The Evolution of C++
Past, Present, and Future

Bjarne Stroustrup

Morgan Stanley, Columbia University

www.stroustrup.com

Overview

- *The past*: Why did C++ succeed?
  - By answering questions before people asked them
  - Not by following the herd
- *The present*: How is standardization shaping C++?
  - Aiming for stability through compatibility
  - Having a hard time choosing and keeping a direction
- *The future*: What do we need to do?
  - Focus our efforts to serve the C++ current and future community
  - Don’t diffuse efforts trying to please everybody
- *The near future*: How we manage until the future comes?
  - Find ways of using features from ISO Technical Specifications
  - Develop guidelines

The Evolution of C++

- A philosophical talk
  - Principles/ideals
  - Plus practice/examples
- *Not*
  - Long lists of features
  - Really cool stuff you can use tomorrow
- Why did C++ succeed?
  - What must we do to sustain that success?
  - "Being lucky" is not a plausible explanation for the 35+ years
  - My focus is C++ itself, rather than the broader IT industry

“Dream no little dreams”

- Change the way people think about code
- Future C++
  - Type- and resource-safe
  - Significantly simpler and clearer code
  - As fast or faster than anything else
  - Good at using “modern hardware”
  - Significantly faster compilation catching many more errors

"there is nothing more difficult to carry out, nor more doubtful of success, nor more dangerous to handle, than to initiate a new order of things."†

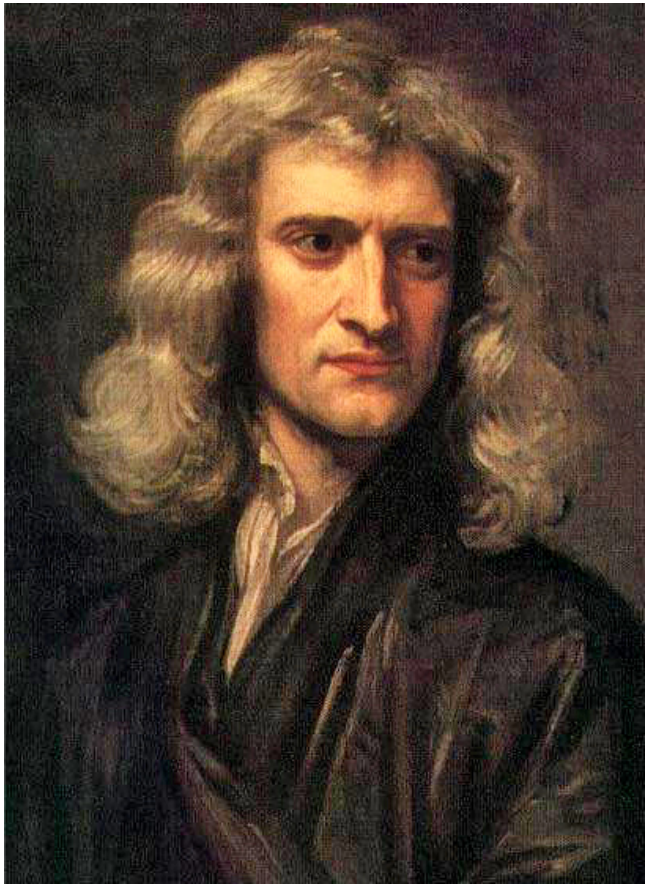
† As quoted in TC++PL3

"there is nothing more difficult to carry out, nor more doubtful of success, nor more dangerous to handle, than to initiate a new order of things. For the reformer makes enemies of all those who profit by the old order, and only lukewarm defenders in all those who would profit by the new order."

Developers love minor changes that help a little with current problems, but many oppose anything that might upset status quo

"The best
is the enemy of
the good"

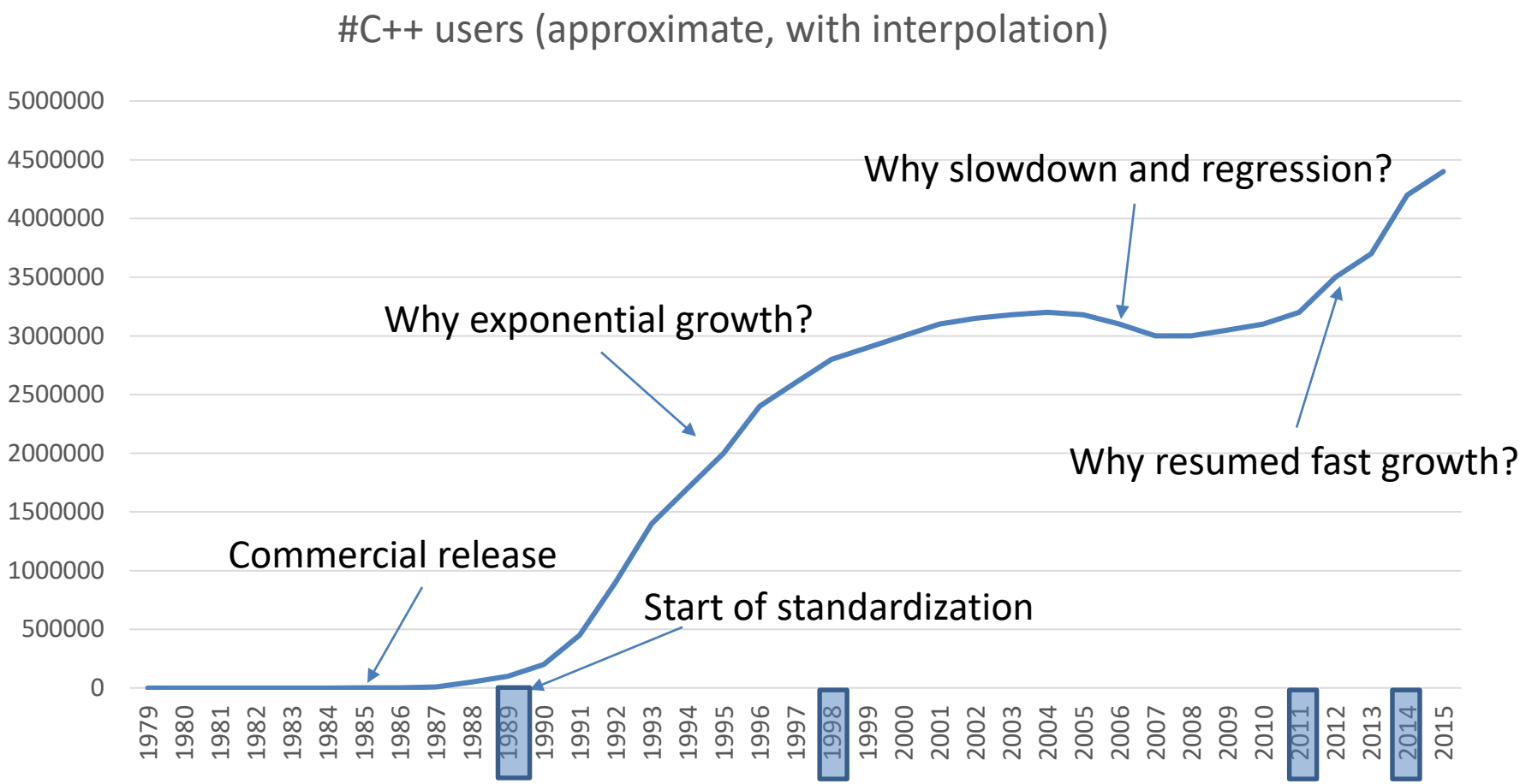
- Don't just dream
  - Take concrete, practical steps
  - Now!



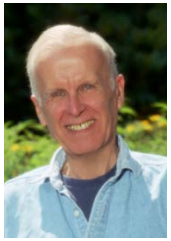
"If I have seen a little further it is by standing on the shoulders of Giants."

- Thanks!
  - Kristen Nygaard
  - Dennis Ritchie
  - Alex Stepanov
  - Christopher Strachey
  - And many, many more

C++: Success


#C++ users (approximate, with interpolation)
5000000
4500000
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
Why slowdown and regression?
Why exponential growth?
Why resumed fast growth?
Commercial release
Start of standardization
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015

The roots of C++



Domain-specific abstraction

Fortran

Cobol

General-purpose abstraction

Simula

Java

Direct mapping to hardware

C++

C++11

C++14

C#

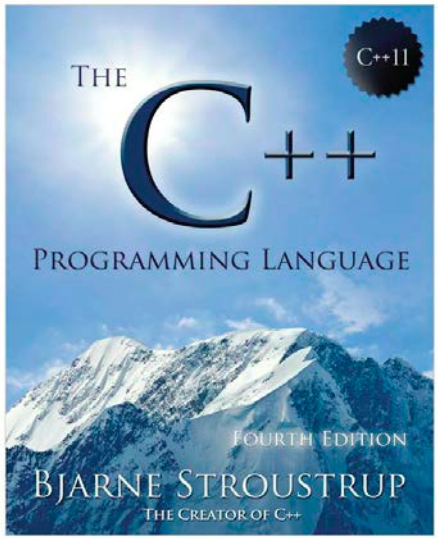
Assembler

BCPL

C

Morgan Stanley

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, …

Example: Choose your level of abstraction

- *High-level*: Say what you want done

```
vector<string> v;
for (string s; cin>>s; ) v.push_back(s);
```

- *Low-level*: Say how you want something done

```
char* strings[maxs];
for (int s=0; s<maxs; ++s) {
        char buf[max];
        int i;
        for (i=0; i<max-1; ++i) getc(buf[i]);  // remember to check for overflow
        buf[i] = '\0';
        strings[s] = (char*)malloc(i);          // remember to free
        memcpy(strings[s], buf, i);
}
```

- The high/low distinction is fluid

What matters? (Software development)

- 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
  - Performance
  - 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

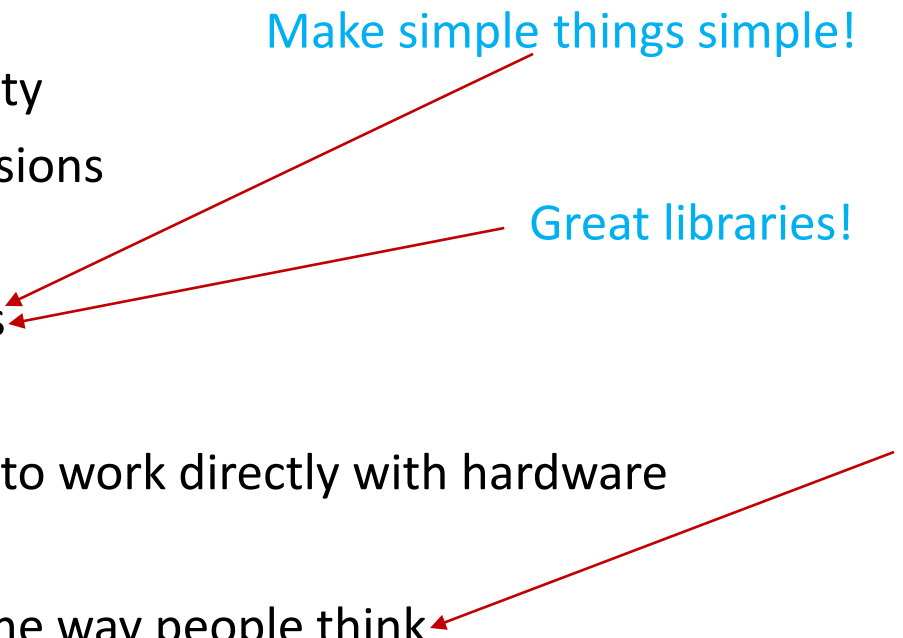
Language feature design

- A language needs good fundamental features
  - mechanisms supporting design and programming techniques
  - Language and standard library
- A (good) language is not just a set of good features
  - The overall framework must be sound
  - Don’t just take the first solution that seems to work
- 90+% of the work is integration
  - Features need lots of “glue and polishing” to work well together
  - Takes more time and thought than most people are willing to believe

- People spend most time discussing individual features
  - Especially pre-acceptance

C++ Design rules of thumb

- 2003 list
  - Maintain stability and compatibility
  - Prefer libraries to language extensions
  - Prefer generality to specialization
  - Support both experts and novices
  - Increase type safety
  - Improve performance and ability to work directly with hardware
  - Fit into the real world
  - Make only changes that change the way people think
- Reflecting decisions from the earliest days
  - See “Design and Evolution of C++” and my HOPL papers



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

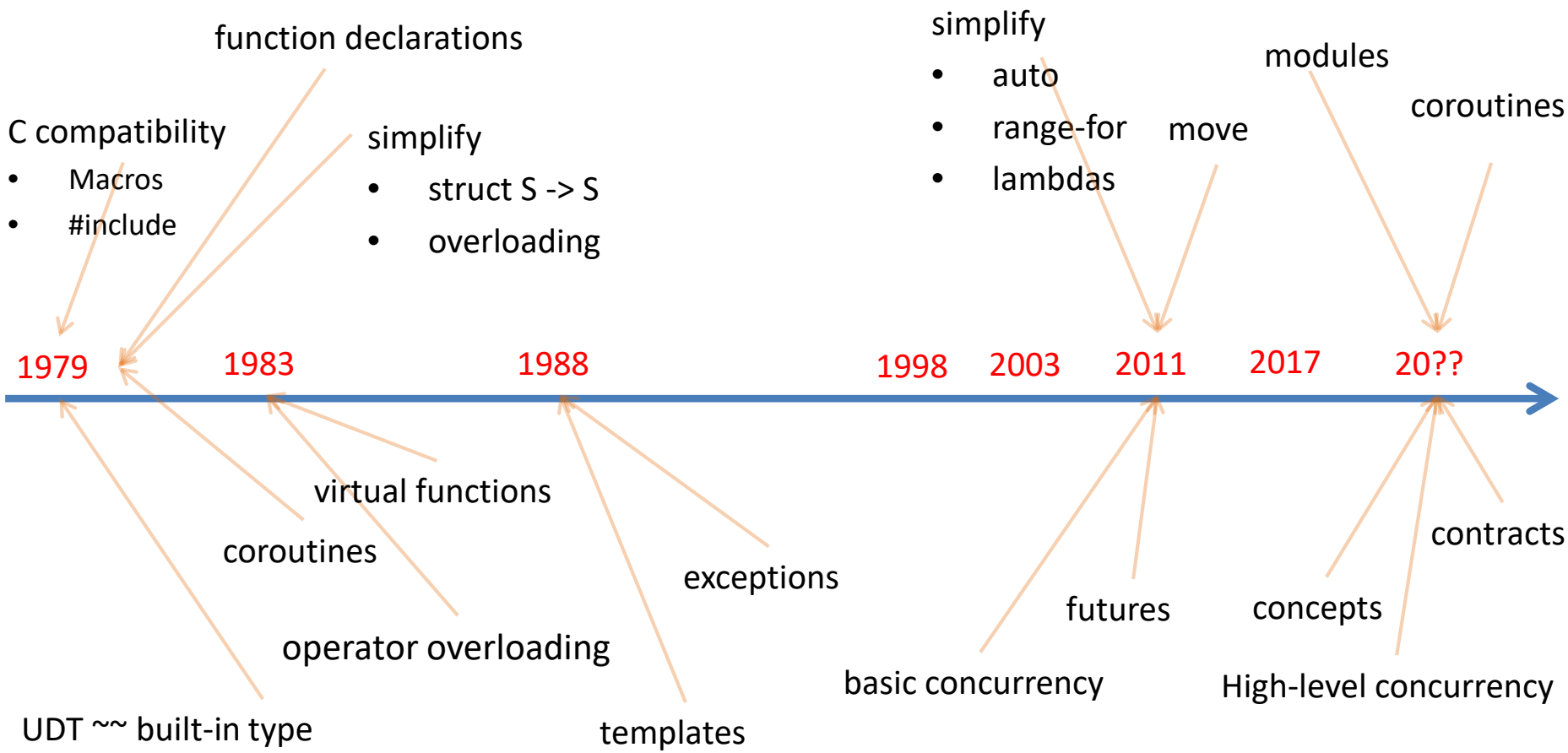
```
for (auto& x : v) x=0;
```

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

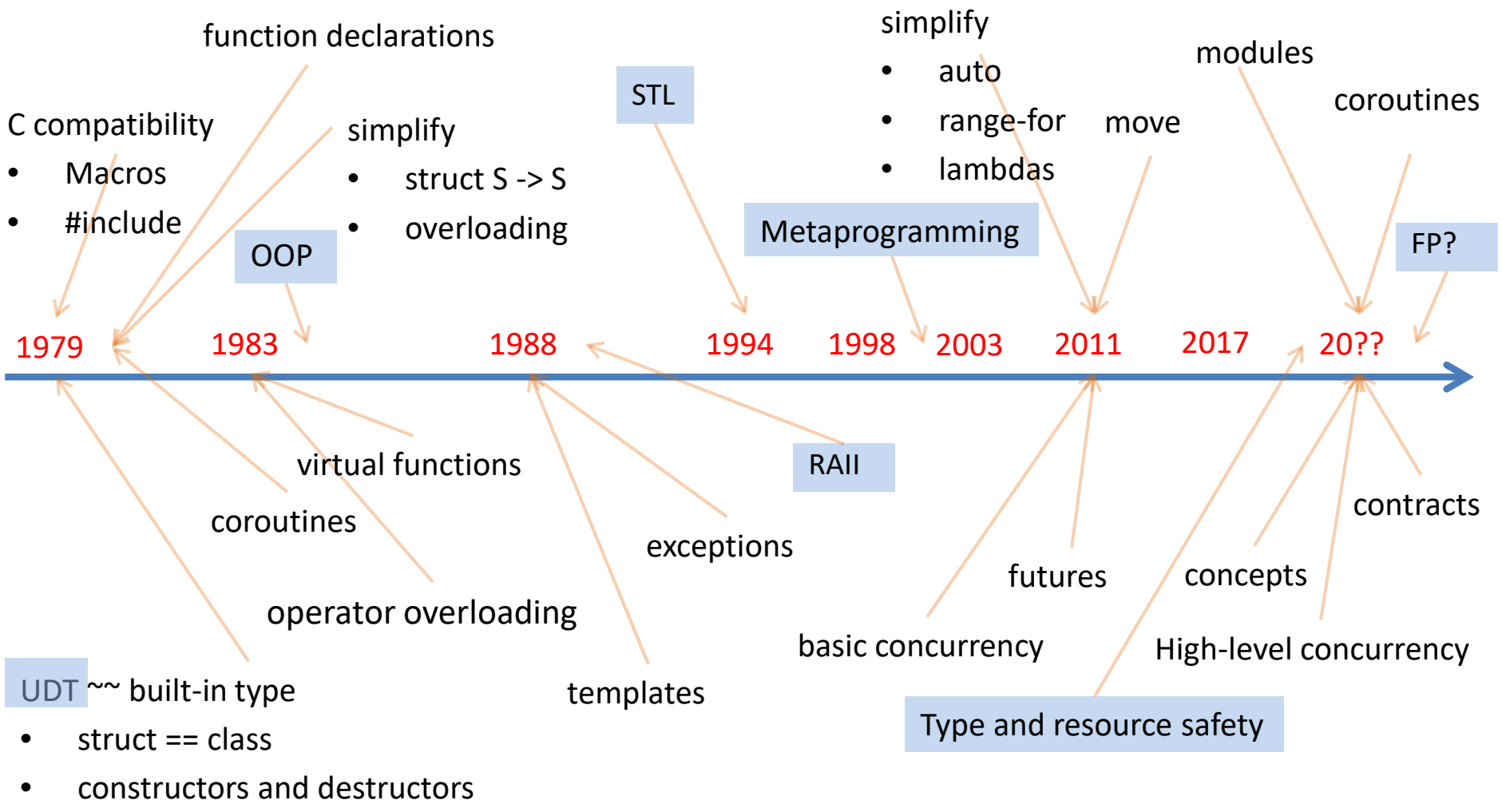
```
for (i=0; i<=max; j++) v[i]=0;     // Ouch! And double Ouch!!
```

Morgan Stanley

Major design decisions: Key language features

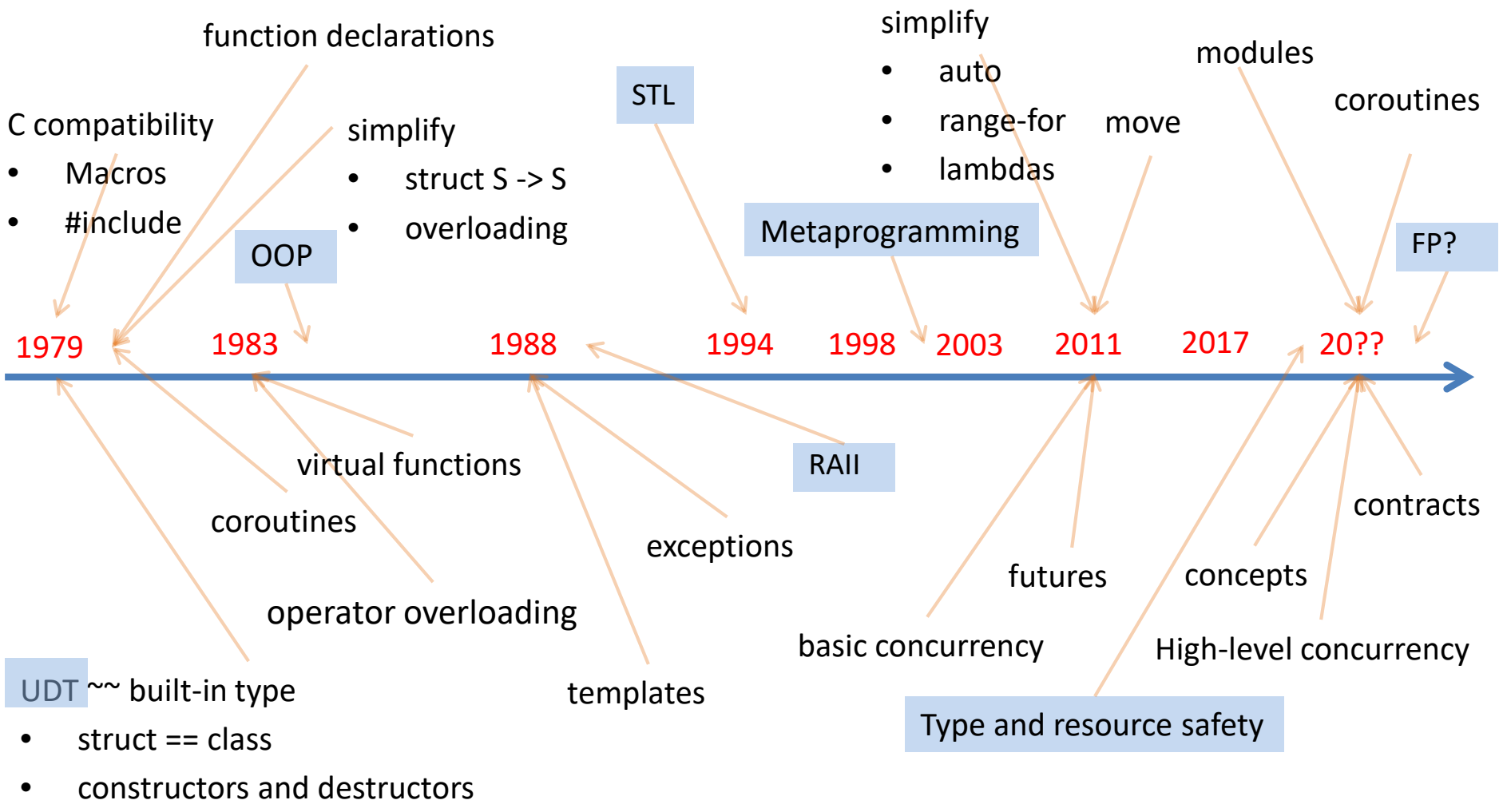


Major design decisions:
How does it change the way we write code?



Morgan Stanley

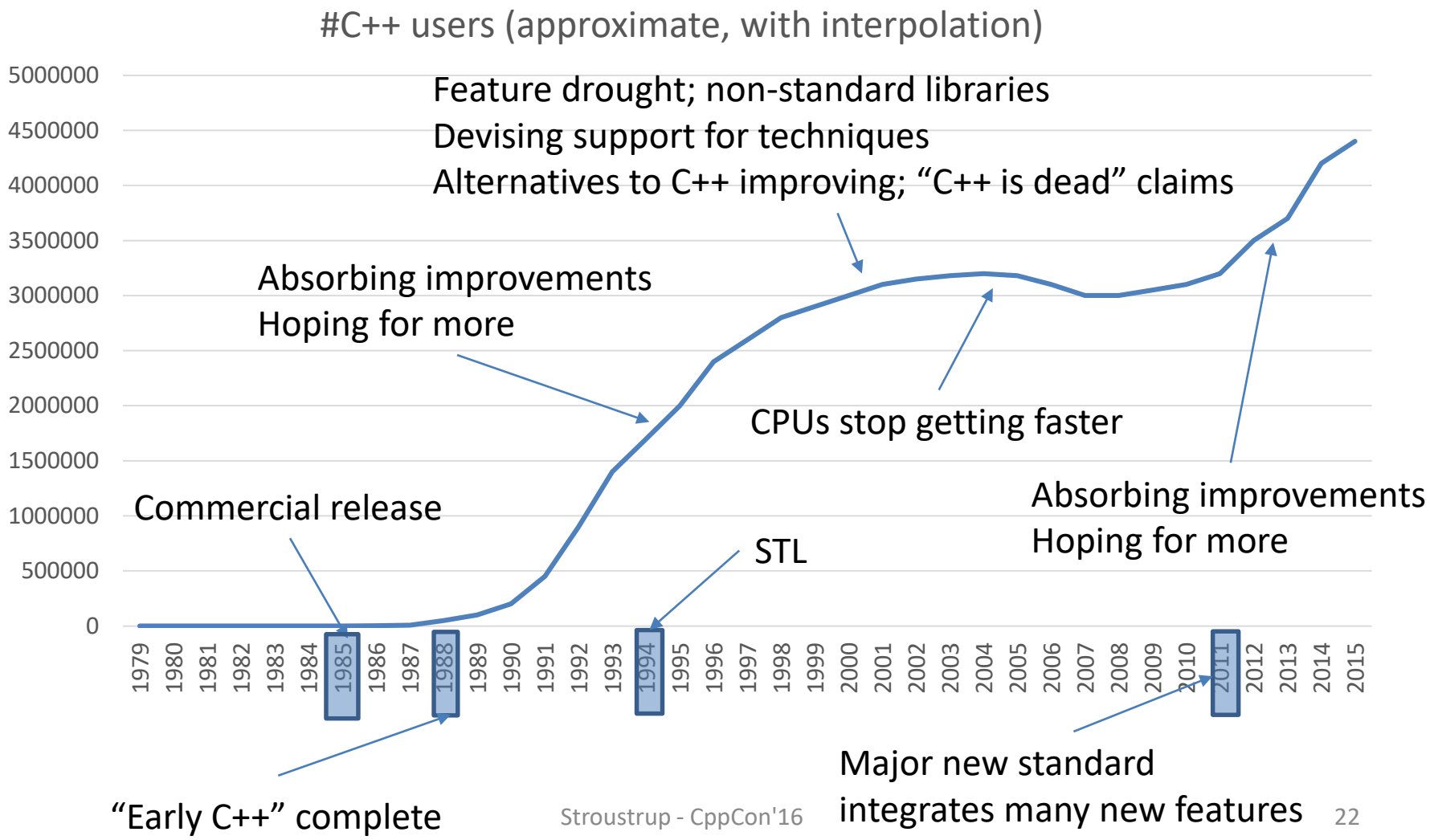
Major design decisions: Evolution is bursty



Language evolution

- Major changes come in bursts
  - To enable new ways of thinking about code
  - Slow, steady, continuous change is (by itself) mostly churn and polishing
- Most changes aren't major
  - Minor changes sustain status quo, but don't change its fundamentals
  - Minor changes are useful and comforting
  - Some minor changes are necessary to complement and complete major changes
- Significant/major change is necessary
  - Our problems change, we change, so the language must change
  - Major change is unsettling
- Significant/major change requires decisions
  - It is a bet on what the future of software development will be
  - Understanding of new programming styles typically lags enabling features

C++: Success


#C++ users (approximate, with interpolation)
5000000
4500000
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
Feature drought; non-standard libraries
Devising support for techniques
Alternatives to C++ improving; "C++ is dead" claims
Absorbing improvements
Hoping for more
CPUs stop getting faster
Commercial release
STL
Absorbing improvements
Hoping for more
"Early C++" complete
Major new standard
integrates many new features

Language and Library: An apology

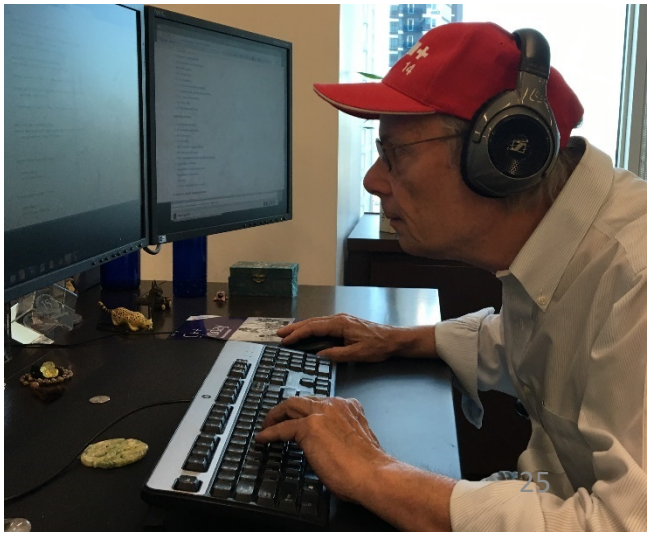
- We need great libraries!
- The standard library is now >50% of the standard
  - It will grow
  - It will become an increasingly large fraction of the standard
  - The standard library can enhance and simplify the language
  - The standard library should strive to meet the same stringent criteria as the language
- This talk is primarily about the language
  - Doing language and library is simply too much for one talk

Where do we go from here?

- Remember: “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
- Remember: “The best is the enemy of the good”
  - Don’t just dream
    - Support directed change
    - Take concrete, practical steps
    - Now!

Developer: What makes a good extension?

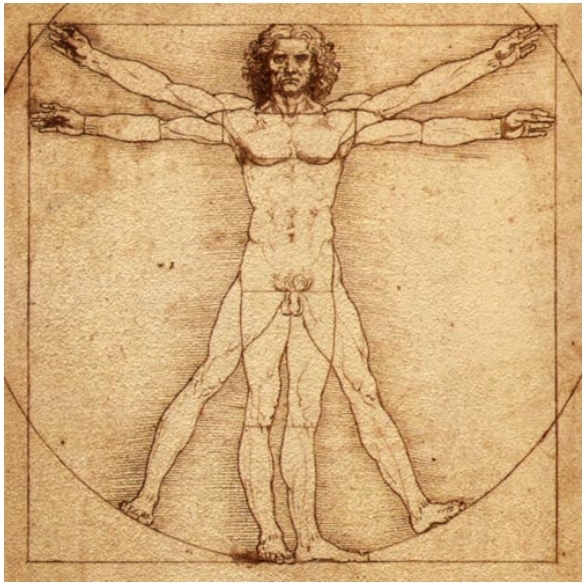
- "Help people like me with my next project"
  - Solve a specific problem
  - Don't surprise long-term and non-expert programmers
  - Isolate change
  - Cause no breakage
  - Available in my compiler "tomorrow"
- Developer experience can be very misleading
  - Lack of experience with long-term planning
  - Aims set by others
  - Short-term evaluation of consequences
  - Deliver on time
  - Focus on details
  - Risk adversity



Designer: What makes a good extension?

- "Significantly help the user community over the next decade++"
  - Address a general/fundamental concern
  - Change the way people think
  - Make the language more regular and easier to use
  - Be theoretically sound
  - Improve C++'s reputation
- Theory/literature can be very misleading
  - Focus on
    - novelty
    - Focus on "complicated problems"
    - Focus on advanced features
  - Disregard of existing code and existing programmers

developer experience ***and*** designer perspective



- Both are essential
  - Long-term ***and*** short-term view
  - Theory **and** practice
  - Ideal ***and*** experience
  - Balance
- The C++ community is huge and diverse
  - It is easy to be parochial
  - Consider the current C++ implementation and tools infrastructures
  - Consider transition
  - Consider teachability

Example: major vs. minor feature

- My definition
  - Major == changes the way people think about code
  - Note: Combinations of minor features can have major impact
- Literal separators (minor)

```
int price = 12'300'500.99;
```

- auto (borderline: minor/major)

```
auto x = 7.3;
template<class Iter>
void too_simple_unique(Iter p, Iter q) { auto x = *++p; if (x!=*p) *q++ = x; }
```

- Concepts (major)

```
void sort(Sortable& c);
vector<string> vs;
// fill vs
sort(vs);
```

Example: Isolated vs. pervasive impact

- Isolated: Simpler global variables (accepted for C++17)

  inline int x = f(); // *no duplication of x if included in multiple headers*

  - No new keyword, no grammar impact, no type system impact
  - Obviously, no impact on old code
  - Makes it trivial to add global variables in header-only libraries
- Pervasive: unified function call (rejected for C++17)

  f(x); // *try x.f() if f(x) is invalid*

  x.f(); // *try f(x) if x.f() is invalid*

  - No impact on old code
  - Eliminates the need for duplicate functions
    - e.g., **begin(x)** and **x.begin()**
  - Eliminates the need for many forwarding and adapter functions
    - E.g. **[&foo](auto&& bar) { return foobar(foo,bar); } //** did I avoid copying?
  - Simplifies library design and use

Every extension does some harm

- Implies implementation, tool, learning burden
- Outdates learning materials
  - Papers, books, and videos “live” for decades
- May have a poor benefit/cost ratio
  - “It helps me” is not a conclusive argument
- May help only a small part of the community
  - “but it doesn’t break old code” is not sufficient for acceptance
- May imply cost to people who do not use it
  - Delay more needed features
  - Distract from tool building and optimization
- May hinder efforts to improve programming practice
  - Reinforce/encourage bad habits
  - Be error prone

Sometimes we must break code

- 100% compatibility leads to stasis
  - A very high degree of compatibility is necessary and sufficient
- Break loudly
  - People can fix
  - People can decide not to upgrade
  - People can use compatibility switches

  - **static_assert(2<sizeof(int),"small ints");** // *new: use up-to-date compiler*
  - **int thread_local=7, *nullptr=NULL;** // *old: fix or use old compiler*

  - **auto x {1,2};** // *was* std::initializer_list<int>*, now ill-formed*
  - **auto y {1};** // *was std::initializer_list<int>, now int*

Quiet change, but very rare, and deliberate use of the rule is likely to break loudly

We need direction

- Principles
  - Consistency, coherence
  - Completeness at what the language and library do
  - Interoperability of features (language and library)
  - Zero overhead (but how to apply?)
  - …
- Process to help us follow those principles
  - Not easy to do with lots of individual decisions
  - Overall view/direction + plus care of details
  - Don’t take the first solution that looks OK
- Concrete examples
  - People twist the wording of principles and directions to mean anything

Creating and maintaining direction is difficult

- People want a smaller language
  - With many more features
- People want stability and 100% compatibility
  - With significant improvements
  - Except for the people who want “revolution”
- Different desirable features can be mutually incompatible
  - Or simply not usable in combination
- We can only do a relatively fixed amount of work in a given time
  - The more features we add, the harder it is to integrate them
  - WG21 has no money
- People disagree on basic philosophies
  - Implicit vs. explicit syntax
  - Simple vs. comprehensive
  - Aesthetics
  - Library vs. language

Design by committee

- C++ is a victim of its success
  - "everybody" wants to help
  - People come with a huge range of backgrounds, concerns, and ideas
  - Many have no knowledge of the past evolution of C++
  - Some people dwell on the problems and failures of the past
- A long series of separate decisions about individual language and library features by differing groups of people
  - Will not lead to a coherent language
  - Leads to individual features bloating
  - Major decisions get delayed and diluted
  - By now it is "design by committees" (really)
    - EWG, LEWG, SG1, …
    - 100+ people

Morgan Stanley

WG21

Example: Library or Language?

- We chose library
  - **complex**
  - **thread**
  - **string**
  - **pair** and **tuple**
  - **variant**, **optional**, and **any**
- We chose language
  - **new/delete**
  - Range-**for**
  - Coroutine
  - Concepts

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("the answer",42);          // C++11

– pair z {"the answer"s,42};                    // C++17
```

Direction (some ideas)

- Keep a running discussion about future directions
  - Keep a long list about distant aims (10 year horizon)
  - Keep a short list of next release aims (3 year horizon)
- Articulate aims
  - Zero-overhead
  - General-purpose
  - Static type safety
  - Minimal syntax
  - Increase regularity through small extensions (generalizations)
  - Make simple things simple
  - Improve hardware utilization (“direct map to hardware”)
- Integrate early
  - Move proposals to WP as soon as there is good agreement
  - Fix WP before the next standards release
  - Don’t stall foundational features in TSs

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

- Delaying a decision is a decision
  - Delays often imply increased complexity and decreased coherence
- Don’t confuse familiarity and simplicity
  - Such confusion hinders and delays major improvements

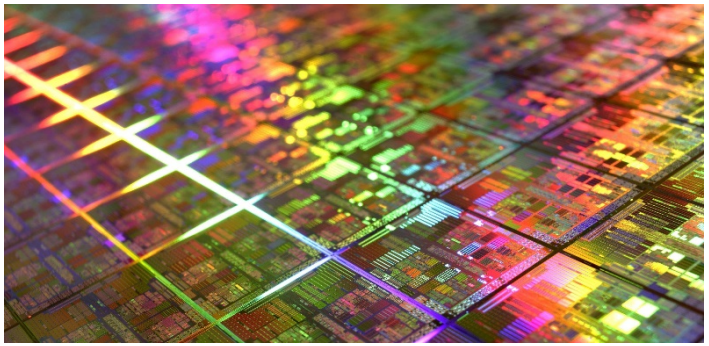
How to characterize C++?

- What is the language for?
  - IMO: A language for defining and using light-weight abstractions, primarily for software infrastructure and resource constrained applications
- Who is the language for?
  - IMO: serious programmers†
- What would make the language better?
  - Define "better"
  - Be specific
- No language can be everything to everybody
  - Direct map to hardware
  - Zero-overhead abstraction mechanisms
  - Primarily industrial
  - Reward good programmers
- We badly need a buzzword ☹
  - ZOAP? ☺

† TC+PL1

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)
- 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

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
- **[[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
  - …

But what do I think ***now***?

- For C++20 (and hopefully available long before 2020)
  - C++17
  - My C++17 list
  - Operator dot ("smart references")
  - Default comparisons (rejected by WG21)
- C++20 and beyond
  - Focus on foundations for tools and library building, explore
    - Package manager
    - IPR (representation of C++ code for analysis and tooling)
    - Static reflection
    - FP-style pattern matching
    - High-level concurrency support
- Not much more
  - Don't dilute the efforts

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

Example: Concepts (TS, GCC 6.1)

- Better specification
- Simpler generic code
- Better error messages

```
void sort(Sortable&);    // the Sortable concept defines what sort() requires

void algo(vector<int>& vi, list<int>& lsti)
{
        sort(vi);        // OK
        sort(lsti);      // error: listi is not Sortable
                         // list<int> is not RandomAccessible
                         // list<int>::iterator does not have [] and +
}
```

- Use concepts in design and comments
- Use concepts from the Ranges TS

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
    - 1.2MB with Clang 3.6.1
    - 1.1MB VC++ Dev14
  - And files are #included dozens or hundreds of times

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: Contracts (new proposal with wide support)

- a powerful, minimal facility for specifying contracts
  - **assert()** on steroids
  - Selectively enabled run-time checking

```
void push(queue & q)
        [[ expects: !q.full () ]]       // there must be room for another element
        [[ ensures: !q.empty() ]]       // q can't be empty after adding an element
{
        // ...
        [[ assert: q.is_ok() ]];        // q's invariant is (re)established at this point
}
```

Example: Contracts

- There are three "levels" of contracts:
  - Audit, default, axiom
  - Roughly: expensive compare to operation, cheap, no runtime impact

```
template<RandomAccessIterator Iter>
void bsearch(Iter p, Iter q)
        [[expects axiom: is_reachable(p,q)]]   // impossible to verify at run time
        [[expects: p<=q]]                      // cheap: assumes that q is reachable from p
        [[expects audit: is_ordered(p,q)]]     // expensive
{
        If (p==q) return;
        Iter mid;
        // …
        [[assert: *mid<=*(mid+1)]];     // cheap: when we pick a new mid
        // …
}
```

Example: structured bindings (C++17)

- 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';
```

Example: the sum is greater than the parts

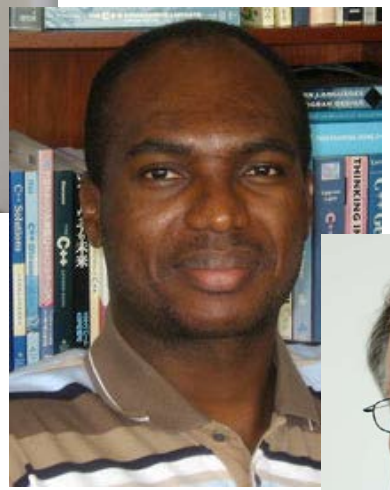
- But I can't test/use combinations of TS features
  - Modules (Microsoft), concepts (GCC), structured bindings (Clang)

```
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

Morgan Stanley

C++ Core Guidelines

- Guides us towards the features we are still waiting for
  - Prepare, practice, develop style
- You can write type- and resource-safe C++
  - No leaks
  - No memory corruption
  - No garbage collector
  - No limitation of expressiveness
  - No performance degradation
  - ISO C++
  - Tool enforced

Caveat: Not yet deployed at scale ☹

- Work in progress
  - C++ Core Guidelines: https://github.com/isocpp/CppCoreGuidelines
  - GSL: Guidelines Support Library: https://github.com/microsoft/gsl
  - Static analysis support tool: Work started: Visual Studio, Clang Tidy
  - "Help wanted" – MIT license

Core Guidelines Status

- The number and quality of rules have increased
  - 423 Rules, 542 examples, 313 pages, 144 contributors,
  - 700+ issues (most closed), Russian, Portuguese, and Korean translations
- Beginning industrial use
  - Subsetting and supersetting as expected
  - GSL is in use (Microsoft and Clang tools)
- Some use in teaching/academia
- Tool support is way behind what we expected back in 2015
  - Apologies (**** happens)
  - We are working on that
  - Would you like to build some tool?
- Expect more and better still
- Thanks to all who helped
  - see the GitHub

Type- and resource safety

- Low-level (static analysis support)
  - **owner<T>** alias
  - Dangling pointer elimination
- Higher level (library support)
  - Abstractions built using the lower-level facilities
    - **vector**, **unique_ptr**, **map**, **shared_ptr**, **graph**, …
  - Optional range checking (use the GSL)
    - **span**, **string_span, not_null**
- Rule support
  - E.g., “don’t cast” and “use **variant**”
- Application
  - Built of the abstractions
  - Verifiably safe

Core Guidelines

- In: Introduction
- P: Philosophy
- I: Interfaces
- F: Functions
- C: Classes and class hierarchies
- Enum: Enumerations
- R: Resource management
- ES: Expressions and statements
- Per: Performance
- CP: Concurrency
- E: Error handling
- Con: Constants and immutability
- T: Templates and generic programming
- CPL: C-style programming
- SF: Source files
- SL: The Standard library

Supporting sections:

- A: Architectural Ideas
- N: Non-Rules and myths
- RF: References
- Pro: Profiles
- GSL: Guideline support library
- NL: Naming and layout
- FAQ: Answers to frequently asked questions
- Appendix A: Libraries
- Appendix B: Modernizing code
- Appendix C: Discussion
- Glossary
- To-do: Unclassified proto-rules

I.13: Do not pass an array as a single pointer

Reason
(pointer, size)-style interfaces are error-prone. Also, a plain pointer (to array) must rely on some convention to allow the callee to determine the size.

Example
Consider:

```
void copy_n(const T* p, T* q, int n);    // copy from [p:p+n) to [q:q+n)
```

What if there are fewer than **n** elements in the array pointed to by **q**? Then, we overwrite some probably unrelated memory. What if there are fewer than **n** elements in the array pointed to by **p**? Then, we read some probably unrelated memory. Either is undefined behavior and a potentially very nasty bug.

Alternative
Consider using explicit spans:

```
void copy(span<const T> r, span<T> r2);    // copy r to r2
```

I.13: Do not pass an array as a single pointer (continued)

Example, bad

Consider:

```
void draw(Shape* p, int n); // poor interface; poor code
Circle arr[10];
// ...
draw(arr, 10);
```

Passing **10** as the **n** argument may be a mistake: the most common convention is to assume [**0**:**n**) but that is nowhere stated. Worse is that the call of **draw()** compiled at all: there was an implicit conversion from array to pointer (array decay) and then another implicit conversion from **Circle** to **Shape**. There is no way that **draw()** can safely iterate through that array: it has no way of knowing the size of the elements.

Alternative

Use a support class that ensures that the number of elements is correct and prevents dangerous implicit conversions.

I.13: Do not pass an array as a single pointer (continued)

Exception

Use **zstring** and **czstring** to represent a C-style, zero-terminated strings. But when doing so, use **string_span** from the GSL to prevent range errors.

Enforcement

- (Simple) ((Bounds)) Warn for any expression that would rely on implicit conversion of an array type to a pointer type. Allow exception for **zstring**/**czstring** pointer types.
- (Simple) ((Bounds)) Warn for any arithmetic operation on an expression of pointer type that results in a value of pointer type. Allow exception for **zstring**/**czstring** pointer types.

T.10: Specify concepts for all template arguments

Reason

Correctness and readability. The assumed meaning (syntax and semantics) of a template argument is fundamental to the interface of a template. A concept dramatically improves documentation and error handling for the template. Specifying concepts for template arguments is a powerful design tool.

Example

```
Template<typename Iter, typename Val>
//    requires Input_iterator<Iter>
//          && Equality_comparable<Value_type<Iter>,Val>
Iter find(Iter b, Iter e, Val v)
{
        // …
}
```

Note

"Concepts" are defined in an ISO Technical specification: concepts. A draft of a set of standard-library concepts can be found in another ISO TS: ranges Currently (July 2016), concepts are supported only in GCC 6.1. Consequently, we comment out …

ES.20: Always initialize an object

Reason

Avoid used-before-set errors and their associated undefined behavior. Avoid problems with comprehension of complex initialization. Simplify refactoring.

Example

```
void use(int arg)
{
        int i;      // bad: uninitialized variable
        // …
        i = 7;      // initialize I
}
```

No, **i=7** does not initialize **i**; it assigns to it. Also **i** can be read in the … part.

Note

The *always initialize* rule is deliberately stronger than the *an object must be set before used* language rule. The latter, more relaxed rule, catches the technical bugs, but:

- It leads to less readable code
- It encourages people to declare names in greater than necessary scopes
- It leads to harder to read code
- It leads to logic bugs by encouraging complex code
- It hampers refactoring

<<much more>>

Related Talks (actually, all the talks are related ☺)

- C++17 and beyond
  - Alisdair Meredith: *C++17 in breath*
  - Davidson, Guillemot, and Wong: *WG21-SG14 – making C++ better for games, embedded systems, and financial developers*
  - David Sankel: *variant<>: past, present, and future*
  - Bryce Lelbach: *The C++17 Parallel Algorithms Library and beyond*
  - Gor Nishanov: *C++ Coroutines: Under the cover*
  - Anthony Williams: *The continuing future of concurrency in C++*
  - Kerr and McNellis: *Putting coroutines to work with the windows runtime*
  - Manuel Klimek: *Deploying C++ modules to 100s of millions of lines of code*
  - Richard Smith: *There and back again: an incremental C++ modules design*
  - Gabriel Dos Reis*: C++ modules: The state of the union*
  - ???
- Guidelines
  - Neil MacIntosh: *The Guidelines support library: One year later*
  - Herb Sutter: *Leak-freedom in C++… By Default*
  - Niall Douglas: *Better Mutual Exclusion on the Filesystem using Boost.AFIO (GSL use)*

Summary

- C++ is successful on a large scale
  - Millions of users
  - Deep in our infrastructure
- C++'s success must be sustained
  - We can't rest on our laurels
- Focus on significant changes
  - Have an articulated direction
- Use the recent features (now)
  - Experiment before production use
  - The TSs are where the major improvements hide
- Use the Core guidelines (now)
  - As a guide to style
  - As a direction