CPP-Summit 2019

全球C++软件技术大会

C++ Development Technology Summit

Boolan
高端IT互联网教育平台

关注"博览Boolan"服务号
发现更多 会议·课程·活动

The Continuing Evolution of C++

Bjarne Stroustrup

Morgan Stanley, Columbia University

www.stroustrup.com

CPP-Summit 2019

Morgan Stanley

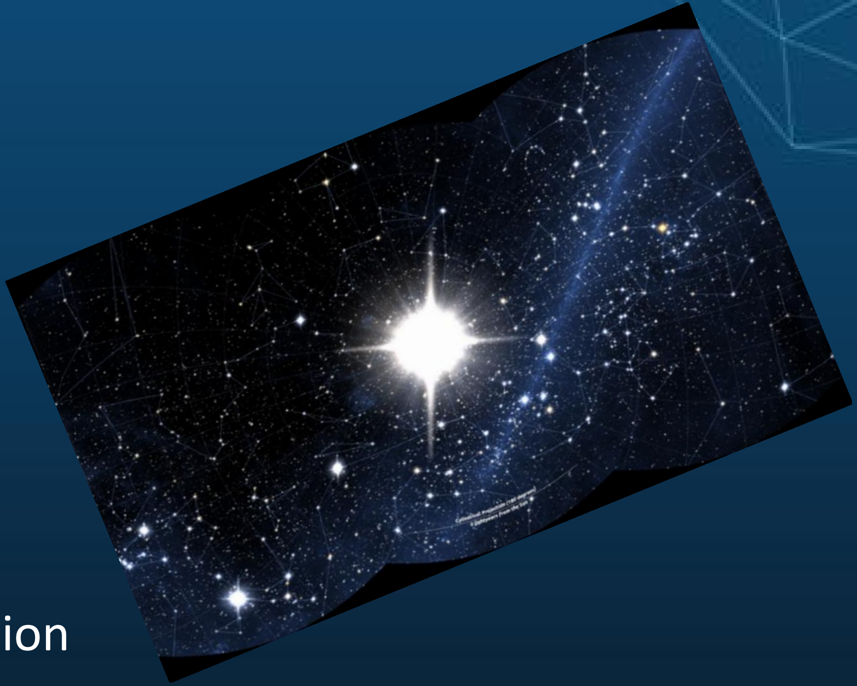
Abstract

- **The Continuing Evolution of C++**
- The development of C++ started in 1979. Since then, it has grown to be one of the most widely used programming languages ever, with an emphasis on demanding industrial uses. It was released commercially in 1985 and evolved through one informal standard (“the ARM”) and several ISO standards: C++98, C++11, C++14, C++17, and soon C++20. How could an underfinanced language without a corporate owner succeed like that? What are the key ideas and design principles? How did the original ideas survive almost 40 years of development and 30 years of attention from a 300+ member standards committee?
- What is the current state of C++ and what is likely to happen over the next few years? What are the problems we are trying to address through language evolution?
- 70 minutes

Morgan Stanley

Overview

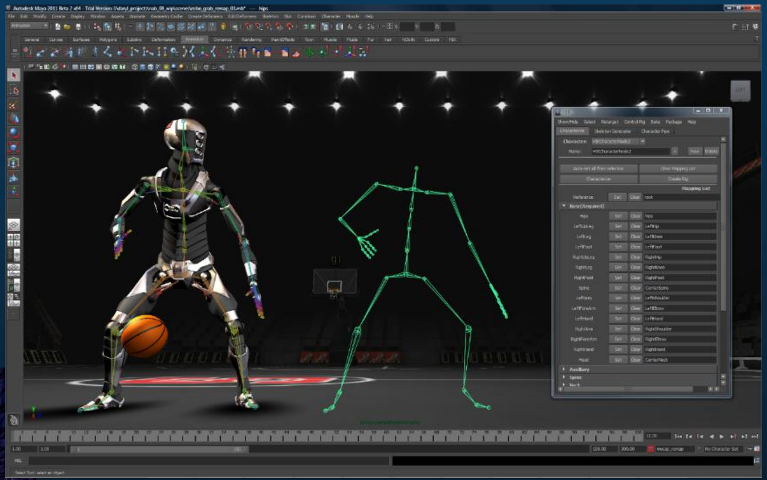
- C++'s role
  - Origins and Fundamentals
  - Example: Resource safety
- Standardization
  - C++11 … C++20 …
  - Example: Generic programming and Compile-time computation
- Design philosophy
  - Example: Concurrency and parallelism
- Guidelines for modern C++ programming
  - Example: Memory safety



Morgan Stanley

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

UNREAL ENGINE



C#

Java

AlphaGo

amazon

aMaDEUS

Google

CPP-Summit 2019

Morgan Stanley

C++ community

- About 4.5 million developers (surveys)
  - Seems to be growing by about 100,000 developers per year
  - Worldwide
    - North America, South America, Western Europe, Eastern Europe, Russia, China, India, Australia, …
  - Most industries
    - Finance, games, Web applications, Web infrastructure, data bases, telecommunications, aerospace, automotive, microelectronics, medical, movies, graphics, imaging, scientific, embedded systems, …

Morgan Stanley

2019 and 1979

C++ inside

Power and connectivity

Morgan Stanley

Then – early 1980s

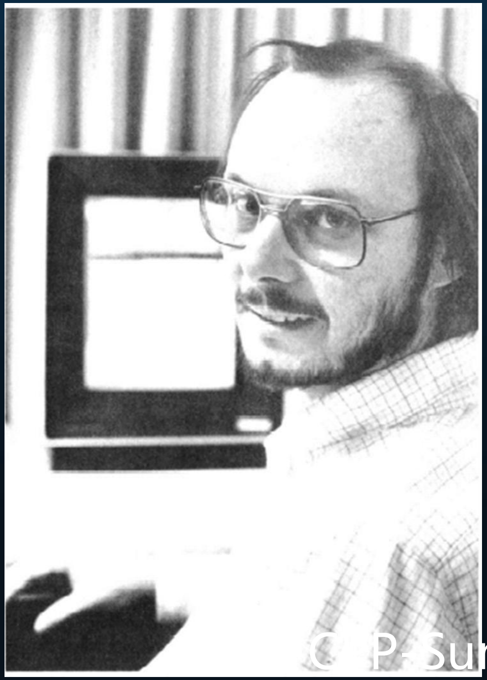
- Ken and Dennis had only just proved that semi-portable systems programming could be done (almost) without assembler
  - C didn't have function prototypes
  - Lint was state of the art static program analysis

- Most computers were <1MB and <1MHz
  - PDP11s were cool
  - VT100s were state of the art
  - A "personal computer" about $3000 (pre-inflation $$$)
  - The IBM PC was still in the future
- "Everybody" "knew" that "OO" was useless
  - too slow, too special-purpose, and too difficult for ordinary mortals

Morgan Stanley

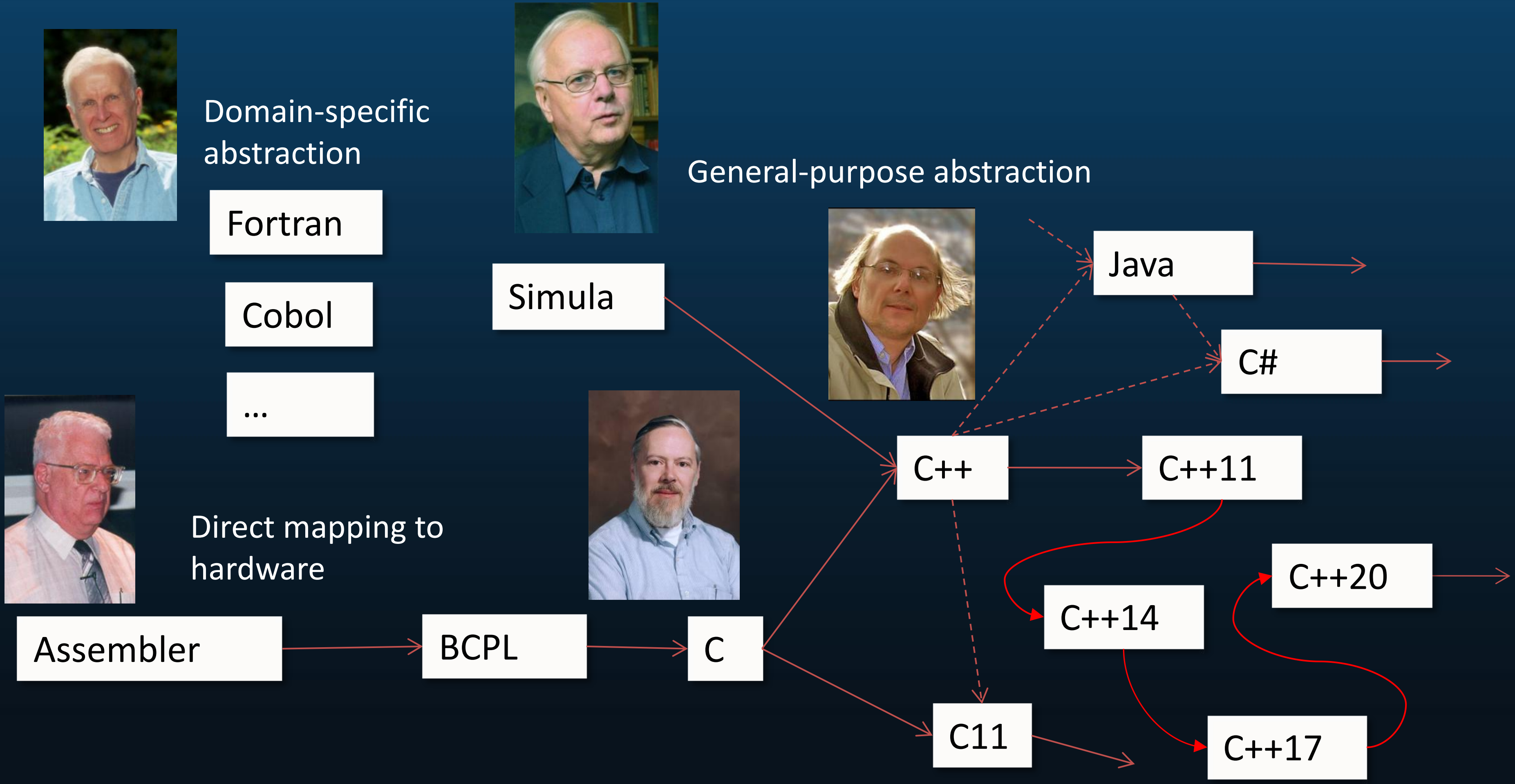
The origin of C++ – Bell Labs 1979

- I wanted to build Unix cluster and/or multiprocessor system
  - Close-to-optimal low-level programming and hardware manipulation
  - A way of describing "parts" and their communication
- No language could do both
  - Idea: C with Classes == C + Simula
  - Idea: Keep facilities general and flexible
    - don't just solve particular problems



Morgan Stanley

Programming Languages


Domain-specific abstraction
Fortran
Cobol
…
Simula
General-purpose abstraction
Java
C#
C++
C++11
C++20
C++14
C++17
C11
Direct mapping to hardware
Assembler
BCPL
C

Morgan Stanley

What matters?

- System development
  - Not just programming language
- Technical aspects
  - Stability and evolution
  - Tool chains
  - Teaching and learning
  - Technical community
  - Concise expression of ideas
  - Coherence
  - Completeness
  - Compact data structures
  - Lots of libraries
  - …

People

Being the best at one or two things isn't sufficient

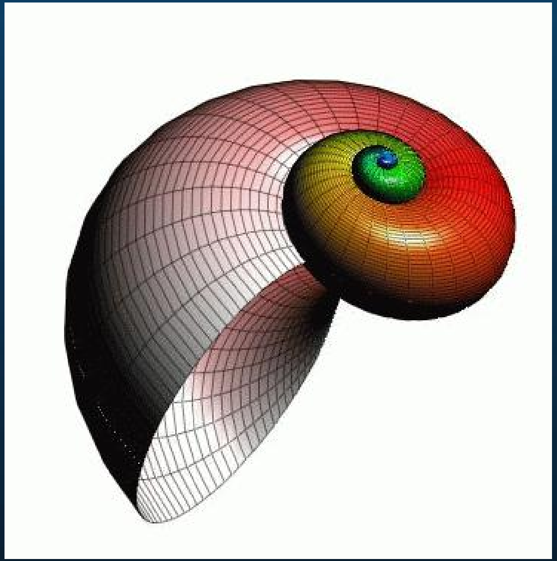
- A language must be good enough for everything in its domain
- Don't get obsessed by a detail or two

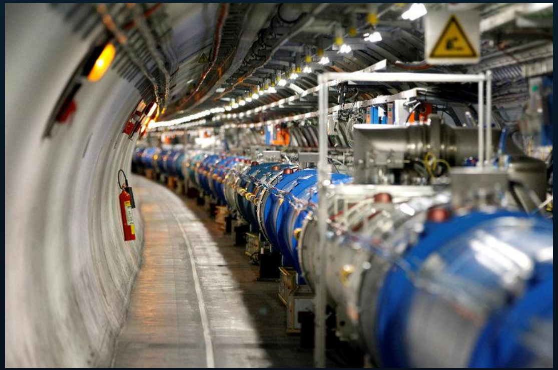
C++'s role

- A language for
  - 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

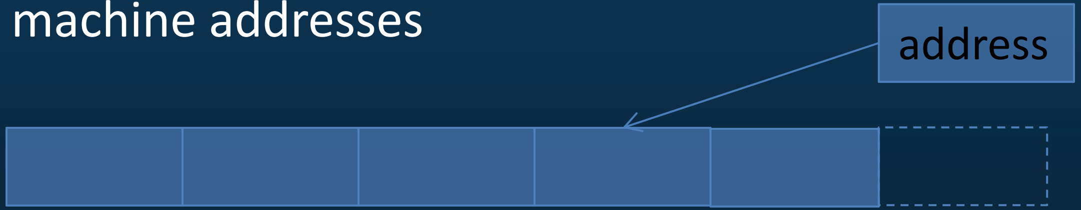


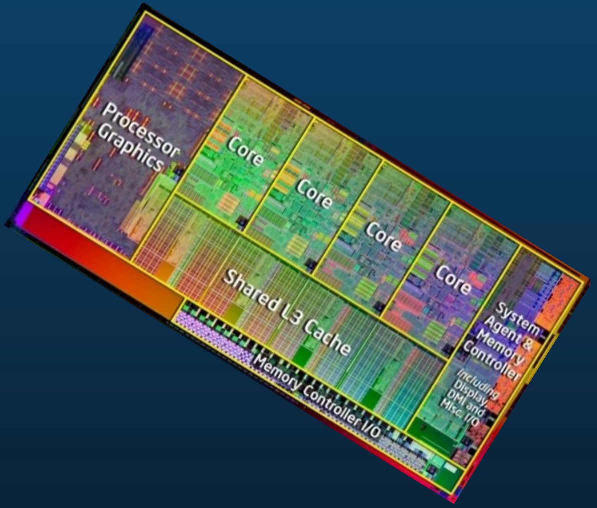


Morgan Stanley

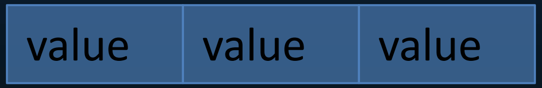
C/C++ machine model

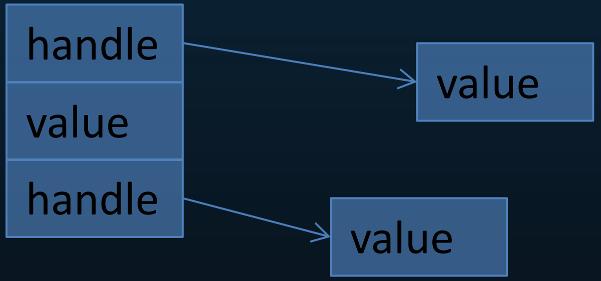
- Primitive operations maps to machine instructions
  - +, %, ->, [], (), …
- Memory is a set of sequence of objects
  - Pointers are machine addresses





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



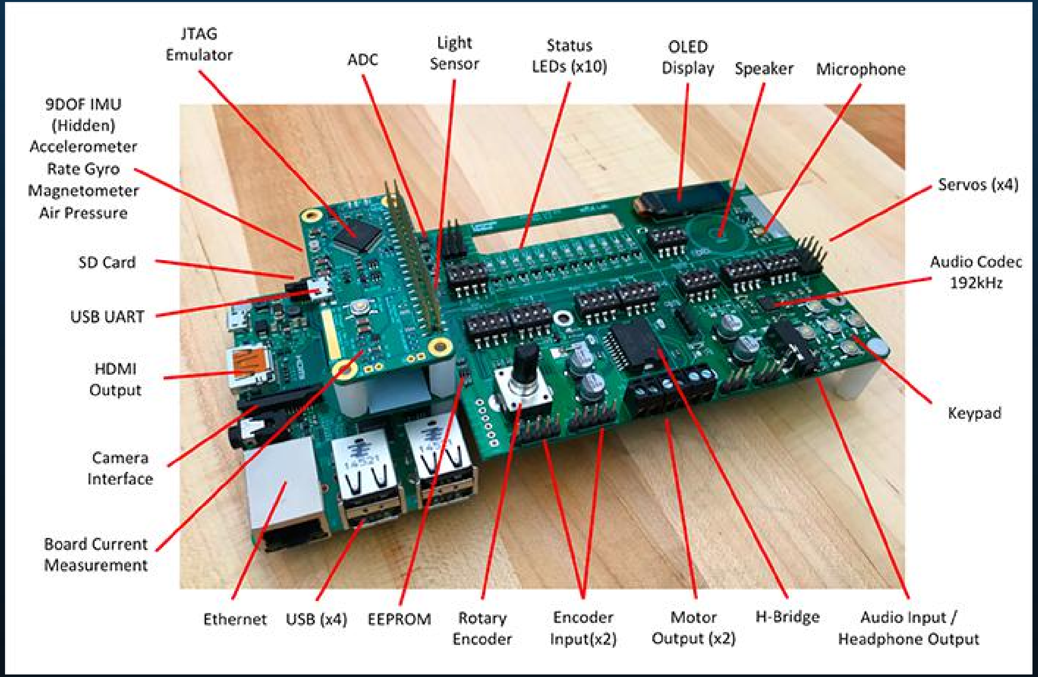


- The simplicity of this mapping is one key to C and C++'s success
  - It's a simple abstraction of hardware

Morgan Stanley

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

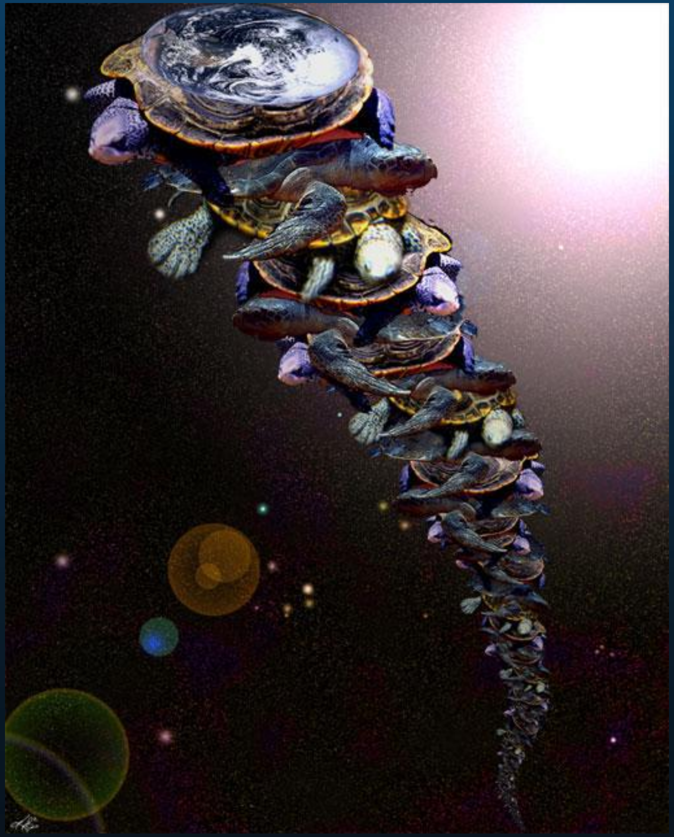


Morgan Stanley

It’s abstraction all the way down

- Abstraction
  - Can simplify understanding
  - can simplify use
  - Can simplify optimization
  - Even your hardware is an abstraction

- Always abstract from concrete examples
  - Maintaining performance
    - Or you get bloat
  - Keep simple things simple
    - Or you get only “expert only” facilities



Morgan Stanley

Case studies

- Resource Management
  - Classes: constructors, destructors, copy, and move
- Generic Programming
  - Templates, concepts, and compile-time computation
- Concurrency
  - Threads, locks, coroutines, and algorithms
- Memory safety
- All
  - Have their roots in the earliest C++
  - Are pervasive in modern C++
  - Are used to combine elegance and efficiency
  - Support direct expression of ideas

Resource management

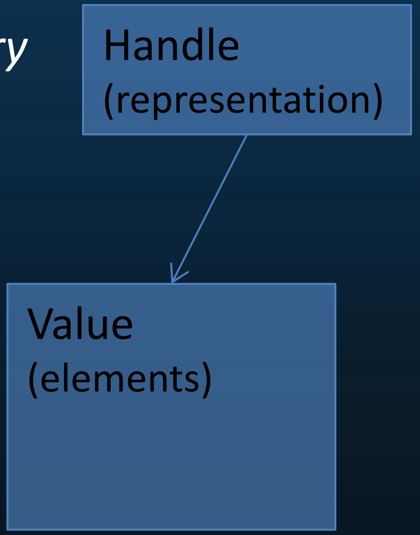
- A resource is something that must be acquired and released
    - Explicitly or implicitly
    - A resource leak eventually makes the computer fail
    - Excessive resource retention makes the computer slow
- Examples
    - Memory
    - Locks
    - file handles
    - Sockets
    - thread handles
- Avoid manual resource management
    - No leaks!
    - Minimal resource retention

Morgan Stanley

Classes, constructors, and destructors

```
template<Element T>
class Vector {       // vector of Elements of type T
public:
        Vector(initializer_list<T>);    // acquire memory for list elements and
initialize
        ~Vector();                      // destroy elements; release memory
        // …
private:
        T* elem;             // representation, e.g. pointer to elements plus #elements
        int sz;              // #elements
};

void fct()
{
        Vector <double> v {1, 1.618, 3.14, 2.99e8};     // vector of 4 doubles
        Vector<string> vs {"Strachey", "Richards", "Ritchie"};
        // …
} // memory and strings released here
```



Morgan Stanley

Resources and Pointers

- Many uses of pointers in local scope are not exception safe

```
void f(int n, int x)
{
        Gadget* p = new Gadget{n};   // look I’m a java programmer! ☺
        // …
        if (x<100) throw std::run_time_error{“Weird!”};        // leak
        if (x<200) return;                                     // leak
        // …
        delete p;                        // and I want my garbage collector! ☹
}
```

  – It leaks
    - Rule: No “Naked New”; no “naked pointers”!

Morgan Stanley

Resources and Pointers

- A **std::shared_ptr** implicitly releases its object at when the last **shared_ptr** to it is destroyed

```
void f(int n, int x)
{
        auto p = make_shared<Gadget>(n);   // manage that pointer!
                                            // return a shared_ptr<Gadget>
        // …
        if (x<100) throw std::run_time_error{“Weird!”}; // no leak
        if (x<200) return;                               // no leak
        // …
}
```

  - **shared_ptr** provides a form of garbage collection
    - General resources are correctly handled
  - I don’t want to create any garbage!

Morgan Stanley

Resources and Pointers

- But why use a pointer at all?
- If you can, just use a scoped variable
  - Often a handle

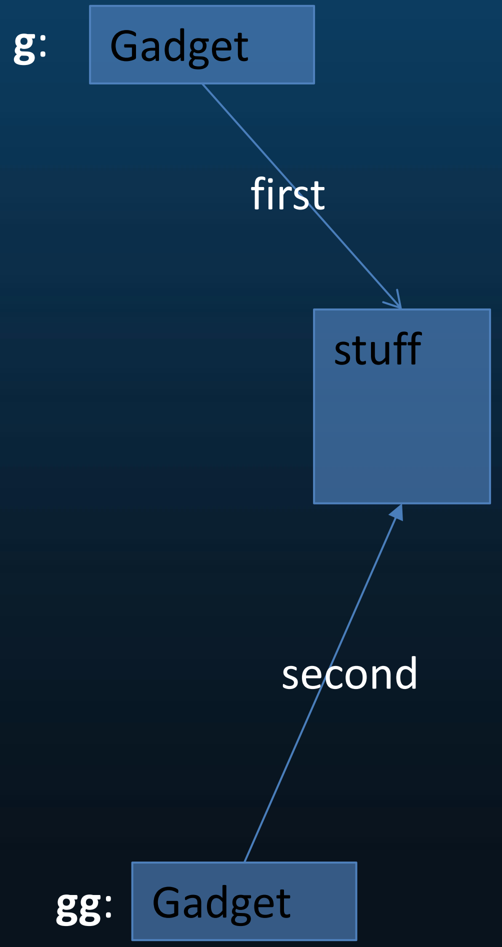
```
void f(int n, int x)
{
        Gadget g {n};
        // …
        if (x<100) throw std::run_time_error{“Weird!”};        // no leak
        if (x<200) return;                                     // no leak
        // …
}
```

Copy elision and Moves

- Control the complete object life cycle
  - Creation, copy, move, destruction

```
Gadget f(int n, int x)
{
        Gadget g {n};       // g may be huge
                            // g may contain non-copyable objects
        // …
        return g;           // no leak, no copy
                            // no pointers
                            // no explicit memory management
}

auto gg = f(1,2);           // move the Gadget out of f
```



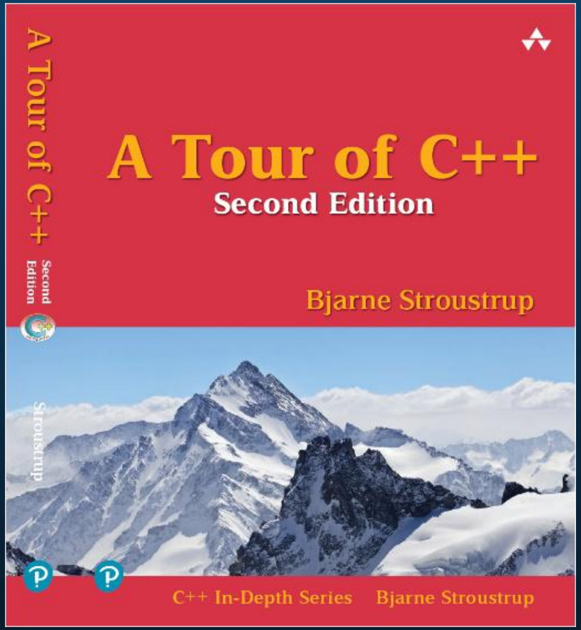
Morgan Stanley

C++

- }
  - “my favorite C++ feature” – *Roger Orr*

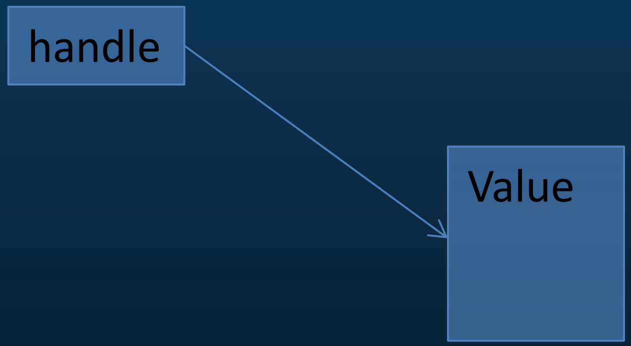
- The scope, guarded by destructors
  - Constructor/destructor pair is the real core of C++
  - Added to C together with
    - Classes
    - member functions
    - public/private
    - function declarations

    in the first month or so of the development of C++ (in 1979)
  - The key to
    - The standard library (containers, strings, files, threads, locks, …)
    - Error handling using exceptions (added in 1988)



Resources

- Make resource release implicit and guaranteed (RAII)
- All C++ standard-library containers manage their elements
  - **vector**
  - **list, forward_list** (singly-linked list), …
  - **map, unordered_map** (hash table),…
  - **set, multi_set, …**
  - **string**
- Some C++ standard-library classes manage non-memory resources
  - **thread, lock_guard, …**
  - **istream, fstream, …**
  - **unique_ptr, shared_ptr**
- A container can hold a non-memory resource
  - This all works recursively



GC is neither sufficient nor ideal

Morgan Stanley

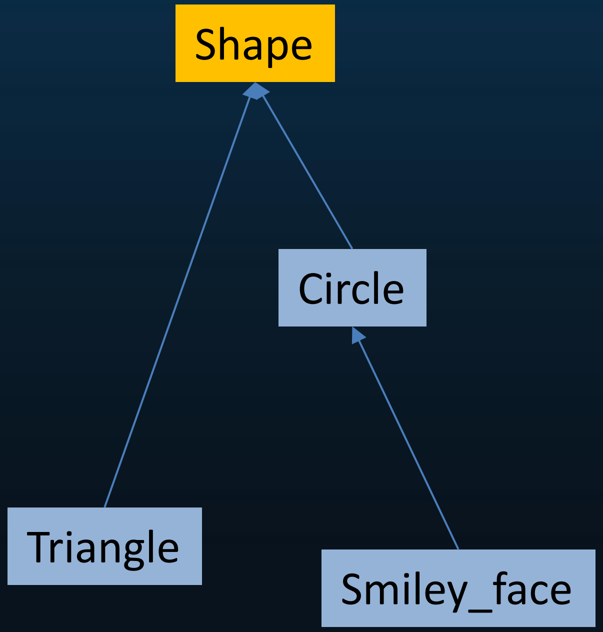
Object-oriented programming

- Sometimes, pointer-semantics is essential
  - You need pointers/references for run-time polymorphism

```
void draw_all(range auto& s)  // Ye good olde Shape example
        requires derived_from<Value_type<s>, Shape>
{
        for (auto& x : s) s->draw();
}

void user(Point p2, Point p3)
{
        vector<shared_ptr<Shape>> lst = {
            make_shared<Circle>(Point{0,0}, 42),
            make_shared<Triangle>(Point{20,200}, p2, p3),
            // …
        };
        // …
        draw_all(lst);
};
```

Use “smart” pointers to avoid leaks



C++ is not just an OOPL

- C++ is not *just* an OO language
  - For any definition of OO
  - It was never intended to be "just OO"
- Direct mapping to hardware
- Value semantics
  - **a = b** implies **a==b** and a modification of **a** doesn't affect **b**
- Separation of function and data
  - **sqrt(2)**
  - **z = a+b**
  - **sort(v)**
- Generic programming
  - generic types, algorithms, and actions
  - First macros (failed), then templates

Not revisionist history:
This was all documented
in 1981, 1991, …

Standardization

- Since 1989
- A necessary evil?
  - “You can’t have a major programming language controlled by a single company”
    - Actually, you can: Java, C#, …
- There are many kinds of standardization
  - ISO, ECMA, IEEE, W3C, …
- Long-term stability is a feature
  - You need a standards committee
- Vendor neutral
  - Important for some major users
  - Deprives C++ or development funds
- Dangers
  - Design by committee
  - Stagnation
  - Divergent directions of design
  - Overelaboration

"the committee"

1990

2011

2014

2017

Evolution: C++98 … C++20

- C++98 – a solid workhorse
  - Exceptions, templates, …
- C++11 – a major improvement
  - Libraries and language features
  - Concurrency, random numbers, regular expressions, …
  - Lambdas, generalized constant expressions, …
- C++14 – completes C++11
- Technical specifications
  - Concepts, modules, networking, …
- C++17 – adds many minor improvements
  - Currently shipping ☺
- C++20 – will be great
  - Concepts, modules, coroutines, …

Morgan Stanley

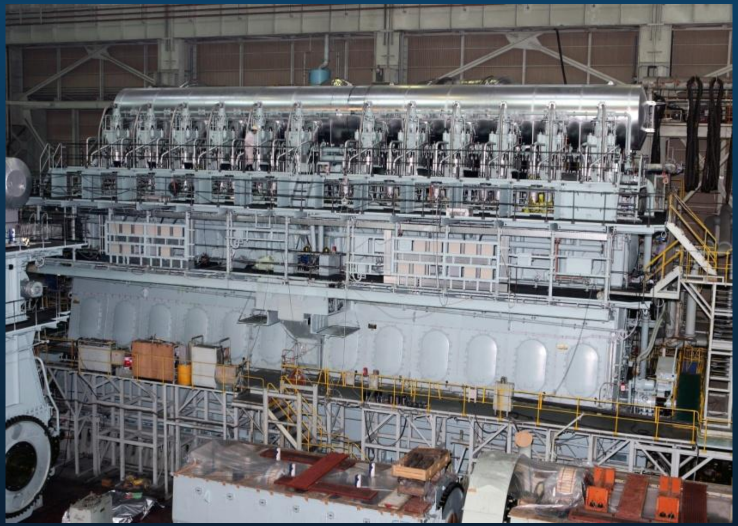
“C++11 feels like a new language”

```
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;   // empty vector of pointers to elements
        for (auto& x : c)
            if (x==v)
                res.push_back(&x);
        return res;                    // potentially huge vector
}

    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";
```

Evolution and stability

- Stability/compatibility is a feature
- Everybody want
  - A smaller, simpler language
  - With just two more features
  - And full backwards compatibility
    - (with many different languages and systems)
  - And interoperability with other languages and systems
- Every successful language becomes “legacy”



Morgan Stanley

The onion principle

- Layers of abstraction
  - The more layers you peel off, the more you cry
- Management of complexity

Morgan Stanley

Make simple things simple!

- Always remember the zero-overhead principle
- Don’t make complex things unmanageable or unaffordable

- Example:

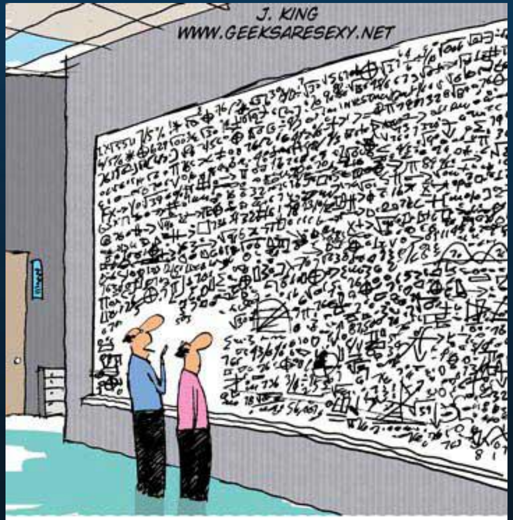
```
int i;
for (i=0; i<max; ++i) f(v[i]);       // C-style loop

for (int i=0; i<max; ++i) f(v[i]);   // C++98: C-style loop

for (int& x : v) f(x);               // C++11: range-for

for_each(begin(v),end(v),f);         // C++11: algorithms

for_each(par_unseq,v,f);             // C++20: parallel algorithm and ranges
```



“...And that, in simple terms, is what’s wrong with your software design.”

Morgan Stanley

Type-rich programming at compile time

- You can't have a race condition on a constant
- Don't need run-time error handling
- Hand-calculating values is error-prone
- macros and template metaprogramming is 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 );     // at compile time
```

Generic programming

- Code that works for all types that meet abstract requirements
  - E.g., is a forward iterator, is integral, is regular, can be sorted
- Requirements defined as concepts
  - A concept is a compile-time predicate on a set of types and values

```
template<typename R>
concept Sortable_range =
    random_access_range<R>               // has begin()/end(), ++, [], +, …
    && permutable<iterator_t<R>>         // has swap(), etc.
    && indirect_strict_weak_order<R>;    // has <, etc.
```

  - Use

```
void sort(Sortable_range auto&);
sort(vec);          // OK: sort a vector with ordered elements
sort(lst);          // error: trying to sort a list with ordered elements
```

Morgan Stanley

Generic Programming: Templates

- 1980: Use macros to express generic types and functions
- 1987 (and current) aims:
  - Extremely general/flexible
    - “must be able to do much more than I can imagine”
  - Zero-overhead
    - vector/Matrix/… to compete with C arrays
  - Well-specified interfaces
    - Implying overloading, good error messages, and maybe separate compilation
- “two out of three ain’t bad”
  - But it isn’t really good either ☹
  - It has kept me concerned/working for 20+ years
  - Concepts! (now available)

Morgan Stanley

Generic Programming

- Problem:
  - C++98 templates offer compile-time duck typing
    - Encourages write-only, complex, unmaintainable code
    - Delivers appalling compiler error messages
  - Templates are useful and wildly popular
    - Flexible
    - Unsurpassed performance
    - Metaprogramming
    - Compile-time computation
- Solution
  - Generalized constant expressions: **constexpr** (C++11, …)
  - Precisely specified flexible interfaces: **concept**s (C++20)

Morgan Stanley

Generic programming

- Selection based on abstract requirements

```
void sort(Sortable_range auto& container);    // container must be sortable

template<typename R>
concept Forward_sortable_range =
        forward_range<R>
        && sortable<iterator_t<R>>;

void sort(Forward_sortable_range auto& seq);  // random access not required

sort(vec);      // OK: use sort of Sortable_range
sort(lst);      // OK: use sort of Forward_sortable_range
```

Flexibility
composability

- We don’t have to say
  - “**Forward_sortable_range** is less strict than **Sortable_range**”
  - we compute that from their definitions

Morgan Stanley

Generic programming

- GP is “just programming”
  - A concept specifies an interface
  - A type specifies and interface plus a layout
  - In principle, there is little difference between **sort(v)** and **sqrt(x)**
  - “as close to ordinary programming, but not closer”
- By default **sort()** uses **<** for comparison
  - We can specify our own comparison

```
template<random_access_range R, class Cmp = less>
        requires sortable_range<R, Cmp>
constexpr void sort(R&& r, Cmp cmp = {});

sort(v, [](const auto& x, const auto& y) { return x>y; });
sort(vs, [](const auto& x, const auto& y) { return lower_case_less(x,y); });
```

Concepts (C++20)

- A concept is a compile-time predicate

```
On a set of types and values
template<typename T>
concept sortable =
        range<T>
        && permutable<T>                        // T has [], +, etc.
        && indirect_strict_weak_order<T>;       // Value_type<T> has <, etc.

template<typename T>
concept range = requires(T a) {           // simplified
        typename T::Iterator;       // must have an iterator type
        { begin(a) } -> Iterator;   // has a beginning
        { end(a) } -> Iterator;     // has an end
        input_iterator<T::Iterator>;  // the iterator must be an input_iterator
}
```

Morgan Stanley

Where do we go from here?

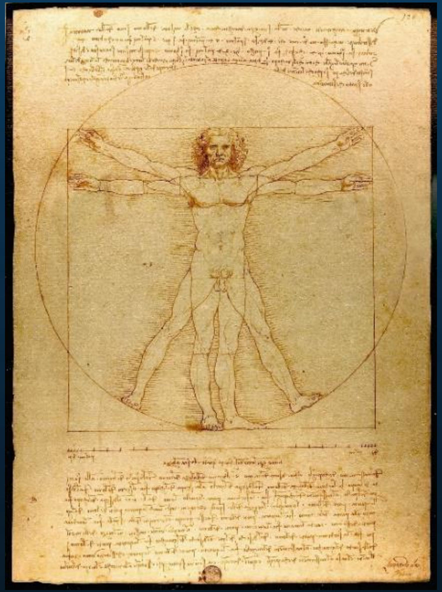
- “Dream no little dreams”
  - My aims include
    - Complete type- and resource safety
    - As fast or faster than anything else
    - Good on “modern hardware”
    - Significantly faster compilation catching many more errors
- “The best is the enemy of the good”
  - Don’t just dream
    - Support directed change
    - Take concrete, practical steps
    - Now!

Make C++ a much better tool
for building demanding applications

Morgan Stanley

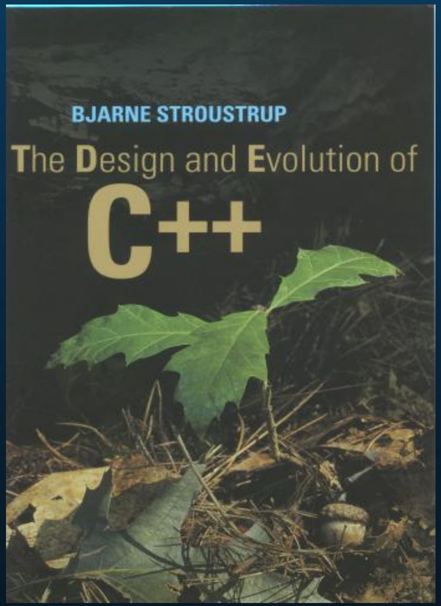
Why “philosophy”?

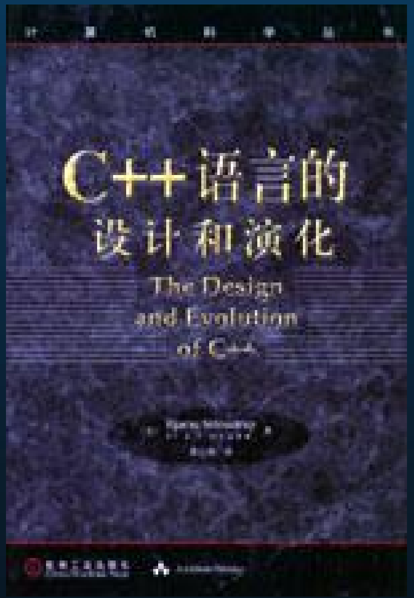
- To maintain a direction towards a coherent language
  - Extremely hard in a large committee
- A language is more than a collection of features
  - Even if every feature is good, a language can be a mess
  - Every feature carries a cost
    - Learning, teaching, size of books, …
    - Specification (incl. bug fixing and maintenance)
    - Implementation
    - Dealing with feature interactions
- Without philosophy language design becomes hacking
- The design rules are mostly non-technical
  - For C++ (until now), see “The Design and Evolution of C++”



D&E General Rules (1994 and before)

- C++'s evolution must be driven by real problems
- Don't get involved in a sterile quest for perfection
- C++ must be useful *now*
- Every feature must have a reasonably obvious implementation
- Always provide a transition path
- C++ is a language, not a complete system
- Provide comprehensive support for each supported style
- It is more important to enable good programming than to prevent errors
- Don't try to force people





Read D&E for an explanation/interpretation of these rules

Morgan Stanley

D&E Language-technical rules

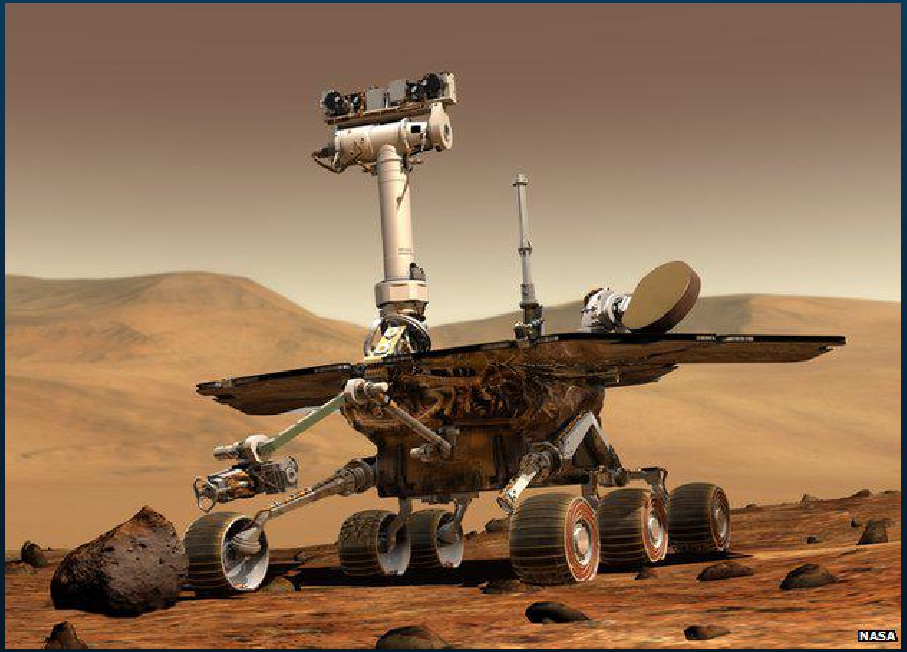
- No implicit violations of the static type system
- Provide as good support for user-defined types as for built-in types
- Locality is good
- Avoid order dependencies
- If in doubt, pick the variant of a feature that is easiest to teach
- Syntax matters (often in perverse ways)
- Preprocessor usage should be eliminated
- Zero-overhead principle
  - "What you don't use you don't pay for"
  - "You can't write it better by hand"

Note: 1994 and before

Morgan Stanley

Concurrency and parallelism

- What's in the standard is foundational
  - Use one of the many libraries
- "Raw" concurrency
  - Mostly for systems work
- Lock-free programming
- Multi-threading
  - Threads and locks
- Vectorization
- Coroutines
- No *standard* distributed computation



Concurrency and parallelism

- **jthread**: Joining thread (RAII)

```
void user()
{
        jthread t1 { my_task1 };
        jthread t2 { my_task2 };
        // …
} // jthreads implicitly join here
```

Concurrency and parallelism

- What if you decide that the result of a thread isn't needed?
  - E.g., **find_any()** after some thread found "it"

```
auto my_task = [] (stop_token tok)
      {
         while (!tok.stop_requested()) {  // is a result still needed?
            // … do work ...
         }
      };

void user()
{
      jthread t1 { my_task }; // stop_token implicitly supplied by jthread
      jthread t2 { my_task };
      // …
      if (t1_no_longer_needed) t1.request_stop();
      // …
}
```

Concurrency and parallelism

- "Double-locked initialization" using atomics

```
mutex mx;               // expensive OS supported synchronization
atomic<bool> initx;     // relatively cheap atomic variable
int x;                  // shared variable
if (!initx) {
        lock_guard lck {mx};
        if (!initx) x = 42;
        initx= true;
}
// … use x …
```

- No data race

Concurrency and parallelism

- Simple locking (RAII)

```
mutex m1;
int sh1;          // shared data

mutex m2;
int sh2;          // some other shared data

void obvious()
{
        // …
        scoped_lock lck1 {m1,m2}; // acquire both locks
        // manipulate shared data:
        sh1+=sh2;
} // release both locks
```

Parallel algorithms

- Don’t fiddle with threads and locks if you don’t have to

```
sort(v);
sort(unseq,v);          // try to vectorize
sort(par,v);            // try to parallelize
sort(par_unseq,v);      // try to vectorize and parallelize

void scale(vector<double>& v, int s)
{
        // …
        for_each(unseq, v, [s](integral auto& x) { x *= s; });
        // …
}
```

Morgan Stanley

Coroutines

```
generator<int> fibonacci()     // generate 0, 1, 1 ,2, 3, 5, 8, 13, 21 …
{
        int a = 0; // initial values
        int b = 1;
        while (true) {
                int next = a+b;
                co_yield a;             // return next Fibonacci number
                a = b;                  // update values
                b = next;
        }
}

int main()
{
        for (auto v: fibonacci())
                cout << v << '\n';
}
```

Fast pipelines and generators
Simple asynchronous programming

Morgan Stanley

What would you like your code to look like in 5 years?

- "Much like today"
  - A very poor answer
- Attack problems with a "cocktail" of approaches
  - Language design
    - To express things more directly
  - Coding rules
    - General and domain specific, modern C++
  - Library
    - Enhance expressiveness
  - Static analysis
    - Enforcement
- Targets
  - Type and resource safety
  - Known over-complex techniques
  - Known bug sources

C++ Core Guidelines

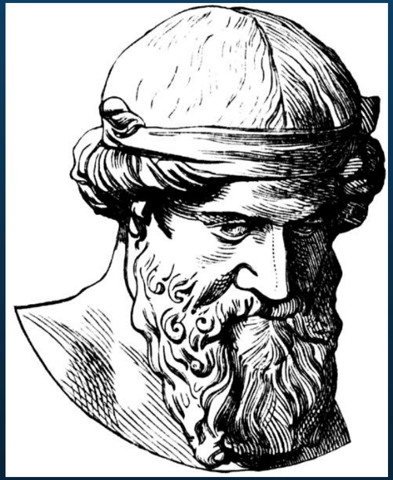
- You can write type- and resource-safe C++
  - No leaks
  - No memory corruption
  - No garbage collector
  - No limitation of expressibility
  - No performance degradation
  - ISO C++
  - Tool enforced (eventually)
- Work in progress
  - C++ Core Guidelines: https://github.com/isocpp/CppCoreGuidelines
  - GSL: Guidelines Support Library: https://github.com/microsoft/gsl
  - Static analysis support tools (work in progress)

Morgan Stanley

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*



Morgan Stanley

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*

Morgan Stanley

Dangling pointers – my most dreaded bug

- One nasty variant of the problem

```
void f(X* p)
{
        // …
        delete p;  // looks innocent enough
}

X* q = new X;    // looks innocent enough
f(q);
// … do a lot of work here …
q->use();        // Ouch! Read/scramble random memory
```

- Solution:
  - Never let a pointer outlive the object it points to
  - Easy to say, hard to do at scale

Morgan Stanley

Other nasty problems

- Other ways of breaking the type system
  - **unions**: use **std::variant**
  - Casts: don't use them
  - …
- Other ways of misusing pointers
  - **nullptr** dereferencing: use **not_null<T>**
  - Range errors: use algorithms and **span<T>**
- …
- "Just test everywhere at run time" is ***not*** an acceptable answer
  - In particular, "smart pointers" is ***not*** a complete solution
  - We want comprehensive guidelines
  - Statically enforced

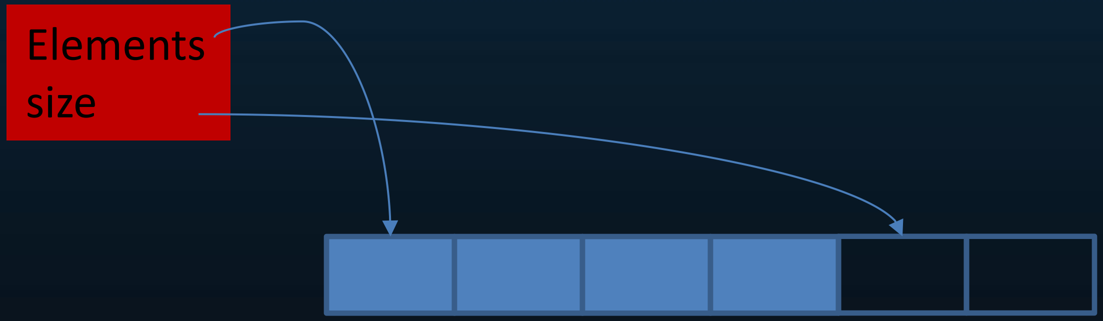
Morgan Stanley

Span

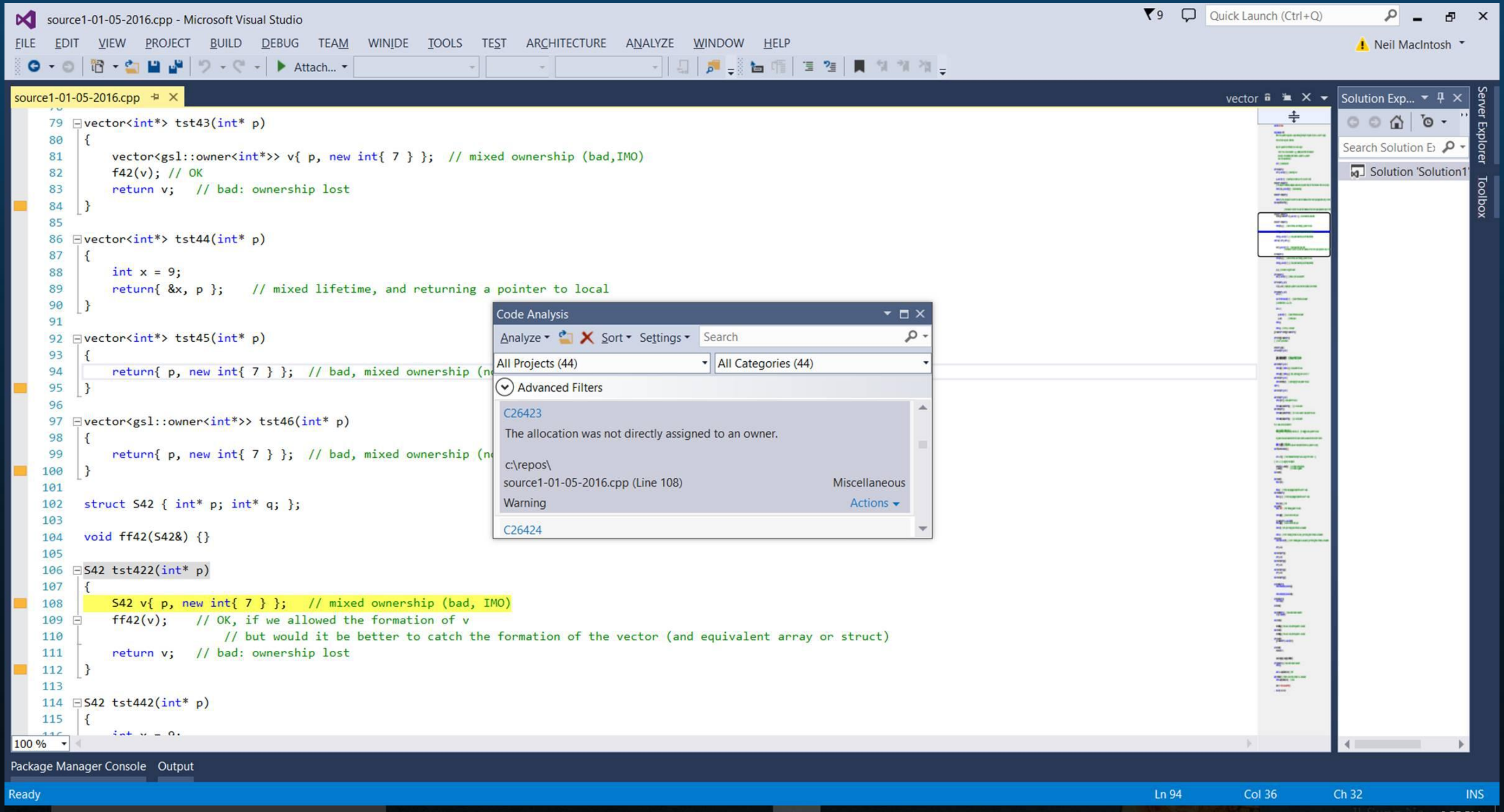
- In the standard library
- Non-owning potentially run-time checked reference to a continuous sequence

```
int a[100];
span s {a};     // note: template argument deduction
for (auto x : s) cout << x << '\n';
```

- From the GSL
- On GitHub



Static analyzer (currently integrated)



```
 79  vector<int*> tst43(int* p)
 80  {
 81      vector<gsl::owner<int*>> v{ p, new int{ 7 } };  // mixed ownership (bad,IMO)
 82      f42(v); // OK
 83      return v;   // bad: ownership lost
 84  }
 85
 86  vector<int*> tst44(int* p)
 87  {
 88      int x = 9;
 89      return{ &x, p };    // mixed lifetime, and returning a pointer to local
 90  }
 91
 92  vector<int*> tst45(int* p)
 93  {
 94      return{ p, new int{ 7 } };  // bad, mixed ownership (n
 95  }
 96
 97  vector<gsl::owner<int*>> tst46(int* p)
 98  {
 99      return{ p, new int{ 7 } };  // bad, mixed ownership (n
100  }
101
102  struct S42 { int* p; int* q; };
103
104  void ff42(S42&) {}
105
106  S42 tst422(int* p)
107  {
108      S42 v{ p, new int{ 7 } };   // mixed ownership (bad, IMO)
109      ff42(v);    // OK, if we allowed the formation of v
110                  // but would it be better to catch the formation of the vector (and equivalent array or struct)
111      return v;   // bad: ownership lost
112  }
113
114  S42 tst442(int* p)
115  {
```

C26423
The allocation was not directly assigned to an owner.

c:\repos\
source1-01-05-2016.cpp (Line 108) Miscellaneous
Warning Actions

C26424

Morgan Stanley

Challenges

- The world is changing
- We can do much better
  - We **must** do much better
  - Our civilization depends on software
- Must be done at industrial scale
  - Billions of lines of code
  - Several million developers
  - We can’t do it all at once
- Stability is a key feature
- Maintaining a coherent direction is extremely hard

Morgan Stanley

Summary

- C++ is true to its principles
  - Direct hardware access
  - Zero-overhead abstraction
  - Static typing
  - More expressive than ever
- Major progress -> C++11 -> C++14 -> C++17 -> C++20
- GCC, Clang, Microsoft, … are currently shipping
      - free downloads
  - You can now say most things simpler and more directly than in C++98
  - The code runs faster
- Use C++ better
  - Core guidelines

C++11: 10+ years of experience added

- Concurrency support
  - Memory model
  - Atomics and lock-free programming
  - Threads, mutex, condition_variable, futures, …
- Move semantics
- Generalized constant expression evaluation (incl. **constexpr**)
- Lambdas
- **auto**
- Range-**for**
- **override** and **final**
- **=delete** and **=default**
- Uniform initialization (**{}** initializers)
- User-defined literals
- **nullptr**
- **static_assert**
- Variadic templates
- Raw string literals (**R"(…)"**)
- **long long**
- Member initializers
- …

- **shared_ptr** and **unique_ptr**
- Hash tables
- **<random>**
- **<chrono>**
- …

Morgan Stanley

C++14: “Completes C++11”

• Function return type deduction	e.g. **auto f(T& x) { return g(x); }**
• Relaxed **constexpr** restrictions	e.g. **for** in **constexpr** functions
• Variable templates	
• Binary literals	e.g., **0b0101000011111010**
• Digit separators	e.g., **0b0101'0000'1111'1010**
• Generic lambdas	e.g., **[](auto x) { return x+x; }**
• Lambda capture expressions	
• **[[deprecated]]**	
• …	
• Shared mutexes	
• User-defined literals for time	
• Type-based tuple addressing	e.g., **get<int>(t)**
• …	

Stability/compatibility is a feature

C++17: “a little bit for everyone”

- 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();**
- 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]]**
- Fold expressions for parameter packs — E.g., **auto sum = (args + …);**
- …
- File system library
- Parallelism library — E.g. **sort(par_unseq,v.begin(),v.end())**
- Special math functions — E.g., **riemann_zeta()**
- **variant**, **optional**, **any**, **string_view**
- …

Morgan Stanley

C++20: A “major release”

- Something that changes the way we think about programming
  - Make simple things simple!
  - Don’t make complicated things unnecessarily complicated or expensive
- What will we get?
  - Concepts (shipping: GCC, Microsoft, and Clang)
  - Modules (shipping: Microsoft; work in Clang and GCC)
  - Coroutines (shipping: Microsoft and Clang, soon GCC)
  - Better parallel algorithms
  - Joining thread and stop tokens
  - Ranges (using concepts)
  - Span (safe, efficient access to contiguous sequences)
  - Format (type-safe printf-style formatting)
  - Source_location, math constants, bit operations, …

Morgan Stanley

Ranges library

- Think "STL 2.0 using concepts"
- Simplify use

```
vector<string> v;
// …
sort(v);
```

- Infinite sequences and pipes

```
std::vector<int> v(42);
std::span foo = v | view::take(3);
```

- And much more
- On GitHub

Eric Niebler

Casey Carter

Morgan Stanley

Modules

Morgan Stanley

Modules history

- 1994: D&E: “Furthermore, I am of the opinion that Cpp must be destroyed.”
  - CPP == C preprocessor, not C++ ☺
  - Mentions **include** (no **#**) as the obvious alternative to **#include** with “module semantics”
- 2005-2012: Daveed Vandevoorde (EDG):
- 2011: Doug Gregor (Apple), C/Objective-C approach presentations
- Stroustrup and Dos Reis: IPR 2007
- Dos Reis (Microsoft): 2014
- Richard Smith (Google): 2016 (based on the C/Objective-C approach, 2017 Atom
- Gaby & Richard: 2017: Joint proposal
- Nathan Sidwell (Facebook) GCC enters the fray
- Gaby, Richard, Nathan, and Daveed: Feb 2019: Get it done now!
- Kona, Feb 2019: Voted in!!!

Morgan Stanley

Modularity and transition

```
import A;
import B;
```

Is the same as

```
import B;
import A;
```

Import is not transitive

```
module;
#include "xx.h"    // to global module
export module C;
import "a.h"                    // "modular headers"
import "b.h"
import A;
export int f() { … }
```

Module partitions

Morgan Stanley

Modules

- Better code hygiene: modularity (especially protection from macros)
- Faster compile times (hopefully factors rather than percent)

```
 export module map_printer;                // we are defining a module

 import iostream;
 import containers;
 using namespace std;

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