A short introduction to the aims and status of modern C++

Bjarne Stroustrup
Morgan Stanley
Columbia University
www.stroustrup.com
Overview

• C++ aims and means
• C++20
  – Time to celebrate
• Concepts
• Modules
• Span
• Concurrency and parallelism
The value of a programming language is in the quality of its applications.
C++ community

• About 4.5 million developers (surveys)
  – Seems to be growing by 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, ...
C++ in two lines

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

• Zero-overhead abstraction
  – Classes, inheritance, templates, concepts, aliases, …
  – Future: Complete type- and resource-safety, concepts, modules, concurrency, …
C++ ideals/aims

• Write type-safe and resource-safe C++
  – No leaks
  – No memory corruption
  – No garbage collector
  – No limitation of expressibility
  – No performance degradation
  – ISO C++
  – Guaranteed: tool enforced (eventually)

• This cannot be done while allowing arbitrary code
  – C++ Core Guidelines: https://github.com/isocpp/CppCoreGuidelines
The onion principle

- Management of complexity
  - Make simple things simple!
- Layers of abstraction
  - The more layers you peel off, the more you cry
Engineering

- Principled and pragmatic
- Progress gradually guided by feedback
- There are always many tradeoffs
  - Choosing is hard
C++: stability and evolution

• Evolution is necessary
  – Newer features and techniques leads to simpler, safer, and faster code

• Stability/compatibility is a feature
  – Old code will run
  – But don’t repeat all the old mistakes

• Use C++ as a modern language
  – Type safe
    • Don’t mess with casts, raw pointers, void*, etc.
  – Resource safe
    • RAII
  – Modular
    • Avoid macros
    • Avoid #include

This will take time
Resource management: Constructors and destructors

```cpp
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"}; // vector of strings
    Vector<pair<string, jthread>> vp {{"t1", t1}, {{"t2", t2}}; // vector of {name,value} pairs
    // ...
} // memory, strings, and threads released here
```

Handle (rep)
Value (elements)
“the committee”
C++20 is here
C++20 trip reports

• “semi-official” on reddit
  – https://www.reddit.com/r/cpp/comments/f47x4o/202002_prague_iso_c_committee_trip_report_c20_is/

• Herb Sutter
  – https://herbsutter.com/

• Focused on the latest developments and features
  – Not on C++20 as a whole
  – Not on general principles
C++20

• Major language features
  – Modules
  – Concepts
  – Coroutines
  – Improved compile-time programming support
• Major standard-library components
  – Ranges
  – Dates
  – Formats
  – Parallel algorithms
  – Span

• Many minor language features and standard-library components
• A dense web of interrelated mutually-supporting features

By “major” I mean “changes how we think”
Generic programming: The backbone of the C++ standard library

- Containers
  - vector, list, stack, queue, priority_queue, ...
- Ranges
- Algorithms
  - sort(), partial_sort(), is_sorted(), merge(), find(), find_if(),...
  - Most with parallel and vectorized versions
- Concurrency support (type safe)
  - Threads, locks, futures, ...
- Time
  - time_points, durations, calendars, time_zones
- Random numbers
  - distributions and engines (lots)
- Numeric types and algorithms
  - complex
  - accumulate(), inner_product(), iota(), ...
- Strings and Regular expressions
- Formats
Generic Programming

• Write code that works for all suitable argument types
  – `void sort(R);`  // pseudo declaration
    • `R` can be any sequence with random access
    • `R`’s elements can be compared using `<`
  – `E* find_if(R,P);`  // pseudo declaration
    • `R` can be any sequence that you can read from sequentially
    • `P` must be a predicate on `R`’s element type
    • `E*` must point to the found element of `R` if any (or one beyond the end)

• That’s what the standard says
  – “our job” is to tell this to the compiler
  – C++20 enables that
Generic Programming

• Write code that works for all suitable argument types

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

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

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

A concept:
• Specifies what is required `r`’s type

Implicit:
• Type of container
• Type of element
• Number of elements
• Comparison criteria
Generic Programming

• Write code that works for all suitable argument types
  – Many/most algorithms have more than one template argument type
  – We need to express relationships among template arguments

```cpp
template<input_range R, indirect_unary_predicate<iterator_t<R> Pred>
  Iterator_t<R> ranges::find_if(R&& r, Pred p);

list<int> lsti;
// ... fill lsti ...
auto p = find_if(lsti, greater_than{7});

vector<string> vs;
// ... fill vs ...
auto q = find_if(vs, [](const string& s) { return has_vowels(s); });
```
Overloading

• Overloading based on concepts

  void sort(ForwardSortableRange auto&);
  void sort(SortableRange auto&);

  void some_code(vector<int> vec&, list<int> lst)
  {
    sort(lst);  // sort(ForwardSortableRange auto&)
    sort(vec)   // sort(SortableRange auto&)
  }

• We don’t have to say
  – “Sortable_range is stricter/better than ForwardSortableRange”
  – we compute that from their definitions

Design principles:
• Don’t force the user to do what a machine does better
• Zero overhead compared to unconstrained templates
Concepts

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

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

template<typename R>
concept Forward_sortable_range =
  forward_range<R> // has begin()/end(), ++; no [] or +
  && sortable<iterator_t<R>>; // can compare and swap elements
```

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

• A concept is a compile-time predicate
  – A function runs at compile time yielding a Boolean
  – One or more arguments
  – Can be built from fundamental language properties: use patterns

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

There are libraries of concepts
<concepts>: equality_comparable
Types and concepts

• A type
  – Specifies the set of operations that can be applied to an object
    • Implicitly and explicitly
    • Relies on function declarations and language rules
  – Specifies how an object is laid out in memory

• A **single-argument** concept
  – Specifies the set of operations that can be applied to an object
    • Implicitly and explicitly
    • Relies on use patterns
      – reflecting function declarations and language rules
  – Says nothing about the layout of the object

Ideal:
Use concepts where we now use types, except for defining layout
Generic Programming is “just” programming

• Why?
  – From 1988 to now “template programming” and “ordinary programming” have been very different
    • Different syntax
    • Different look-up rules
    • Different source code organization
    • “Expert friendly” programming techniques
  – We don’t need two different sets of techniques (and notations)
    • Unnecessary complexity
    • Make simple things simple!
  – “ordinary programming” is expressive and familiar
Generic Programming

• will change the way we think about Programming
Modules and transition

• Source organization
• Header file conversion
  – Header and module coexistence
• Build systems
  – Build2
  – Cmake prototype
Modules

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

```cpp
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';
    }
```
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
Compile speeds

The compile time was taken on an Intel Core i7-6700K @ 4 GHz using msvc 19.24.28117, average of 100 compiler invocations after preloading the filesystem caches. The time shown is the additional time on top of compiling an empty main function.
Coroutines

generator<int> fibonacci() // generate 0,1,1,2,3,5,8,13 ...
{
    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';
}
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
Span

• Non-owning potentially run-time checked reference to a continuous sequence
  
  ```cpp
  int a[100];
  span s {a}; // note: template argument deduction
  for (auto x : s) cout << x << '\n';
  ```

• From the GSL

• On GitHub

Neil Macintosh
Concurrency and parallelism

• C++20
  – Atomics
  – Lock-free programming
  – Fences and barriers
  – Type-safe Posix/windows thread, mutex, etc.
  – Future and promise
  – Parallel algorithms
  – Coroutines (synchronous and asynchronous)

• C++23?
  – Executor model (readers and writers, push and pull)
    • We have consensus, but not in time for C++20
  – Networking (now a TS)
    • Depends on executors
  – Standard library support for coroutines
    • Implementation likely to depend on executors
What we want

• **Ease of programming**
  – Writing correct concurrent code is hard
  – Modern hardware is concurrent in more ways than you imagine

• **Uncompromising performance**
  – But for what?

• **Portability**
  – Preferably portable performance

• **System level interoperability**
  – C++ shares threads with other languages and with the OSs
Threading

• A thread represents a system’s notion of executing a task concurrently with other tasks

• You can
  – start a task on a thread
  – wait for a thread for a specified time
  – control access to some data by mutual exclusion
  – control access to some data using locks
  – wait for an action of another task using a condition variable
  – return a value from a thread through a future
Future and promise

future+promise provides a simple way of passing a value from one thread to another

- No explicit synchronization
- Exceptions can be transmitted between threads
Parallel algorithms

- algorithms giving the option of parallel and/or vectorised execution of standard-library algorithms
  - e.g., `sort(par,b,e)` and `sort(unseq,b,e)`
- All the traditional STL algorithms, e.g., `find(seq,b,e,x)`,
  - but no `find_all(par,b,e,x)` or `find_any(unseq,b,e,x)`
- parallel algorithms:
  - `For_each`
  - `Reduce` // parallel accumulate
  - `Exclusive scan`
  - `Inclusive scan`
  - `Transform reduce`
  - `Transform exclusive scan`
  - `Transform inclusive scan`
  - ...

Stroustrup - C++20 - Barcelona 2020
**SIMD vector** (in parallelism TS)

- `simd` is a data-parallel type. The width of a given `simd` is a constant expression.

```cpp
template<class T, class Abi> // Abi is size, simd is single dimensional
class simd {
public:
    simd() noexcept = default
    template<class U> simd(U&& value) noexcept;
    template<class U> simd(const simd<U, simd_abi::fixed_size<size()>**)& noexcept;
    // no copy constructor or copy assignment
    template<class G> explicit simd(G&& gen) noexcept;
    template<class U, class Flags> simd(const U* mem, Flags f);
    template<class U, class Flags> copy_from(const U* mem, Flags f);
    template<class U, class Flags> copy_to(U* mem, Flags f);
    reference operator[](size_t);value_type operator[](size_t) const;
    // unary @, binary @, and @= for all @ where it makes sense
};
```

[Link](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/n4796.pdf)
C++23 – we have a plan

• Top priorities:
  – Library support for coroutines
  – A modular standard library
  – Executors
  – Networking

• Also make progress on
  – Reflection
  – Pattern matching
  – Contracts

• After that
  – Everything else
C++

• C++20
  – Competed February 15, 2020
  – Most features shipping somewhere
  – Expected: essentially all features shipping by all major vendors in 2020
  – Is going to make a major difference to the way we think and program
  – Compatible / stable

• Use C++ as a modern language
  – Aim for complete type-safety and resource-safety
  – Enforce coding guidelines
Notable features in C++20

- **Modules**.
- **Coroutines**.
- **Concepts**.
- **Ranges**.
- Compile-time programming support:
  - `constinit`, `consteval`
  - `is_constant_evaluated`
  - `constexpr` allocation, `vector`, `string`, `union`, `try and catch`, `dynamic_cast` and `typeid`
- `format("For C++{}", 20)`.
- `operator<=>`,
- **Feature test macros**.
- `std::span`.
- **Synchronized output**.
- `source_location`.
- `atomic_ref`.
- `atomic::wait`, `atomic::notify`, `latch`, `barrier`, `counting_semaphore`, etc.
- `jthread` and `stop_*`. 