CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

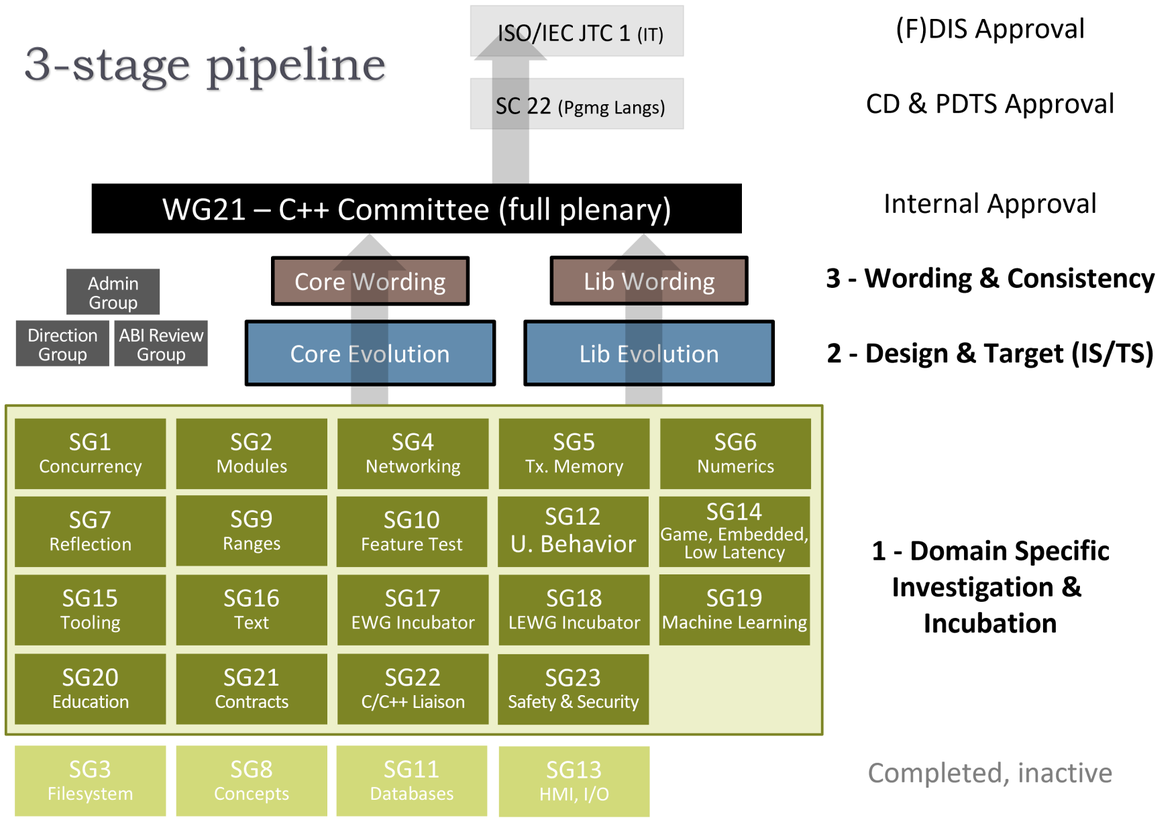
Advanced C++ Course

Future C++

CSCS

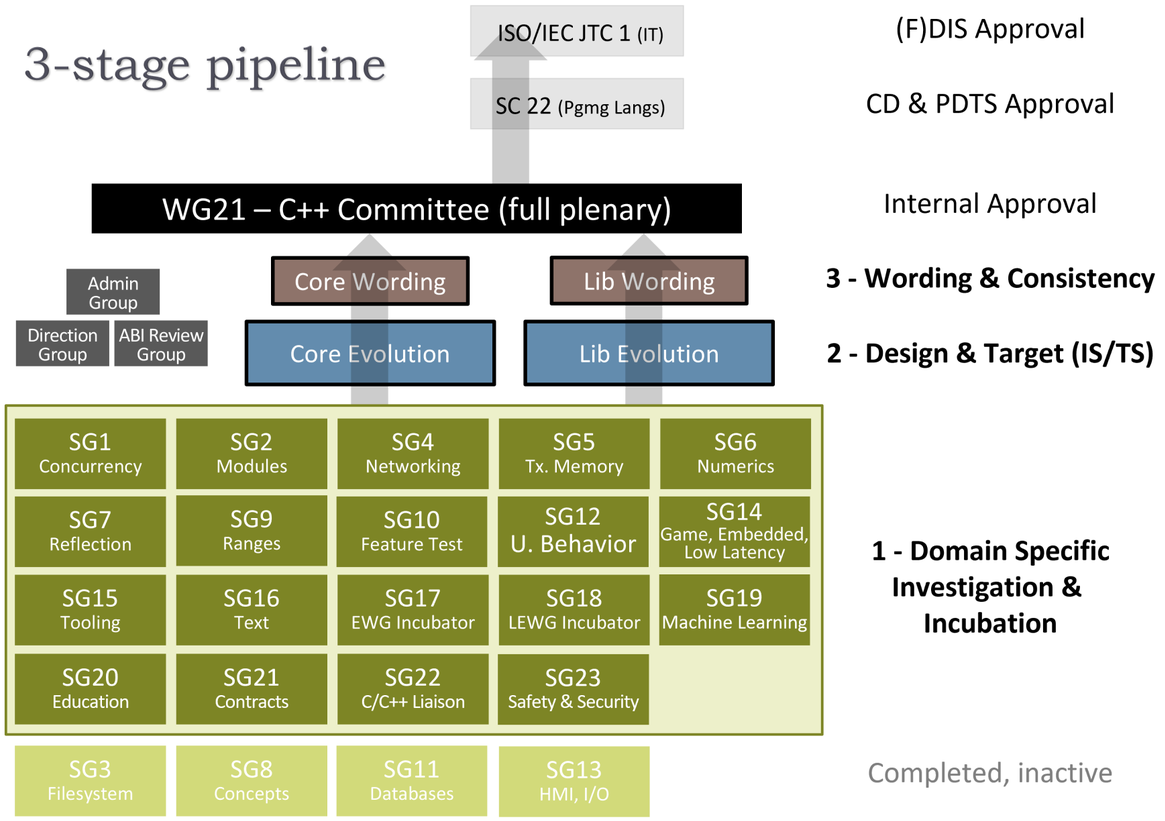
C++ standardization process

- C++ standardized under the ISO in "Working Group 21" (WG21)
  - C is WG14
- From proposal to standardization:
  - "Study groups" evaluate proposals, discuss merits, improve design
  - "(Library) Evolution Working Group" evaluates how well the design fits C++ as a whole
  - "Core/Library Working Group" evaluates wording
  - Full committee vote with "National Bodies" (NB)



C++ standardization process

- More information
- Draft of the standard available at https://eel.is/c++draft
- Proposals available at https://www.open-std.org/jtc1/sc22/wg21/docs/papers/
- https://wg21.link/P0000 takes you to the latest revision of proposal P0000



Moving to newer standards

- C++23 recently finalized and compiler support is also getting finalized, yet many are still stuck with C++17
- Easier than ever to use newer compilers, but even with system compilers getting C++20 support is now relatively easy
- Biggest obstacle to upgrading is NVIDIA compilers
  - From experience: if you can separate your CUDA kernels from your regular source files you will have an easier time moving to newer standards
- Sticking with GCC and clang will get you furthest (honorable mention to MSVC which has been improving rapidly lately)
- Following best practices with CI helps make upgrades as painless as possible
  - Testing with warnings (and errors) enabled with *multiple* compilers helps catch problems early
- Excellent resource for checking compiler support: https://en.cppreference.com/w/cpp/compiler_support

What's new in C++?

- The past sessions have covered advanced C++ topics up to C++23
- Features that didn't fit previous sessions, but are good to know about:
  - modules
  - `std::format` and `std::print`
  - `std::expected`
- Features useful for HPC that are targeted for C++26:
  - `std::simd`
  - `std::execution`
  - `std::linalg` + `std::mdspan`

Modules

- Problem: header includes error prone and slow
  - "Stateful" headers possible with macros
  - Parsing transitive headers may be very expensive (https://github.com/s9w/cpp-lit): simply enabling C++20 may increase compilation times!
- C++ modules (C++20) moves away from textual inclusion to a model where exports are explicit, macros don't leak, and build times aren't insane (though they can still be insane because of templates)
- GCC 14, clang 16, and CMake ~3.26 are starting to have usable (but experimental) support for modules
  - https://godbolt.org/z/von5MfK7T
  - CMake 3.28 (unreleased) will have non-experimental support for modules: https://gitlab.kitware.com/cmake/cmake/-/issues/18355
- The modules `std` and `std.compat` were added in C++23

```
import std;
auto main() -> int { std::println("Hello, world!"); }
```

`std::format` and `std::print`

- C's `printf` : convenient but not type safe
- C++'s iostreams: clunky and stateful but type safe
- What if we could have both (and go a bit further)?
  - C++20 introduced `std::format`
  - C++23 introduced `std::print` and `std::println`
  - https://godbolt.org/z/MqKPxzzxM

```
std::print(
    "Hello {}!\npadding: {:#08x}\nalignment: {:>30}\nprecision: {:.2f}\n",
    "Bjarne", 42, "short", std::numbers::pi_v<float>);
```

  - user-generated `static_assert` messages: https://wg21.link/p2741
- What to do until C++23 is well supported?
  - Use fmt

`std::expected`

- C++ lacks a language level multi-return type
  - store either result in case of **success**
  - or store error information in case of **failure**

Workarounds

- C style flags: `int fun(args..., Result& r)`
  - need to construct `Result` type before (assignable)
  - cumbersome and error-prone to check
  - limited error information
  - not composable
- `std::optional`
  - is *value-or-nothing* (no error information)
- exceptions
  - undesired (inversion of control flow)
  - disabled for certain hardware/applications
  - expensive to unwind
  - error-prone (forget to check)

`std::expected<T, E>` : Overview

- Introduced in C++23
- similar to a `union` / `std::variant` behind the scenes
- no dynamic memory allocation
- *value-or-error* semantics (never empty)
- `T` is the `value_type` which is the *expected* type
- `E` is the `error_type` which is the *unexpected* type
- explicit access of result (no implicit converison)
- access of *value* when result is *unexpected*: throw exception
- access of *error* when result is *expected*: throw exception

`std::expected<T, E>` : Basic Usage

Construction (by callee)

```
auto foo(std::string const& s) noexcept -> std::expected<double, std::string> {
    // wrap C-style parser for example
    double r;
    if (parse(s, &r))
        return r;
    else
        return std::unexpected(std::string{"could not parse"});
}
```

- interface familiar from `std::optional`
- use `std::unexpected<E>` to represent unexpected value (`E` == `std::string` here)

Operations (by caller)

```
if (auto r =foo(s); r.has_value())
    std::cout << r.value() << "\n";
else
    std::cout << r.error() << "\n";
```

- explicit access of value/error
- `value_or(U other)` akin to `std::optional`

`std::expected<T, E>` : Monadic Operations (map, bind)

- named after a concept from category theory (familiar from haskell)
- increase composability (functional programming style)
  - signature `expected<T1,E1>::op(F&& f) -> expected<T2, E2>`
  - apply function `f` on the current result (pass `*this` as argument to `f`)

result	`F: T1 -> T2` `G: E1 -> E2`	`F: T1 -> expected<T2, E1>` `G: E1 -> expected<T1, E2>`	return type
expected	`transform(F&&)`	`and_then(F&&)`	`expected<T2, E1>`
unexpected	`transform_error(G&&)`	`or_else(G&&)`	`expected<T1, E2>`

- extended example at https://godbolt.org/z/dEGYa6fGW

std::simd

- SIMD: single instruction multiple data
- Modern processors: wide registers with machine instructions acting on the multiple data
- SSE, SSE2, AVX, AVX2, AVX512, SVE, SVE2
- Auto-vectorization sometimes not good enough
- Requires hardware-specific code, usually compiler intrinsics

```
// AVX: Compute 2.0^2
// Arbor's implementation, https://github.com/arbor-sim/arbor/blob/master/arbor/include/arbor/simd/avx.hpp
__m256d exp2int(__m128i n) {
    n = _mm_slli_epi32(n, 20);
    n = _mm_add_epi32(n, _mm_set1_epi32(1023<<20));

    auto nl = _mm_shuffle_epi32(n, 0x50);
    auto nh = _mm_shuffle_epi32(n, 0xfa);
    _mm256_insertf128_si256(_mm256_castsi128_si256(nl), nh, 1);

    return _mm256_castps_pd(
        _mm256_blend_ps(_mm256_set1_ps(0),
        _mm256_castsi256_ps(nhnl),0xaa));
}
```

- Many libraries for abstracting hardware specifics
  - vir::stdx::simd
  - Agner Fog's Vector Types
  - E.V.E.
  - xsimd
  - Vc
  - highway
  - kokkos

- Standardization efforts for C++26
  - std::experimental::simd (data parallelism TS v2)
  - std::(experimental::)simd implementation from Intel in progress (permutation operations and support for complex numbers)
  - std::simd prototyping https://github.com/mattkretz/simd-prototyping/
  - https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2803r0.pdf

`std::simd`

- in `std::experimental` (gcc: since v11) for now
- `std::simd<T, Abi>`
  - behaves mostly like object of type `T` but operates on multiple values (element-wise)
  - Abi (template-template parameter) determines vector width (`size`) and ABI (i.e. how parameters are passed to functions)
    - `std::simd_abi::native<T>` : most efficient for given hardware
    - `std::simd_abi::fixed_size<T, N>` : user-defined width

Scalar

```
double f(double x) {
    return x * 2.;
}
```

SIMD

```
std::simd<double> f(std::simd<double> x) {
    return x * 2.;
}
```

`std::simd` : Loop parallelization

```
void f(std::vector<float>& data) {
    using V = std::native_simd<float>;
    std::size_t i = 0
    for (; i + V::size() <= data.size(); i += V::size()) {
        v(&data[i], std::element_aligned);
        v = std::sin(v);
        v.copy_to(&data[i], std::element_aligned);
    }
    for (; i < data.size(); ++i) {
        data[i] = std::sin(data[i]);
    }
}
```

select simd type/Abi

loop in `V::size()` increments
load data (pay attention to alignment `memory_alignment_v<V>`)
store data

You often need an “epilogue”

- efforts to integrate simd with parallel algorithms (P0350)

```
std::transform(std::execution::simd, data.begin(), data.end(), data.begin(),
    [](auto x) {
        return std::sin(x);
    });
```

`std::simd` : Generic enough?

```
template<typename T>
T f(T x) {
    return x * 2.;
}
```

- works, but:
  - be careful with constants (`2.f` insted of `2.`)
  - what about conditionals?

```
template <typename T>
T f(T x) {
    std::where(x > 0.f, x) *= 2.f;
    return x;
}
```

- `std::where`
  - takes a `std::simd_mask<T, Abi>` as first argument (data-parallel type with the element type bool)
  - returns a `std::where_expression<simd_mask, simd>` : abstracts the notion of selected elements
  - works for scalars, as well

`std::simd` : Conclusions

- SIMD width is selected at compile time (Abi)
- Speed-up is often a factor of `size()` , but may be less, depending on hardware details
- Requires refactoring of code (loops, elimination of conditionals, introduction of `where` expressions)
- P0350 may help for simple cases

`std::execution`

- Until now: parallel algorithms (CPU only)
- Third party vendor solutions:
  - Thrust (CPU, NVIDIA, AMD)
  - nvhpc (NVIDIA)
  - SYCL (CPU, NVIDIA, AMD)
- Other third party libraries:
  - Kokkos
  - Alpaka
- `std::execution` aims to put generic building blocks in C++ standard

`std::execution` hello world

```
using namespace std::execution;

scheduler auto sch = thread_pool.scheduler();

sender auto begin = schedule(sch);
sender auto hi = then(begin, []{
    std::cout << "Hello world! Have an int.";
    return 13;
});
sender auto add_42 = then(hi, [](int arg) { return arg + 42; });

auto [i] = this_thread::sync_wait(add_42).value();
```

`std::execution`

- Performance portable building blocks
- *Senders* represent work
- *Schedulers* represent where work runs
- *Algorithms* represent what work to do
- Reference implementation can already be used today: stdexec
- CSCS developing pika: builds functionality on top of `std::execution`
- Targeted for C++26
- Proposal: https://wg21.link/p2300
- stdexec is available on compiler explorer: https://godbolt.org/z/T3MqhPGex

`std::execution` : not only for asynchrony

- Schedulers (executors) finally get us a step closer to heterogeneous execution of parallel algorithms
- Blocking overloads of parallel algorithms much simpler to reason about
- Proposal: https://wg21.link/p2500

```
std::for_each(
    std::execute_on(scheduler, std::execution::par),
    begin(data),
    end(data),
    f);
```

`std::linalg`

- Decades of existing practice in BLAS
- No more ZGERC, instead `matrix_rank_1_update_c(std::par, x, y, A)`
- Works with `std::mdspan` as inputs
- Execution policies for parallelization
- GPU support dependent on `std::execution` support for parallel algorithms
- Targeted for C++26
- Only covers BLAS, not LAPACK
- Proposal: https://wg21.link/p1673

`std::linalg` cholesky

```
template<in-matrix InMat, class Triangle, in-vector InVec, out-vector OutVec>
void cholesky_solve(InMat A, Triangle t, InVec b, OutVec x)
{
  using namespace std::linalg;
  if constexpr (std::is_same_v<Triangle, upper_triangle_t>) {
    // Solve Ax=b where A = U^T U
    // Solve U^T c = b, using x to store c.
    triangular_matrix_vector_solve(transposed(A), opposite_triangle(t), explicit_diagonal, b, x);
    // Solve U x = c, overwriting x with result.
    triangular_matrix_vector_solve(A, t, explicit_diagonal, x);
  } else { /* ... */ }
}
```

`std::execution` and `std::linalg` use case: DLA-Future

- DLA-Future: distributed linear algebra built on what is currently being standardized
  - https://github.com/eth-cscs/DLA-Future
- `std::execution` for
  - asynchrony (senders)
  - heterogeneous execution (CPU, CUDA, HIP)
  - currently covered by stdexec and pika
- `std::linalg` for
  - BLAS and LAPACK
  - currently covered by cuBLAS, rocBLAS, cuSOLVER, rocSOLVER, and DLA-Future because DLA-Future needs asynchronous versions of `std::linalg`
- networking
  - currently covered by MPI and pika

`std::execution` and `std::linalg` use case: DLA-Future

```
using Memory = CPU;
auto sched = cpu_scheduler;

dlaf::Matrix<T, CPU> m1;
dlaf::Matrix<T, CPU> m2;

sender auto a = dlaf::comm::recv_tile(ij);
sender auto b = m1.read(ij);
sender auto c = m2.readwrite(ij);

sender auto s = std::execution::when_all(a, b, c) |
    on(cpu_scheduler, dlaf::general_multiplication) |
    std::execution::then([]() { std::print("matrix-matrix multiplication done\n"); });
std::this_thread::sync_wait(s);
```

Thank you!