

# Meeting Cpp 2021 - Trip report

---



---

Author    Guillaume Dua

---

Date      10/11/2021 - 12/11/2021

---

## Table-of-content

- Meeting Cpp 2021 - Trip report
  - Table-of-content
  - **Introduction**
  - 📅 10/11/2021
    - (AMA) **with Hana Dusikova**
    - **Nicolai Josuttis - "C++20 - My Favourite Code Examples"**
    - (rebroadcasted) **Jonathan's Boccara - "Meta polymorphism"**
    - **Slobodan Dmitrovic - "How to approach learning and teaching C++ ?"**
    - **Andreas Fertig - "C++20 Templates: The next level"**
    - (Book room) **Guy Davidson and Kate Gregory - "Beautiful C++"**
    - (AMA) **with Sean Parent**
  - **Klaus Igleberger - "Breaking Dependencies: Type Erasure - A Design Analysis"**
    - Daniel Withopf - *Physical Units for Matrices: How Hard Can It Be?*
  - 📅 11/11/2021
    - **Christian Eltschig - "Writing sustainable software. The how and the what!"**
    - **Marc Mutz - "C++20 Coroutines As An API Principle"**
    - **Phil Nash - "Zen and the art of Code Lifecycle maintenance"**
    - **Kris Jusiak - "++namedtuple - Python style named tuples in C++20"**
    - **Jeff Garland - "C++23 Standard Library Preview"**
  - 📅 12/11/2021
    - **Jens Weller - "Improving hiring in C++"**
    - **Matthias Killat - "Lock-free programming for real-time systems - how compare-exchange will become your new best friend"**
    - **Joel Falcou & Denis Yaroshevskiy - "EVE: A new, powerful open source C++20 SIMD library"**
    - **Ivica Bogosavljevic - "My program was running fast six months ago. What happened"**
    - (Closing keynote) **Jens Weller - "Meeting C++ update"**
    - **Diversity and Inclusion Panel**
    - **Lighting talks 2022**
      - **Deniz Bahadir - "Compile-time checks for user-defined literals"**
      - **Tina Ulbrich - "C++ quizz : Squid game edition"**
      - **Marc Mutz - "QStringView : past, present, future"**
      - **Jens Weller - "Butterflies and C++"**
    - **Conclusion**

## Introduction

This week, I attended - *with 108 other C++ enthusiasts* - the virtual C++ conference [MeetingCpp-2021](#).


Is this really a trip report then ? Since le latter was fully online, maybe not.

But the goal here remain unchanged : to give readers an overview on what the various talks I attended to was, from a very personal perspective. I hope reading the next lines will give you make you want to participate to the next edition, and watch the talks once available on the [Meeting CPP Youtube channel](#).

About the **Hubilo** platform :

Beside a previous Meeting C++ meetups, this was the first conference hosted on Hubilo I experienced (comparing to other plateforms, like Remo for instance).

The global UI/UX design & some of the platform features made my experience very smooth, especially being able to prepare attendances by creating a schedule (*so you don't have to check the schedule anymore once the conference started, nor write down when to switch from a track to another*), but also take notes in advance, which is a partuculary convenient feature for AMA sessions.

 10/11/2021

Jens Weller and all the staff started the conference with a warm welcome.

### (AMA) with Hana Dusikova

After a nice coffee-break, I connected to the "Asking me anything" session hosted by [Hana Dusíková](#).

Back in 2019, I've had a great time attending to her [keynote at CPPP-2019](#), "[A State of Compile Time Regular Expressions](#)", for which you can find my [trip-report here](#).

This was a very interactive hour, with people's asking relevant question in the Q&A chat, divided in three kind :

- Questions about the C++ committee, as Hana is part of SG7 (compile-time)  
How does it internally work, how decisions are made, how things are discussed, etc.
- Questions about the C++ current and next standards, especially about TMP mechanisms & libraries
- Questions about [CTRE](#) - (*compile-time regex expressions*)

I couldn't help myself but to ask the same question I always ask to any experienced C++ developer/designer since a decade, especially to C++ committee members. This one is kinda hard to answer because it implies many factors, but answers are always interesting.

My question :

Q (6 upvotes): "*Many codebases exists since decades, with huge inertia, and suffered from organic-ish evolutions. Which C++ feature(s) in the next standard do you expect the most, to improve theses legacy codebases, and motivate teams to move forward from a standard to another ?*"

Hana replied by mentioning C++ features like [std::spans](#) & [conceptss](#), which I totaly agree with.

So far, [conceptss](#) are such a game-changer for - *my?* - C++ designs. If you are interested in, you can check the paper I wrote about [designing with concepts here](#).

About how to promote new standards in teams, Hana underlined the need to explain to decision makers (managers, tech leads, etc.) how efficient post-modern latest standards are.

## Nicolai Josuttis - "C++20 - My Favourite Code Examples"

To be honest, it was really hard to choose between this talk, and [Rainer Grimm](#)'s one about C++20 hidden pearls.

However, I felt like beyond cool features presentations, what I wanted - *at the moment* - was concrete examples.

Because of - *or thanks to* ? - my job, I mainly deal with legacy code bases, or those that have been around for decades. So what I expected from this talk was well-designed code example that I can get inspiration from, to promote the C++20 standard on a bunch of existing projects.

And ... that's exactly what I got ! 😊

Here is a non-exhaustive list of mentioned features, illustrated across a bunch of relevant examples on which the speaker made evolutions on each slides (which is something I particularly appreciate, as an effective way to both demonstrate stuffs but also not overload attendees minds).

- Ranges (range library concepts, subrange & sentinels, customization point objects)  
`std::ranges`'s `range_value_t<T>`, `begin(container)/end`, `ssize`, and various algorithms like `unique_copy` for instance.
- Compile-time support of containers/ranges
- Consteval functions
- NTTPs in C++20 (floating points, lambdas)
- Chrono library additions in C++20  
supports of literals + operations to enable synthax like `13d/09/2021y`,  
`std::chrono`'s `weekday`, `local_days`, `zoned_time`, `current_zone`, and more.
- Spaceship operator
- `std::format`

My question :

Q (5 upvotes) : "*In which case(s) would you prefer to code the printing logic directly into an `std::ostream operator<<(std::ostream &, const T&)` function, instead of using `std::formatter`, which is the customisation point of `std::format` ?*"

Answer from **Nicolai Josuttis** :

Creating a formatter is an alternative here, so you get supports [for custom printing].  
I wait eagerly for GCC to support it in a near future.  
I have no clear answer for that, if you want to use it, then do it.

Note : While writing this report, I realize that I did not mention the conditionally generated formatter for `ostream` support, already available in `fmt/ostream.h`

## (rebroadcasted) Jonathan's Boccara - "Meta polymorphism"

During the lunch-break, a stream opened to the youtube replay of [Jonathan's keynote back in 2020](#).

It's been a while since I had the opportunity to attend one of Jonathan's talk.

After defining polymorphism as a way to decouples components in source-code, the speaker underlined that polymorphism is not only about runtime.

During the keynote, Jonathan demonstrated few ways to achieve polymorphism.

- Using **preprocessor**, so implemented with macros.  
*Which obviously, is not something that you might be willing to do in production code.*
- At **link-time**
- At **run-time**
- At **compile-time**

Beside illustrated scenarios about mocking, the speaker detailed the big **why**, **how** and **what** of such design techniques.

### Slobodan Dmitrovic - "**How to approach learning and teaching C++ ?**"

The 20th slide of this presentation really captures its core idea, with the following quote :

*"Climbing mountain C++ is both a challenging and revwarding task. But once at the top, the biew is breathtaking. I strongly encourage you to take this journey".*  
-- Slobodan Dmitrovic.

As a C++ trainer myself, I was really interested in receiving advices from someone who's been practicing and teaching C++ from about 25 years.

If I do not agree with all the points of view that the speaker promoted during the talk about *how to learn & teach C++*,

this talk was full of valuable & synthetic informations.

I received it like a pragmatic state-of-the art of what teaching C++ is and should be, like how to deal with the underlying complexity of the language, and how to prioritize some feature over others that we can delegate to future times.

The author divides the process of learning C++ into 3 parts :

- The core language,
- The standard library,
- *(and only after)* modern standards.

One thing I totaly agree with, and the speaker highlitghted, is the fact that a live training sessions with a trainer might help attendees to achieve things they might literaly take years to understand & learn otherwise.

Here is a list of - *many* - points that caught my attention, divided according to the two part of the talk, which are **learning** and **teaching**.

#### **Learning :**

- Learn idiomatic ways to do things in C++  
By opposition to resourses that teach C with classes
- How to approach learning C++
  - Not by guessing : that's impossible (comparing to other languages)

- Not by drawing parallels between C++ and other languages
- Get good basics at first, that fit basic/common use-cases, then move forward
- Promote why peoples should learn C++

Proceed in three major & distincts steps :

- Focus on basic facilities,
- then move to classes & templates,
- and finally the C++ standard library.

### Teaching :

- How to approach teaching C++ (to beginners)
  - Do not overwhelm attendees with informations like design pattern and guidelines
  - Show the beauty of C++
  - Make clear distinction between basics, and corner cases
  - Promote portable C++, not domain-specific C++
 So, avoid focusing on specific domain of applications.

How to structure, as a C++ trainer, your approach ?

- Carefully choose topics and their scope
- Carefully decide an order : respect topics dependencies
- Keep the theoretical part to a minimum, but not less
- Support theory with lots of examples, with increasing complexity
- Avoid forward-referencing as much as possible
- Forecast the point when how much informations are too much informations

How to face common challenges (as a trainer) ?

- It's better to teach deprecated ways to do things like raw array & pointers, but only to promote modern/better way after
- Carefully pick guideline you want to promote

### Andreas Fertig - "**C++20 Templates: The next level**"

An alternative name for this talk might have be :

*"How to improve templates usage by constraining theses with concepts"*

As I already attending Daniela Engert's talk "*A (short) tour of C++ modules*" a few weeks ago at MeetingCpp (meetups), I chosed to go the [Andreas Fertig](#)'s one about templates in C++20.

You may be familiar with the name, as he's the author of the convenient online tool [Cplusplus](#), that I often use during training to demonstrate what's the generated code behind templates and some syntactic sugar.

After the usual explanation about Andreas's name, the speaker started his talk by giving a short reminder of one of the big four C++ 20 features : concepts.

Then the Andreas presented some way to implement the same constrained functions from the use of `std::enable_if` to a require-clause; highlighting how much simpler/neater this is.

Speaking of concepts, a very concise yet relevant example of the 5 kind of way to constraint a type or value using concept or requires-clause hit me :

```
template <C1 T>
requires C2<T>
C3 auto Fun(C4 auto param) requires C5<T>
{}
```

For different kinds of requirements :

- Simple requirements  
Check if an expression is valid and will compile
- Nested requirements Evaluates the boolean value of an expression
- ad-hoc constraints (requires-requires) Requirements are written directly after the requires clause

A very interesting point during the talk was also how to test created concepts.

Abbreviated function templates, like `<concept_name> auto && arg`

- C++ 20 NTP improvement
- Explicitly templated lambdas

My question :

Q : "As a library developer, would you rather create the egg or the chicken first ?  
Meaning, would you rather create for instance an `std::invocable` concept first, then an `std::invoke` function which's first parameter is constraint using `invocable`; or define `invocable` as `requires{invoke(...)}` ?"

Andreas's answer :

A : "Good question.  
If you figured out your requirements and can create a concept first, then you might be willing to do so."

(Book room) **Guy Davidson and Kate Gregory - "*Beautiful C++*"**

After Andreas's talk, I joined the room that promoted the book [Beautiful C++](#) by J. Guy Davidson and Kate Gregory.

To be honest, I had no idea Guy and Kate were about to release such a book, therefore was very curious about it.

As always, it's very pleasurable to discuss with theses two, who have such enthusiastic way to share.  
Thus I had a very nice time participating in this book QnA session.

My question :

Q : If you have only one sentence that'll make me buy your book in the next minute, what would it be ?

Guy's answer :

A : "Amaze your friends !" *with a funny face and thumbs up.*

Coffee spilled, book bought.

## (AMA) **with Sean Parent**

Last time I attended a **Sean Parent's** talk was at the [itC++ conference 2021](#) a few months ago. After the talk, Sean made himself available in the lobby for casual talks, and answer some questions.

This time, two topics that emerged from the QnA chat triggered me more than others :

The way to work on codebases/projects, for which Sean answered with the following advises :

- Focus on what the code is doing, and what it does wrong (in opposition to blindly trusting the specs/documentation)
- Rely on sophisticated debuggers

But also an interesting opinion about coroutines, which is that such feature does not interface very well with the rest of the language

## **Klaus Igleberger - "*Breaking Dependencies: Type Erasure - A Design Analysis*"**

After a quick diner, I attended the first talk of the evening, which was about *type erasure & software design patterns*.

Why I looked forward this talk in particular ? There are two main reasons :

- I really appreciated previous Klaus's talks, and am always amazed by his trainer's skill. The way it makes listening - *thus learning* - such an easy task, with no noticeable loss of attention at any point, yet filling your mind with a bunch of new ideas & technics.
- Uncoupling component, promoting SRP, SoC & LSP principles in particular, and using designs like type erasure in particular is what I do the most on daily basis.

Here are the key ideas of the talk :

- Dependencies is the core problem of software development that developers, designers and software architect face.
- Classes hierarchy using inheritance & perils of never-ending YAIL, which often - *if not always* - scales poorly.
- Design pattern as an answer the inheritance problem, and tight-coupling in general.

Like in his previous talk earlier that year, Klaus underlined that design pattern has nothing to do with OOP; - *which is something I value, and promote in all design trainings I give.*

The speaker concluded his talk by giving the following summary :

Type-erasure is :

- a template constructor
- non-virtual interface

- external polymorphism + bridge + prototype

Why you should use it :

- reduce dependencies
- enable value semantics
- improve performance
- improve readability thus maintenance

### Daniel Withopf - *Physical Units for Matrices: How Hard Can It Be?*

As the last talk of this first day of conference, Daniel Withopf demonstrated some approach to solve maths and physic issues using C++.

To be honest, not sure I really understand all the details related to maths & physic, but the library that the author presented looked pretty neat.

What do we get from the TypeSafeMatrix ?

- Expressive & enforced names for vector/matrix entries
- Compile-time protection against out-of-bounds access
- Abstraction of underlying linear algebra libraries



11/11/2021

### Christian Eltzschig - ***"Writing sustainable software. The how and the what!"***

For this talk, Christian Eltzschig described some design and principles to create sustainable softwares; especially in safe, thus constrained environments that makes heap-allocations and vtable impossible.

The main axis the speaker chosed is how to design future-proof components, taking into account both the environment limitation but also the fact that a design is never completed ?

After enumerating some beginners mistakes, and defining what over-abstractions are and why we should avoid it, the speaker explained how to design element in a simpler yet resilient way.

To me, this part sound really close to a promotion of the YAGNI principle, as well as the IBA paradigm.

What interested me in particular in this talk, are the following guideline to design future-proof components; as theses are principles that I promote on daily basis at work :

- Provide intuitive interfaces, that any user who's not familiar with can use without reading any documentation : IDE auto-completion is enough here.
- Avoid misusages by-design, by promoting Design-by-contract; thus enforcing contract violation checks.

Which results in easy to use APIs, yet hard - *if not impossible* - to misuse.

A nice advantage provided here with such way to design & implement component is the ease to maintain theses.

By promoting SRP & SoC principles on designs, we improve readability, thus make code easier to maintain.

Also, the speaker highlighted how important ownership & lifetime management are, thus the RAIL principle. Finally, the author demonstrated some monadic way to deal with concurrency, yet providing an intuitive



interface to user-defined types.

## Marc Mutz - "C++20 Coroutines As An API Principle"

Coroutine is one of the big four features shipped with C++20. However, as I did not play enough with yet, I attended this conference to best understand the challenges that designers & developers face using it.

This talk had quite a **Qt** flavor. However, the speaker made it quite understandable, focusing his talk on design principles and underlying issues developers may face when using coroutines.

The entry point of the talk was a simple statement : owning containers in API are often not a perfect solution, but non-owning interfaces have their own issues; with lifetime management on top.

What I'll remember from the talk is some best-practice on how to write safe coroutines (*re-entrance, frame allocation, etc.*), but also why and when choose coroutine among other available options.

Also, the speaker pinpointed important facts :

- Coroutines are forward-only, meaning that by-design it can't "go-back", thus random-access are not an option. Any attempt to violate that principle is a non-sense.
- Coroutines can help avoiding the use of owning containers in APIs
- Since coroutines are ordinary fonctions
  - It can be virtual
  - It can be DLL-exported
  - Its can be used in APIs
- Coroutines's magic is in the return type
- Fewer promises types are easier to learn, implement, document, but requires type erasure - *thus memory allocation*
  - fine-tuning behaviours requires per-coroutine promise types
  - and inlines functions as well for performances
- Coroutines can help with implementing lazy sequences, thus allow less work to be wasted when needing only a subset of results

But also, issues some warnings :

- Coroutines have their own issues
  - lifetime, with dangling references

As a design *decision-making* helper, the author concluded with the following advise :

- Either data is stored and contiguously, then you should return a view and have random-access, or it does not then you should use a coroutine.

## Phil Nash - "Zen and the art of Code Lifecycle maintenance"

Phil began its talk with an interesting observation : all of the 115 talks he did before can be sorted in four major categories, which are :

- Testing
- Functional programming
- Error handling

- Simplicity

Which all have a common thread, that is software quality.

The speaker underlined that as a matter of fact, quality is undefinable : everybody have their own vision of what software quality is. For instance, do we speak about external or internal quality ? Is quality relative to some person ? If so, the end-user ?

To solve this definition issue, he opposed his own definition to the CISQ (*Consortium for IT Software Quality*) one :

- **CISQ's**

"Security, reliability, performances/efficiency, maintainability"

- **Phil's**

"Correctness, reliability, performances/efficiency, malleability/evolvability + applicability, reasonability"

With such starting point, the author then defined the following two key concepts :

- Applicability, as **doing the right thing**
- Correctness -> as **doing the thing right**

And made a nice parallels with **Allan Kelly's** talk "*Do it right, then do the right thing*".

// TODO : INSERT SLIDE HERE

Then, he wonders how does the definition elements intersect with each others : for instance, *reasonability* & *evolvability*, giving precises illustration examples and sharing analysis. Here are a only a few, for instance.

### ***correctness & reliability*** :

Correctness can be defined as an expected behaviour, while reliability as ways to handle what's unexpected. The conjunction of theses two leads to :

- **Code coverage**  
Per line/statement, using TDD (Test Driven Development)
- **Data-coverage**  
PBT (property-based testing), Fuzz testing, Manual testing (testers)

If error-paths are often hidden, we still have multiple ways to deal with it; - *that deserved to be used way more* :

- **IO errors**  
With exceptions, error codes used in conjunction with the `nodiscard` attribute, `std::optional`, `std::error_code`, etc.
- **Logic errors** / contracts  
`asserts` : which works way better with function with proper name (`is_valid_<smthg>`, user-defined type with checks in constructor, etc.

Phil also mentioned - *among many other interesting points* - that :

- The intersection of **reasonability & evolvability** should lead to promoting a lower complexity (any kind of dependencies); which result in components being easier to test, to reason about, and to change.
- The intersection of **reasonability + efficiency** underlines the gap between high level code and low-level machines.

In general, the author highlighted how important static analysis tools are, and why we should use these.

## Kris Jusiak - "+ +namedtuple - Python style named tuples in C++20"

As a C++ tmp lover, really looked forward this talk. Also because the past months, I start playing - & *benchmarking* - with various way to implement tuples.

### Motivation :

Just like python, use descriptive field names instead of interget indices, so it provides context on what values are.

Ultimately, using such a utility will result in cleaner and more maintainable code.

Also, this will fix `std::tuple` interface issue like the - *somehow* - ambiguous `std::get<T>` invoked with a value of type `std::tuple<T, T>`.

Starting with a goal/dream vision, the speaker structured his talk as a devloper journey, listing wished features first, then implementing features one after another; - *with plenny of code snippets*.

This resulted in a very enjoyable talk to attend at, easy to understand thanks to the iterative logic.

Speaking of a dream interface, the later was :

```
constexpr auto t = tuple(x = 4, y = 2);
static_assert(4 == t.x);
static_assert(2 == t.y);
t.z // error

static_assert(requires { t.x; });
static_assert(requires { t.z; }); // error

// Extension
constexpr auto t2 = tuple(t, z = 42);

// Query
std::vector<decltype(tuple_value)> ts{t};
db.query("select x, y from ", ts);
```

But also add JSon simple serialization.

After excluding static-reflection (as not being currently part of the standard) and type annotation, the author first target the core feature : having square brackets operator to access tuple elements by name - *and nice compiler errors as well, in case of misuseage*.

However, beside theses interface consideration, Kris added the following requirements :

- Still have `std::get<N>`

- Still have structured-binding `auto [x,y] = tuple_value;`
- Still can be packed (memory)

The speaker then demonstrated how to implement such requirements in a really enjoyable way, with lots of - *cool* - code samples.

Icing on the cake, he shared the complete implementation [available here](#)) so everybody can play & experiment with.

Kris concluded his talk with a bunch of usage examples, that best demonstrated how convenient `namedtuple` are to code with;

And highlighting the following points :

- Python-style namedtuples are powerful & flexible
- The additional challenges when it comes to translate features from a scripting languages to a system language
- C++20 greatly improved template-metaprogramming
- C++ needs reflection - *so much*

## Jeff Garland - "C++23 Standard Library Preview"

Even if I try to read as many proposals as I can to always keep an eye on how C++ is evolving, my expectation for this talk was similar to a child the night before xmas : a bunch of well-wrapped gifts waiting at the foot of the tree, and a hint of magic floating in the air.

Technically speaking, some features of the STL that matter the most to me are the `range` & `format` libraries extensions, as well as getting more `constexpr` components; so I can get rid of many duct-tape flavored like code I used for the last past years.

[Slides \(from CppCon2021 available here](#)

For this talk - *with was full of highly valuable informations* - , Jeff Garland - as part of the **LEWG** (*Library Evolution group*) - began with a reminder of the shipping process of the committee, and how the COVID pandemic impacted work.

In short : C++23 will be a smaller release, focusing more on bug fixes rather than new features - *comparing to C++20 that was so large*.

Priorities :

- Fix bugs shipped in C++20
- (not) networking TS
- Support of modules -> which granularity / size ?
- Concurrency support - co-routines / `std::generator` ?
- Addition to C++20 features : ranges, format

Outline of the talk :

- String processing
- Standard library modules support
- I/O additions
- Ranges changes & additions

- Constexpr all the things
- Utilities - stacktrace, expected, bute\_swap, monadic optional
- (not part of the talk) containers
- (not part of the talk) c interfaces

Then the speaker drawn a nice picture of which features are gonna be shipped with the next C++ standard.

- String processing
  - `std::basic_string::contains`
  - `std::string` and `nullptr`
  - `resize_and_overwrite`
  - `string_view` range constructor (*construct a string\_view from a range*)
- Standard library modules support
  - `import std, import std.compat` (*std + c global namespace things*)
  - With very good compilation speed.  
*Import beats include between 7 to 60 times, which is a significant advantage !*
- I/O additions
  - additions to `format`
    - Compile-time checks, like the **fmt** library already has
    - Fix backported to C++20
  - `std::print`  
x3 faster than `std::cout`, but also still faster than `printf`

```
#include <print>

print("{} answer is {}", "The", 42)

FILE* fp = fopen("junk.txt", "w+");
print("{} answer is {}", "The", 42);

println(stderr, "{} answer is {}", "The", 42);
```


- `std::format/formatters` specializations for the STL (*like, ranges*)

```
std::vector v{10, 20, 30, 40, 50, 60};
std::string str = std::format("{:02X}", fmt::join(v, " "));
```

- `spanstream`
  - `#include <spanstream>`
  - `ispanstream, ospanstream`

- Ranges changes & additions
  - **ABI & API breaks** (*like, `split_view`*)
  - Fix C++20 **issues**
    - `istream_view`
    - `join_view`
    - `split_view` -> renamed to `lazy_split_view`
    - `split_view` now do what you'd expect !
  - new **range algorithms**
    - `shift_left`, `shift_right`
    - `fold_left`, `fold_right`
    - `starts_with`, `ends_with`
    - `iota`
    - `find_last`
  - new **adaptors & views**
    - `adjacent`, `adjacent_transform`
    - `cartesian_product`
    - `chunk`, `chunk_by` `v | std::views::chunk(2); // [1,2] [3, 4] [5]`
    - `join_with`
    - `slide` `v | std::views::slide(2); // [1,2] [2,3] [3,4] [4,5]`
    - `zip`, `zip_transform`
  - convert ranges to container with `ranges::to`
  - pipe support for user views
  - support interoperation with non-range algorithms
- **constexpr** all the things
  - `std::unique_ptr`
  - `std::variant`
  - `std::optional`
  - `std::type_info`
  - `cmath` functions
  - `to_chars`, `from_chars`
- **utilities**
  - `stacktrace` (based on `boost.stacktrace`. Challenges : no universal portability)
  - `expected`
  - `byte_swap`
  - `monadic optional`
  - `is_scoped_enum`
  - `to_underlying`
  - `atexit()`
- (not part of the talk) containers
- (not part of the talk) c interfaces

Looks exciting, right ? The video is already available [here on the Meeting-CPP youtube channel](#).

 12/11/2021

## Jens Weller - "*Improving hiring in C++*"

Who never heard "*C++ Devs are just hard to find*" ? Like many, Jens got use to hearing this for many years. On the other hand, C++ developers get their emails boxes literally spammed with request from recruiters, including when they are not looking for a job.

Starting from this observation, the speaker detailed multiples issues related to hiring C++ developers, and - *of course* - promoted he's promising solution : Meeting C++ job fairs.

The later allows :

- Direct recruitments - *in opposition to service/consulting companies*
- By relevant & valuable companies

For me, whether as someone who was used to integrating new team members, or who has already looked for a job, this topic is really interesting and worth digging into.

## Matthias Killat - "*Lock-free programming for real-time systems - how compare-exchange will become your new best friend*"

What a dense talk about lock-free programming ! I was really curious about this one before attending, as I don't do much concurrent programming.

The last past years, most of the codebases I worked on handled concurrency on an higher design level than the one I worked on (*like multi-levels distributed computing for instance*).

After defining what real-time systems and lock-free programming are, Matthias detailed common technics like atomics & CAS (compare & swap).

Inherent requirements/challenges/constraints for real-time systems - *thus, with multiples concurrent processes performing tasks* - are :

- Task have priorities & deadlines
- Tasks are often periodic
- Real-time OS (RTOS) guarantees priority-based scheduling of processes & threads
- RTOS cannot guarantee time bounds in completing of user algorithms

Which underlines the need of fair scheduling, as required to both :

- Meet the deadline  
*If not, then system failure(hard real-time)*
- Allow system-wide progress

Concurrency in real-time systems :

- Requires synchronizations of shared data
- Thus sync primitives (*mutex, semaphore, cv, promise/futures, read-write lock*)
- Problems :
  - blocking may lead to lockout and deadlocks
  - priority inversion (*when high priority thread waits for resources held by low priority threads*)

About lock-free programming, the speaker defined such concept as a way to create algorithms without any blocking primitives,  
which results in the following advantages :

- No deadlocks, improve robustness against partial failure
- Guarantee progress or at least 1 thread over the others
- No priority inversion
- Fewer context switches (*no blocking wait*)

... but also different disadvantages :

- Data sync is much hard
- More complex algorithms
- Not always faster

### How to do lock-free programming ?

After a quick summary of what atomics are (C++11 STL feature, basic building block of lock-free algos as not involving locks for small datas - 64 bits) -, *and challenges related to this feature* -, Matthias detailed other technics.

### Compare & swap (CAS)

- Major building block of lock-free algos
- `std::compare_exchange_strong`
  - Atomically try to set the current value to the desired one - *returning true if successful*.
- `std::compare_exchange_weak`
  - Can fail even if the condition is true

As a recurring pattern for lock-free algorithms, **CAS loop** is a tool of choice. The speaker summarized such pattern with the following workflow :

- load current value
- compute new value locally
- try to update the old value atomically (CAS)
- retry if concurrent modification is detected
- Starvation is possible but veru unlikely in practice

### Lock-free data exchange between threads

Another interesting sub-topic of this talk was lock-free exchange buffer - *that use only one slot for datas, thus can be generalized (queue)*.

The speaker demonstrated what requirements such interface have, and how to implement it.

Here is a quick description of the latters :

- How to share data - *of generic type T* - between threads without locks ?
- With an arbitrary number of concurrent readers/writers
- For which failures to write are allowed but should be rare & well-defined (e.g out-of-memory)

Finally, the speaker warned about common issues of lock-free programming like [ABA](#).



## Joel Falcou & Denis Yaroshevskiy - "EVE: A new, powerful open source C++20 SIMD library"

As I attend the C++FrUG meetups regularly, I knew in advance that this talk was gonna be really cool - *despite SIMD is not my cup of tea though* - because of Joel.

What really marked me was the communicative enthusiasm that both speakers have about their new library.

If you watched/attended to other Joel's talk before, you may be surprised with this one : once is not custom, it is easy to understand and won't hammer your brain with template-metaprogramming tasty - *yet complex* - tricks.

Also, a few months ago, I checked another of Joel's library - that btw EVE depends on, which is [kumi](#) : a really cool & convenient tuple implementation.

So, **about EVE** :

It's an MIT-licensed, C++20 library for SIMD (*same instruction, multiple datas*) computation;

As a wrapper/abstraction around SIMD intrinsics (+library of core types, algorithms, 250+ numerical functions including operators overloading & maths stuffs), it applies the same functions to multiples data in all the registers.

The key point here is that it exposes SIMD in a uniform way across both ARM (neon) and x86 (from sse2 to avx-512). "*Clean abstractions for clean code*" so.

Comparing to other similar libraries, its advantages are :

- STL like algorithm support, including zip to operate on multiple ranges
- ARM support (many libraries only support x86)
- A very comprehensive math library
- Production quality

### What requirements are ?

- C++20
- Clang or GCC (Latest until module)

Note that msvc-clang/msvc are not supported yet, though speakers announced it "soon~ish".

### Why another SIMD library ?

It answers a common issue : there are 1001 flavors of SIMD (set of instructions AVX, SSE (+versions), (ARM NEON, ASIMD), powerPC, etc.). Every instruction set have their particular idioms to know.

### As-is, what can someone use to achieve such kind of vectorization ?

- Compiler's auto-vectorization (Ok for non-fancy maths functions)
- Special pragmas & special compilers
- `std::execution::unseq`, `hpx` => implementation defined (by compilers)
- Specialized tools : Halide, simdjson
- Write it yourself

### What kind of algorithms are available ?

- `all_of/any_of/none_of`
- `find/find_if`
- `equal, mismatch`
- `transform`
- etc.

An interesting design choice is that the library have sets of decorators so users can choose between accurate result (at the price of speed), or speed. With lots of relevant/interesting demos, including crispy technical/implementation details.

FYI, [slides available here](#), and a [minimalistic demo here](#) - on *godbolt* - of vectorized `find_if` for x86 and arm architectures.

## Ivica Bogosavljevic - "My program was running fast six months ago. What happened"

For this talk, Ivica started with a list of common factors that often - *if not always* - result in performances issues.

- Architectural issues (requires to make many changes, that involves multiples teams)
  - Careful design is important to avoid performance issues
- Software consists of logical components or modules
- API design -> minimize the number & size of messages between components
  - Avoid "chatty" components -> in favor of bulk data processing
    - overheads
      - functions calls
      - inhibit compiler optimizations
      - critical sections protection
      - instruction cache misses
    - especially when components are moved to multiples locations
    - example : malloc
- Data copying & data conversions

But also general problematics - *that you might be familiar with* :

- **Ressources contention** - *waiting on a resource (data from the network, waiting to enter critical section).*

For instance, a logger (which is a shared component, that involves mutexes, and wide utilization). By design, such component is a bottleneck, and likely to generate some dominos-effect (*components waiting for components, and so on.*).

- **Compiler optimization are fragile** - *as relying on pattern matching & heuristics,* thus can slow-down your system

For instance, any algorithm that relies on vectorization & inlining, is by design not resilient to change : *adding/changing 1 line can break optimizations.*

- **Hardware issues** - *"Hardware-friendliness"*

Larger program or larger datas set are less hardware-friendly (*more instruction & data cache misses, failure to use CPU vectorization units*)

The speaker concluded his speech with various strategies to best help mitigate previously mentioned issues, for instance decomposition large classes into smaller classes or using the [ECS](#) design pattern - *that I love !*).

### (Closing keynote) **Jens Weller - "Meeting C++ update"**

For this closing keynote, Jens shared informations about the future of Meeting C++.

You might already know, Meeting-CPP 's gonna be 10 years old in Decembre, and a special meetup will take place for this occasion - *December the 9th from 4pm to the end of the night*, [meetup link here](#).

Also, the speaker detailed his vision about the components of MeetingCPP, which are :

- The conference event
- Meeting C++ online
- The new book section
- Surveys
- Job fairs
- Book & tool fair
- Trainings

Before giving warm thanks to all the events sponsors, speakers & attenders,

Jens mentioned plans for the next year, especially afor the 10th Meeting-CPP conference, but also his will to help founding new C++ user groups all over the world.

He concluded his talk with something that sound like a promise : *"See you next year in Berlin !"*.

### **Diversity and Inclusion Panel**

This session was hosted by Chandler Carruth, Patricia Haas, Hana Dusikova, Tina Ulbrich, Filipe Mulonde, and Jens Weller.

After giving stats about the support tickets for this year (*28 student and 11 support tickets !*), the hosts discussed - *often involving with the chat too*, about diversity at work, but also in the whole C++ community.

Mentioned was made of many kind of discriminations (like skin color, sexual orientation, sex, age, health conditions & disabilities, etc.), so this session had a nice depth; and - *like most non-tech talks* - was very interesting.

If you are interested in, the video is already available [here on the Meeting-CPP Youtube channel](#).

---

### **Lighting talks 2022**

The Meeting C++ conference closed with four lighting talks.

All talks are available as one single video [here on the Meeting-CPP Youtube channel](#).

#### **Deniz Bahadir - "Compile-time checks for user-defined literals"**

This first lighting talk, from Deniz was pretty cool despite how fast it was.

As titlted, it's about compile-time checks for user-defined literals, giving a nice demo to validate IPV4 adresses.

The demo's [code is available here on godbolt](#), and the talk's [video here](#).

### **Tina Ulbrich - "C++ *quizz* : *Squid game edition*"**

Feel up for a - very - stressful, almost impossible challenge ? Like creepy music ? Check the talk here on the [MeetingCpp Youtube channel](#).

[Quizz source code](#) are available here

### **Marc Mutz - "QStringView : *past, present, future*"**

The complete talk is available [here](#), and mention topics like considering QStringView as a replacement to QString, the difference between owning and non-owning containers when it comes to design choices, and why coroutines is an interesting choice for QStringTokenizer.

### **Jens Weller - "*Butterflies and C++*"**

For [this talk](#), Jens shared his love of butterflies and how he got involved in observing & preserving them, inherent challenges of such tasks, but also his will to create a project/tool that will detecting butterflies in photos, for monitoring purposes.

---

## **Conclusion**

What a great conference, I hope you have enjoyed reading it so far, and it made you want to participate in any C ++ conference, and the next edition of the Meeting-CPP in particular.

Next for me will be [CPPP 2021](#) (1-3 December 2021 - *fully Online*), I hope to see you all there !

Guillaume "Guss" Dua.