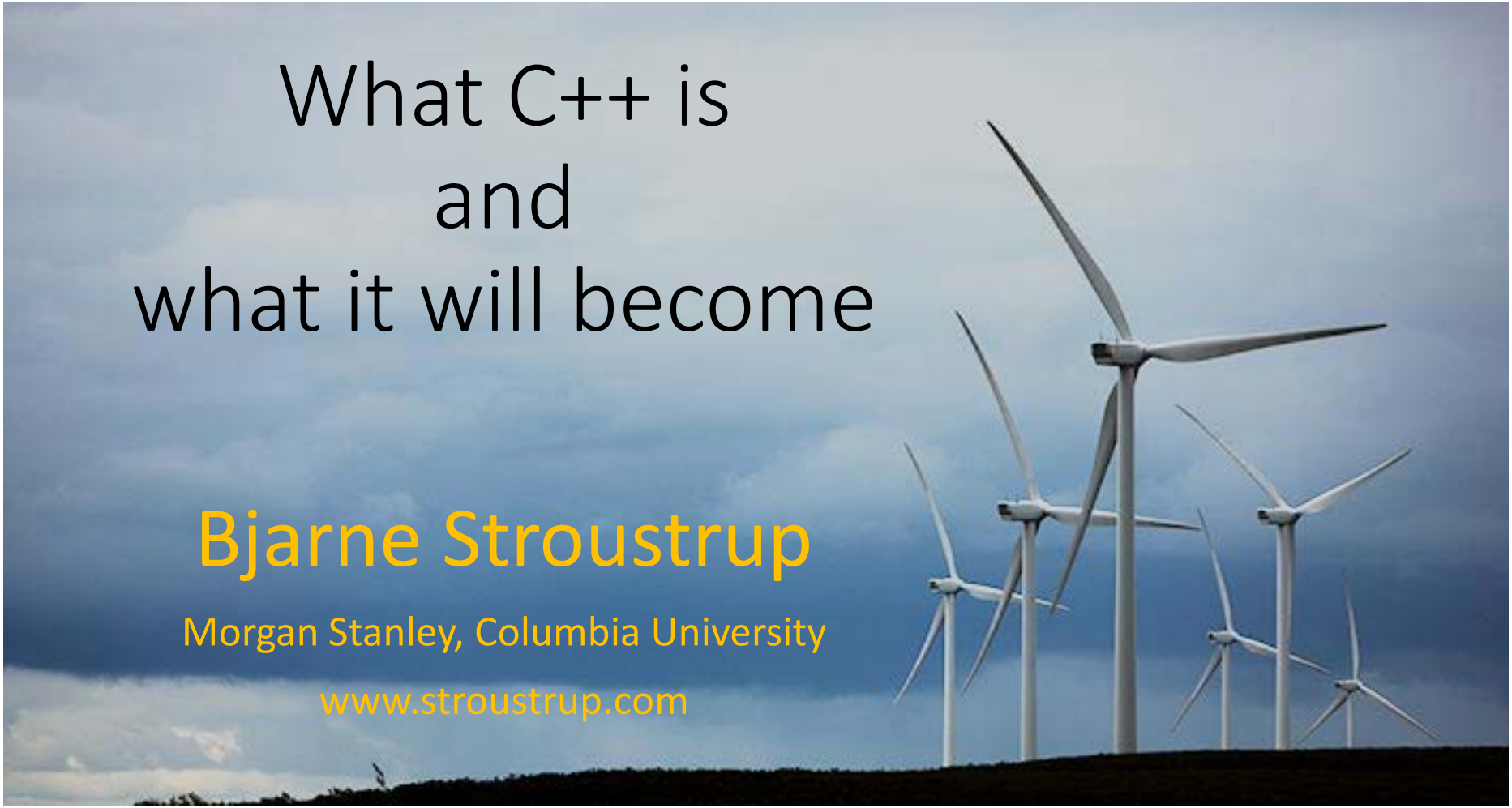


What C++ is and what it will become

Bjarne Stroustrup

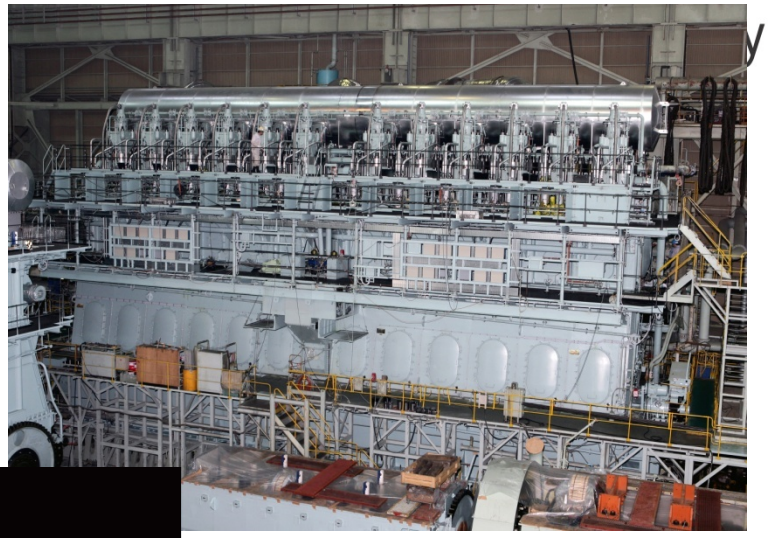
Morgan Stanley, Columbia University

www.stroustrup.com

Overview

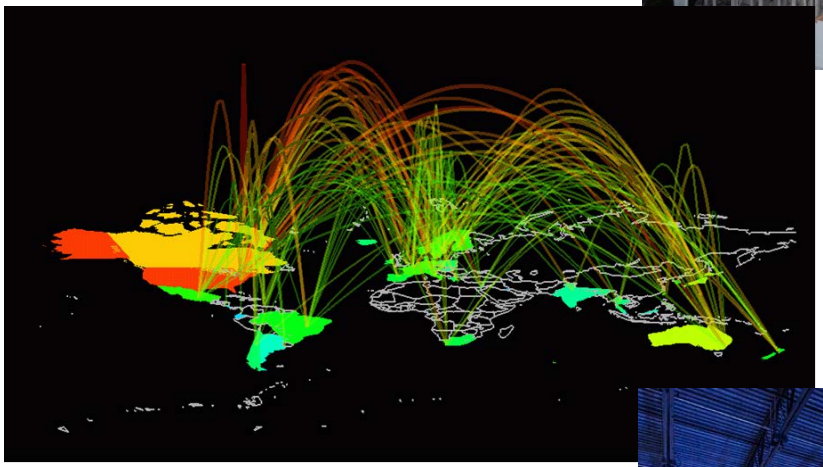
- What is the essence of C++ that we mustn't compromise?
  - Direct access to hardware
  - Zero-overhead abstraction
  - Stability and portability
- What is the likely near future?
  - C++11, C++14, C++17
  - Modules
  - Concepts
  - Contracts
- What can we do better now?
  - Design
  - Experimentation
  - Guidelines

"The value of a programming language is in the quality of its applications"



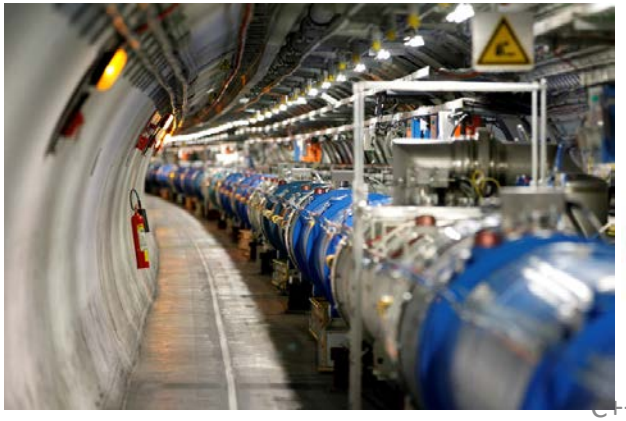
OpenOffice.org

Microsoft .net



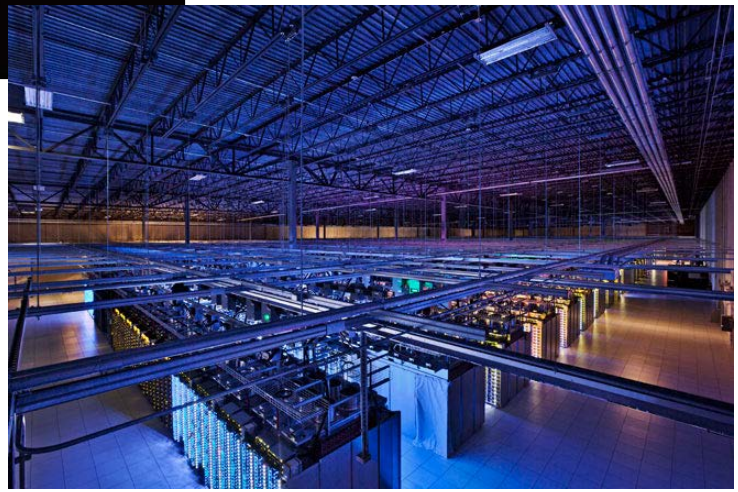
Java

C#

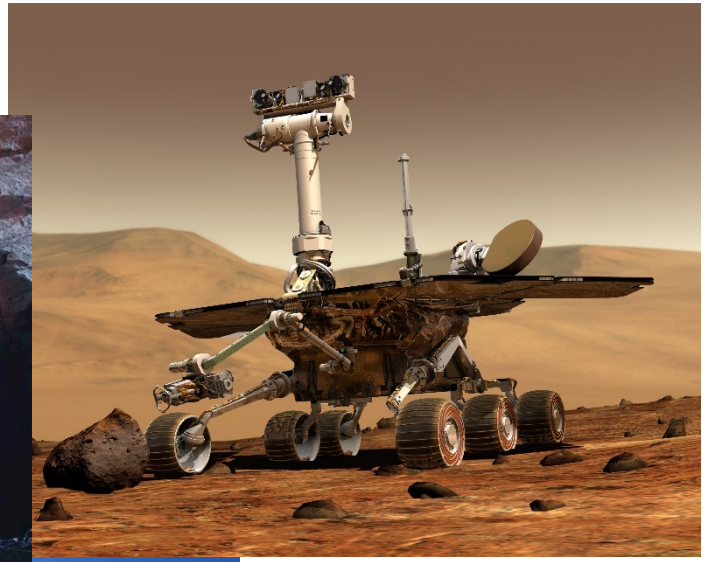


amazon

Google



Morgan Stanley



Morgan Stanley

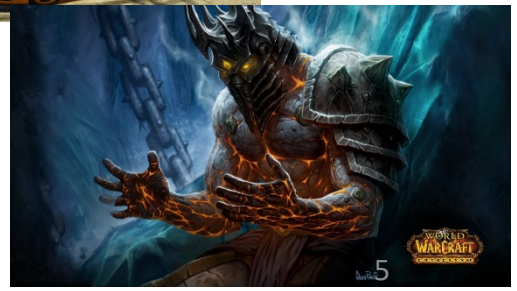
AlphaGo

TOMTOM

GARMIN.

MySQL

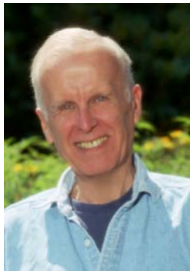
Dropbox



aMaDEUS
Your technology partner

PayPal

Programming Languages



Domain-specific abstraction

Fortran

Cobol

General-purpose abstraction

Simula

C#

Java

Direct mapping to hardware

Assembler

BCPL

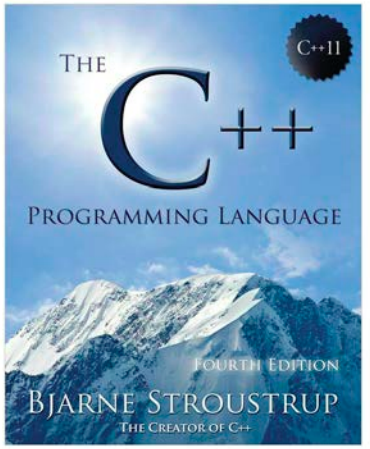
C

C++

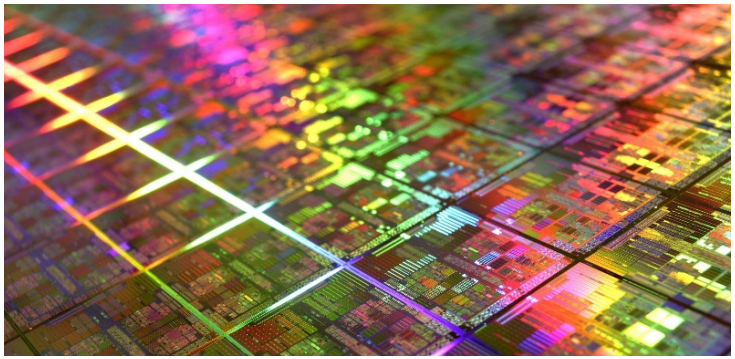
C++11

C++14

C++ in two lines



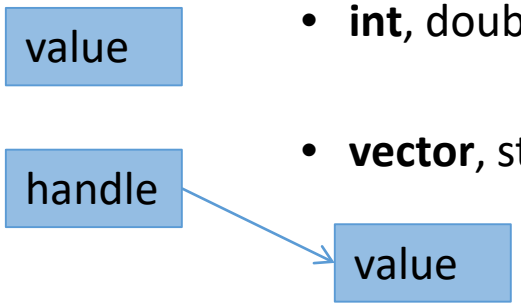
- Direct map to hardware
  - of instructions and fundamental data types
  - Initially from C
  - Future: use novel hardware better (caches, multicores, GPUs, FPGAs, SIMD, …)



- Zero-overhead abstraction
  - Classes, inheritance, generic programming, …
  - Initially from Simula (where it wasn't zero-overhead)
  - Future: Type- and resource-safety, concepts, modules, concurrency, …

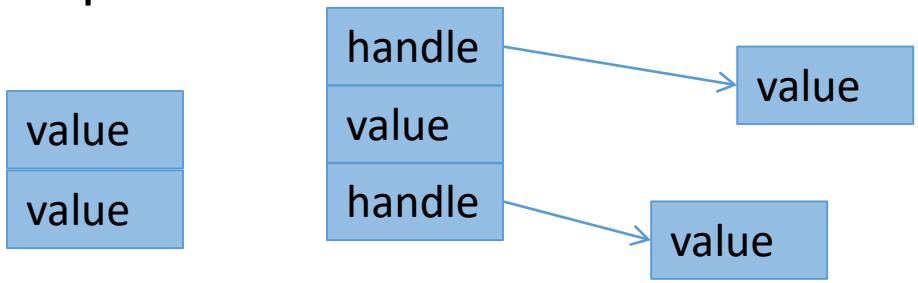
Map to Hardware

- Primitive operations => instructions
  - +, %, ->, [], (), …



- **int**, double, complex<double>, Date, …
- **vector**, string, thread, Matrix, …

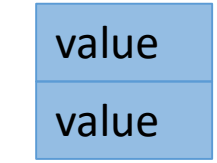
- Objects can be composed by simple concatenation:
  - Arrays
  - Classes/structs

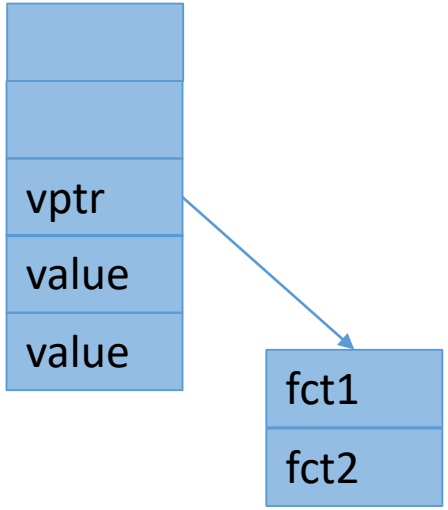


- The simplicity of this mapping is one key to C and C++'s success

Zero-overhead abstraction

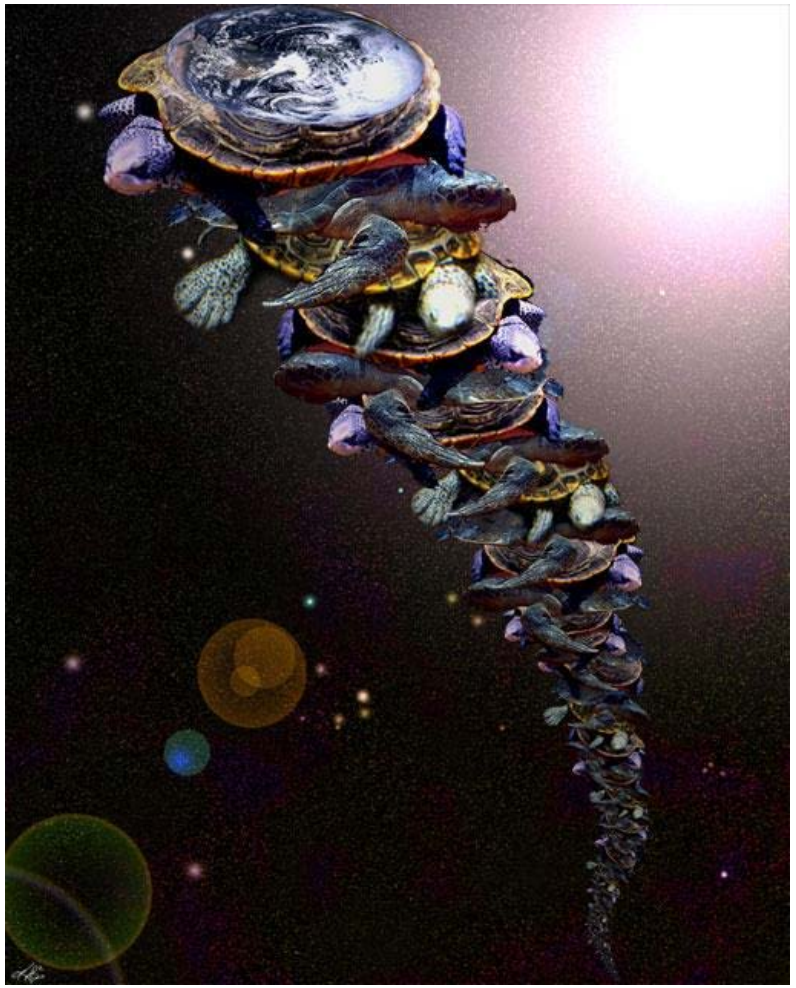
- What you don't use, you don't pay for
- What you do use, you couldn't hand code any better.
  - So you can afford to use the language features
- Examples
  - Point, complex, date, tuple
    - No memory overhead
    - No indirect function call
    - No need to put on free store (heap)
    - Inlining
  - Compile-time computation
    - Pre-compute answers





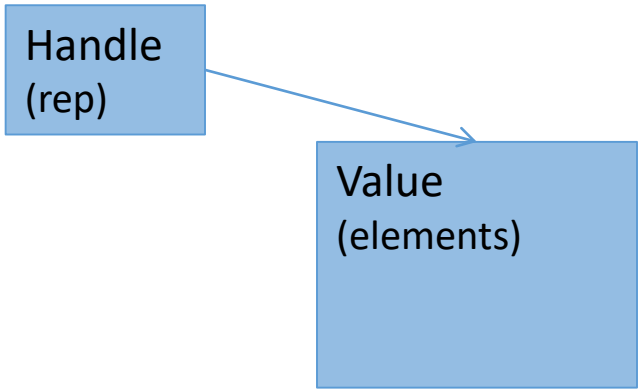
Zero-overhead abstraction

- The C/C++ machine model is itself an abstraction
  - It's abstractions all the way down!



Constructors and destructors

```
template<Element T>
class vector {      // vector of Elements of type T
        vector(initializer_list<T>);  // acquire memory for list elements and initialize
        vector(int n);        // acquire memory for n default elements and initialize
        ~vector();            // destroy elements; release memory
        // …
        vector_rep rep;   // representation
};

void fct()
{
        vector <double> v {1, 1.618, 3.14, 2.99e8};  // vector of 4 doubles
        vector <string> v2(100);                     // vector of 100 strings
        // …
} // memory and strings released here
```



Resource management

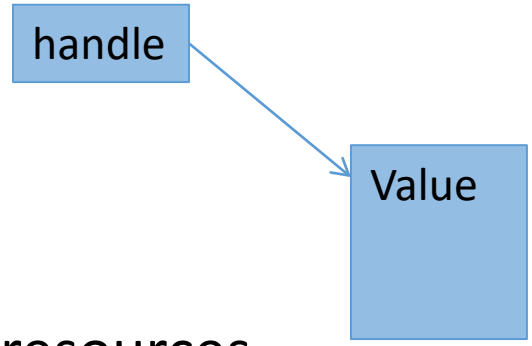
- A resource is something that must be acquired and released
  - explicitly or implicitly
- Examples: memory, locks, file handles, sockets, thread handles

```
void f(int n, string name)
{
        vector<int> v(n);              // vector of n integers
        fstream fs {name,"r"};         // open file <name> for reading
        // …
} // memory and file released here
```

- We must avoid manual resource management
  - We don't want leaks
  - We want to minimize resource retention

Resource Management

- All the standard-library containers manage their elements
  - **vector**
  - **list, forward_list** (singly-linked list), …
  - **map, unordered_map** (hash table),…
  - **set, multi_set, …**
  - **string**
  - All support copy and move
- Other standard-library classes manage other resources
  - Not just memory
  - **thread, lock_guard, …**
  - **istream, fstream, …**
  - **unique_ptr, shared_ptr**



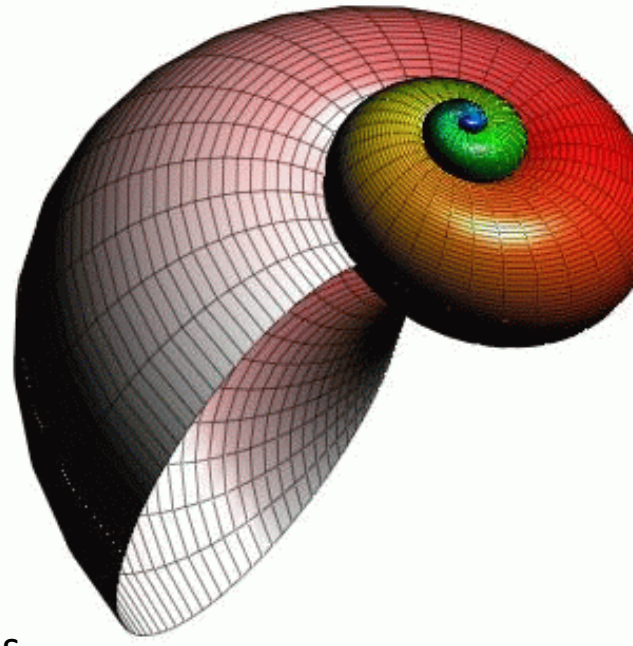
Garbage collection is not sufficient;
We must and can do better

What matters?

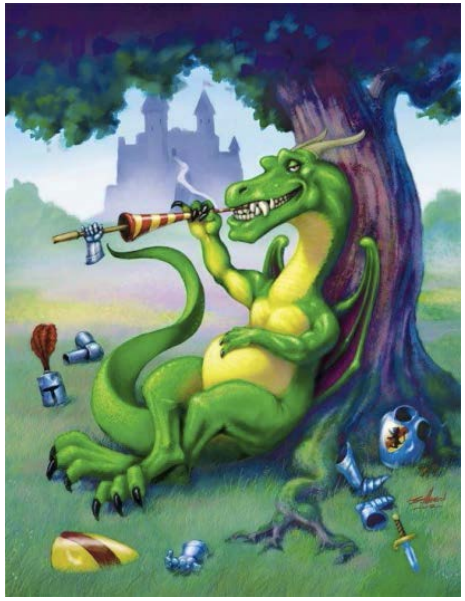
- Far too much for one talk
  - Stability and evolution
  - Tool chains
  - Teaching and learning
  - Technical community
  - Concise expression of ideas
  - Coherence
  - Completeness
  - Compact data structures
  - Lots of libraries
  - …

- Being the best at one or two things isn't sufficient
  - a language must be good enough for everything
    - You can't be sure what "good enough" and "everything" mean to developers
- Don't get obsessed by a detail or two

C++'s role



- C++
  - A general-purpose programming language for
    - Building dependable, affordable software
    - writing elegant and efficient programs
    - for defining and using light-weight abstractions
    - A language for resource-constrained applications
    - building software infrastructure
  - Offers
    - A direct map to hardware
    - Zero-overhead abstraction
- No language is perfect
  - For everything
  - For everybody



Evolution

- C++11 was a major improvement
  - C++14 completes C++11
  - C++17 adds many minor improvements
- Lots of new features
  - Concurrency, random numbers, regular expressions, …
  - Lambdas, generalized constant expressions, …
- Simplification of use
  - Auto, range-for, uniform initialization, moves, …
- Currently shipping
  - Even features beyond C++17

C++98: a solid work horse

- Good OO support
  - Classes
  - Class hierarchies
- Integrated resource management
  - RAII
  - Exceptions
- Support for Generic Programming
  - STL
  - Template metaprogramming
  - The language is creaking under the weight of the GP success

Example: Make simple things simple

- 1972

```
int i;
for (i=0; i<max; i++) v[i]=0;
```

- 1983

```
for (int i=0; i<max; ++i) v[i]=0;
```

- 2011

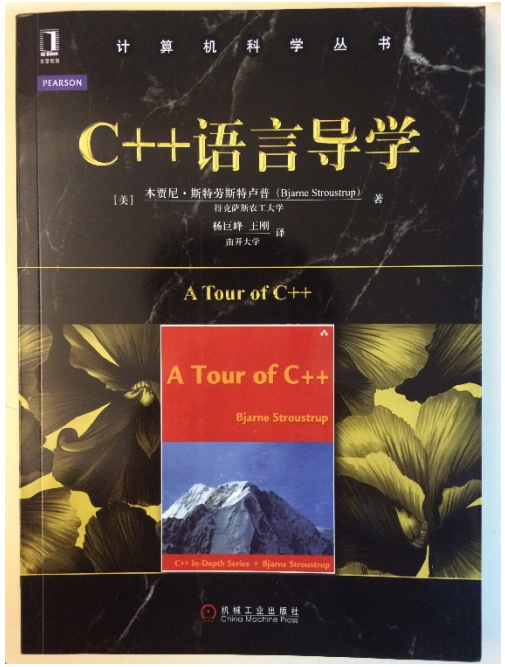
```
for (auto& x : v) x=0;
```

- Note: the simpler code is as fast, and safer than the old

```
for (i=0; i<=max; j++) v[i]=0;     // Ouch! And double Ouch!!
```

C++11: 10+ years of experience added

- "C++11 feels like a new Language"
  - Resource management pointers: **unique_ptr**, **shared_ptr**
  - Concurrency support: **thread**, **mutex**, **future**, etc.
  - Generalized and guaranteed constant expression evaluation: **constexpr**
  - Uniform initialization using **{}**-lists
  - Type deduction from initializer: **auto**
  - Range-**for** statement
  - Null pointer keyword: **nullptr**
  - Strongly-typed enums: **enum class**
  - Compile-time assertions: **static_assert**
  - Move semantics
  - Lambdas
  - Variadic templates
  - **tuple**s
  - Type and template aliases: **using**
  - Raw string literals
  - Controls of defaults: **=default** and **=delete**
  - Override controls: override and final
  - …



Range-**for**, **auto**, and move

- As ever, what matters is how features work in combination

```
template<typename C, typename V>
vector<Value_type<C>*> find_all(C& c, V v)  // find all occurrences of v in c
{
        vector<Value_type<C>*> res;
        for (auto& x : c)
            if (x==v)
                res.push_back(&x);
        return res;
}

string m {"Mary had a little lamb"};
for (const auto p : find_all(m,'a'))            // p is a char*
    if (*p!='a')
        cerr << "string bug!\n";
```

Example: Concise expression of ideas

- Auto and lambda
  - Avoid repetition of type
  - Preserve inlining opportunities
  - Improve locality
  - No, you don't have to use them everywhere
    - (every good new feature will be overused and misused)

```
for (auto& x: v)            // iterate through all elements
        cout << x << '\n';

sort(v, [](auto x, auto y) { return x>y; });    // specify sorting criterion

set_onclick([&context]{ /* ??? */ });          // specify what's to be done on click
```

C++14: completing C++11

- "A deliberately minor release"
  - Function return type deduction: **auto square(double d) { return d*d; }**
  - More general **constexpr** evaluation
  - Variable templates
  - Binary literals: **0b0001001000110100**
  - Digit separators: **1'234'567, 0b0001'0010'0011'0100**
  - Generic lambdas
  - Standard-library literal suffices: **12s** (12 seconds), **"Hello!"s** (a **std::string**)
  - Tuple addressing via type: **get<int>(t)**
  - …

Example: Compile-time computation

- **constexpr** brings type-rich programming to compile time
  - If you know the answer, just use it
  - It's hard to run faster than a table lookup
  - You can't have a race condition on a constant
  - macros or template metaprogramming can be very complicated and error-prone

```
constexpr int isqrt(int n)          // evaluate at compile time for constant arguments
{
        int i = 1;
        while (i*i<n) ++i;
        return i-(i*i!=n);
}

constexpr int s1 = isqrt(9);        // s1 is 3
constexpr int s2 = isqrt(1234);     // s2 is 35

cout << weekday{jun/21/2016} << '\n';              // Tuesday
static_assert( weekday{jun/21/2016}==tue );        // we can do that at compile time
```

C++17: many small improvements

- Approved last week
  - Template argument deduction for constructors
  - Guaranteed copy elision
  - Order of evaluation guarantees
  - Compile-time if
  - Inline variables
  - Structured bindings
  - **[[fallthrough]]**
  - Standard-library vocabulary types: **variant**, **any**, **optional**, **string_view**
  - File system library
  - Some parallel algorithms
  - Mathematical special functions

Example: Simplify

- Make many forwarding functions redundant
  - Why **make_pair()**, **make_tuple()**, …?
    - They deduce template argument types
  - Are you sure your “make functions” don’t make spurious copies?
  - Being explicit about template argument types can be a bother
    - And error prone

  - **pair<string,int> x("the answer",42);** // *C++98*

  - **auto y = make_pair(string("the answer"),42);** // *C++11*

  - **pair z {"the answer"s,42};** // *C++17*

Example: structured bindings

- Simpler multiple return values (try it in Clang 4.0)
  - Giving local names to struct members
  - Less need for uninitialized variables (important)
- Simpler error-code checking

```
map<int,string> mymap;
// …
auto [iter, success] = mymap.insert(value);
        // types are: iter is a mymap<int,string>:: iterator, success is a bool
if (success) f(*iter);
```

- Simpler loops

```
for (const auto& [key, value] : mymap)
        cout << key << " -> " << value << '\n';
```

Where do we go from here?

- "Dream no little dreams"
  - My aims include
    - Type- and resource safe
    - As fast or faster than anything else
    - Good on "modern hardware"
    - Significantly faster compilation catching many more errors

- "The best is the enemy of the good"
  - Don't just dream
    - Support directed change
    - Take concrete, practical steps
    - Now!

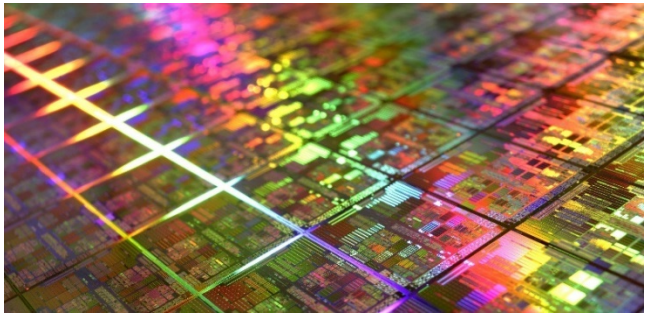
Some philosophy

- We will make errors
  - Make them early so that we can fix them
- Maximize successes
  - Rather than minimizing failures
- Any change carries risk
  - Doing nothing is also risky
- Integrate early
  - And be willing to back out if wrong
- Be confident
  - On average we have succeeded

- It is more important to support good programming than to prevent bad programming
  - D&E
- Don’t confuse familiarity and simplicity
  - Such confusion hinders and delays major improvements

Now let's look ahead

- My high-level aims for C++~~17 and beyond~~
  - Improve support for large-scale dependable software
  - Support higher-level concurrency models
  - Simplify core language use and address major sources of errors.
- Preserve C++'s fundamental strengths
  - Direct map to hardware
  - Zero-overhead abstraction
- Avoid:
  - Abandoning the past
    - stability – backwards compatibility – is a feature
  - Failing to address new challenges
    - e.g., not supporting new hardware (e.g., GPUs, FPGAs)
  - Small-feature creep



My top-ten list for C++17 (in early 2015)

- Concepts
  - Concept-based generic programming, good error messages
- Modules
  - Fast compilation through cleaner code
- Ranges (library)
- Uniform call syntax
- Co-routines
  - Fast and simple
- Networking (library, asio)
- Contracts
- SIMD vector and parallel algorithms (mostly library)
- Library "vocabulary types"
  - *such as* ***optional***, ***variant***, ***string_span***, *and* ***span***
- A "magic type" **stack_array**

It's hard to make predictions, especially about the future

So what can we do *now*?

- Get ready for C++17
  - Upgrade to C++14 if you haven't already
  - Try out new features that'll help further
    - C++17 has nothing major, but lots of minor improvements
      - Structured binding, template argument deduction for constructors, …
      - variant, optional, …
    - I hope for rapid implementation compliance
- Try out the TSs now shipping
  - Concepts, Ranges, Networking, Coroutines, Modules, …
- Use the Core Guidelines
  - Improve them
  - Improve tool support

Generic Programming is “just” Programming

- Traditional code

```
double sqrt(double d);      // C++84: accept any d that is a double

double d = 7;
double d2 = sqrt(d);        // fine: d is a double

vector<string> vs = { "Good", "old", “templates" };
double d3 = sqrt(vs);       // error: vs is not a double
```

Generic Programming is "just" Programming

- Generic code using a concept (Sortable)

```
void sort(Sortable& c);  // Concepts: accept any c that is a Sortable container

vector<string> vs = { "Hello", "new", "World" };
sort(vs);        // fine: vs is a Sortable container

double d = 7;
sort(d);         // error: d is not a Sortable container
                 //  (double does not provide [], begin(), etc.)
```

Concepts

- Concepts are compile-time predicates
  - They give us precisely specified interfaces
- Error handling is simple (and fast)

```
template<typename Cont>
        requires Sortable<Cont>    // Sortable is a Sequence with random access
                                   // with elements that you can compare using <
                void sort(Cont& c);

vector<double> vec {1.2, 4.5, 0.5, -1.2};
list<int> lst {1, 3, 5, 4, 6, 8,2};

sort(vec);          // OK
sort(lst);          // Error at (this) point of use
```

- Actual error message

  error: 'list<int>' does not satisfy the constraint 'Sortable'
  - More information upon request

Concepts: overloading

- But what if we do want to sort a list?

```
template<Sortable Cont>              // shorthand: Cont is a type that is Sortable
        void sort(Cont& container);

template<Sequence Seq>
        void sort(Seq& seq)    // sort a sequence that doesn't offer random access
        {
                vector<Value_type<Seq>> v {begin(seq),end(seq)};
                sort(v);
                copy(begin(v),end(v),seq);
        }

sort(vec);      // OK: use sort of Sortable
sort(lst);      // OK: use sort of Sequence
```

- We don't say **Sequence < Sortable**
  - we compute that from their definitions

Example: Use a "module" (current style)

- Today: #include and macro proliferation

```
#include <iostream>              // what's in here? Affects date.h?
#include "Calendar/date.h"       // what's in here?

int main()
{
        using namespace Chrono;
        Date date { 22, Month::Sep, 2015 };
        std::cout << "Today is " << date << '\n';
}
```

- 176 bytes of user-authored text expands to
  - 412KB with GCC 5.2.0 – about 412KB (about 235% expansion)
  - 1.2MB with Clang 3.6.1 – about 1.2MB (about 685% expansion)
  - 1.1MB VC++ Dev14 – about 1.1MB (about 615% expansion)
- And .h files are often #included dozens or hundreds of times
  - (your compiler is really, really good/fast, but it has an impossible task)

Example: Use a module

(TS, Microsoft is shipping beta)

- Code hygiene
- Fast compilation

```
import std.io;
import calendar.date;

int main() {
        using namespace Chrono;
        Date date { 22, Month::Sep, 2015 };
        std::cout << "Today is " << date << '\n';
}
```

Example: Define a module

- Not rocket science
- Can be introduced gradually

```
import std.io;
import std.string;

module calendar.date;

namespace Chrono {
    export
        struct Date {
            // … the conventional members …
        };

        export std::ostream& operator<<(std::ostream&, const Date&);
        export std::string to_string(const Date&);
}
```

Example: the sum is greater than the parts

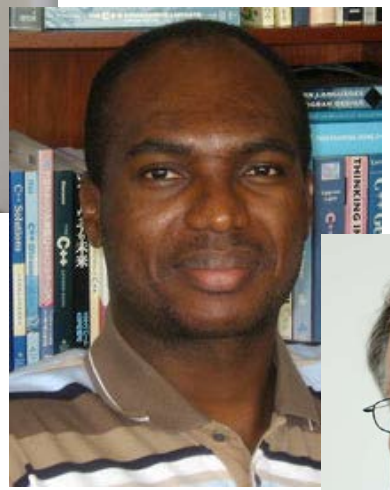
- But I can't test/use combinations of TS features
  - Modules (Microsoft), concepts (GCC), structured bindings (Clang)

```
import iostream;
using namespace std;

module map_printer;

export
template<Sequence S>
void print_map(const S& m)
        requires Printable<KeyType<S>> && Printable<ValueType<S>>;
{
        for (const auto& [key,val] : m)        // break out key and value
                cout << key << " -> " << val << '\n';
}
```

A few contributors (and thanks to many more)

J. Daniel Garcia

Andrew Sutton



Gabriel Dos Reis

Herb Sutter

Alex Stepanov

Bjarne Stroustrup

Mike Spertus

John Lakos

What can we do *now*?

- What would you like your code to look like in 5 years time?
  - "Just like what I write today" is a poor answer
- Use C++14
  - GCC, Clang, Microsoft, …
  - You can now say most things simpler and more directly than in C++98 – and it runs faster
- Use C++17 libraries (already available)
  - Asio, File system
- Experiment with
  - Concepts (GCC)
  - Modules (Microsoft)
- Work on or support standardization or implementation efforts
  - Contracts, …
- Use C++ better
  - Core guidelines

Guidelines: High-level rules

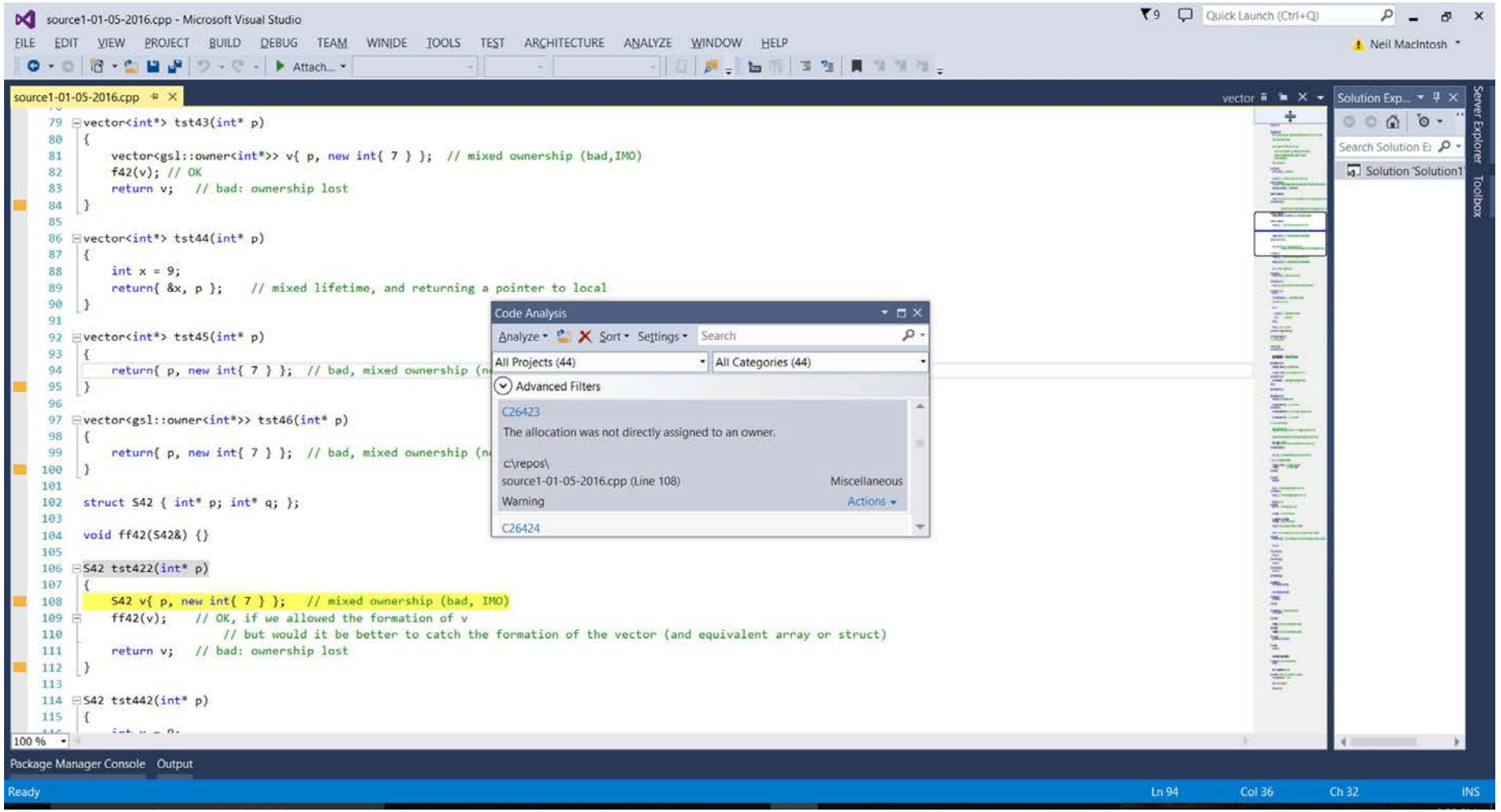
- Provide a conceptual framework
  - Primarily for humans
- Many can't be checked completely or consistently
  - *P.1: Express ideas directly in code*
  - *P.2: Write in ISO Standard C++*
  - *P.3: Express intent*
  - *P.4: Ideally, a program should be statically type safe*
  - *P.5: Prefer compile-time checking to run-time checking*
  - *P.6: What cannot be checked at compile time should be checkable at run time*
  - *P.7: Catch run-time errors early*
  - *P.8: Don't leak any resource*
  - *P.9: Don't waste time or space*

Guidelines: Lower-level rules

- Provide enforcement
  - Some complete
  - Some heuristics
  - Many rely on static analysis
  - Some beyond our current tools
  - Often easy to check “mechanically”
- Primarily for tools
  - To allow specific feedback to programmer
- Help to unify style
- Not minimal or orthogonal

  - *F.16: Use **T*** or **owner<T*>** to designate a single object*
  - *C.49: Prefer initialization to assignment in constructors*
  - *ES.20: Always initialize an object*

Static analyzer (integrated)



C++ Core Guidelines

- You can write type- and resource-safe C++
  - No leaks
  - No memory corruption
  - No garbage collector
  - No limitation of expressibility
  - No performance degradation
  - ISO C++
  - Tool enforced
- Work in progress
  - "Help wanted" – MIT license
  - C++ Core Guidelines: https://github.com/isocpp/CppCoreGuidelines
  - GSL: Guidelines Support Library: https://github.com/microsoft/gsl
  - Static analysis support tool: In Microsoft Visual Studio
    - Work started for Clang Tidy

Caveat: Not yet deployed at scale ☹

Summary

- C++ is true to its principles
  - Direct hardware access
  - Zero-overhead abstraction
  - Static typing
- C++11/C++14/C++17 represent major progress
  - GCC, Clang, Microsoft, …
  - You can now say most things simpler and more directly than in C++98 – and it runs faster
- Use C++ better
  - Core guidelines
- Experiment
  - Concepts (GCC), Modules (Microsoft), Coroutines (Microsoft and clang), Networking (everywhere), Ranges (everywhere), …
- Support the standardization and implementation efforts
  - Contracts, …

Likely C++17 feature list (language)

- Structured bindings. E.g., **auto [re,im] = complex_algo(z);**
- Deduction of template arguments. E.g., **pair p {2, "Hello!"s};**
- More guaranteed order of evaluation. E.g., **m[0] = m.size();**
- Guaranteed copy elision
- Auto of a single initialize deduces to that initializer. E.g., **auto x {expr};**
- Compile-time if, e.g., **if constexpr(f(x))** …
- Deduced type of value template argument. E.g., **template<auto T>** …
- **if** and **switch** with initializer. E.g., **if (X x = f(y); x)** …
- Dynamic memory allocation for over-aligned data
- **inline** variables (Yuck!)
- **[[fallthrough]]**, **[[nodiscard]]**, **[[maybe unused]]**
- Lambda capture of *this. E.g. **[=,tmp=*this]** …
- Fold expressions for parameter packs. E.g., **auto sum = (args + …);**
- Generalized initializer lists
- …

Likely C++17 feature list (library)

- This not a library talk, so no details
  - File system library
  - Parallelism library
  - Special math functions. E.g., **riemann_zeta()**
  - **variant**, **optional**, **any**, **string_view**
  - Many minor standard-library improvements
  - …
- The standard library is now >50% of the standard
- We need great libraries!