Has to be compile-time evaluated



// 3 ↑↑ 3 = 7625597484987
```

Pure Template Functional Programming in C++

Template Pow “function”

```
template <UInt A, UInt N>
struct Pow
{
   static constexpr UInt value = A * Pow<A, N - 1>::value;
};


template <UInt A>
struct Pow<A, 0>
{
   static constexpr UInt value = 1;
};
```

Template Pow “function”

```
template <UInt A, UInt N>
struct Pow
{
    static constexpr UInt value = A * Pow<A, N - 1>::value;
};
```

“Recursion”

```
template <UInt A>
struct Pow<A, 0>
{
    static constexpr UInt value = 1;
};
```

“Pattern matching”

Template Tower “function”

```
template <UInt A, UInt B>
struct Tower
{
   static constexpr UInt value = Pow<A, Tower<A, B - 1>::value>::value;
};


template <UInt A>
struct Tower<A, 1>
{
   static constexpr UInt value = A;
};
```

Template ArrImpl “function”

```
template <bool brackets_rec, UInt C, UInt A, UInt S, UInt B>
struct ArrImpl
{
   static constexpr UInt value = 1;
};


template <UInt C, UInt A, UInt S, UInt B>
struct ArrImpl<true, C, A, S, B>
{
   static constexpr UInt value
       = ArrImpl<false, 0, A,
                 ArrImpl<true, C - 1, A, S, B>::value,
                 S>::value;
};


... // More template code
```

“Evaluation”

```
template <UInt A, UInt S, UInt B>
struct Arr
{
   static constexpr UInt value = ArrImpl<false, 0, A, S, B>::value;
};


auto result = Arr<3, 2, 3>::value;          // 3 ↑↑ 3 = 7625597484987
```

Template ⇨ constexpr = 👍

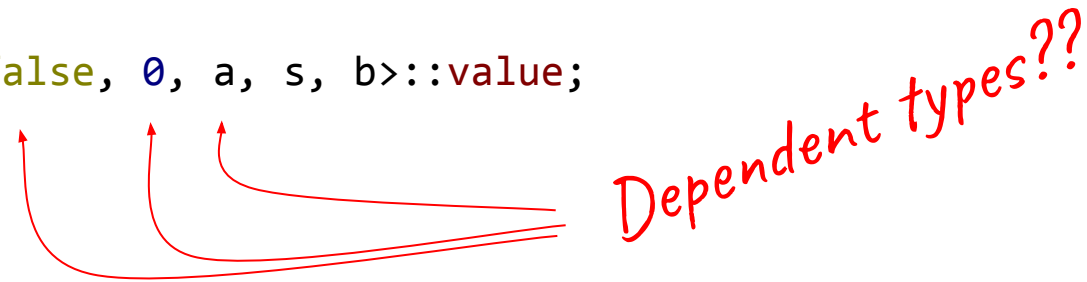
```
template <UInt A, UInt S, UInt B>
struct Arr
{
   static constexpr UInt value = arr_impl(false, 0, A, S, B);
};
```

Constexpr ⇨ template = ☠

```
template <UInt A, UInt S, UInt B>
struct Arr
{
   static constexpr UInt value = arr_impl(false, 0, A, S, B);
};

// Won’t compile:
constexpr UInt arr(UInt a, UInt s, UInt b)
{
   return ArrImpl<false, 0, a, s, b>::value;
}
```

Constexpr ⇨ template = ☠

```
template <UInt A, UInt S, UInt B>
struct Arr
{
   static constexpr UInt value = arr_impl(false, 0, A, S, B);
};

// Won't compile:
constexpr UInt arr(UInt a, UInt s, UInt b)
{
   return ArrImpl<false, 0, a, s, b>::value;
}
```



Template constexpr!

```
template <UInt A, UInt S, UInt B>
struct Arr
{
   static constexpr UInt value = arr_impl(false, 0, A, S, B);
};

template <UInt A, UInt S, UInt B>
constexpr UInt arr()
{
   return ArrImpl<false, 0, A, S, B>::value;
}
```

* * *

Algebraic Data Types

product types

a, b, c — class, struct, pair, tuple

sum types

T | U | V — optional, variant, expected

Meeting C++ 2018
Phil Nash
Option(al) is not a failure

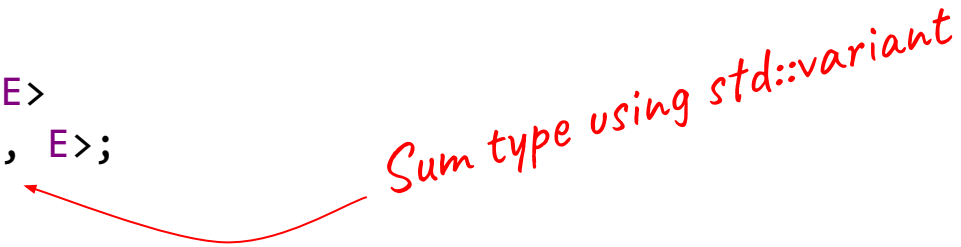
https://www.youtube.com/watch?v=iWDjoD3MdXs

expected<T, E>: Proposal P0323R2

```
expected<double, error_condition>
safe_divide(double i, double j)
{
    if (j == 0)
    {
        return make_unexpected(arithmetic_error::divide_by_zero);
    }
    else
    {
        return i / j;
    }
}
```

Custom Expected<T, E>

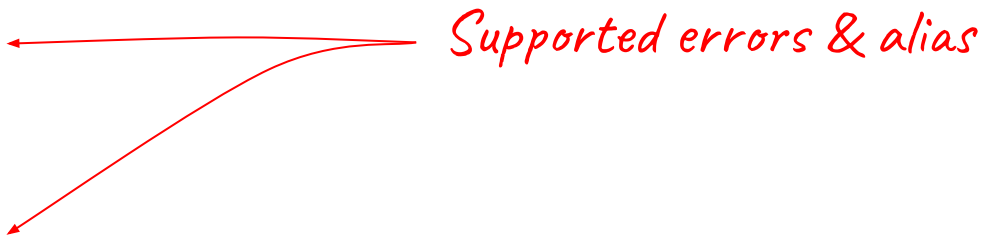
```
template <typename T, typename E>
using Expected = std::variant<T, E>;
```




```
enum class Errors {
   ZeroTowerError    = 0,
   ZeroDivisionError = 1
};

using UInt    = uint64_t;
using SafeInt = Expected<UInt, Errors>;


constexpr SafeInt make_pure (const UInt& v)   { return { v }; }
constexpr SafeInt make_error(const Errors& e) { return { e }; }
```

Custom Expected<T, E>

```
template <typename T, typename E>
using Expected = std::variant<T, E>;


enum class Errors {
    ZeroTowerError    = 0,
    ZeroDivisionError = 1
};

using UInt    = uint64_t;
using SafeInt = Expected<UInt, Errors>;


constexpr SafeInt make_pure (const UInt& v)   { return { v }; }
constexpr SafeInt make_error(const Errors& e) { return { e }; }
```



Custom Expected<T, E>

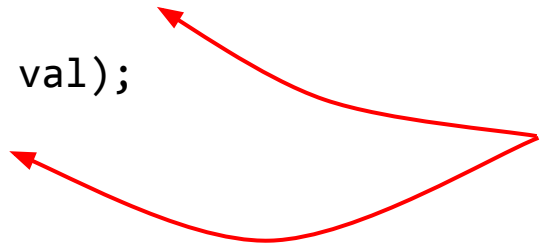
```
template <typename T, typename E>
using Expected = std::variant<T, E>;


enum class Errors {
   ZeroTowerError    = 0,
   ZeroDivisionError = 1
};

using UInt    = uint64_t;
using SafeInt = Expected<UInt, Errors>;


constexpr SafeInt make_pure (const UInt& v)   { return { v }; }
constexpr SafeInt make_error(const Errors& e) { return { e }; }
```

Extracting values of variant<>

```
std::variant<UInt, Errors> val = make_pure(10);


UInt result = std::holds_alternative<UInt>(val)
              ? std::get<UInt>(val)
              : throw std::runtime_error("Error got.");
```

Pattern Matching?

```
struct SafeIntVisitor
{
   void operator()(const UInt& i)   { cout << "Result: " << i; }
   void operator()(const Errors& e) { cout << "Error:  " << e; }
};

std::visit(SafeIntVisitor(), val);
```



Visiting std::variant

```
inspect (val)
{
  <UInt>   i: cout << "Result: " << i;
  <Errors> e: cout << "Error:  " << e;
};
```

Pattern Matching!

```
struct SafeIntVisitor
{
   void operator()(const UInt& i)   { cout << "Result: " << i; }
   void operator()(const Errors& e) { cout << "Error:  " << e; }
};

std::visit(SafeIntVisitor(), val);
```

Pattern matching
Proposal P1260R0

```
inspect (val)
{
  <UInt>   i: cout << "Result: " << i;
  <Errors> e: cout << "Error:  " << e;
};
```

Object-oriented programming is an exceptionally bad idea which could only have originated in California.

-- Edsger Dijkstra

Functional Design, Patterns and Approaches in C++

Monadic error handling with Expected<T, E>

```
template <typename T, typename E>
using Expected = std::variant<T, E>;


enum class Errors {
  ZeroTowerError    = 0,
  ZeroDivisionError = 1
};

using UInt    = uint64_t;
using SafeInt = Expected<UInt, Errors>;


constexpr SafeInt make_pure (const UInt& v)   { return { v }; }
constexpr SafeInt make_error(const Errors& e) { return { e }; }
```

Unsafe tower() function

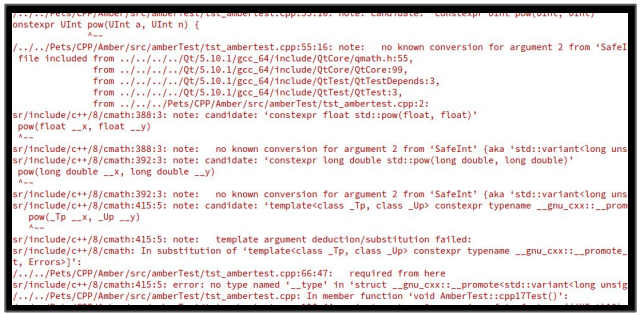
```
UInt pow(UInt a, UInt n) { ... }

UInt tower(UInt a, UInt b) {
   return b == 0
            ? throw std::invalid_argument("b == 0")
            : b == 1
                ? a
                : pow(a, tower(a, b - 1));
}
```

safe_tower() function

```
UInt pow(UInt a, UInt n) { ... }

SafeInt safe_tower(UInt a, UInt b) {
    return b == 0
             ? make_error(Errors::ZeroTowerError)
             : b == 1
                 ? a
                 : pow(a, tower(a, b - 1));
}
```

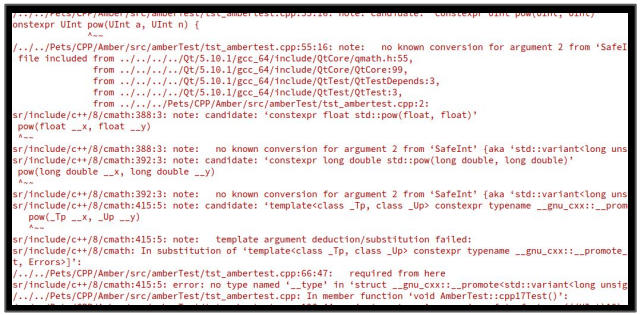
safe_tower() function

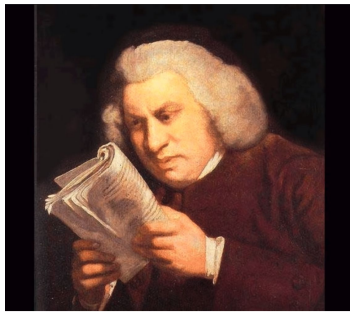
```
UInt pow(UInt a, UInt n) { ... }

SafeInt safe_tower(UInt a, UInt b) {
    return b == 0
               ? make_error(Errors::ZeroTowerError)
               : b == 1
                   ? a
                   : pow(a, tower(a, b - 1));
}
```



safe_tower() function

```
UInt pow(UInt a, UInt n) { ... }

SafeInt safe_tower(UInt a, UInt b) {
    return b == 0
               ? make_error(Errors::ZeroTowerError)
               : b == 1
                   ? a
                   : pow(a, tower(a, b - 1));
}
```





safe_tower() function

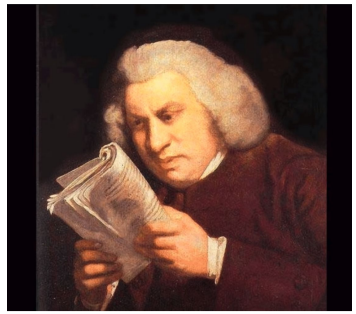
```
UInt pow(UInt a, UInt n) { ... }

SafeInt safe_tower(UInt a, UInt b) {
   return b == 0
            ? make_error(Errors::ZeroTowerError)
            : b == 1
                ? make_pure(a)
                : make_pure(pow(a, tower(a, b - 1)));
}
```

safe_tower() function

```
UInt pow(UInt a, UInt n) { ... }

SafeInt safe_tower(UInt a, UInt b) {
    return b == 0
               ? make_error(Errors::ZeroTowerError)
               : b == 1
                   ? make_pure(a)
                   : make_pure(pow(a, tower(a, b - 1)));
}
```

safe_tower() function

```
UInt pow(UInt a, UInt n) { ... }

SafeInt safe_tower(UInt a, UInt b) {
    return b == 0
               ? make_error(Errors::ZeroTowerError)
               : b == 1
                   ? make_pure(a)
                   : make_pure(pow(a, tower(a, b - 1)));
}
```

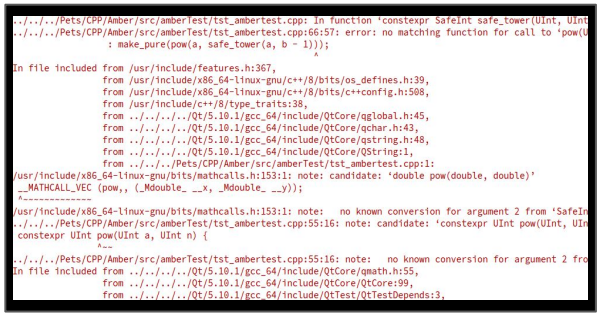


safe_tower() function

```
UInt pow(UInt a, UInt n) { ... }

SafeInt safe_tower(UInt a, UInt b) {
   return b == 0
            ? make_error(Errors::ZeroTowerError)
            : b == 1
                ? make_pure(a)
                : make_pure(pow(a, safe_tower(a, b - 1)));
}
```

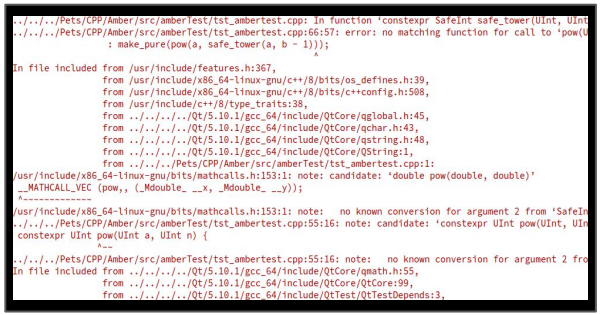
safe_tower() function

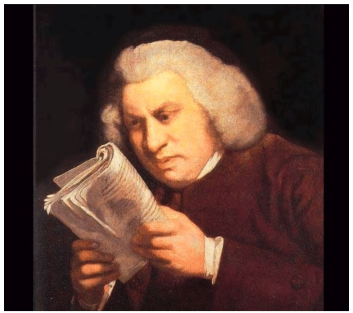
```
UInt pow(UInt a, UInt n) { ... }

SafeInt safe_tower(UInt a, UInt b) {
    return b == 0
               ? make_error(Errors::ZeroTowerError)
               : b == 1
                   ? make_pure(a)
                   : make_pure(pow(a, safe_tower(a, b - 1)));
}
```



safe_tower() function

```
UInt pow(UInt a, UInt n) { ... }

SafeInt safe_tower(UInt a, UInt b) {
    return b == 0
               ? make_error(Errors::ZeroTowerError)
               : b == 1
                   ? make_pure(a)
                   : make_pure(pow(a, safe_tower(a, b - 1)));
}
```





Monadic binding*

```
UInt pow(UInt a, UInt n) { ... }

SafeInt safe_tower(UInt a, UInt b) {
    return b == 0
             ? make_error(Errors::ZeroTowerError)
             : b == 1
                 ? make_pure(a)
                 : mbind(safe_tower(a, b - 1),
                         [=](UInt v) { return make_pure(pow(a, v)); }
                        );
}
```

* Almost

Monadic binding*

```
UInt pow(UInt a, UInt n) { ... }

SafeInt safe_tower(UInt a, UInt b) {
   return b == 0
            ? make_error(Errors::ZeroTowerError)
            : b == 1
                ? make_pure(a)
                : mbind(safe_tower(a, b - 1),
                        [=](UInt v) { return make_pure(pow(a, v)); }
                       );
}

SafeInt mbind (SafeInt, Func<SafeInt, UInt>)

mbind :: SafeInt -> (UInt -> SafeInt) -> SafeInt
```

* Almost

mbind() function

```
template <typename Lambda>
SafeInt mbind(SafeInt safe_val, Lambda lambda)
{
   return std::holds_alternative<E>(safe_val)
               ? safe_val
               : lambda(std::get<T>(safe_val));
}
```

Monadic chaining*, error path

```
auto mul10 = [](UInt i) { return make_pure(i * 10); };


SafeInt v1 = safe_tower(2, 0);
SafeInt v2 = mbind(v1, mul10);




std::visit(SafeIntVisitor(), v1);      // error : 0
std::visit(SafeIntVisitor(), v2);      // error : 0
```

* Almost

Monadic chaining*, success path

```
auto mul10 = [](UInt i) { return make_pure(i * 10); };


SafeInt v1 = safe_tower(2, 3);
SafeInt v2 = mbind(
                  mbind(v1, mul10),
                  mul10
                 );


std::visit(SafeIntVisitor(), v1);      // result: 16
std::visit(SafeIntVisitor(), v2);      // result: 1600
```

* Almost

Monadic chaining*, do-notation dream

```
auto mul10 = [](UInt i) { return make_pure(i * 10); };


SafeInt v1 = safe_tower(2, 3);
SafeInt v2 = mbind(
                  mbind(v1, mul10),
                  mul10
                 );


SafeInt v2 = do {
    UInt i1 <- safe_tower(2, 3);
    UInt i2 <- mul10(i1);
    return mul10(i2);
}
```

mbind is hidden under "<-"

* Almost

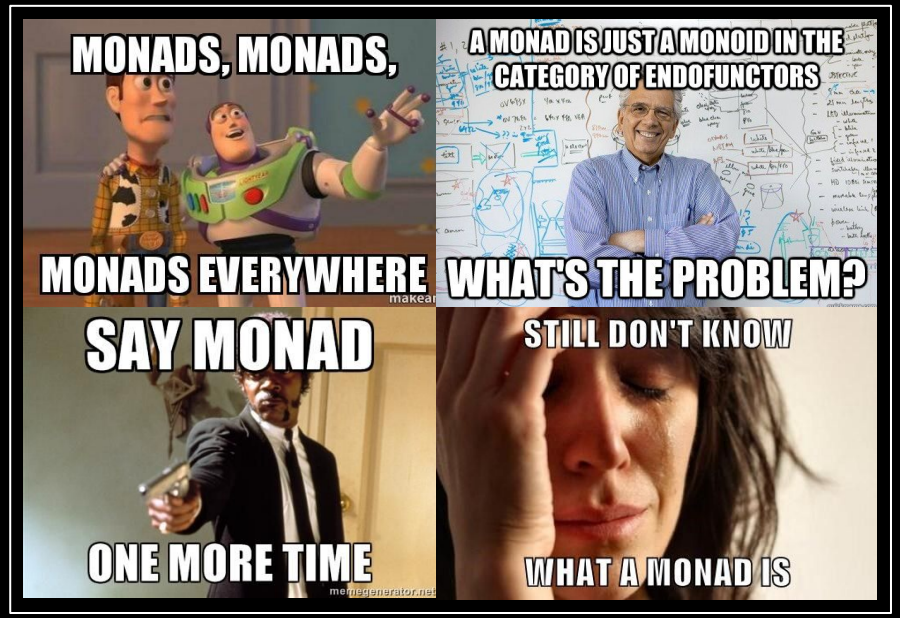
Do-notation dreams

```
expected<int, Errors> safe_calculate(int a, int b) {
    do {
        UInt v <- safe_tower(a, b);
        int r  <- safe_divide(v, b);
        return r;
    }
}
```

Haskell-like "do"

```
expected<int, Errors> safe_calculate(int a, int b) {
    UInt v = try safe_tower(a, b);
    int r  = try safe_divide(v, b);
    return r;
}
```

Proposal P0650R1
"C++ Monadic Interface"



MONADS

It’s what we like memes to be about

Monads everywhere: expected<>

```
expected<int, Errors> safe_calculate(int a, int b) {
    do {
        UInt v <- safe_tower(a, b);
        int r  <- safe_divide(v, b);
        return r;
    }
}
```

```
mbind :: expected<A, E> -> (A -> expected<B, E>) -> expected<B, E>
```

Monads everywhere: optional<>

```
optional<int> same_age_persons_count(int personId) {
   do {
       Person  p  <- get(personId);
       Persons ps <- find( [=](Person x){ return x.age == p.age; } );
       return ps.size();
   }
}
```

```
mbind :: expected<A, E> -> (A -> expected<B, E>) -> expected<B, E>
mbind :: optional<A>    -> (A -> optional<B>)    -> optional<B>
```

Monads everywhere: future<>

```
future<int> calculate(int a, int b) {
   do {
       int x <- heavy_calc(a);
       int y <- heavy_calc(b);
       return heavy_calc(x + y);
   }
}
```

```
mbind :: expected<A, E> -> (A -> expected<B, E>) -> expected<B, E>
mbind :: optional<A>    -> (A -> optional<B>)    -> optional<B>
mbind :: future<A>      -> (A -> future<B>)      -> future<B>
```

Monads everywhere: async<> (coroutines?)

```
async<Result> request_person(int personId) {
   do {
       auto person <- request_person(personId);
       auto confs  <- request_conferences(personId);
       bool saved1 <- save(person);
       bool saved2 <- save(confs);
       return {saved1 && saved2, person, confs};
   }
}
```

```
mbind :: expected<A, E> -> (A -> expected<B, E>) -> expected<B, E>
mbind :: optional<A>    -> (A -> optional<B>)    -> optional<B>
mbind :: future<A>      -> (A -> future<B>)      -> future<B>
mbind :: async<A>       -> (A -> async<B>)       -> async<B>
```

Monads everywhere: parser<>

```
parser<Person> person() {
   do {
       _          <- many(space);
       first_name <- name();
       _          <- many(space);
       last_name  <- name();
       return {first_name, last_name};
   }
}
```

```
mbind :: expected<A, E> -> (A -> expected<B, E>) -> expected<B, E>
mbind :: optional<A>    -> (A -> optional<B>)    -> optional<B>
mbind :: future<A>      -> (A -> future<B>)      -> future<B>
mbind :: async<A>       -> (A -> async<B>)       -> async<B>
mbind :: parser<A>      -> (A -> parser<B>)      -> parser<B>
```

* * *

Postmodern immutable data structures

cppcon | 2017
THE C++ CONFERENCE • BELLEVUE, WASHINGTON

```
const auto v0 = vector<int>{};
const auto v1 = v0.push_back(15);
const auto v2 = v1.push_back(16);
const auto v3 = v2.set(0, 42);

assert(v2.size() == v0.size() + 2);
assert(v3[0] - v1[0] == 27);
```

immutable data structure

one with all methods
marked const

persistent data structure

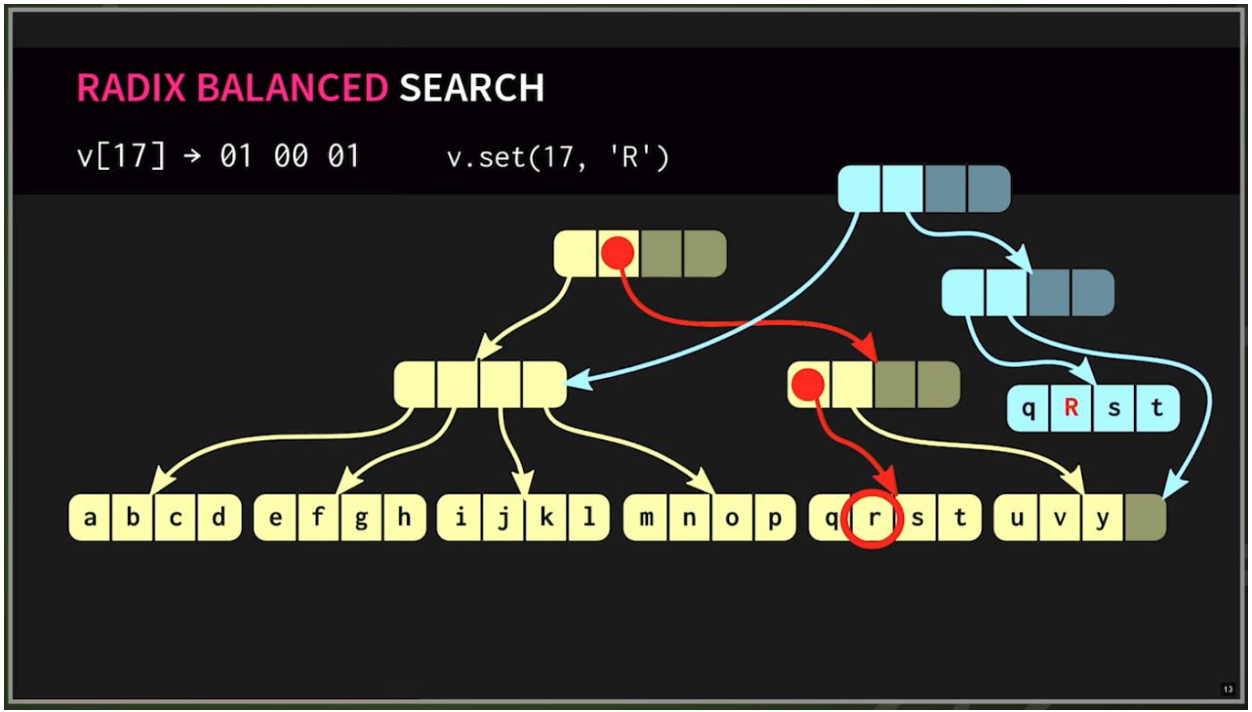
old values are preserved

6

Postmodern immutable
data structures

CppCon.org

https://www.youtube.com/watch?v=sPhpeIUfu8Q


RADIX BALANCED SEARCH
v[17] → 01 00 01
v.set(17, 'R')
q R s t
a b c d e f g h i j k l m n o p q r s t u v y

cppcon the c++ conference

Ranges for the Standard Library

www.CppCon.org

Group Dates into Months

```
auto dates_in_year(int year) {
    return view::iota(date{year,greg::Jan,1},
                      date{year+1,greg::Jan,1});
}

int main() {
    auto year = dates_in_year(2015);
    // Group into months:
    auto months = year | view::group_by([](date a, date b) {
        return a.month() == b.month();
    });

    // Print the first day of each month:
    RANGES_FOR(auto month, months)
        cout << front(month) << '\n';
}
```

https://www.youtube.com/watch?v=mFUXNMfaciE

Pythagorean Triples in Ranges

```
auto triples =
  for_each(iota(1), [](int z) {
     return for_each(iota(1, z+1), [=](int x) {
        return for_each(iota(x, z+1), [=](int y) {
           return yield_if(x*x + y*y == z*z,
              make_tuple(x, y, z));
     });
   });
 });
```

Pythagorean Triples in Ranges

```
auto triples =
  for_each(iota(1), [](int z) {
     return for_each(iota(1, z+1), [=](int x) {
        return for_each(iota(x, z+1), [=](int y) {
           return yield_if(x*x + y*y == z*z,
              make_tuple(x, y, z));
     });
   });
 });
```

Pythagorean Triples in Ranges

```
auto triples =
  for_each(iota(1), [](int z) {
     return for_each(iota(1, z+1), [=](int x) {
        return for_each(iota(x, z+1), [=](int y) {
           return yield_if(x*x + y*y == z*z,
              make_tuple(x, y, z));
     });
   });
 });


for(auto triple : triples | view::take(10)) {
 cout << '('
      << get<0>(triple) << ','
      << get<1>(triple) << ','
      << get<2>(triple) << ')' << '\n';
}
```

Output:

```
(3,4,5)
(6,8,10)
(5,12,13)
(9,12,15)
(8,15,17)
(12,16,20)
(7,24,25)
(15,20,25)
(10,24,26)
(20,21,29)
```

Pythagorean Triples in Ranges, reformatted

```
auto triples =
   for_each(iota(1),          [] (int z) { return
   for_each(iota(1, z+1),     [=](int x) { return
   for_each(iota(x, z+1),     [=](int y) { return
   yield_if(x*x + y*y == z*z, make_tuple(x, y, z));
 }); }); });
```

Looks familiar...

```
auto triples =
   for_each(iota(1),          [] (int z) { return
   for_each(iota(1, z+1),     [=](int x) { return
   for_each(iota(x, z+1),     [=](int y) { return
   yield_if(x*x + y*y == z*z, make_tuple(x, y, z));
 }); }); });
```

```
iota :: list<int>

for_each :: list<int> -> (int -> list<int>) -> list<int>
```

for_each → mbind...

```
auto triples =
   mbind(iota(1),              [] (int z) { return
   mbind(iota(1, z+1),         [=](int x) { return
   mbind(iota(x, z+1),         [=](int y) { return
   yield_if(x*x + y*y == z*z, make_tuple(x, y, z));
 }); }); });
```

```
iota :: list<int>

for_each :: list<int> -> (int -> list<int>) -> list<int>

mbind    :: list<int> -> (int -> list<int>) -> list<int>
```

Monads, again??

```
auto triples =
   mbind(iota(1),              [] (int z) { return
   mbind(iota(1, z+1),         [=](int x) { return
   mbind(iota(x, z+1),         [=](int y) { return
   yield_if(x*x + y*y == z*z, make_tuple(x, y, z));
 }); }); });
```

```
auto triples =
 do {
    int z <- iota(1);
    int x <- iota(1, z+1);
    int y <- iota(x, z+1);

    guard (x*x + y*y == z*z);

    return make_tuple(x, y, z);
 }
```

Monadic Pythagorean Triples in Haskell

```
auto triples =
   mbind(iota(1),              [] (int z) { return
   mbind(iota(1, z+1),         [=](int x) { return
   mbind(iota(x, z+1),         [=](int y) { return
   yield_if(x*x + y*y == z*z, make_tuple(x, y, z));
 }); }); });
```

```
auto triples =
 do {
    int z <- iota(1);
    int x <- iota(1, z+1);
    int y <- iota(x, z+1);

    guard (x*x + y*y == z*z);

    return make_tuple(x, y, z);
 }
```

```
triples = do

    z <- [1..]
    x <- [1..z]
    y <- [x..z]

    guard (x*x + y*y == z*z)

    pure (x, y, z)
```

It’s inevitable. This is terrifying.

```
mbind :: expected<A, E> -> (A -> expected<B, E>) -> expected<B, E> // error handling

mbind :: optional<A>    -> (A -> optional<B>)    -> optional<B>    // absent values

mbind :: future<A>      -> (A -> future<B>)      -> future<B>      // parallelism

mbind :: async<A>       -> (A -> async<B>)       -> async<B>       // callback hell
                                                                      elimination
mbind :: parser<A>      -> (A -> parser<B>)      -> parser<B>      // parsing

mbind :: list<A>        -> (A -> list<B>)        -> list<B>        // combinatorics
                                                                      (kind of)
mbind :: reader<A>      -> (A -> reader<B>)      -> reader<B>      // environment data

mbind :: writer<A>      -> (A -> writer<B>)      -> writer<B>      // write-only data

mbind :: state<A>       -> (A -> state<B>)       -> state<B>       // monadic state

mbind :: io<A>          -> (A -> io<B>)          -> io<B>          // effects control
```

Functional composition with ranges

```
for(auto triple : triples | view::take(10)) {
  cout << '('
       << get<0>(triple) << ','
       << get<1>(triple) << ','
       << get<2>(triple) << ')' << '\n';
}
```

Output:

```
(3,4,5)
(6,8,10)
(5,12,13)
(9,12,15)
(8,15,17)
(12,16,20)
(7,24,25)
(15,20,25)
(10,24,26)
(20,21,29)
```

Functional composition with ranges

```
for(auto triple : triples | view::take(10)
                          | view::transform(mul10_middle)) {
 cout << '('
      << get<0>(triple) << ','
      << get<1>(triple) << ','
      << get<2>(triple) << ')' << '\n';
}
```

```
auto mul10_middle = [](const tuple<int, int, int>& t) {
   const auto& [x, y, z] = t;
   return make_tuple(x, y * 10, z);
};
```

Output:

```
(3,40,5)
(6,80,10)
(5,120,13)
(9,120,15)
(8,150,17)
(12,160,20)
(7,240,25)
(15,200,25)
(10,240,26)
(20,210,29)
```

Functional composition with ranges

```
for(auto triple : triples | view::take(10)
                          | view::transform(mul10_middle)
                          | view::reverse) {
 cout << '('
      << get<0>(triple) << ','
      << get<1>(triple) << ','
      << get<2>(triple) << ')' << '\n';
}


auto mul10_middle = [](const tuple<int, int, int>& t) {
   const auto& [x, y, z] = t;
   return make_tuple(x, y * 10, z);
};
```

Output:

```
(20,210,29)
(10,240,26)
(15,200,25)
(7,240,25)
(12,160,20)
(8,150,17)
(9,120,15)
(5,120,13)
(6,80,10)
(3,40,5)
```

Limitations
of Functional Programming in C++

“Problems” of Functional Programming in C++

- Compilation time
- Hard debugging
- Performance

Issues of Functional Programming in C++

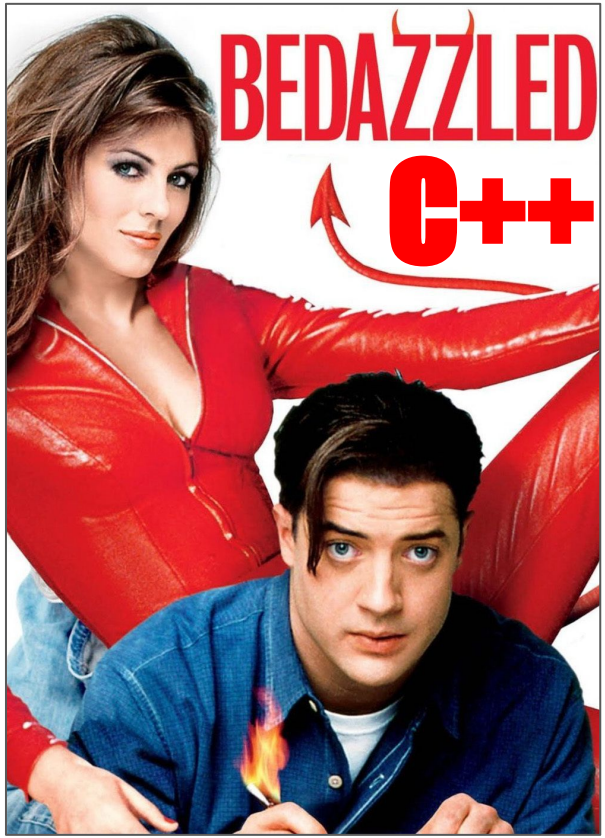
- Absence of currying
- Dangerous closures
- Lambdas are verbose
- Tuple is extremely verbose

Real Problems of Functional Programming in C++

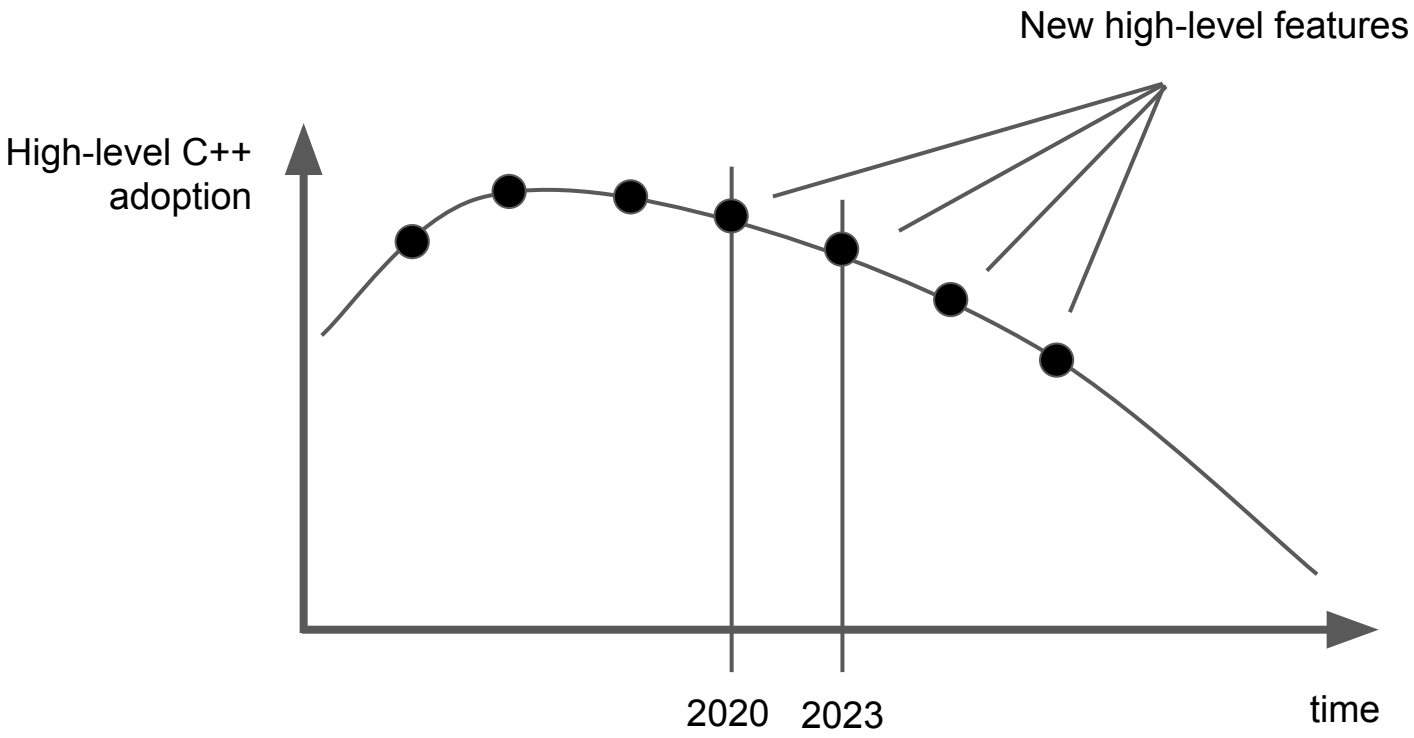
- Poor type system
  - No real Algebraic Data Types
  - No real function type
  - Lambdas have unique types
  - No real pattern matching (yet)
    - Won’t be expression??
  - Type inference is weak
- Composition of functions is complicated
- No native support of monads (in 2019)

Real Problems of Functional Programming in C++

- Poor type system
  - No real Algebraic Data Types
  - No real function type
  - Lambdas have unique types
  - No real pattern matching (yet)
    - Won’t be expression??
  - Type inference is weak
- Composition of functions is complicated
- No native support of monads (in 2019)



Prediction for high-level C++



GitHub List: Functional Programming in C++

- Books
- Articles
- Papers
- Talks
- QA
- Libraries
- Showcase projects

☞ https://github.com/graninas/cpp_functional_programming

C++ Siberia 2019

Thanks for watching!

Let’s talk?

Alexander Granin
graninas@gmail.com

???

Community Driven

- **Community-Driven Software**
  - No chief architect
  - We all care

It's your fault, Alexander!
when "monads" is not the way you want it !
You didn't come to propose it 5 years ago.

Meeting C++ 2018
Closing Keynote
Nicolai Josuttis
50 shades of C++