C++ Club UK Meeting 140

Gleb Dolgich

2021-12-02

Contents

Giusy

This episode is dedicated to our cat Giuseppe who passed away two weeks ago. Miss you Giusy.

Follow-up: Jason Turner on abidiff

Jason Turner posted an episode of his **C++ Daily** YouTube channel called *Detecting ABI Changes With abidiff*, in which he experiments with getting `abidiff` tool installed and attempts to use it to detect ABI changes in a test program. The tool is written using libabigail that we talked about last time. For library vendors it could be a good idea to incorporate a tool like `abidiff` in their CI pipeline, so that any breaks are flagged early before the library is made available to the clients.

November mailing

The November 2021 committee mailing is out (Reddit). I'll mention just one paper this time.

Portable assumptions

P1774R4

This paper proposes a portable way to assume something without executing it, but instead allowing the compiler to use it for optimizations. The new syntax replaces similar built-in facilities in the popular compilers:

```
__assume(expr); // MSVC, ICC
__builtin_assume(expr); // Clang
if (expr) {} else { __builtin_unreachable(); } // GCC
[[assume(expr)]] // Proposed (portable)
```

Side effects are allowed in the expression because it's not evaluated (except for GCC, but that can be worked around). If the expression doesn't evaluate to `true`, the behaviour is undefined.

This looks like something out of the contracts proposal. It is also a very easy way to accidentally introduce undefined behaviour into your program. But no worries, I'm sure everyone will be extra diligent.

C++23: Near the Finish Line

This Reddit post outlines the current plan for C++23. The author analyzes the current library papers and their chances of getting into C++23. For each paper its age in months is included, but it is there for your amusement only – it should not reflect on the readiness of the paper. The "feature-complete" deadline for C++23 is 7 February 2022. The plan is outlined in P2489.

After all I've read I was a bit surprised to see P2300 `std::execution` as one of the papers aimed for C++23. Some commenters in the thread seem to think it is not yet ready and should not be rushed, despite the fact that an implementation (libunifex) has been in use at Facebook for some time now. Ben Craig says that it's not guaranteed that P2300 ends up in C++23, but the poll outcome was to aim for it, so the meetings will be scheduled accordingly. To be accepted, this paper needs the strongly-opposed committee members to change their minds. We'll see soon enough if that was a realistic expectation.

There is a ton of changes and improvements in Ranges. P2214 A Plan for C++23 Ranges has been split into multiple papers and they all seem to be on their way into C++23, which is good. Same with coroutine-based generator paper P2168 `std::generator`. Another coroutine-related paper aiming for C++23 is P1056 `std::lazy`, which I must have completely missed till now. It adds `std::lazy` template to enable creation and composition of coroutines representing asynchronous computation. The only operation you can do with the lazy is to await on it. It'll be interesting to see how to use it in practice. I don't think coroutine support improvements and additions will be as extensive in C++23 as initially planned, so anyone wanting to use coroutines without descending into low-level details will have to resort to third-party libraries like Lewis Baker's excellent CppCoro.

I'm glad to see P2093 Formatted Output in the list of things in C++23, seems like a natural extension of `std::format` feature we have in C++20.

P0009 `std::mdspan` has been going for 74 months! It adds `std::mdspan`, a multidimensional array view template with supporting classes, and `std::submdspan` template that allows slicing of `std::mdspan`.

Now we come to the section where the committee says "no". Some of the papers that are not going to be in C++23 are:

- [P0447 `std::hive`] — come on, seriously? 18 revisions, 61 months, and still not ready? I'm so old I remember when it was called `std::colony`. Disappointed. A very interesting data structure that would have been a good addition to the Standard Library. Oh well, there is always `plf::hive`.
- P1385 and P1673 (Linear algebra) — some commenters think it should not be in the standard because the API is fluid and there are constant new versions and additions to the linear algebra library landscape, others disagree and say that BLAS is the standard API and it should be in C++.
- P1288 Coroutine Concepts And Metafunctions — 37 months, one revision, probably not ready, which is a pity.

At the top of the comment thread Jonathan Müller (foonathan) says:

> Just a friendly reminder: Please remember that everyone involved in the standardization process is trying to improve C++. Personal attacks on individuals, conspiracy theories about the standardization process, etc. will not be tolerated here.

To that redditors predictably say:

> You take away all the fun.
>
> I just want to say that I am perfectly happy to tolerate as-hominem attacks and expressions of anger. How else is the true feeling of the user community going to be heard?
>
> We're not silencing dissent: you're free to discuss the proposal; just don't attack the authors. You're also free to voice criticism on committee decisions; just don't spread theories about an organized effort to silence proposals.
>
> Well, that's just a silly conspiracy theory; I don't believe that the committee members are nearly organized enough to silence proposals.

Regarding random number generation, this redditor says:

> P0205 not making it in is a tragedy. Every standard version without a correct way of seeding an engine is a mistake.

I believe we discussed it before. Without this proposal seeding random number generator properly is an expert-level task, which it shouldn't be. Consider the current 'trivial' way:

```
template <typename EngineT>
void seed_non_deterministically_1st(EngineT& engine)
{
    std::random_device device{};
    engine.seed(device());
}
```

The proposal author says:

> This code is severely flawed. If EngineT is `std::mt19937`, it has a state size of 19968 bits. However, if an `unsigned int` is 32 bits (as is the common case on many platforms today), then of the up to 2^{19968} states, at most 2^{32} (that is one 2^{-19936}-th) can possibly be chosen!

This is how you can seed random number generator better today (and the code is not without flaws either!):

```
template <typename EngineT, std::size_t StateSize =
    EngineT::state_size>
void seed_non_deterministically_2nd(EngineT& engine)
{
    using engine_type = typename EngineT::result_type;
    using device_type = std::random_device::result_type;
    using seedseq_type = std::seed_seq::result_type;
    constexpr auto bytes_needed = StateSize *
        sizeof(engine_type);
    constexpr auto numbers_needed = (sizeof(device_type) <
        sizeof(seedseq_type))
         ? (bytes_needed / sizeof(device_type))
         : (bytes_needed / sizeof(seedseq_type));
    std::array<device_type, numbers_needed> numbers{};
    std::random_device device{};
    std::generate(std::begin(numbers), std::end(numbers),
        std::ref(device));
    std::seed_seq seedseq(std::cbegin(numbers),
        std::cend(numbers));
    engine.seed(seedseq);
}
```

And this is the proposed *proper* way:

```
template <typename EngineT>
void seed_non_deterministically_3rd(EngineT& engine)
{
    std::random_device device{};
    std::seed_adapter adapter{device};
    engine.seed(adapter);
}
```

But we can't have nice things, and so this won't be available at least until C++26.

If you like to read emotionally-charged Reddit comments, here are some links into the thread:

- why no reflection, also RIP reflection
- why algebra
- senders/receivers
- networking
- what the hell is 'iota'?
- the metrics for the listed proposals are wrong
- ranges is a 'dead horse'
- graphics in the standard library vs. oh god please no

A quick update: the author just published a new revision of the original post which contains references to prior work for each paper, to better illustrate how the standardization process works.

Bitpacker

GitHub (Header only, Boost license, C++14/C++17/C++20)

A library to do type-safe and low-boilerplate bit level serialization, binary-compatible with Python bitstruct module. This is useful for embedded projects where bit-level encoding is needed so it's not possible to use other serialization libraries like FlatBuffers, ProtoBuffers, or Cereal.

Note that the library doesn't currently support floating point values. To define serialization format, the library uses a DSL, something akin to `printf`.

```cpp
std::array<uint8_t, 7> make_message_py(const MessageData&
    data) {
    return bitpacker::pack(BP_STRING("u12b1b1u14s24"),
                   data.voltage,
                   data.error,
                   data.other,
                   data.pressure,
                   data.time);
}
```

Reddit points to another similar library: CAN-Helpers, which makes it easier to get and set CAN signals from CAN messages.

Use of built-in exception types vs. own types

Redditor asks if they should use standard exception types or add their own exception types. The options presented are:

- Only use `std::exception` and derived classes.
- Use `std::runtime_error` and `std::logic_error` and derived classes.
- Use more specific classes like `std::invalid_argument`, `std::domain_error`, `std::overflow_error` and derived classes.
- Don't derive own exception classes and only use the standard ones.
- Don't use standard exception classes and create own exception hierarchy.

The replies divided between:

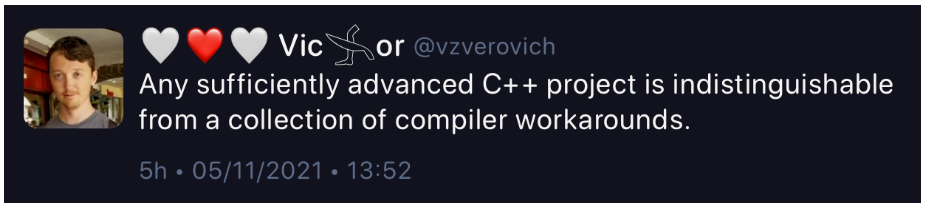
- Don't use exception hierarchy and just derive a single exception class from the standard one.
- Whatever you do, don't use `std::logic_error`, as it indicates an unrecoverable bug in the program that can only lead to calling `std::terminate`. (Related: contracts can't come soon enough.)
- We should prefer wide to deep exception hierarchies, and if you need to have additional data in an exception, you can use Boost.Exception (Interestingly, the latest advice from Boost is to use Boost.LEAF for error handling, exceptions or not.)

By the way, my source says that the exception class hierarchy in C++ was probably a mistake.

Book: Programming with C++20

Andreas Fertig has finished his book and you can order it on Leanpub. Pay what you want, the suggested price is $40, minimum $20. There is a free sample available, but even without it I expect this to be a great resource for learning the new C++20 features.

Twitter



🤍❤️🤍 Vic🛩or @vzverovich

Any sufficiently advanced C++ project is indistinguishable from a collection of compiler workarounds.

5h • 05/11/2021 • 13:52

Thanks for joining me today. Until next time!