21st Century C++

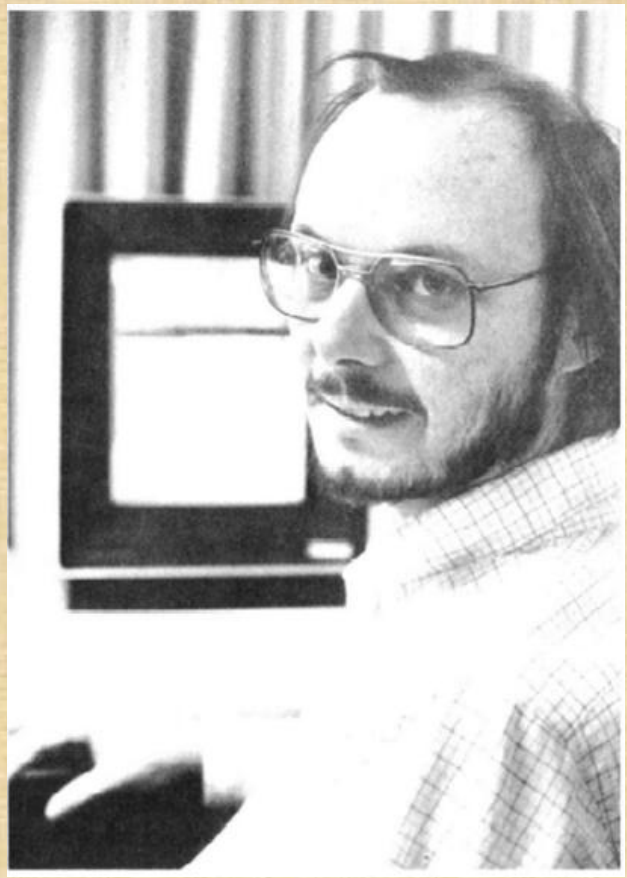
Bjarne Stroustrup

Columbia University

Bjarne@stroustrup.com

A good design starts with a problem

- I wanted to build a distributed Unix
  - In 1979
  - No language could do all I needed for that
- Efficient manipulation of hardware
  - Like C
- Abstraction to manage complexity
  - Like Simula
  - A flexible “strong” static type system



Bjarne Stroustrup, about 1986

Key idea: Direct representation of ideas in code

- Examples
  - **Math**: tensor, polynomial
  - **Engineering**: Matrix, Fourier transform
  - **Graphics**: Shading, gnome, path
  - **Biology**: DNA_string, protein
  - **Telecommunications**: Buffer, channel
  - **Aerospace**: Electric motor, route
  - **Automotive**: vehicle, car, pedestrian
  - **Finance**: instrument, transaction
  - **Computer Science**: map, task, image, file, edge
  - …
- And make that affordable
- Your imagination is the limit
- Note: most good software is invisible

A good design rests on sound principles

- Flexible static type system
  - Extensibility
  - Zero overhead
  - Minimal explicit type conversion
- Resource management
  - No leaks
- Error handling
  - Guarantees
- Flexible concurrency support
  - No one style

Contemporary C++
can deliver that better than
any earlier C++

A good tool evolves to meet demands

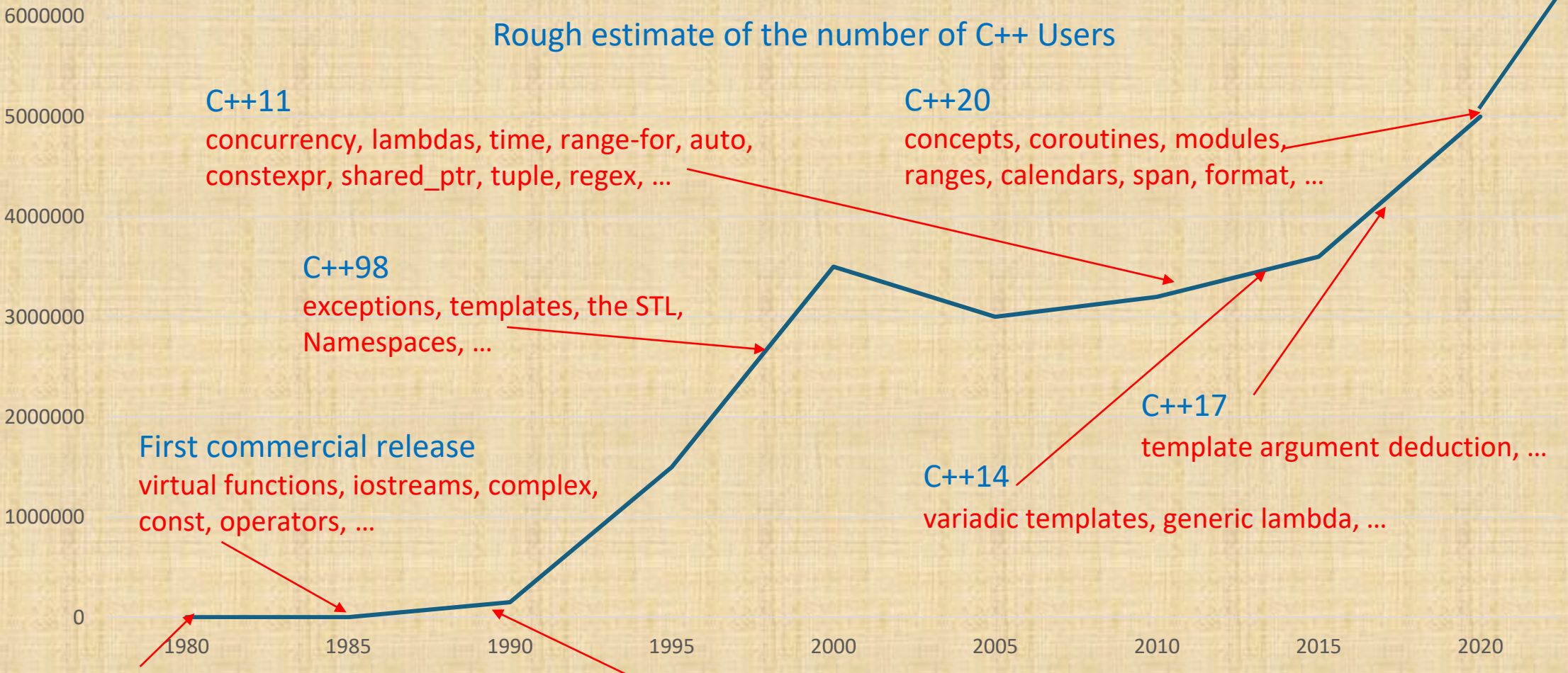
- Why evolve?
  - The world changes
  - The problems change
  - We change
- Good engineering relies on feedback and refinement
  - E.g., Generic programming, Compile-time programming, Modules

- "Anyone who claims to have a perfect language is either a salesman or a fool, or both"
  - – Bjarne Stroustrup 1980s onward
- "Even I could design a prettier language"
  - – Bjarne Stroustrup 1980s onward

Stability and evolution

- ***Stability***: what used to work well, still works well
- ***Evolution***: usually, we can do even better today
- ***Real-world Progress***: How do developers catch up with necessary changes?

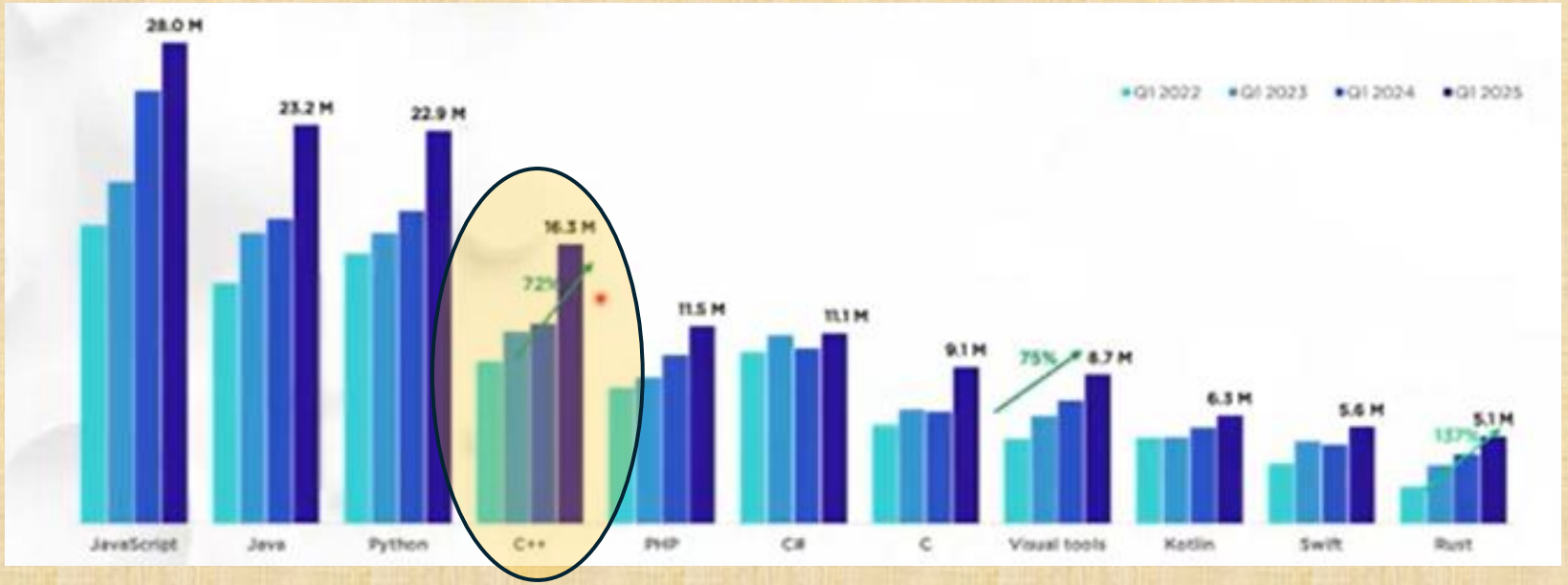
C++ was designed to evolve, and its use grew



It is ***very*** hard to count C++ users

- Who is a developer? A programmer?
  - Do students count
  - Do managers, planners, testers, documenters, etc. count?
  - Most developers use multiple languages
- Not all developers have equal impact
  - yet are all counted as one
  - Enterprise software
  - embedded systems
  - open source or community projects
  - Individual projects
- Many organizations don’t publish their number of developers
- Different organizations use different methods of counting
- Some users and organizations make a lot of noise on the Web
  - Many do not

From www.slashdata.co How many developers?

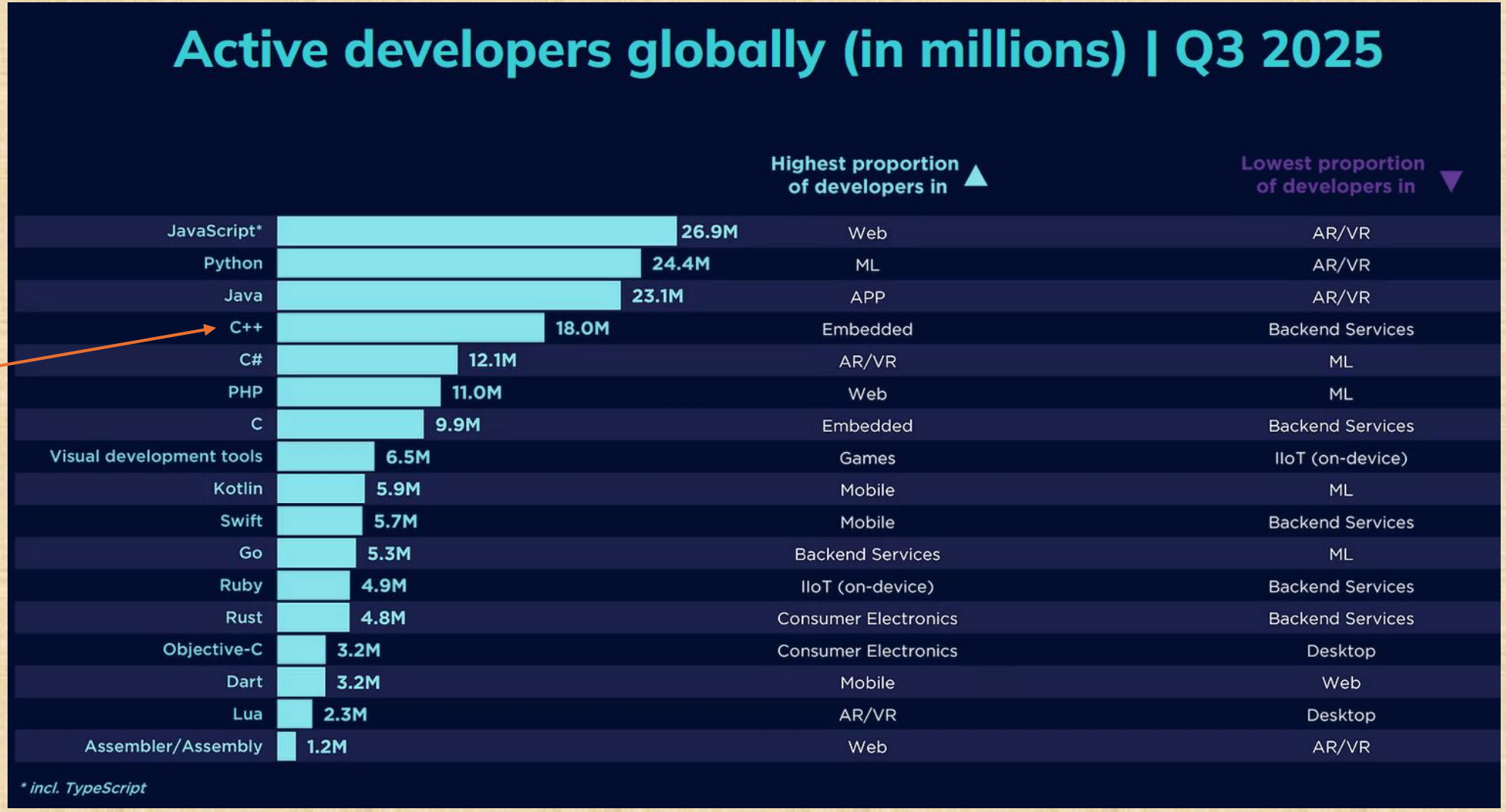


12 million seems to be a conservative estimate

- Many people trust this source
  - 16.3 million C++ developers in 2025
  - 72% growth of the number of C++ developers 4 years (almost 20% per year!)
  - Almost 7 millions new developers in 4 years (almost 2 million per year!)
  - C++, Java, and Python developer populations are growing significantly faster than those of other languages
- Many languages depends on C++
  - Python, Java, Javascript, C, …

And it is continuing (Why?)

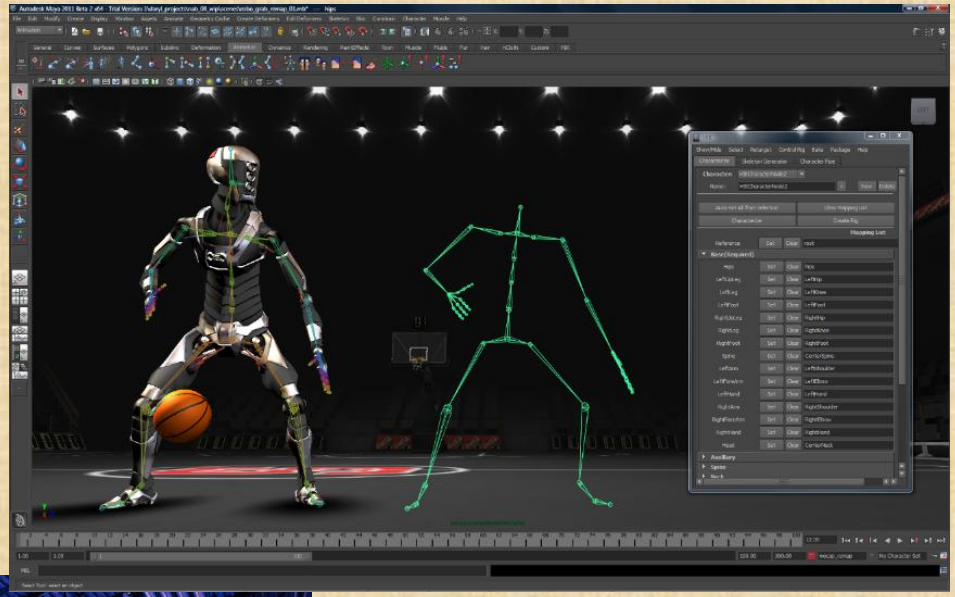
+1.7M in 2025
Amazing!

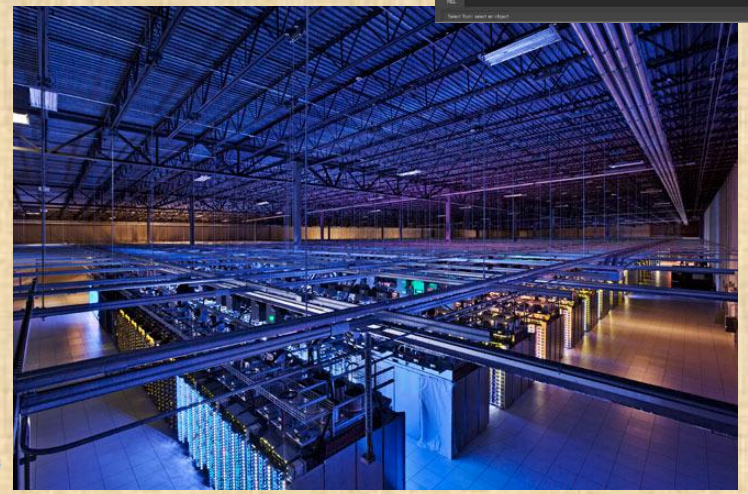


The purpose of my work is

- To significantly improve the state of the art for millions of developers
  - And not just C++ programmers
  - Many of the resulting ideas and techniques find their way into other languages; e.g., C, Java, C#, Python
  - That involves work and innovation in many diverse fields

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



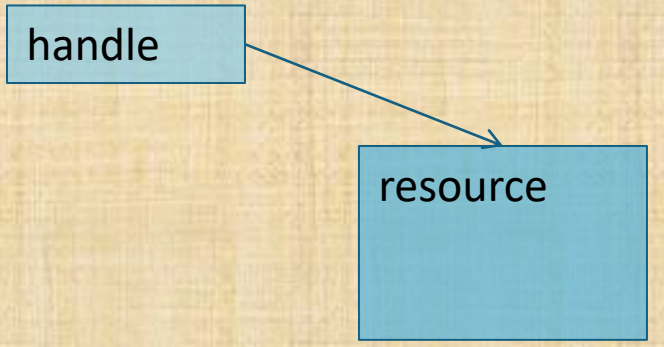


Here, I focus on

- Resource management
  - including control of lifetime and error handling
- Generic programming
  - including concepts
- Modules
  - including eliminating the preprocessor
- Guidelines and enforcement
  - to guarantee that we write “21st Century C++”?

Resource management

- A resource is anything you must acquire and later release (give back)
  - Explicitly or implicitly
  - Examples: memory, strings, locks, file handles, sockets, thread handles, transactions, shaders, …
- Avoid resource leaks
  - Avoid manual release
  - No **free()**, **delete**, and similar resource releases in application code
- Every resource is represented by a handle
  - Responsible for access and release
  - No **malloc()**, **new**, and similar pointer-returning resource acquisitions in application code
- Every resource handle is rooted in a scope
  - And handles can be moved from scope to scope



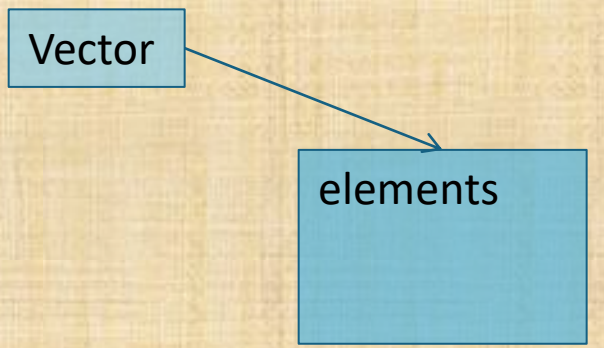
Raise the level of abstraction

Type parameter

The real start of C++: Constructor/destructor

```
template<typename T>
class Vector {              // vector of elements of type T
public:
    Vector(initializer_list<T>);        // acquire memory; initialize elements – constructor
    ~Vector();                          // destroy elements; release memory – destructor
    // …
private:
    T* elem;                // pointer to elements
    int sz;                 // number of elements
};

void fct()
{
    Vector<double> constants {1, 1.618, 3.14, 2.99e8};
    Vector<string> designers {"Strachey", "Richards", "Ritchie"};
    // …
    Vector<pair<string,jthread>> vp { {"producer",prod}, {"consumer",cons}};
}
```

Lower level



Rules apply recursively

Control of lifetime

- Necessary for simple and efficient resource management
  - Construction
    - Before first use establish the invariant (if any)
    - Constructor
  - Destruction
    - After last use release every resource (if any)
    - destructor
  - Copy
    - Copy: **a=b** implies **a==b** (regular types)
    - Copy constructor: **X(const X&)**
    - Copy assignment: **X::operator=(const X&)**
  - Move
    - Move resources between scopes
    - Move constructor: **X(X&&)**
    - Move assignment: **X::operator=(X&&)**

Output unique lines #1

// the AWK program **(!a[$0]++)** *relies on implicit I/O and loops*

```
import std;
using namespace std;

int main()          // print unique lines from input
{
        unordered_map<string,int> m;                    // hash table
        for (string line; getline(cin,line); )
                if (m[line]++ == 0)
                        cout << line << '\n';
}
```

No explicit
- allocation/deallocation
- Sizes
- Error handling
- Type conversions
- Pointers
- Preprocessor use

Quite efficient
- Tuneable

Output unique lines #2 – collect the lines

```
import std;
using namespace std;

vector<string> collect_lines(istream& is)       // collect unique lines from input
{
        unordered_set<string> s;                // hash table
        for (string line; getline(is,line); )
                s.insert(line);
        return vector{from_range, s};           // copy set elements into a vector
}

auto lines = collect_lines(cin);
for (auto& s : lines)
        cout << s << "\n";
```

The vector's element type is deduced

The vector is moved, not copied

Output unique lines #3 – eliminate copying

```
vector<string> collect_lines(istream& is)     // collect unique lines from input
{
        unordered_set<string> s {from_range,istream_view<Line>{is}};  // construct the set directly from input
        return vector{from_range,s};
}

auto lines = collect_lines(cin);
```

- Use move a constructor for **vector**
  - Faster and simpler than using **new**, a pointer, and **delete**
- Or even better: construct the **vector** in **lines** (since 1983)
  - “copy elision”

Eliminate copying: important detail

- The standard library doesn't have a **Line** type
  - So, I built one

```
struct Line : string { };
istream& operator>>(istream& is, Line& ln) { return getline(is, ln); }
```

Kristen Nygaard

- Note: Conventional OOP
  - Don't get obsessed trying to use only new features
  - Many older features are still essential and the often the best for what they are good at

Tuning

- Is essential for some code
  - “do not optimize prematurely”
  - Always measure before and after optimization
  - Design interfaces to allow optimization if needed
    - well defined
    - Type rich
    - With enough information to enable checking and optimization

- Complexity management
  - Make simple things simple!
- Layers of abstraction
  - The more layers you peel off, the more you cry

Tuning: many opportunities. Don’t be (too) clever

```
vector<span<char>> collect_lines8(istream& is, span<char> buf)
{
    // read into single sufficiently large non-local character buffer
    //        just one allocation, don't initialize chars
    // return spans rather than strings from readlines()
    //        avoid new free store use
    // put into unordered_set or flat_set
    //        avoid linked structures
    // copy spans (not strings) into vector
    // return vector<span<char>>
}
```

Concerns

- Estimate size of buffer
- Complex code. Correct?
- Really fast? always measure
- Anything I could vectorize?
- Read and write faster than iostream?

Resources and errors

- In general
  - Don’t leak a resource
  - Don’t leave a resource in an invalid state

Easy if every resource has a resource handle.
Essentially impossible if not.

- When an error happens
  - Before exiting a function
    - Put every object accessed into a valid state
    - Release every object that the function owns

Resource Handles and Pointers

- Don’t use a built-in pointer as a resource handle
  - “No naked new”

```
void f(int n, int x)
{
        Gadget g {n};     // a Gadget may hold resources, e.g., memory, locks and file handles
        Gadget* pg = new Gadget{n};                          // explicit new: don't!
        // …
        if (x<100) throw std::runtime_error{"Weird!"};       // leak *pg; g not leaked
        if (x<200) return;                                   // leak *pg; g not leaked
        // …
        delete pg;                  // it is hard to remember to delete consistently
}
```

- Local objects are simpler and typically faster than explicit new

Have an articulated error-handling policy

- Use error codes and tests for failures that are common and can be handled locally
  - Ensure that failure to check for an error gives termination or exception, rather than wrong results
- Use exceptions for failures that are rare ("exceptional") or cannot be handled locally
  - To handle errors in constructors, operators, and other functions that don't have an easy way to return an error indicator (e.g., **Matrix x = y+z;**)
  - To automatically propagate errors up the call chain to a handler
  - The alternative is "error-code hell" where every caller up the call stack must remember to test
  - Don't use pointers as resource handles – consistently use scoped resource handles (RAII)
- In some important applications, unconditional immediate termination isn't an option.

Error handling

- Rely on RAII
  - Exceptions can be cheaper and faster than error codes even for small systems: C++ Exceptions for Smaller Firmware

Imagine what this code would look like if using error-code-based error handling

```
void fct(jthread& prod, string name)
{
        ifstream in { name };
        if (!in) { /* … */ }                // possible failure expected
        // …
        vector<double> constants {1, 1.618, 3.14, 2.99e8};          // memory exhaustion possible
        vector<string> designers {"Strachey", "Richards", "Ritchie"};   // nested construction
        auto dmr = "Dennis “s + "M. " + designers[2];
        // …
         jthread cons { receiver };
         pair<string,jthread&> pipeline[] = { {"producer", prod}, {"consumer", cons}};
        // …
}
```

Generic Programming

- A key foundation of contemporary C++
  - Shorter, more readable code
  - More direct expression of ideas
  - Zero-overhead abstraction
  - Type safe
- Pervasive in the standard library
  - Containers and algorithms
  - Concurrency: threads, locks, and more
  - Memory management: allocators, resource-management pointers, and more
  - I/O
  - Strings and regular expressions
  - …

Alex Stepanov

Concept-based Generic Programming

- Write code that works for all suitable argument types

```
void sort(Sortable_range auto& r);

vector<string> vs;
// … fill vs …
sort(vs);

array<int,128> ai;
// … fill ai …
sort(ai);

list<int> lsti;
// … fill lsti …
sort(lsti);                    // error: a list doesn't offer random access
```

A concept:
- Specifies what is required r's type

Implicit:
- Type of container
- Type of element
- Number of elements
- Comparison criteria

Templates

- C++'s support for generic programming is based on three aims
  - ***Extremely general/flexible***: "it must be able to do much more than I can imagine"
  - ***Zero-overhead***: Abstractions such as **vector** and **Matrix**, can compete with C arrays
  - ***Well-specified interfaces***: Implying type safety, overloading, and good error messages

- Only this century did we figure out how to achieve all three aims simultaneously
  - Bjarne Stroustrup: Concept-based Generic Programming. October 2025.
  - Gabriel Dos Reis and Bjarne Stroustrup: Specifying C++ Concepts. POPL06. January 2006.

Templates – a classic (C++98) example of use

```
template<typename Random_access_iterator, typename Compare = std::less>
void sort (Random_access_iterator first, Random_access_iterator last, Compare = Compare{})
{ … }
```

```
vector<string> v = { "CPL", "BCPL", "C", "C++" };

sort(begin(v), end(v));                          // sort the vector
sort(begin(v), begin(v)+size(v)/2);              // sort the first half of the vector
sort(begin(v), end(v), greater<string>{});       // sort strings in accenting order

int a[] = { 3,1,4,2,6,9,0,-1};

sort(begin(a), end(a));      // sort the array
sort(begin(a), end(a), [](int x, int y) { return abs(x)<abs(y); });          // sort absolute value
```

a standard-library function object that calls < when invoked

a lambda expression generating a function object

Templates – a classic (C++98) example of use

- This style of generic programming used for this **sort()** is
  - very general, flexible, and traditional
  - a bit verbose
  - not type checked until late (at “template instantiation time”), implying awful error messages
  - Close to optimally efficient
  - Successful at a huge scale
  - Violates the fundamental aim (and general principle) that interfaces should be precisely specified

We must do better

Concepts for specifying constraints

- A concept is a compile-time predicate
  - A function run at compile time yielding a Boolean
  - Often built from other concepts

- Specify the requirements for **sort()**
  - First try

There are libraries of concepts
<ranges>: random_access_range and sortable

```
template<typename R>
concept Sortable_range =
        random_access_range<R>          // has begin()/end(), ++, [], +, …
        && sortable<iterator_t<R>>;     // can compare and swap elements

void sort(Sortable_range auto& r);
```

Concepts

- That simple **Sortable_range** isn't general enough for a foundation library
  - We need to specify a comparison criteria

```
template<typename R, typename Compare = ranges::less>
concept Sortable_range =
        random_access_range<R>                  // has begin()/end(), ++, [], +, …
        && sortable<iterator_t<R>, Compare>;    // compare elements using Compare

template<Sortable_range R, typename Compare = ranges::less>
void sort(R& r, Compare cmp = {});
```

  - Concepts can take several arguments
    - They are not just types of types
  - The standard-library algorithms and concepts are very general
    - often with many parameters and overloads

Use

- Now we can directly express sorting a container

```
vector v = { 1,5,2,8,-1 };

sort(v);                        // sort using <
sort(v, ranges::greater{});     // sort using >

list lst = { 1,5,2,8,-1 };
sort(lst);                      // error: a list isn't a Sortable_range – no random access
```

Compatibility/stability
is an important feature

Use – but what if we wanted to sort lists?

```
template<typename R, typename Compare = ranges::less>
concept Forward_sortable_range =
        forward_range<R>                          // has begin()/end(), … but not ++, [], +
        && sortable<iterator_t<R>, Compare>;      // compare elements using Compare

template<Forward_sortable R, typename Compare = ranges::less>// list version
void sort(R& r, Compare cmp = {})
{
    vector v(range,r);
    sort(v,cmp);
    ranges::copy(v,r);
}
```

- Use

```
vector v = { 1,5,2,8,-1 };
List lst = { 99, 66, 77 }
sort(lst);              // use the forward version
sort(v);                // use the random-access version
```

Concepts

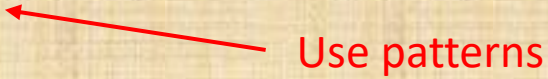
- We have always had the notion of concepts. For example,
  - ***C built-in types***: arithmetic and floating (from 1972 or so)
  - ***STL concepts***: iterators, sequences, and containers (since the early 1990s)
  - ***Mathematical concepts***: monad, group, ring, and field (for a couple of centuries)
  - ***Graph concepts*****:** edge, vertex, graph, and DAG (since 1736)
- No generic program could work unless the programmer had an idea of the concepts involved clearly in mind.
  - What is relatively new is that we can define them to be used in code.
- C++ concepts are compile-time functions (predicates) that can take type arguments.
  - Simple
  - General
  - We must learn to use concepts effectively

Concepts are often built out of other concepts

```
template<typename R, typename Compare = ranges::less>
concept Sortable_range =
    random_access_range<R>                  // has begin()/end(), ++, [], +, …
    && sortable<iterator_t<R>, Compare>;    // compare elements using Compare
```

Concepts can be built out of basic language features

```
template<typename T, typename U = T>
concept equality_comparable = requires(T a, U b) {
        {a==b} -> Boolean;
        {a!=b} -> Boolean;
        {b==a} -> Boolean;
        {b!=a} -> Boolean;
}
```



- Use patterns
- The **requires** operator is a low-level mechanism for checking whether a construct is valid C++.
  - It is essential for expressing low-level requirements but best avoided in high-level code where named concepts (often built using **requires**) are more comprehensible and maintainable.

Use patterns

- Concepts specify what a template must be able to do with its arguments
  - Not exactly what an argument type must be
- For example: in **requires(X a, Y b) {a+b;}**, the **+** in **a+b** could be provided as any of
  - **X operator+(X,Y);** *// if a is an X and b is a Y*
  - **X X::operator+(const Y&);** *// if a is an X and b is of a class derived from Y*
  - **Y operator+(const X&, const Y&);** *// if an X can be implicitly constructed from a Y and a Y can be implicitly constructed from a X*
  - **Y operator+(Y,X&);** *// if a Y can be implicitly constructed from a X and b is an X*
  - *… and many more …*
- That's important.
  - Handles mixed-mode arithmetic
  - Handles implicit conversions
  - Interface stability (as the definition of **+** changes)

Concept benefits

- Support good design
  - Like classes does
  - Usually, one level of abstraction higher
- Readability, maintainability
  - Overuse of unconstrained types (**auto** and **typename**) is a source of problems
- Enable massive improvements in error messages
- Overloading
  - Like functions, but much simpler
  - Different implementations/versions of the same idea can be represented by their common concept

Simplify design – example: conditional properties

- Concepts provide simple, elegant expression of conditions

```
template<typename T> class Ptr {
        // ...
        T* operator->() requires is_class_v<T>;          // offer -> (only) if T is a class
};

template<typename T, typename U> class Pair {
        // ...
        // a constructor (only) for types that can be converted to the members:
        template<convertible_to<T> TT, convertible_to<U> UU>
        Pair(const TT&, const UU&);
};
```

Modules

Finally!
Mentioned in D&E 1994

- Order dependence

```
#include "a.h"
#include "b.h"
```

- Can have a different meaning from

```
#include "b.h"
#include "a.h"
```

- **#include** is textual inclusion
- Implies
  - **#include** is transitive
  - much repeated compilation
  - Subtle bugs

- Modularity

```
import a;
import b;
```

- Same as

```
import b;
import a;
```

- **import** is not transitive
- Implies
  - Much compilation can be done once only

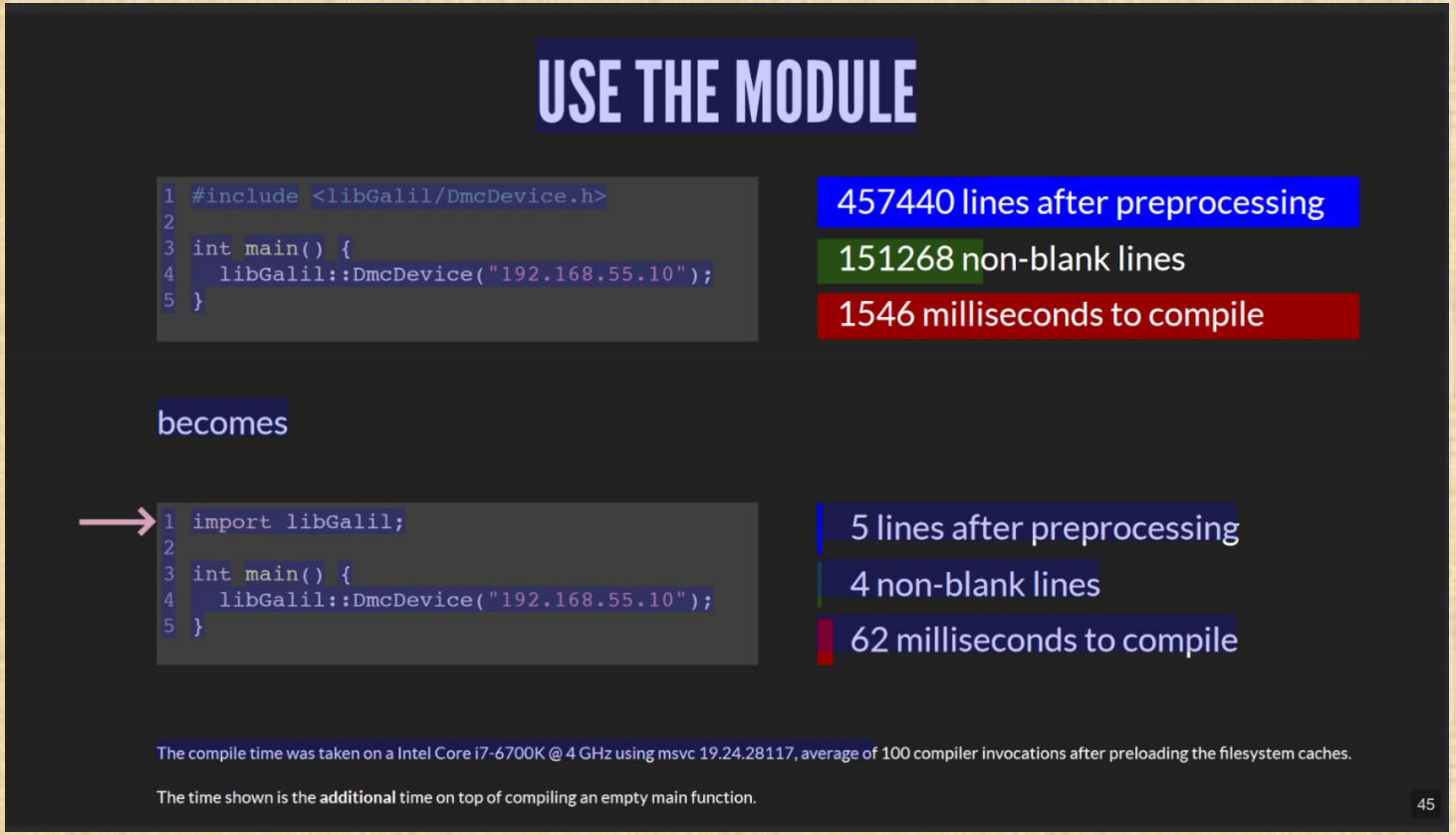
Modules

- Better code hygiene: modularity (especially protection from macros)
  - => Faster compile times (factors rather than percents)
  - Same source code organization for templates, classes, and functions

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

import iostream;
import containers;
using namespace std;

export                                  // this template is the only entity exported
template<Sequence S>
void print_map(const S& m) {
        for (const auto& [key,val] : m)       // access key and value
                cout << key << " -> " << val << '\n';
}
```

Imports are not transitive

Modules – not just for the standard library



- A 25 times speedup
  - Don't expect that in all cases
  - Daniela Engert: Contemporary C++ in Action

Watch this video.
It's dynamite

Do novel features matter?

- I haven’t seen resource leaks after basic testing for ye
- I once re-did a 1990s design to increase reliability
  - Not a toy: It had been in production use for 20+ years
  - Key technique: simplify!
  - The new design ***accidentally*** got 100 times fasters
- I don’t use anything more advanced than what you see in this talk
  - In production code
  - Experiments are less constrained
- The techniques are very widely applicable
  - Resource management
  - Error handling
  - Generic programming
  - …

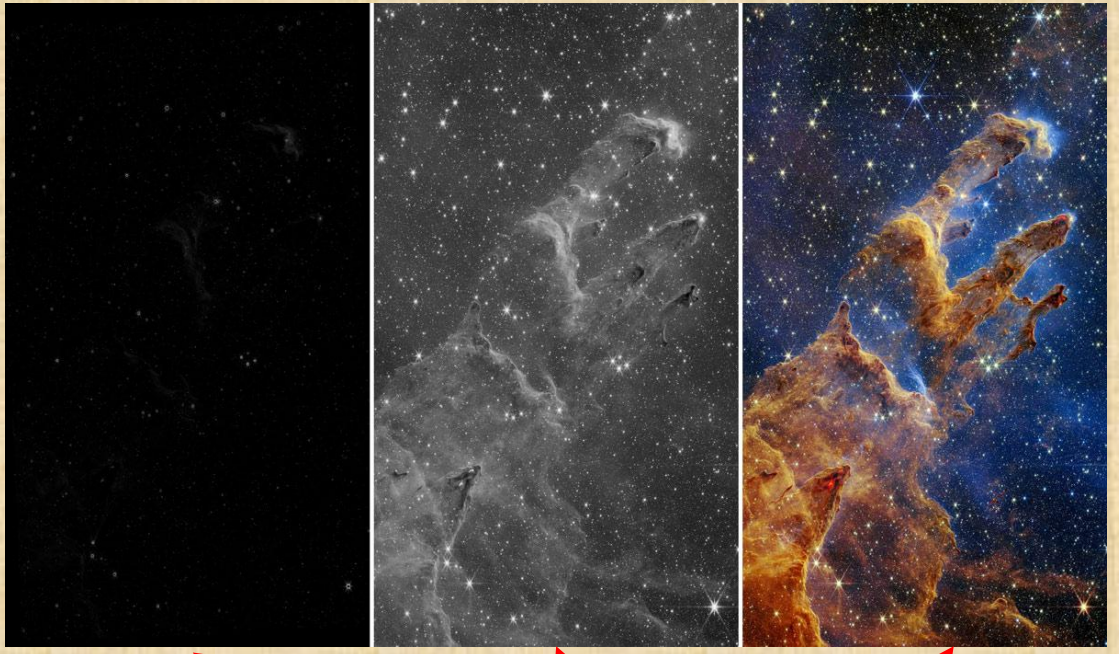
Concurrency

- For efficiency
  - Therefore, many different styles
  - Many optimization techniques
- Broad support
  - Threads and locks
  - Sharing
  - Parallel algorithms
  - Cooperative cancellation
  - Futures
  - Coroutines
  - …

Essential, but requires its own talk

Don’t be stuck in the 20th century

- Contemporary styles yield major benefits
  - In most cases
- Upgrading code is hard
  - But can be most beneficial
  - But can often be done incrementally
- New code doesn't have to follow outdated styles



But how?

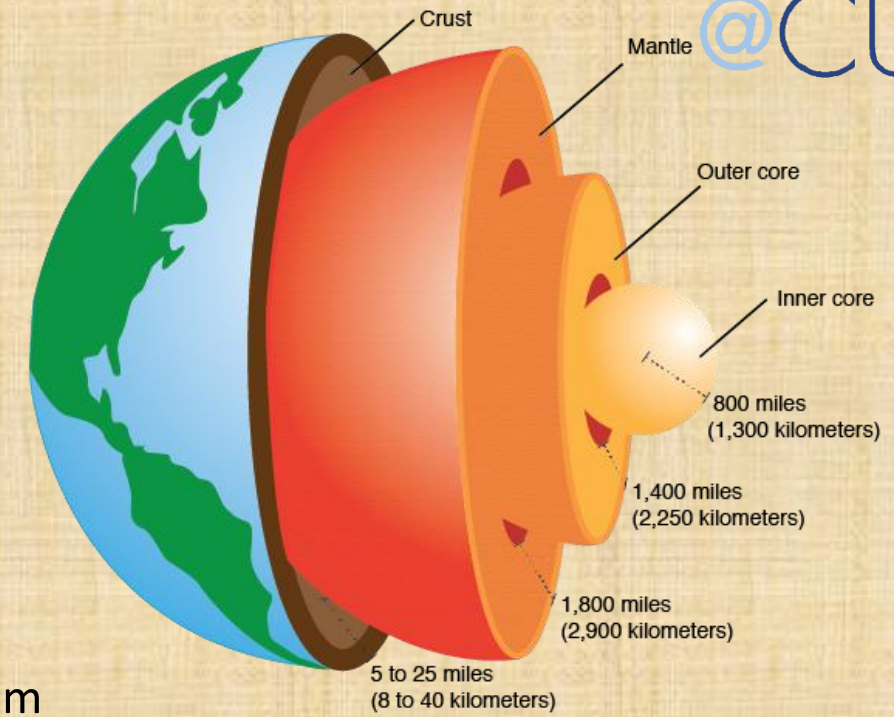
- Avoiding sub-optimal techniques is difficult
- Old code
- Old habits

Guidelines and enforcement

- We can’t change the language
  - We can change the way it is used
  - Stability/compatibility is a major feature
  - Gradual adoption of novel features and techniques is essential
- Guidelines
  - Individual rules can be used or not
  - Enforcement is incomplete
  - Available now: The C++ Core Guidelines
- Profiles
  - Enforced coherent sets of guidelines rules
  - Being worked on in WG21 and elsewhere – not yet available
  - Addresses technical debt
  - Address safety and simplicity concerns
  - Helps focus education

Core Rules

- Some people are not able to apply all rules
  - At least initially
  - Gradual adoption is very common
- Many people will need additional rules
  - For specific needs
- We initially focus on the core rules
  - The ones we hope that everyone eventually could benefit from
- The core of the core
  - No uninitialized variables
  - No range or nullptr violations
  - No leaks
  - No dangling pointers
  - No type violations through pointers
  - No invalidation



Example: don’t subscript pointers

- Instead, use an abstraction with enough information to range check
  - Requires run-time support

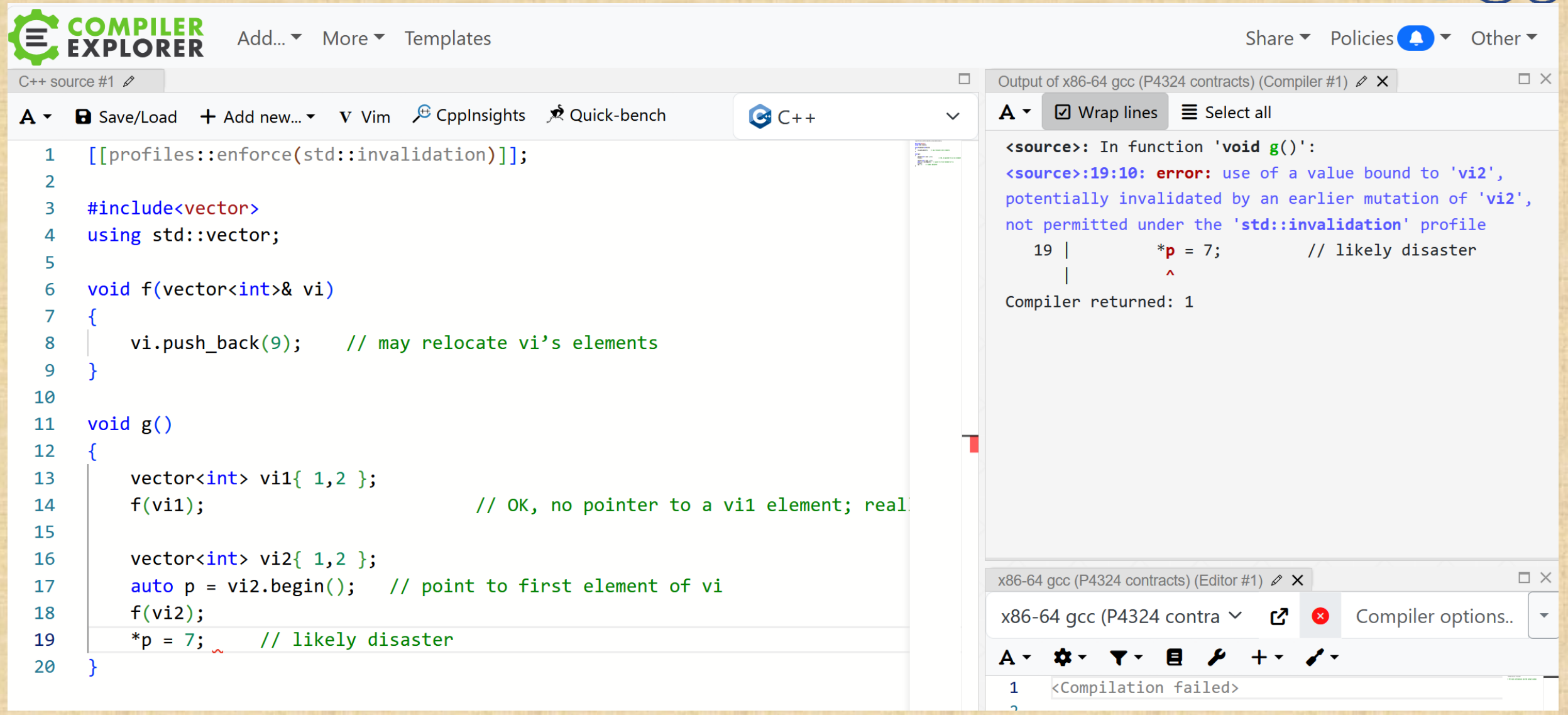
- Common style

```
void f(int* p, int n);

int a[100];
// …
f(a,100);       // OK? (depends on the meaning of n)
f(a,1000);      // likely disaster
```

- “Make simple things simple”
  - Simpler than “old style”
  - Shorter
  - At least as fast

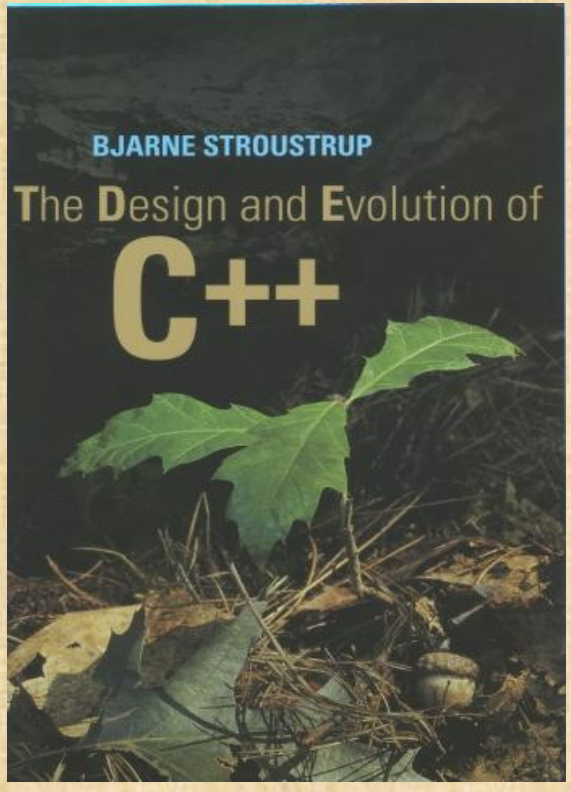
- Better

```
void f(span<int> s);

int a[100];
// …
f(span<int>{a});        // verbose
f(a);
f({a,1000});            // checkable
```

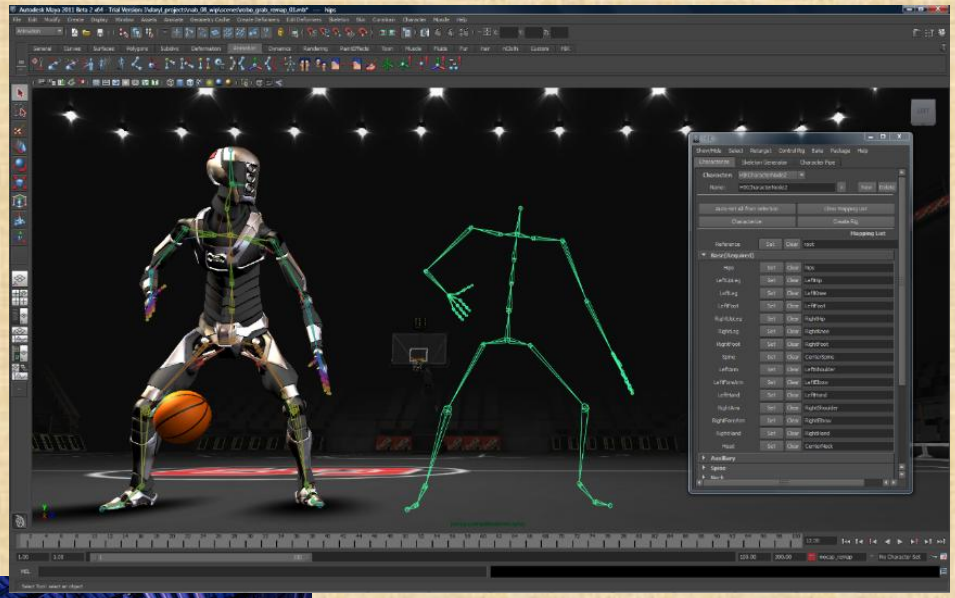

COMPILER EXPLORER
Add...
More
Templates
Share
Policies
Other
C++ source #1
Save/Load
Add new...
Vim
CppInsights
Quick-bench
C++
[[profiles::enforce(std::invalidation)]];
#include<vector>
using std::vector;
void f(vector<int>& vi)
{
vi.push_back(9); // may relocate vi's elements
}
void g()
{
vector<int> vi1{ 1,2 };
f(vi1); // OK, no pointer to a vi1 element; real
vector<int> vi2{ 1,2 };
auto p = vi2.begin(); // point to first element of vi
f(vi2);
*p = 7; // likely disaster
}
Output of x86-64 gcc (P4324 contracts) (Compiler #1)
Wrap lines
Select all
<source>: In function 'void g()':
<source>:19:10: error: use of a value bound to 'vi2', potentially invalidated by an earlier mutation of 'vi2', not permitted under the 'std::invalidation' profile
19 | *p = 7; // likely disaster
| ^
Compiler returned: 1
x86-64 gcc (P4324 contracts) (Editor #1)
x86-64 gcc (P4324 contra
Compiler options..
<Compilation failed>

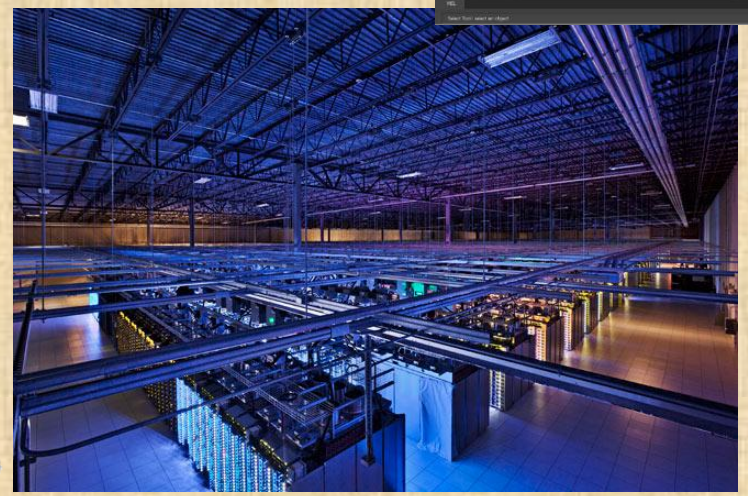
The C++ model

- Static type system
  - Equal support for built-in types and user-defined types
  - Value and reference semantics
- Systematic and general resource management (RAII)
- Efficient object-oriented programming
- Flexible and efficient generic programming
- Compile-time programming
- Direct use of machine and operating system resources
- Concurrency support through libraries (supported by intrinsics)
- Eventually eliminate the C preprocessor



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





References

- B. Stroustrup: 21st century C++ Blog@CACM January 2025
- B. Stroustrup: Concept-based Generic programming October 2025
- B. Stroustrup: A type-safety profile. February 2026.
- C++ Core Guidelines GitHub
- Profiles
  - B. Stroustrup: Safety Profiles: Type-and-resource Safe programming in ISO Standard C++
  - B. Stroustrup: Profiles syntax
  - B. Stroustrup: Profile invalidation - eliminating dangling pointers
  - Herb Sutter: Lifetime safety: Preventing common dangling
- Khalil Estell: C++ Exceptions for Smaller Firmware. CppCon 2024.
- Daniela Engert: Contemporary C++ in Action CppCon 2022.
  - A client server application for displaying video frames with timing constraints
- B. Stroustrup: A Tour of C++ (3rd Edition). Addison-Wesley. 2022.
- B. Stroustrup: Programming: Principles and Practice using C++. Addison-Wesley. 2023.
- B. Stroustrup's HOPL papers
  - A History of C++: 1979-1991. March 1993
  - Evolving a language in and for the real world: C++ 1991-2006. June 2007
  - Thriving in a Crowded and Changing World: C++ 2006–2020. June 2020.

Videos (https://www.stroustrup.com/videos.html)

- CppCon 2026 opening keynote: Profiles for Simplicity and Guarantees - Bjarne Stroustrup. September 2026.
- CppCon 2026 plenary: Language Designers Panel - Bjarne Stroustrup, Guido van Rossum, Mads Torgersen & Emma Tracey. September 2026.
- Ida Bechtle for cultRepo: C++ Documentary. The Story of C++: The World's Most Consequential Programming Language | The Official Story. Based on interviews with 14 people. June 2026.
- Italian C++ Community and the universities of Florence and Pisa: Concept-based Generic Programming and AMA. May 2026.
- Don't be too clever: Optimization through simplification. Short talk at the STAC (Strategic Technology Analysis Center) conference in New York City. May 2025.
- C++ as a 21st century language. Closing keynote of using std::cpp 2025 in Madrid. March 2025.
- Hrvoje Kukina Podcast #29: Bjarne Stroustrup: C++, Programming, Software Development, Philosophy. January 2025.
- Ryan Peterman interviews: Creator of C++: Bell Labs, Negative Overhead Abstraction, Mistakes | Bjarne Stroustrup . April 2026.
- Ryan Donovan for Stackoverflow: He designed C++ to solve your code problems. Podcast. February 2026.