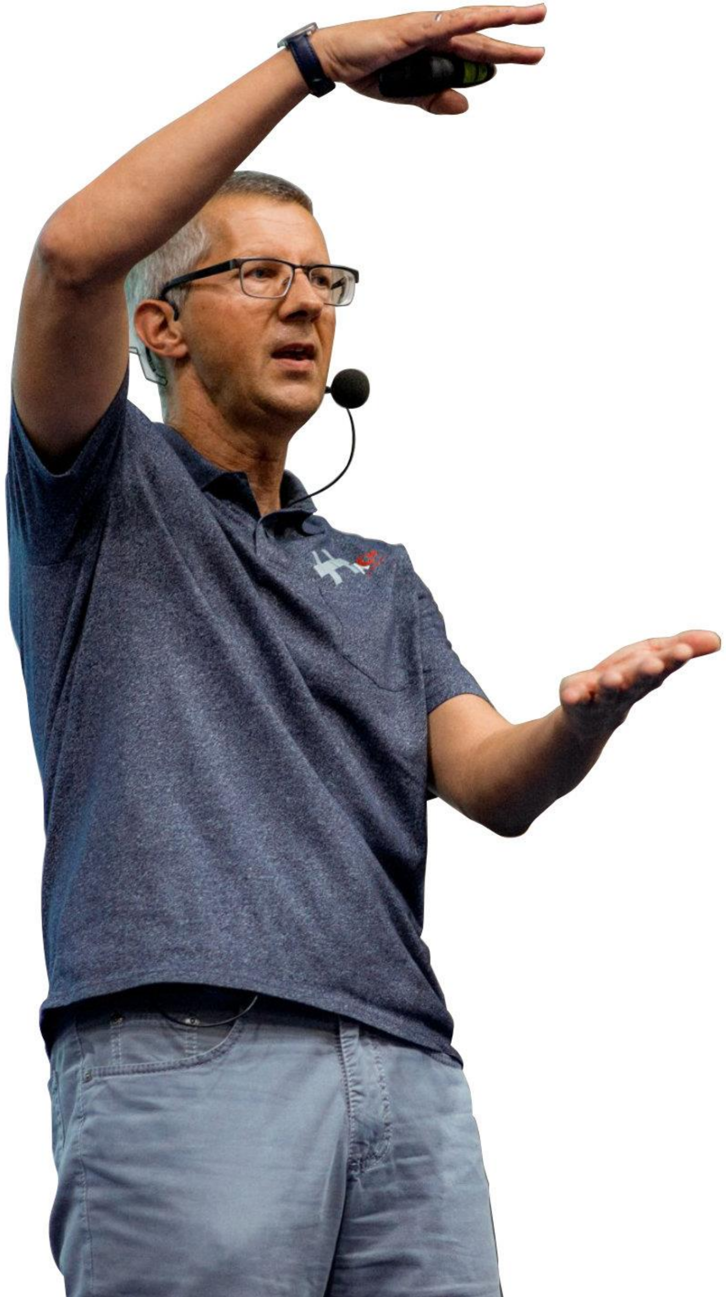


C++26:

An Overview

Rainer Grimm

Training, Mentoring, and
Technology Consulting

C++26

Core Language

- Reflection
- Contracts
- Placeholder
- `static_assert` extension
- Template improvements
- `delete` with reason

Library

- `string` and `string_view`
- Format extensions
- `std::inplace_vector`
- Range improvements
- `constexpr` extensions
- Linear algebra support
- `std::submdspan`
- Debugging support

Concurrency

- `std::execution`

C++26

Core Language

- Reflection
- Contracts
- Placeholder
- `static_assert` extension
- Template improvements
- `delete` with reason

Library

- `string` and `string_view`
- Format extensions
- `std::inplace_vector`
- Range improvements
- `constexpr` extensions
- Linear algebra support
- `std::submdspan`
- Debugging support

Concurrency

- `std::execution`

Reflection

Reflection is the ability of a program to examine, introspect, and modify its structure and behavior.

```cpp
int main() {
    constexpr auto r = ^^int;
    typename[:r:] x = 42;         // Same as: int x = 42;
    typename[:^^char:] c = '*';   // Same as: char c = '*';

    static_assert(std::same_as<decltype(x), int>);
    static_assert(std::same_as<decltype(c), char>);
    assert(x == 42);
    assert(c == '*');
}
```

- **`^^`: Reflection Operator** creates a reflection value from its operand (`^^int` and `^^char`)
- **`[:refl:]`: Splicer** creates a grammatical element from a reflection value (`[:r:]` and `[:^^char:]`)
- **`Reflection Value`** is a representation of program elements as a constant expression

Reflection

- Reflection
  - Proposal [P2996R5]
  - is a minimal viable product
  - supports many metafunctions

- Metafunctions
  - are declared `consteval`
  - accept the reflection type `std::meta::info`

- Reflection Operator (`^^`)
  - creates `std::meta::info`

[daveed.cpp]
[getSize.cpp]

Contracts

A **contract** specifies interfaces for software components in a precise and checkable way.

- The software component are functions and methods that must fulfill preconditions, postconditions, and invariants.
  - A **precondition**: a predicate that is supposed to hold upon entry in a function.
  - A **postcondition**: a predicate that is supposed to hold upon exit from the function.
  - An **assertion**: a predicate that is supposed to hold at its point in the computation.
- Contracts are based on the proposal P2961R2.

Contracts

```
int f(int i)
    pre (i >= 0)
    post (r: r > 0) {
    contract_assert (i >= 0);
    return i+1;
}
```

`pre` and `post`

- adds a precondition (postcondition). A function can have an arbitrary number of preconditions (postconditions). They can be intermingled arbitrarily.
- are contextual keywords
- are positioned at the end of the function declaration

`post`

- can have a return value. An identifier must be placed before the predicate, followed by a colon.

`contract_assert`

- is a keyword. Otherwise, it could not be distinguished from a function call.

[contract.cpp]

Placeholders

Placeholders are a nice way to highlight variables that are no longer needed.

Placeholder

- is the underscore(_)
- can be used as often as you like
- does not emit a warning when not used
- is frequently used in Python

[placeholder2.cpp]

`static_assert` extension

Syntax of `static_assert`

- C++11: `static_ assert(compile time predicate, unevaluated string)`
- C++17: `static_ assert(compile time predicate)`
- C++26: `static_ assert(compile time predicate, user-defined type)`
  - the user-defined type must have the following properties:
    - has a `size()` method that produces an integer
    - has a `data()` method that produces a pointer of character type such that
    - the elements in the range `[data(), data()+size())` are valid. (p2741r3)

static_assert26.cpp

Template Improvements

Pack Indexing enables the index access on parameter packs.

Pack indexing

- May be your favorite template improvement if you are template metaprogramming friend
- is based on the proposal P2662R3

packIndexing.cpp

`delete` with Reason

With C++26, you can specify a reason for your `delete`.

- `delete` with reason
  - will become best practice
  - is based on the Proposal [p2573r2]

[deleteReason.cpp]

C++26

Core Language

- Reflection
- Contracts
- Placeholder
- `static_assert` extension
- Template improvements
- `delete` with reason

Library

- `string` and `string_view`
- Format extensions
- `std::inplace_vector`
- Range improvements
- `constexpr` extensions
- Linear algebra support
- `std::submdspan`
- Debugging support

Concurrency

- `std::execution`

string and string_view

- Testing for success or failure of <charconv> functions
  - to_chars and from_chars was inconvenient to test: if(res.ec == std::errc{})
  - res can be directly compared with bool: if(res)

- Interfacing stringstreams with std::string_view

```
// implicitly convertable to string_view
const mystring str;

stringstream s1(""sv);
stringstream s1(str);
s2.str(""sv);
```

- Concatenation of strings and string views

```
std::string calculate(std::string_view prefix)
{
  return prefix + get_string(); // NO ERROR
}
```

string and string_view

- Arithmetic overloads of `std::to_string` and use `std::format`

```cpp
auto loc = std::locale("uk_UA.UTF-8");
std::locale::global(loc);
std::cout.imbue(loc);
setlocale(LC_ALL, "C");

std::cout << "iostreams:\n";
std::cout << 1234 << "\n";
std::cout << 1234.5 << "\n";

std::cout << "\nto_string:\n";
std::cout << std::to_string(1234) << "\n";
std::cout << std::to_string(1234.5) << "\n";

setlocale(LC_ALL, "uk_UA.UTF-8");

std::cout << "\nto_string (uk_UA.UTF-8 C locale):\n";
std::cout << std::to_string(1234) << "\n";
std::cout << std::to_string(1234.5) << "\n";
```

```
iostreams:
1 234
1 234,5

to_string:
1234
1234.500000

to_string (uk_UA.UTF-8 C locale):
1234
1234,500000
```

```cpp
std::cout << std::format(std::locale{"uk_UA.UTF-8"}, "{:L}", 1234.5) << '\n';
```

`std::inplace_vector`

`std::inplace_vector`

- dynamically-resizable vector with compile-time fixed capacity
- contiguous embedded storage in which the elements are stored within the vector object itself
- drop-in replacement for `std::vector`

- When `std::inplace_vector?` (P0843R8)
  - memory allocation is not possible
  - memory allocation imposes an unacceptable performance penalty
  - allocation of objects with complex lifetimes in the static-memory segment is required
  - `std::array` is not an option, e.g., if non-default constructible objects must be stored
  - a dynamically-resizable array is required within `constexpr` functions
  - the storage location of the `inplace_vector` elements is required to be within the `inplace_vector` object itself (e.g. to support `memcpy` for serialization purposes)

`std::format`

- Pointers
  - Before C++26, only `void`, `const void`, and `std::nullptr_t` pointer types are valid.
  - If you want to display the address of an arbitrary pointer, you must cast it to `(const) void*`.
- Newline
  - `println()`

Ranges Improvements

The ranges library will get new functions:

- `std::ranges::generate_random`
- `std::ranges::concat_view`

- `std::ranges::generate_random(fltArray, g, d)`
  - uses the generator `g` and the distribution `d` to create the random numbers
  - is equivalent to the following loop

```
for(auto& el : fltArray)
el = d(e);
```

`constexpr` Extensions

More algorithm become `constexpr`

- `std::stable_sort`
- `std::stable_partition`
- `std::inplace_merge`
- This is also true for their counterparts in the ranges library.

Linear Algebra Support

`<linalg>` is a free function linear algebra interface based on the BLAS.

- BLAS: **B**asic **L**inear **A**lgebra **S**ubprograms is a specification that prescribes a set of low-level routines for performing common linear algebra operations
  - vector addition
  - scalar multiplication
  - linear combinations
  - matrix multiplication
- These operations are the de facto standard low-level routines for linear algebra libraries.

std::submdspam

std::submdspan

```
template<class T, class E, class L, class A,
         class ... SliceArgs)
auto submdspan(mdspan<T,E,L,A> x, SliceArgs ... args);
```

```
int* ptr = ...;
int N = ...;
mdspan a(ptr, N);

// subspan of a single element
auto a_sub1 = submdspan(a, 1);
static_assert(decltype(a_sub1)::rank() == 0);
assert(&a_sub1() == &a(1));

// subrange
auto a_sub2 = submdspan(a, tuple{1, 4});
static_assert(decltype(a_sub2)::rank() == 1);
assert(&a_sub2(0) == &a(1));
assert(a_sub2.extent(0) == 3);
```

```
// subrange with stride
auto a_sub3 = submdspan(a, strided_slice{1, 7, 2})
static_assert(decltype(a_sub3)::rank() == 1);
assert(&a_sub3(0) == &a(1));
assert(&a_sub3(3) == &a(7));
assert(a_sub3.extent(0) == 4);

// full range
auto a_sub4 = submdspan(a, full_extent);
static_assert(decltype(a_sub4)::rank() == 1);
assert(a_sub4(0) == a(0));
assert(a_sub4.extent(0) == a.extent(0));
```

Debugging Support

C++26 has three functions to deal with debugging.

- `std::breakpoint:` pauses the running program when called and passes the control to the debugger
- `std::breakpoint_if_debugging`: calls `std::breakpoint` if `std::is_debugger_present` returns `true`
- `std::is_debugger_present`: checks whether a program is running under the control of a debugger

C++26

Core Language

- ☐ Reflection
- ☐ Contracts
- ☐ Placeholder
- ☐ `static_assert` extension
- ☐ Template improvements
- ☐ `delete` with reason

Library

- ☐ `string` and `string_view`
- ☐ Format extensions
- ☐ `std::inplace_vector`
- ☐ Range improvements
- ☐ `constexpr` extensions
- ☐ Linear algebra support
- ☐ `std::submdspan`
- ☐ Debugging support

Concurrency

- ■ `std::execution`

std::execution

`std::execution` provides "*a Standard C++ framework for managing asynchronous execution on generic execution resources*". (P2300R10)

- `std::execution`
  - previously known as executors or senders/receivers
  - stdexec is the reference implementation of this proposal. It is a complete implementation, written from the specification in this paper, and is current with \R8.
  - Has three key abstractions: schedulers, senders, and receivers, and a set of customizable asynchronous algorithms.

godbolt

std::execution

The “Hello word” program of the proposal P2300R10.

```
using namespace std::execution;

scheduler auto sch = thread_pool.scheduler();                               // 1

sender auto begin = schedule(sch);                                          // 2
sender auto hi = then(begin, []{                                            // 3
    std::cout << "Hello world! Have an int.";                               // 3
    return 13;                                                              // 3
});                                                                         // 3
sender auto add_42 = then(hi, [](int arg) { return arg + 42; });            // 4

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

std::execution

- Execution resources
  - represent the place of execution
  - don‘t need a representation in code

- Scheduler
  - represent the execution resource
  - The scheduler concept is defined by a single sender algorithm: `schedule`.
  - The algorithm schedule returns a sender that will complete on an execution resource determined by the scheduler.

```
execution::scheduler auto sch = thread_pool.scheduler();
execution::sender auto snd = execution::schedule(sch);
// snd is a sender (see below) describing the creation of a new execution resource
// on the execution resource associated with sch
```

std::execution

- Sender factories
  - execution::schedule
  - execution::just
  - execution::just_error
  - execution::just_stopped
  - execution::read_env
- Sender consumer
  - this_thread::sync_wait
- Sender adaptors
  - execution::continues_on
  - execution::then
  - execution::upon_*
  - execution::let_*
  - execution::starts_on
  - execution::into_variant
  - execution::stopped_as_optional
  - execution::stopped_as_error
  - execution::bulk
  - execution::split
  - execution::when_all

std::execution

- Sender describe work
  - send some values if a receiver connected to that sender will eventually receive said values

- Receivers stops the workflow
  - it supports three channels: value, error, stopped

```
execution::scheduler auto sch = thread_pool.scheduler();
execution::sender auto snd = execution::schedule(sch);
execution::sender auto cont = execution::then(snd, []{
    std::fstream file{ "result.txt" };
    file << compute_result;
});

this_thread::sync_wait(cont);
// at this point, cont has completed execution
```

C++26

Core Language

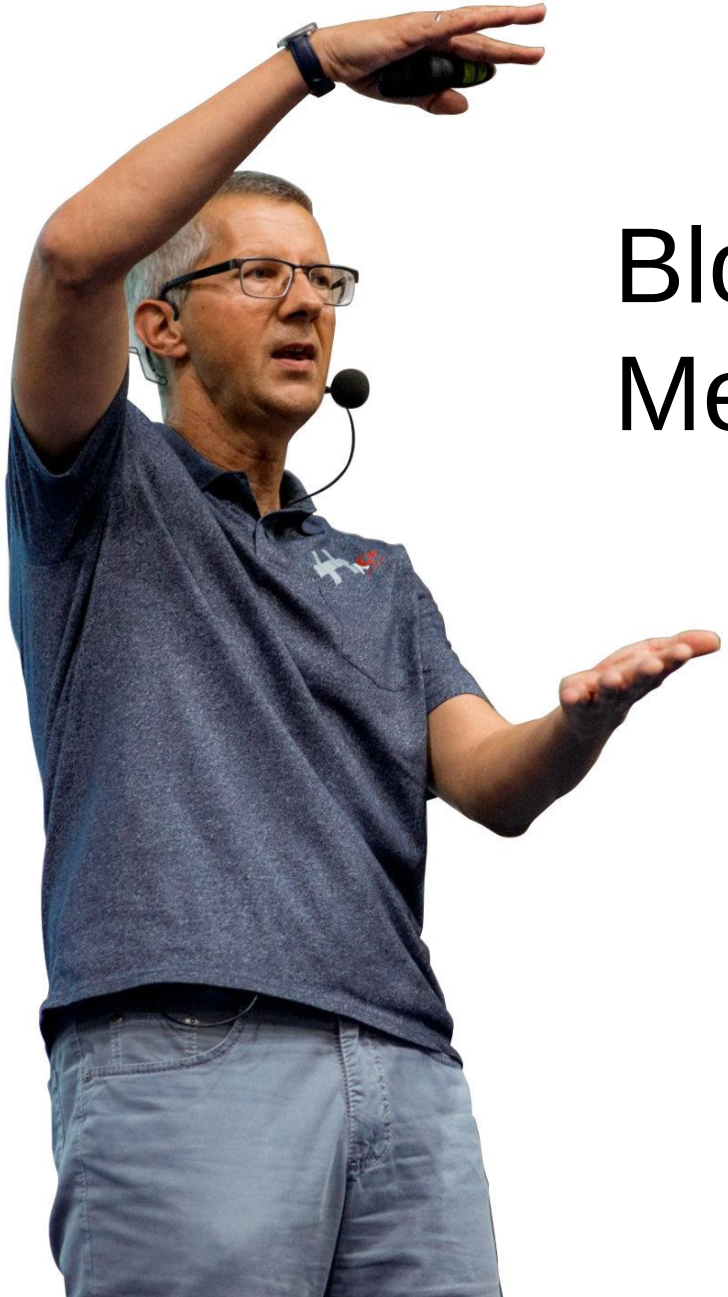
- ☐ Reflection
- ☐ Contracts
- ☐ Placeholder
- ☐ `static_assert` extension
- ☐ Template improvements
- ☐ `delete` with reason

Library

- ☐ `string` and `string_view`
- ☐ Format extensions
- ☐ `std::inplace_vector`
- ☐ Range improvements
- ☐ `constexpr` extensions
- ☐ Linear algebra support
- ☐ `std::submdspan`
- ☐ Debugging support

Concurrency

- ☐ `std::execution`



Blog: www.ModernesCpp.com
Mentoring: www.ModernesCpp.org

Rainer Grimm

Training, Mentoring, and
Technology Consulting