C++ Club UK Meeting 133

Gleb Dolgich

2021-07-08

Contents

June 2021 C++ Standard Commitee Mailing

The June 2021 mailing is out. The following papers caught my attention.

`std::execution`

P2300R0 by Michał Dominiak, Lewis Baker, Lee Howes, Michael Garland, Eric Niebler, Bryce Adelstein Lelbach.

> This paper proposes a Standard C++ model for asynchrony, based around three key abstractions: schedulers, senders, and receivers, and a set of customizable asynchronous algorithms.

It aims to *replace* the existing Executors proposal, which is currently at revision 14:

> This paper is not a patch on top of [P0443R14]; we are not asking to update the existing paper, we are asking to retire it in favor of this paper <...>
>
> This paper is not an alternative design to [P0443R14]; rather, we have taken the design in the current executors paper, and applied targeted fixes to allow it to fulfill the promises of the sender/receiver model, as well as provide all the facilities we consider essential when writing user code using standard execution concepts <...>

According to GitHub notes, it has been discussed in a joint telecon by Library Evolution and Concurrency groups. Among other topics, they discussed how coroutines work with the new proposal (the authors think coroutines cannot express all aspects of the sender/receiver model, like cancellation etc.) One of the main points was the current Networking TS and if it should remain coupled with executors/senders/receivers.

> Some said out that we have been trying to develop a "grand unified" async model for years, and perhaps that is a mistake. Others

> pointed out that it may be a mistake to try and retrofit the Networking TS, which is based on established practice from Boost.Asio, with a new model.

The Networking TS has been in development for a long time, with no end in sight. The committee are now going to have to make a decision between shipping it as is (based on Asio) and abandoning it in favour of something else. Either way the community won't be happy with yet another delay.

Pattern matching using `is` and `as`

P2392R0 by Herb Sutter.

This proposal builds on the existing pattern matching paper and adds the new keywords to the language:

- `is` for matching constraints;
- `as` for all cast scenarios, except `reinterpret_cast`.

In addition, the structured bindings syntax `[]` is generalized to allow nesting and wildcards.

Herb Sutter proposes that the new matching syntax is allowed not only in `inspect` statements of the original pattern matching paper, but throughout the language, including `if` statements and `requires` clauses.

I so hope that we get pattern matching in C++23, and this proposal seems like a sensible idea built on top of that. If it gets pattern matching adopted faster and makes it simpler to use, it's a good thing.

Efficient seeding of random number engines

P0205R1 by Moritz Klammler.

> The purpose of this proposal is to make properly seeding a random number engine in a (non-)deterministic way easy and fast. The changes are non-breaking and can be implemented as a pure library solution with current C++20 features.

C++11 introduced random number generation facility with the header `<random>`. However, there is no way to seed the RNG properly with `std::random_device`. The code most people use is severely flawed. A better way is to use a *seed sequence*, but the code required is long and non-trivial. The proposed way is much simpler and more thorough:

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

```
7 }
```

Stacktrace from exception

P2370R0 by Andrei Nekrashevich and Antony Polukhin.

This paper proposed a new function `std::stacktrace::from_current_exception()` which can be used in a `catch` clause to dump stack trace corresponding to the current exception. It looks like a very useful feature which would bring C++ just a little bit closer to Java (!) which, as we all know, is a language that converts XML configuration files to exception stack traces.

Implementing this could be expensive though, as the stack information would need to be propagated through the exception handling machinery, making it popentially less efficient.

Abort-only contract support

P2388R0 by Andrzej Krzemieński and Gašper Ažman.

This paper proposes a 'minimum viable product' for the much-delayed contracts feature of C++. It would serve as the first phase of introduction of the full contracts, whenever that happens.

The goals of the paper are:

- Be a coherent whole, and provide a value to multiple groups of developers.
- Be small enough to guarantee that it will progress fast through the standardization pipeline.
- Be devoid of any controversial design issues; this is to obtain the maximum consensus.

The proposed precondition can result in either no action or a run-time check that results in program abort if it fails.

I wonder how much discussions and controversy this one will generate.

match(it)

Speaking of pattern matching: match(it) is a lightweight header-only pattern-matching library for C++17 with macro-free APIs. It is single-header, doesn't allocate heap memory, is cross platform, `constexpr`-ready, doesn't have external dependencies, supports composable matching and user-defined patterns. There are many examples, and even migration guides from Rust and Pattern Matching proposal.

The Reddit crowd is very impressed by the library and especially by the absence of macros in the API. Of course this leads to a side discussion on why macros are bad.

This is what redditor **gracicot** says:

> Macro are textual and have no idea what C++ is and what C is. They don't have scope, cannot be isolated, and create all sorts of problems, especially since C++ is using headers to consume declarations from other translation units.

Redditor **Foundry27** disagrees:

> I'd argue that the biggest problem with preprocessor macros is the community's general lack of familiarity with them ("general" meaning "for the general C++ programmer"), and the near-total absence of teaching material for how to use macros in a safe, effective manner.

To this **johannes1971** replies:

> This completely misrepresents the position of 'the community'. The dislike comes from all the bad sh*t macros do, not a general lack of understanding.

I don't know how production-ready this library is, but it surely looks impressive, and since we're unlikely to get the official language pattern matching until at least C++26, this may just become a usable replacement.

```cpp
#include "matchit.h"

constexpr int32_t factorial(int32_t n)
{
    using namespace matchit;
    assert(n >= 0);
    return match(n)(
        pattern | 0 = expr(1),
        pattern | _ = [n] { return n * factorial(n - 1); }
    );
}
```

{fmt} 8.0

The excellent text formatting library {fmt} by Viktor Zverovich reaches version 8. Some of the new features include:

- Compile-time formatting string checking enabled by default (but needs C++20 `consteval`).
- Much faster format string compilation.
- New format string literal `_cf` which replaces the `FMT_COMPILE` macro.
- Initial support for compiling {fmt} as a C++20 module.

The article *A quest for safe text formatting API* by Viktor Zverovich provides some background for the compile-time format string checking in {fmt}, including

dynamic extensible formatting and zero-overhead format string compilation. Apparently what {fmt} does is not trivial to do in Rust in an extensible way.

FTXUI - A C++ functional terminal user interface

Arthur Sonzogni from Paris created a remarkable user interface library based on reactive programming paradigm. Redditors say that the library looks great, and its API is simple and elegant. It is also compatible with WebAssembly, so the examples can be used from the browser. They look pretty impressive, too. The library also supports Linux, macOS and Windows, has keyboard and mouse navigation, implements all the standard controls, and can be used instead of a GUI. The programming constructs it uses remind me of SwiftUI, a declarative UI framework for Apple platforms:

```
vbox({
    hbox({
      text(L"left") | border,
      text(L"middle") | border | flex,
      text(L"right") | border,
    }),
    gauge(0.5) | border,
  });
```

FTXUI is used by several projects, among them **git-tui**, a rich Git console interface by the same author.

Size type convention, or `size_t` confuses people again

A redditor asks, what should they use for length of an array, `size_t` which seems like a natural choice, or some other type.

Most replies tell the original poster (OP) to just use `size_t` as it's used everywhere in the library. But some advise using a signed type, and OP is not sure about this:

> Why would I use a signed type? Array Length and Memory Size can't be negative?

This prompts a link to Bjarne Stroustrup's paper P1428R0: Subscripts and sizes should be signed. Redditor **Untelo** says:

> Negative values resulting from erroneous logic are easy to detect early, and signed overflow has undefined behaviour. This makes signed types safer for representing non-negative integers. <...> It may seem counterintuitive but that is in fact the case. Overflow of integers representing numbers (as opposed to say bitmasks) is almost always a bug. In the case of unsigned types, the compiler and other tooling has no choice but to assume that you the programmer

> intended for overflow to occur. In the case of signed types however, the undefined behaviour allows for more freedom. Trapping (i.e. causing a fault which would crash the program immediately) is a valid manifestation of undefined behaviour. In addition this situation can be detected by a variety of analysis tools either statically or at runtime.

Redditor **ioctl79** says:

> <...> there are existing tools for identifying and trapping UB. If you use unsigned types, you opt out of all of those.

Redditor **Ameisen** replies with a useful information:

> Clang **ubsan** has `-fsanitize=unsigned-integer-overflow`.

And lastly, C++20 has `std::ssize(container)-> std::ptrdiff_t` which you can use to 'fix' the original unfortunate decision of using `std::size_t` for sizes in the standard library.

True love

True love (via Reddit)


The pupil of your eye can expand over 45% when you see something you love.
Code compiled successfully
OK