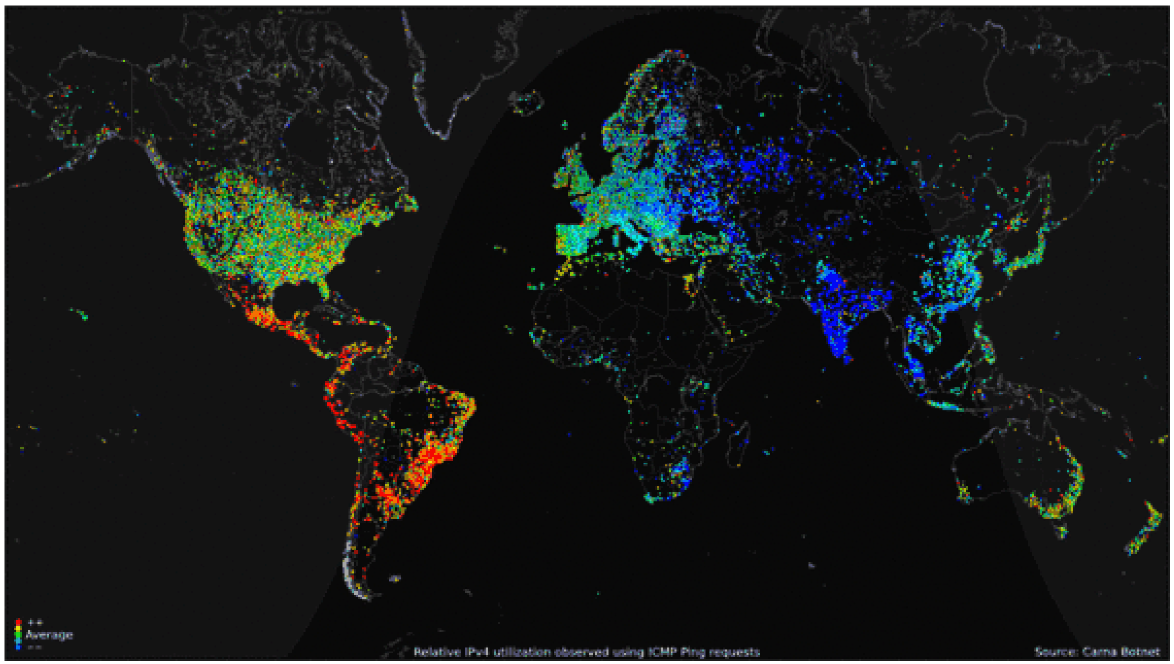
C++20: C++ at 40

stability and evolution



Bjarne Stroustrup

Morgan Stanley, Columbia University

www.stroustrup.com

Morgan Stanley

2019 and 1979

C++ inside

Power and connectivity

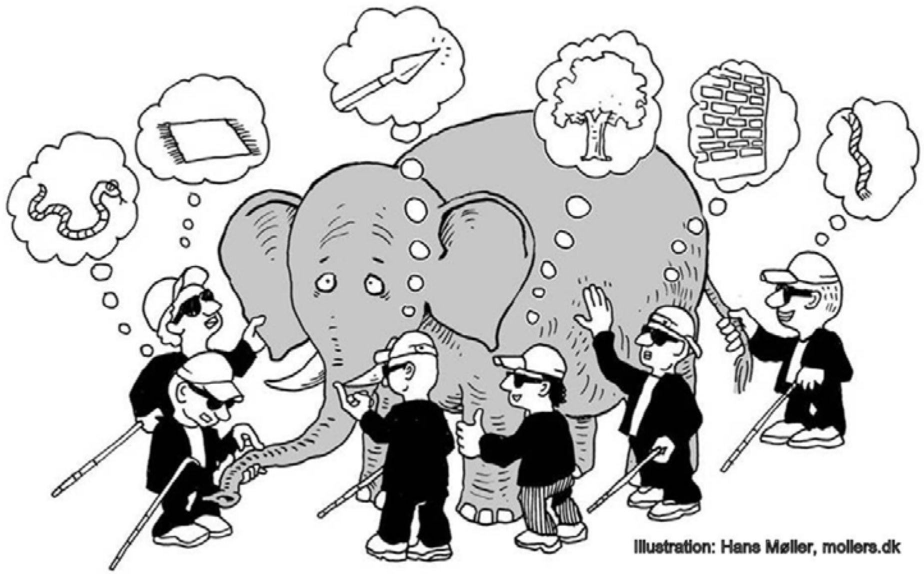
Then – early 1980s

- 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

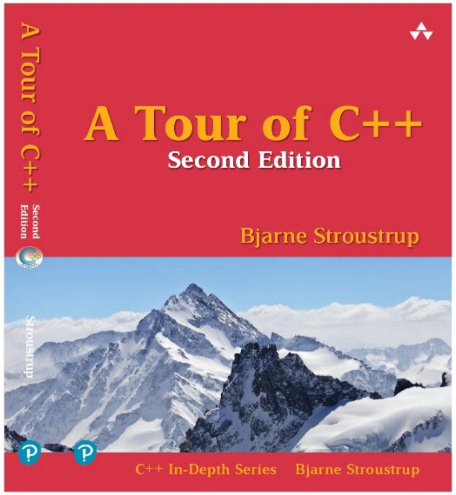
Morgan Stanley

Present and use C++ as a modern language



- This is a talk about C++ as it is today (C++20)
  - new != good != old and new != bad != old
  - I am not labeling examples as C++98, C++14, C++20, etc.
  - I will, occasionally, give a historical perspective; we have come a long way
- This is not a talk about "details"
  - Every technical point mentioned here has a one-hour talk this week

General approach – a recommendation



- Using C++
  - focus on the essentials
  - use “advanced features” only when necessary
- Teaching C++
  - focus on the essentials
  - don't hide the key features and techniques in a mess of information
  - Tell the truth
    - only the truth
    - but not the whole truth at once
    - gradually increase the level of detail
- Distinguish between what’s legal and what’s effective
  - Better tool support is needed
    - E.g. for the C++ Core Guidelines

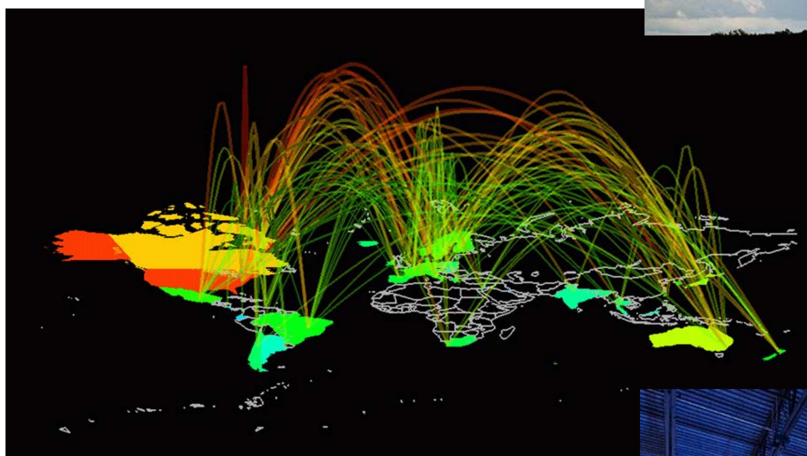
C++: principled and eclectic

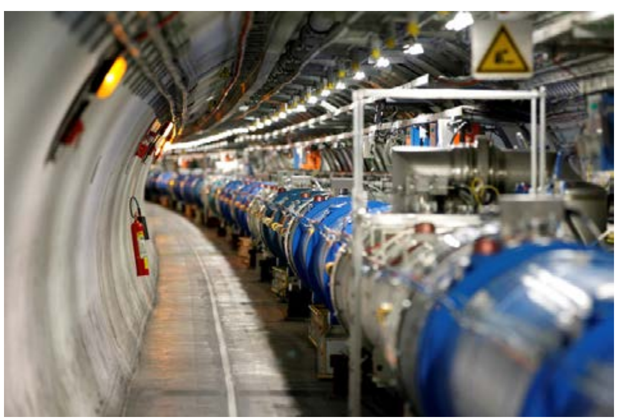
- C++ a general-purpose programming language for the definition, implementation and use of lightweight abstractions
- Language design is not just product development
  - Coherent design philosophy is essential
  - Stable over decades
- Whatever it takes for production code
  - The world is unimaginably diverse
  - Much of the world is messy
  - Many applications require stability over decades
  - Often, C++ is “the only thing that works”
- Simple enough for “casual use”
  - Don’t try to enforce some idea of “theoretical purity”
  - Don’t be “expert only”
  - Make simple things simple

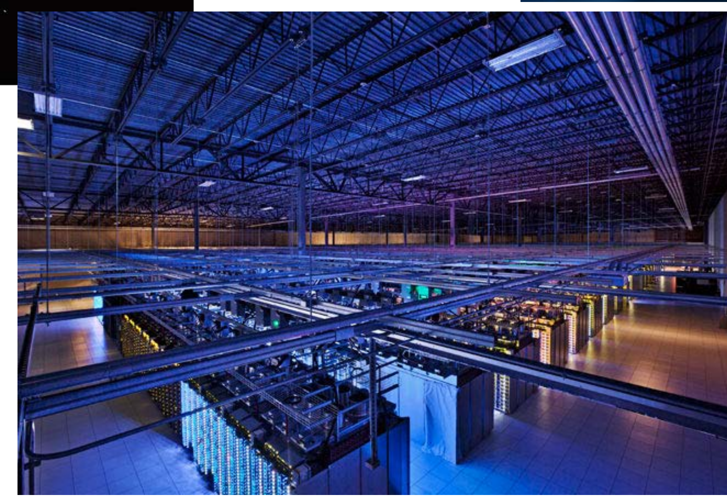
C++ high-level aims (aka principles)

- Evolutionary
  - Stable (backwards compatible)
  - Support gradual adoption
- Make simple things simple
  - Don’t make complicated tasks impossible
  - Don’t make complicated tasks unreasonably hard to do
- Zero-overhead principle
  - What you don't use, you don't pay for (aka ``no distributed fat'')
  - What you do use, you couldn't hand-code any better
- Aim high
  - Significantly change the way we design and implement software
  - Change the way we think

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







We changed the world!

- Programming and design
  - Abstraction: direct expression of ideas
  - Better use of better hardware
  - Code analysis and compiler construction
- Applications
  - Scale and sophistication
  - Engineering, science, …
  - …

2019 and 1979

C++ inside

Power and connectivity

C++ has been driving dramatic changes in many many areas

Key C++ "Rules of thumb"

1. A static type system with equal support for built-in and user-defined types
2. Value ***and*** reference semantics
3. Direct use of machine and operating system resources
4. Systematic and general resource management (RAII)
5. Support composition of software from separately developed parts
6. Support for object-oriented programming
7. Support for generic programming
8. Support for compile-time programming
9. Concurrency through libraries supported by intrinsics
10. …

My guide
for this talk

A language is not just a list of features

- C++ is (most deliberately) evolving
  - Too fast for some, too slow for some
- Maintaining coherency is hard
  - Requires articulated principles
  - Aim for steady gradual improvement (aka evolution)

- There was and is a plan
  - H. Hinnant, B. Stroustrup, R. Orr, D. Vandevoorde, and M. Wong: ***Direction for ISO C++ (R*)***. P0939R*. (The Direction Group)
  - B. Stroustrup: ***Remember the Vasa!*** P0977r0. 2018-03-6.
  - Jan Christiaan van Winkel, Jose Daniel Garcia, Ville Voutilainen, Roger Orr, Michael Wong, Sylvain Bonnal: ***Operating principles for evolving C++***. P0559R0. 2017-01-31. (Heads of National standards delegations)
  - B. Stroustrup: ***Thoughts about C++17***. 2015-05-15.
  - B. Stroustrup: **Evolving a language in and for the real world: C++ 1991-2006**. ACM HOPL-III. 2007.
  - B. Stroustrup: ***The design and evolution of C++***. Addison-Wesley. 1994.

Key C++ "Rules of thumb"

1. A static type system with equal support for built-in and user-defined types
2. Value ***and*** reference semantics
3. Direct use of machine and operating system resources
4. Systematic and general resource management (RAII)
5. Support composition of software from separately developed parts
6. Support for object-oriented programming
7. Support for generic programming
8. Support for compile-time programming
9. Concurrency through libraries supported by intrinsics
10. …

A static type system – the base of all

- Compile-time error detection
  - e.g., **list<int> lst; … sort(lst);** **//** *error: no random access in lists*
  - Run-time error-handling can get expensive and complicated
- Performance
  - Direct expression of ideas simplifies optimization
  - Move computation from run-time to compile-time
- Flexibility through compile-time resolution
  - Overloading
    - e.g., **sqrt(2);**
  - Generic programming
    - e.g., **vector<int> v; … auto p = find(v,42);**
  - Metaprogramming
    - e.g., **conditional<(sizeof(int)<4),double, int> x;**
  - Compile-time evaluation
    - e.g., **static_assert(weekday(August/3/2019)==Sunday);**

Key C++ "Rules of thumb"

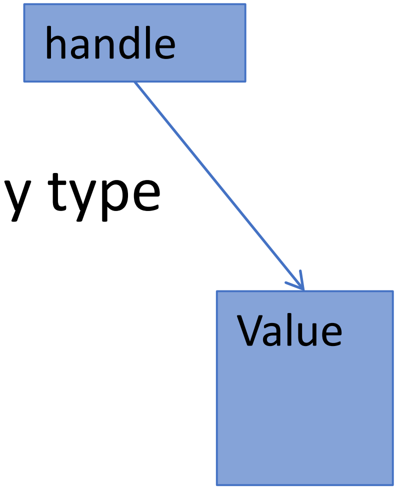
1. A static type system with equal support for built-in and user-defined types
2. Value ***and*** reference semantics
3. Direct use of machine and operating system resources
4. Systematic and general resource management (RAII)
5. Support composition of software from separately developed parts
6. Support for object-oriented programming
7. Support for generic programming
8. Support for compile-time programming
9. Concurrency through libraries supported by intrinsics
10. …

Value *and* reference semantics

- We need both to represent essential concepts

- We can supply value semantics to any type

  x = y + z; *// could be int, or complex<double>, or Matrix, or …*
  x = y; *// x is a copy of y*
  // x and y are independent objects

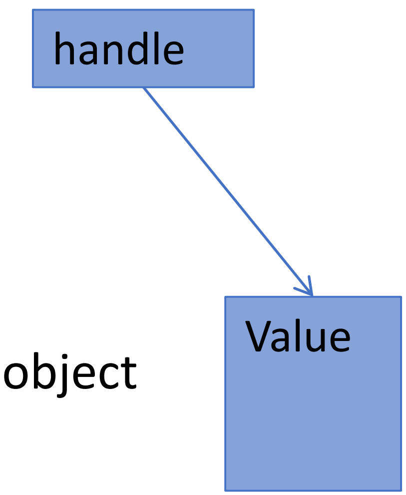


- We can supply pointer/reference semantics for any type

  ***p = x;** *// could be a T*, and shared_ptr<T>, or …*
  p = q; *// p and q points to the same object*

Value *and* reference semantics

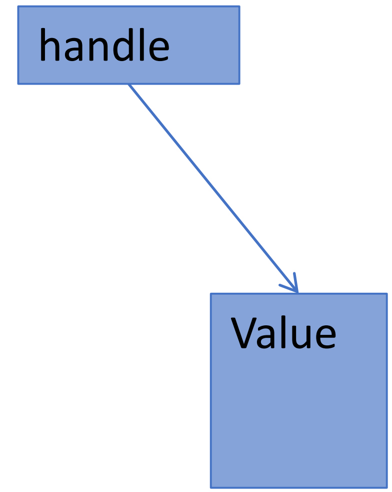
- Value types
  - All our most common types
    - Integers, characters, strings, containers, …
  - Ideal semantics (often regular)
  - Easily allocated on stack
  - Inlining
  - Often implemented using pointers
- Pointers/references
  - All kinds of pointers and references “point” to some object
    - **T***, **T&**, **unique_ptr<T>**, **Forward_iterator**
  - Essential for passing information around efficiently
    - **auto p = find(lst, "something interesting");**
    - **sort(v);**
  - Essential for building non-trivial objects (data structures)
- Both are needed for optimal use of machine resources



Equal support for built-in types and user-defined types

- Regularity is essential for generic programming and much more

```
template<Element T> class Vector {
public:
        Vector(initializer_list<T>);
        // …
        T* elem;            // T* can point to any Element type
                            // user-defined or built-in
};
```



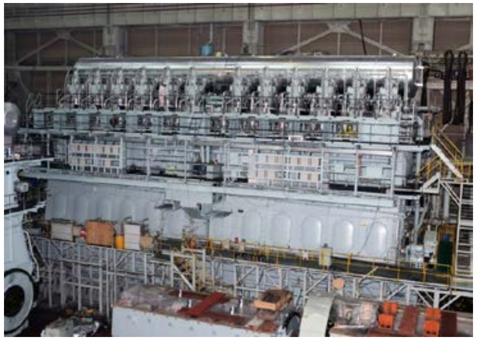

```
// We can parameterize with any Element type:

Vector<int> vi = {1,2,3};                             // built-in
Vector<complex<double>> vc = {{1,2},{3,4},{5,6}};     // user-defined
Vector<Vector<int>> vvi = {{1,2,3}, {4,5,6}, {7,8,9}}; // recursive
```

Morgan Stanley

Key C++ "Rules of thumb"

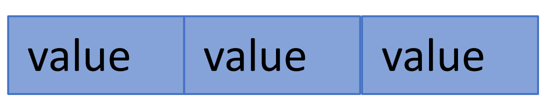
1. A static type system with equal support for built-in and user-defined types
2. Value ***and*** reference semantics
3. Direct use of machine and operating system resources
4. Systematic and general resource management (RAII)
5. Support composition of software from separately developed parts
6. Support for object-oriented programming
7. Support for generic programming
8. Support for compile-time programming
9. Concurrency through libraries supported by intrinsics
10. …

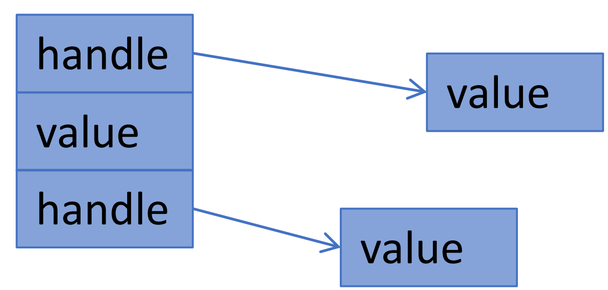


Direct use of machine resources

- Primitive operations maps to machine instructions
  - Arithmetic: +, -, *, /, %
  - Access: ->, [], (), …
  - Bitwise logical: &, |, ^ (exclusive or), ~ (complement), >> and << (shift), rotate
- Memory is a set of sequences of objects
  - Pointers are machine addresses

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





- A simple abstraction of hardware

Direct use of machine resources

- **bitset**
  - Manipulate contiguous sequences of bits of arbitrary sizes
  - &, |, ^ (exclusive or), ~ (complement), >> and << (shift), rotate
- **span**
  - Manipulate contiguous sequences of objects

```
array<byte,1024> a;
// …
span s { a };                  // no explicit element type or size
for (const auto x : s) f(x);   // no range checking and no range error
for (auto& x : s) x = 99;

span s2 {a,512};         // you can give a size if you want to
span<byte> s3 {a};       // you can give an element type if you want to
```

Type inference

The onion principle

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

Key C++ "Rules of thumb"

1. A static type system with equal support for built-in and user-defined types
2. Value ***and*** reference semantics
3. Direct use of machine and operating system resources
4. Systematic and general resource management (RAII)
5. Support composition of software from separately developed parts
6. Support for object-oriented programming
7. Support for generic programming
8. Support for compile-time programming
9. Concurrency through libraries supported by intrinsics
10. …

Constructor/destructor pairs (RAII)

- An archetype of a resource manager: "**Gadget**"
  - A resource is anything that must be acquired and given back
  - A user doesn't need to know which resources **Gadget** uses

```
class Gadget {
        Gadget( /* arguments */ );    // initialize/construct
                                      // incl. acquire any resources needed
        ~Gadget();                    // clean up any mess
                                      // incl. releasing any resources held
        // … copy and move …
        // … rest of user interface …
private:
        // … representation …
};
```

Morgan Stanley

Systematic general resource management

- Every resource must have an owner
  - Responsible for its cleanup (destruction)
  - Don’t use built-in pointers (**T***) to manage ownership
- Anchor resources in scopes

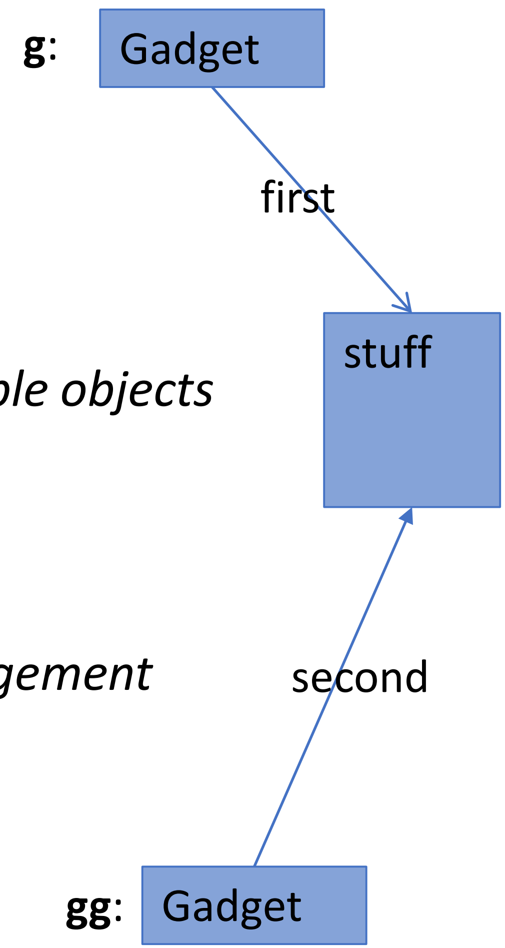
```
void f(int n, int x)
{
	Gadget g {n};		// we don’t need to know which resources g owns
	// …
	if (x<100) throw run_time_error{"Weird!"};		// no leak
	if (x<200) return;					// no leak
	// …
}
```

Systematic general resource management

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

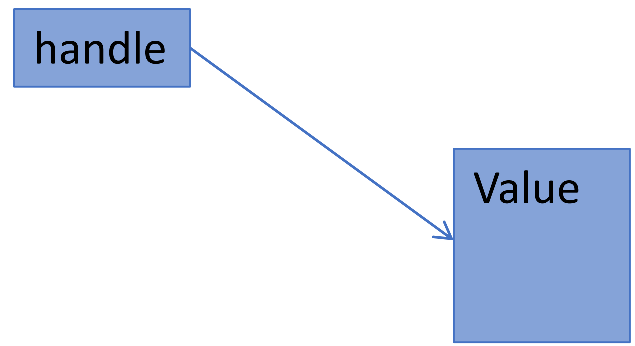
```
Gadget f(int n, int x)
{
      Gadget g {n};      // g may be huge
                         // g may contain non-copyable objects
      // …
      return g;          // no leak, no copy
                         // no pointers
                         // no explicit memory management
}

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



General resource management

- 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, multiset, …**
  - **string, path**



- Many C++ standard-library classes manage non-memory resources
  - **thread**, **jthread**, **shared_mutex**, **scoped_lock**, …
  - **istream**, **fstream**, …
  - **unique_ptr**, **shared_ptr**

GC is neither sufficient nor ideal

- A container can hold a non-memory resource
  - This all works recursively, e.g., **vector<forward_list<pair<string,jthread>>**

Key C++ "Rules of thumb"

1. A static type system with equal support for built-in and user-defined types
2. Value ***and*** reference semantics
3. Direct use of machine and operating system resources
4. Systematic and general resource management (RAII)
5. Support composition of software from separately developed parts
6. Support for object-oriented programming
7. Support for generic programming
8. Support for compile-time programming
9. Concurrency through libraries supported by intrinsics
10. …

Modules

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

import std;                // the order of imports is unimportant
import my_containers;

export
template<forward_range S>
	requires Printable<KeyType<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';
}
```

Modules

- Support clean code
  - Minimizes dependencies
  - Avoids circular dependencies
  - Modularity

    import A;

    import B;

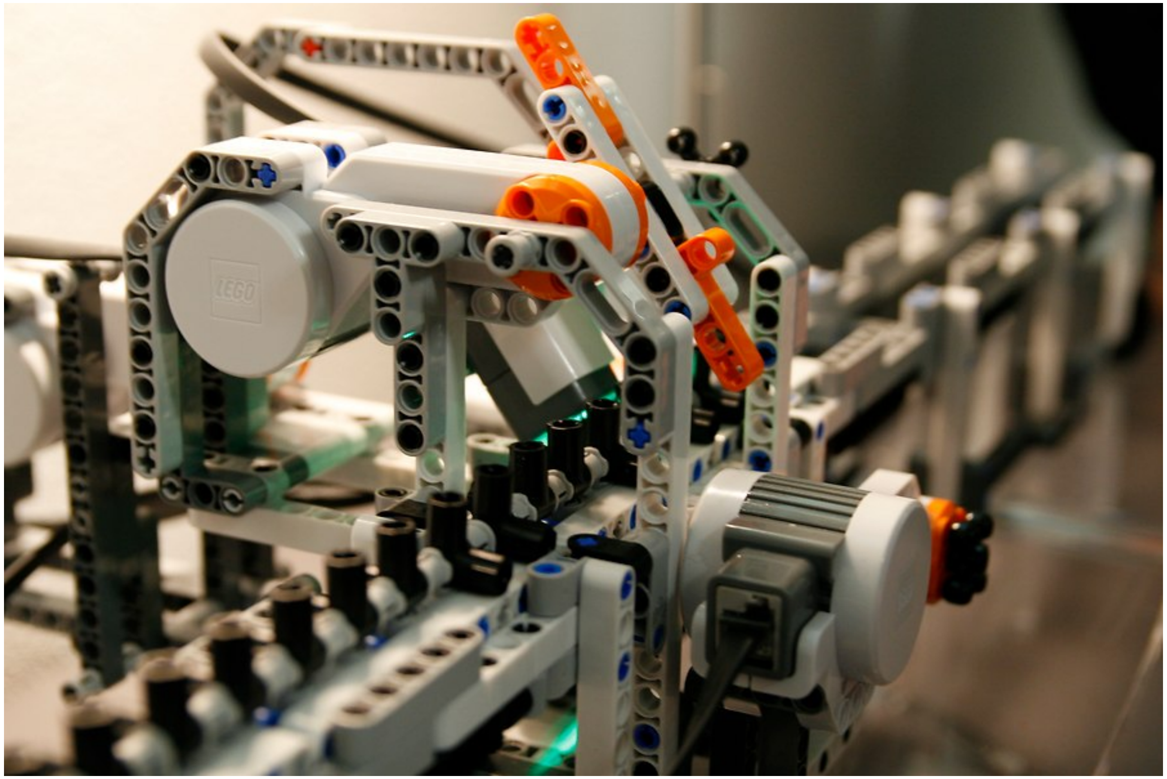
    means the same as

    import B;

    import A;
  - Only the used parts of an imported module are turned into generated code
  - There is only one “copy” of a module, analyzed once

Composition

- All major features support composition
  - Modules
  - Classes
  - Concepts
  - Templates
  - Functions
  - Aliases



Morgan Stanley

Key C++ "Rules of thumb"

1. A static type system with equal support for built-in and user-defined types
2. Value ***and*** reference semantics
3. Direct use of machine and operating system resources
4. Systematic and general resource management (RAII)
5. Support composition of software from separately developed parts
6. Support for object-oriented programming
7. Support for generic programming
8. Support for compile-time programming
9. Concurrency through libraries supported by intrinsics
10. …

Generic programming

- Write code that works for types that meet abstract requirements
  - E.g., is a forward iterator, is integral, is regular, can be sorted
- These requirements are 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
```

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

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); });
```

Key C++ "Rules of thumb"

1. A static type system with equal support for built-in and user-defined types
2. Value ***and*** reference semantics
3. Direct use of machine and operating system resources
4. Systematic and general resource management (RAII)
5. Support composition of software from separately developed parts
6. Support for object-oriented programming
7. Support for generic programming
8. Support for compile-time programming
9. Concurrency through libraries supported by intrinsics
10. …

Object-oriented programming

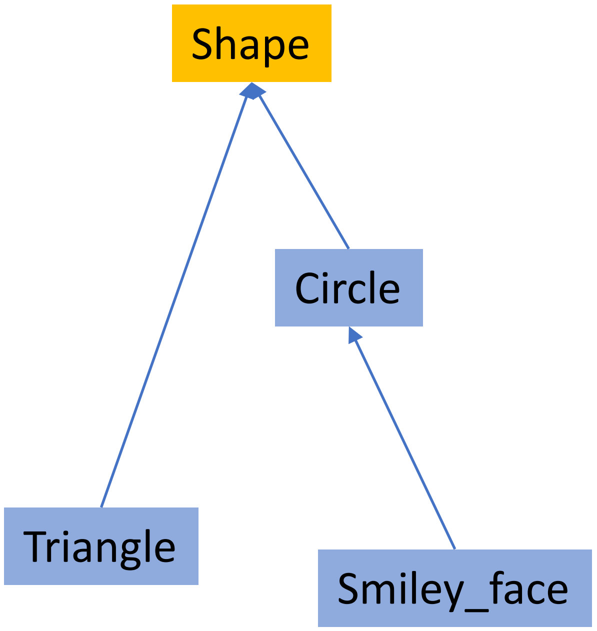
- Hierarchy
  - From the dawn of time (1969)
  - For run-time resolution

Still useful and popular

```
class Shape {
        virtual void draw() =0;       // abstract class
        // …
};

class Circle : public Shape {
        void draw() override;
        // …
};

class Triangle : public Shape {
        void draw() override;
        // …
};
```



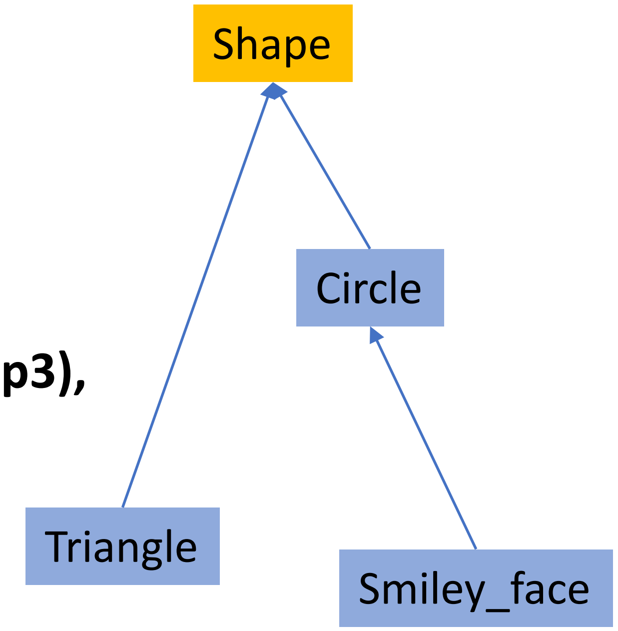
Object-oriented programming

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

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

void some_use(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



Object-oriented programming?

- What if I don't need run-time resolution?
  - Maybe use static resolution?

```
using Vec = vector<variant<Circle,Triangle,Smiley>>;

void draw_all(Vec& vec)
{
	for (auto& v: vec) {
			visit(overloaded {		// set of alternatives
					[](Circle& c) { c.draw(); },
					[](Triangle& t) { t.draw(); },
					[](Smiley& s) { s.draw(); },
			}, v);
	}
}
```

Best when the variant types are of roughly the same size

Oops!

- **overloaded()** didn't make it in time for C++20
- C++ is extensible
  - Build what you need
  - Or – better – use one of the existing libraries

```
template<class... Ts>
struct overloaded : Ts... {        // collect N types
    using Ts::operator()...;       // call for each of the N types
};

// deduce template argument types:
template<class... Ts> overloaded(Ts...) -> overloaded<Ts...>;
```

Key C++ "Rules of thumb"

1. A static type system with equal support for built-in and user-defined types
2. Value ***and*** reference semantics
3. Direct use of machine and operating system resources
4. Systematic and general resource management (RAII)
5. Support composition of software from separately developed parts
6. Support for object-oriented programming
7. Support for generic programming
8. Support for compile-time programming
9. Concurrency through libraries supported by intrinsics
10. …

Direct use of system resources

- 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
```

Morgan Stanley

Direct use of system resources

- “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

Direct use of system resources

- There is always a lower level

```
mutex mx;
atomic<bool> initx;
int x;
if (!initx.load(memory_order_acquire) {
	mx.lock();
	if (!initx.load(memory_order_relaxed) {
			x = 42;
			initx.store(true, memory_order_release);
	}
	mx.unlock();
}
// … use x …
```

- Don’t lower the level of abstraction unless you really need to

C++ is tunable and evolves

- Common scenario
  - pX: See X is faster than / as fast as / almost as fast as C++ !!!
  - pC++: but your C++ version is poor C++, not colloquial
    - try this version; it's as fast as / faster than X
  - pX: That's cheating: that's not pure OO, FP, X !
  - pC++2: pC++'s version is still quite slow
    - here's a much faster version
  - pX: but the X version is much easier / elegant / safer / …
  - pC++2: but I need the performance
- C++ evolves
  - (soon after) pC++3: here is a C++ library that does that
  - (years later) pC++4: ISO C++ now has a feature that does that

C++ is tunable

- Make simple things simple
  - Don't make complicated tasks impossible
  - Don't make complicated tasks unreasonably hard to do
  - The onion principle
- Don't drop to lower levels of abstraction
  - Unless you *really*, *really* need to
  - Hide messy code behind clean interfaces
- Always measure
  - But be careful
    - results on your laptop may not apply to a server
    - And vice versa

Direct use of system resources

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

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

Direct use of system resources

- 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();
      // …
}
```

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; });
        // …
}
```

Oops!

- The parallel versions of range sort didn't make it in time for C++20
- C++ is extensible
  - Build what you need
  - Or – better – use one of the existing libraries

```
// in the standard:
template<class ExecutionPolicy, class RandomAccessIterator>
	void sort(ExecutionPolicy&& exec,
			RandomAccessIterator first, RandomAccessIterator last);

// what I wanted:
void sort(execution_policy auto&& exec, random_access_range r)
{
	sort(exec, r.begin(), r.end());
}
```

Key C++ "Rules of thumb"

1. A static type system with equal support for built-in and user-defined types
2. Value ***and*** reference semantics
3. Direct use of machine and operating system resources
4. Systematic and general resource management (RAII)
5. Support composition of software from separately developed parts
6. Support for object-oriented programming
7. Support for generic programming
8. Support for compile-time programming
9. Concurrency through libraries supported by intrinsics
10. …

Compile-time computation

- Move computation from run-time to compile-time
  - For performance and elegance
  - Do it once, rather than a billion times
  - Don’t need run-time error handlers
  - You can’t have a data race on a constant
- It’s everywhere
  - Overloading and virtual functions
  - Templates
  - Variadic templates
  - Constexpr functions and user-defined types

Compile-time computation

- Type-rich programming at compile time

```
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
```

Compile-time computation

- Not just built-in types

```
cout << weekday{June/21/2016} << '\n';                      // cout << “Tuesday\n”
static_assert( weekday{June/21/2016}==Tuesday );   // At compile time
```

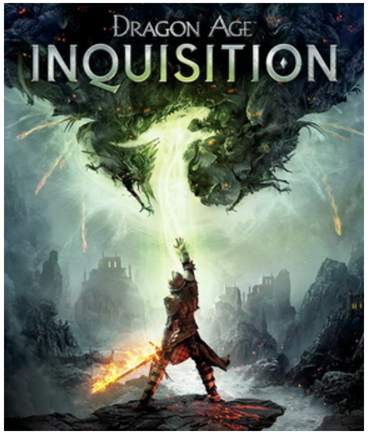
- Compile-time computation tends to be invisible

```
auto z = sqrt(3+2.7i);                              // call sqrt(complex<double>)
auto d = 5min+10s+200us+300ns;                      // a duration
auto s = "This is not a pointer to char"s;          // a string

// implementations:
constexpr complex<double> operator""i(long double d) { return {0,d}; }
constexpr seconds operator""s(unsigned long long s) {return s; }
constexpr string operator""s(const char* str, size_t len) { return {str, len}; }
```

Key C++ "Rules of thumb"

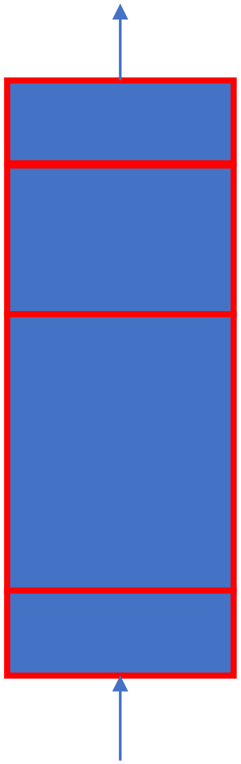
1. A static type system with equal support for built-in and user-defined types
2. Value ***and*** reference semantics
3. Direct use of machine and operating system resources
4. Systematic and general resource management (RAII)
5. Support composition of software from separately developed parts
6. Support for object-oriented programming
7. Support for generic programming
8. Support for compile-time programming
9. Concurrency through libraries supported by intrinsics
10. …



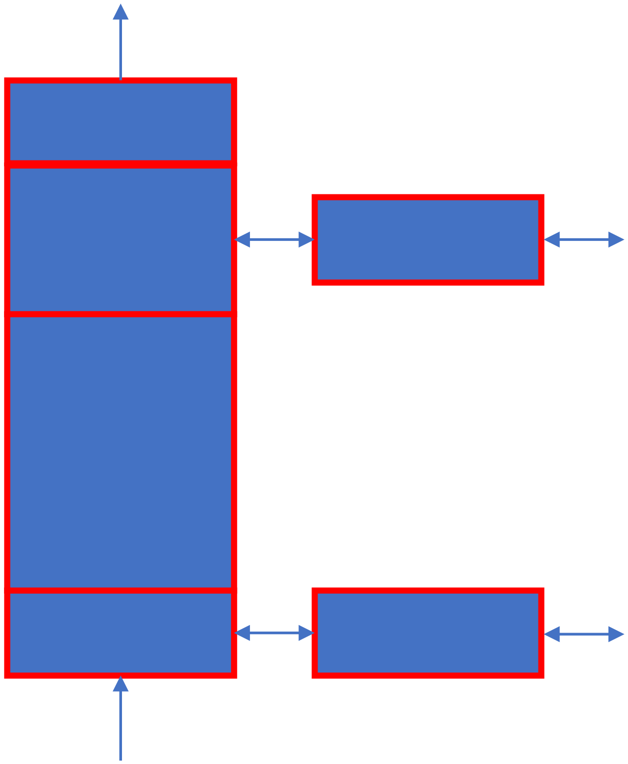
Morgan Stanley

Direct use of hardware

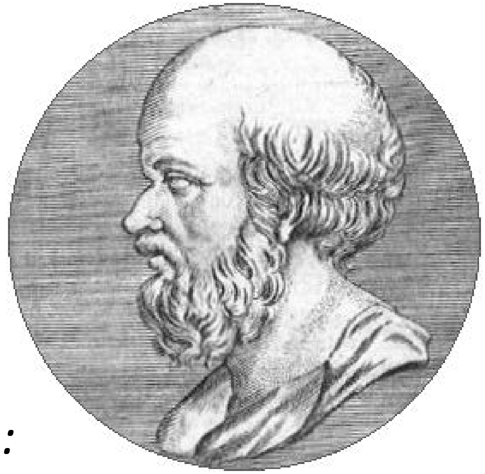
- Functions
  - Stack frames



- Coroutines
  - Invocation frames



Coroutines: Better generators and pipelines



- Lazy evaluation

```
int main()
{
        auto src = seq(2);              // infinite int sequence:
                                        // 2,3,4,5,6,7,8,9,10,11…
        auto s = sieve(src);            // filter out non-primes:
                                        // 2,3,5,7,11,…
        auto t = take(s, 10'000);       // get first 10,000 primes
                                        // 2,3,5 … 104729
        print(t);                       // print them
}
```

Coroutines (a bit of boilerplate)

```
generator<Int> seq(int start)        // generate an infinite sequence
{
    while (true) co_yield start++;   // yield the next int
}

generator<Int> take(generator<Int>& src, Int count)    // Take elements
{
    if (count <= 0)  co_return;
    for (auto v : src) {
        co_yield v;          // yield the next int
        if (--count == 0)    // we're done
            break;
    }
}
```

Coroutines: Better generators and pipelines

```
generator<Int> sieve(generator<Int>& src)        // Eratosthenes
{
      Int p = head(src);
      co_yield p;           // yield the first non-filtered prime
      auto f = filter(src, [p](auto val) { return val % p; });
      co_yield sieve(f); // yield the next prime
                         // stack up the filters
                         // ...->f(2)->f(3)->f(5)->...
}

template <typename Pred>     // Filter out (skip) !pred elements
generator<Int> filter(generator<Int>& src, Pred pred)
{
      for (auto v : src) if (pred(v)) co_yield v;
}
```

Output

2 3 5 7 11 13 17 19 23 29 … 104707 104711 104717 104723 104729

Generic Sieve

- Did you notice that I use **Int** rather than **int**?
  - **using Int = int;**
- What if I want more primes than fits in an **int**?
  - **using Int = long long;**
- Even more primes?
  - **using Int = Big_int;**

- A brute force approach
  - Just illustrating combinations of features and techniques

Coroutines: simpler asynchronous use

- A major use of coroutines is to simplify and speed up asynchronous operations

```
Task<> start()              // Infinite read/write socket task
{
    char data[1024];  // Buffer
    while (true) {
            auto n = co_await socket.async_read_some(buffer(data));
            co_await async_write(socket, buffer(data,n));
    }
}
```

Putting it all together

- Language features are meant to be used in combination
  - And together with libraries
- Examples
  - Reference semantics enables the efficient implementation of advanced types with value semantics (e.g., **jthread** and **vector**).
  - Uniform rules for built-in and user-defined types simplifies generic programming (built-in types are not special cases).
  - Compile-time programming makes a range of abstraction techniques affordable for effective use of hardware.
  - RAII allows use of user-defined types without taking specific actions to support their implementations' use of resources.
  - …

Libraries

- A user shouldn't have to care whether a feature is implemented in the language or in a library
  - Library design is language design
  - Language design is library design
- The standard library should follow the same design guidelines as the language
  - We are not perfect at that
- We need great libraries

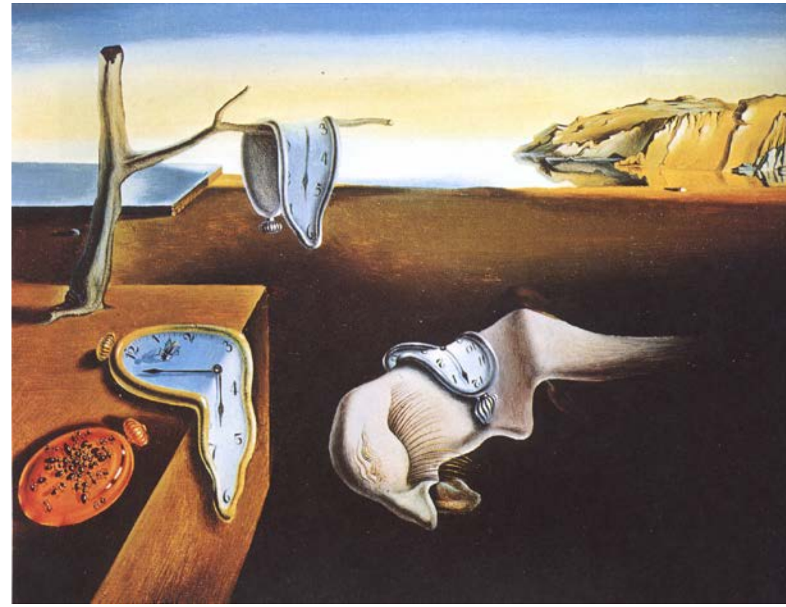
Libraries

- You don’t have to do it all by yourself from scratch
- Std
  - The STL, ranges, concepts
  - Iostreams, locale, format
  - Chrono, dates, time zones
  - Threads, locks, atomics, futures, …
  - Random
  - File system
  - String, regex
  - Variant, tuple, pair, shared_ptr, unique_ptr, traits, …
  - …
- Other (lots and lots)
  - Boost, Qt, Poco, asio, CopperSpice, GSL, Eigen, …
  - https://en.cppreference.com/w/cpp/links/libs

Key C++ "Rules of thumb"

1. A static type system with equal support for built-in and user-defined types
2. Value ***and*** reference semantics
3. Direct use of machine and operating system resources
4. Systematic and general resource management (RAII)
5. Support composition of software from separately developed parts
6. Support for object-oriented programming
7. Support for generic programming
8. Support for compile-time programming
9. Concurrency through libraries supported by intrinsics
10. …

Chrono



- **time_points**, **durations**
- **days**, **months**, **years**
- Time zones

```
int main()
{
   using namespace std::chrono;
   for (auto m = local_days{January/9/2019};
                  year_month_day{m}.year() < 2020y;
                  m += weeks{2}) {
        zoned_time london{"Europe/London", m + 18h};
        cout << london << '\n';
        cout << zoned_time{"America/New_York", london} << '\n';
        cout << zoned_time{"Etc/UTC", london} << '\n';
        cout << '\n';
     }
}
```

Morgan Stanley

Output

2019-01-09 18:00:00 GMT
2019-01-09 13:00:00 EST
2019-01-09 18:00:00 UTC

2019-01-23 18:00:00 GMT
2019-01-23 13:00:00 EST
2019-01-23 18:00:00 UTC

…

What I didn’t mention

Except as implementation details and asides

- Sizes
- Raw pointers
- Allocation and deallocation
- Loop-control variables
- Casts
- Macros

- That’s for lower levels of abstraction (often in implementations)

- How we used to do things (aka now)
- Most “details”

Guidelines

- Write modern C++
  - Not C or 1988-vintage C++ (whenever you can)
- We must distinguish between
  - What's legal and what's good
  - What works and what's maintainable
  - What runs and what's affordable (time and space)?
- 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)

Morgan Stanley

Key C++ "Rules of thumb“

about 40 years old

1. A static type system with equal support for built-in and user-defined types
2. Value ***and*** reference semantics
3. Direct use of machine and operating system resources
4. Systematic and general resource management (RAII)
5. Support composition of software from separately developed parts
6. Support for object-oriented programming
7. Support for generic programming
8. Support for compile-time programming
9. Concurrency through libraries supported by intrinsics
10. …

So, what is C++20?

- The best approximation of C++'s ideals
  - so far
- As big an improvement over C++11 as C++11 was over C++98
  - A major "release"
- Lots of useful features
  - Simpler, more expressive, faster code that compiles faster
  - Modules
  - Concepts
  - Coroutines
  - Ranges
  - Dates
  - Span
  - Better compile-time programming support
  - Many "minor features"
    - Some significant

All available now,
But not yet in all
implementations

So, what is C++20?

- C++ a general-purpose programming language for the definition, implementation and use of lightweight abstractions
- Not a grab bag of features
  - A set of ideals
  - A set of design principles
- A stage in an evolutionary process
  - … -> C++98 -> C++11 -> C++14 -> C++17 -> C++20 -> …
- A process
  - WG21

Remember WG21's hard work

1989 … 2019

The future* **

- C++20
  - The best approximation of C++'s ideals so far
    - Not perfect, of course
- C++23
  - "Completes C++20"
  - Plus
    - Standard modules
    - Library support for coroutines
    - Executors & networking
  - Maybe
    - Static reflection
    - Pattern matching

* First approximation suggested by Ville Voutilainen. Supported by the Direction Group

** It is hard to make predictions, especially about the future
– Niels Bohr

C++20 is great!

- The votes have started
- Large parts are in implementations shipping now
- All major parts will ship in all major implementations in 2020
- Not perfect, of course
- Lots of in-depth talks at CppCon'19