C++ Club UK Meeting 134

Gleb Dolgich

2021-08-26

Contents

WG21 mailing 2021-08

The August 2021 WG21 mailing is out, with its corresponding Reddit thread. The following papers caught my attention.

`basic_string::resize_and_overwrite`

P1072R9

The ninth revision (!) of this proposal has got some fixes and enhancements. This should be a useful feature to make string manipulation more efficient.

Static `operator()`

P1169R2

This paper has received some fixes since it's been reviewed, and seems to be progressing well. It also has been implemented in the EDG (Edge) compiler, which powers Microsoft Visual Studio IntelliSense, and does nothing else, as far as I know.

Conversions from ranges to containers

P1206R6

This proposal received some review fixes too. To remind you, it adds `ranges::to` function that can materialize any range as a specified container, and also adds

tagged constructors and the functions `insert` and `assign` for standard containers and string types.

Formatted output

P2093R8

The eighth revision of the formatted output proposal from the text formatting ninja Viktor Zverovich added lots of poll results for various aspects of the paper. If nothing else, they demonstrate how difficult it is to get a consensus for a new feature in C++.

Stacktrace from exception

P2370R1

This useful feature becomes off by default because it can increase memory consumption by exceptions. The authors also propose a linker flag that can turn the feature on or off.

Expression Function Body

P2425R0

> This paper suggests a way of defining single-expression function bodies, that is aimed at solving one of the issues, which prevented standardizing the Abbreviated Lambdas proposal.

C++ lambda syntax is too verbose. It would be so nice to have a shorter syntax for most common cases. Unfortunately, the previous attempt failed because of various corner cases and subtle issues. This proposal is an attempt to solve one of those issues: differing semantics with regular lambdas, meaning that the same function body would return different types depending on whether it was an abbreviated lambda or a normal one.

To remind you, the issue was caused by the way abbreviated lambdas were defined, which meant that reference semantics was the default, as opposed to value semantics in normal lambdas. So, the following two lambdas would return different types:

```
auto l1 = [](int* p) { return *p; }; // returns a value
auto l2 = [](int* p) => *p;          // returns a reference
```

The paper offers two solutions:

1. The minimal expression has reference semantics by default, and the non-minimal expression has value semantics by default.
2. The minimal expression has value semantics by default, with opt-in reference semantics.

Example:

```
auto l3 = [](int* p) *p;         // returns a reference
auto l4 = [](int* p) return *p; // returns a value
auto l5 = [](int* p) => *p;      // returns a value (alternative 1)
auto l6 = [](int* p) auto(*p);   // returns a value (alternative 2)
```

In the end though, the paper just adds a whole lot more corner cases, so I'm not sure now if it's worth it. Maybe the current lambda syntax is actually OK…

Slides: BSI issues with P2300

P2428R0

The new senders/receivers proposal P2300 `std::execution` has been discussed by the committee; there was a consensus in favour of abandoning the previous executors proposal (P0443) and switching efforts to the new paper. So it's another reset. *Deep sigh.*

These slides show the issues The British Standard body had with the new executors proposal. I'm sure all of these are valid issues, and there are quite a few of them in the slides. But I fear C++ executors won't be executing anything till at least C++26, and then we'll have to wait until C++29 for them to work out the kinks and become usable (*see: coroutines*).

We'll see more on this shortly.

Minimal module support for the standard library

P2412R0 by Bjarne Stroustrup

This paper wasn't in the mailing as it was released just before it went out. Bjarne urges to adopt basic standard library modules as soon as possible so that they get in C++23, the main module being `std`. This would simplify adoption of the C++ modules, with more granular standard library modules possibly coming later.

Reddit offers this insightful comment:

> This seems like a reasonable, well-thought out proposal that would have real practical benefits to C++ programmers now. As such I expect it to be rejected in favor of a proposal that will solve all module related issues forever (which hasn't been written yet) that will never actually happen.

Boost 1.77.0

Boost 1.77.0 has been released. Highlights include:

- A new C++14 reflection library, Describe, by Peter Dimov. Unavoidably macro-based until we have proper reflection in C++ (probably in C++26).

- A new C++14 lambda helper library, Lambda2, by Peter Dimov. If you've been watching in frustration the failed proposals for simplifying lambda syntax, this one could be for you:

```
#include <boost/lambda2.hpp>
#include <algorithm>

using namespace boost::lambda2;

int count_even(int const* first, int const* last)
{
    return std::count_if(first, last, _1 % 2 == 0);
}
```

- Boost.Filesystem v4 with lots of changes and new features.
- Tons of changes and new features in Asio, thanks to Christopher Colhoff - these include cancellation support for individual asyncronous operations.

See also the article on Boost 1.77 by Jens Weller.

ASIO

Vinnie Falco posted excitedly on Reddit about the new features in Boost.Asio 1.77:

> Kohlhoff is a C++ design god.

Vinnie Falco also replied to another redditor:

> Do note that all of these changes are built on Networking TS. In other words, none of the core concepts in Networking TS are incompatibly changed to support these features.

And then we have this redditor saying:

> I hate ASIO and I can't be alone. <...> ASIO has been getting a lot of praise recently due to the new version in Boost. I feel like I can't be the only one who just hates it with a burning passion. <...> Whenever a project of mine requires networking, I end up reaching for ASIO because it's the least worst option we have <...> And so when I want to do networking, I first have to wrap ASIO before I can focus on what it is I actually want to do. So that right there is issue #1: ASIO is just too low-level to do anything useful with. <...> Issue #2: It is just a giant minefield for lifetime issues. <...> Issue #3: ASIO pretty much requires `shared_ptr` everywhere to get cleanup even remotely correct.

The thread has some explanations and alternatives. Most replies say that Asio is a low-level foundation that enables other higher-level libraries to be built on top of it, like Boost.Beast by Vinnie Falco. If you use Asio, you have to buy into it at the application level.

Reditor **Minimonium** says:

> The whole issue of async models is that they're required to be universal. That's why standard executors are so important and why it's so frustrating to see so much ego from P2300. It's understandable that the standardization process sometimes may be hard and frustrating but I'd hope that the committee members would refrain from "I'll take my toys and go play by myself" tantrums.

They continue:

> Initially, the Executors proposal was based on the ASIO design. It has more than a decade of user experience, dozens of thousands of users, and achieved grand success in its field. But for standardization - it's important that an executor model that would be picked would satisfy the requirements of all members. And the ASIO model is more optimized around the IO part, hence the name. The issue though is that for the parallel processing domain - the tradeoffs are not optimal, so a part of the committee designed a new model called Senders/Receivers, which allows to queue continuations in a lightweight manner (which, unfortunately, prevents the use of Coroutines). All is great, we got the Unified Executors paper which seemed to be a consensus among the experts on how C++ must proceed. But Sender/Receiver folks were unsatisfied with the presence of ASIO facilities and, failed to convince people who need ASIO facilities that they don't actually need it, they decided to make a paper that would just scrap everything instead of proceeding with a compromise. Unfortunately, greenfield design tends to confuse people, making them believe that now they know how to solve everything for sure. Due to the lack of experience and the refusal to admit that all people make mistakes - they forgot that the requirements for IO async operations and parallel processing are not quite compatible and composable.

And then they finish with this:

> The P0443 was indeed a collaboration. The P2300 on the other hand is an unconditional departure, which is problematic with respect to disagreements from other members.

Wow, what a mess! This looks like a classic case of "great is the enemy of good".

Niall Douglas pitched in:

> I gotta be honest, I think you haven't spent enough time studying ASIO to use it correctly. <…> sure, ASIO is a legacy design with a lot of backwards compatibility cruft. Rather like C++ itself. So, same as with C++, for new code ignore the 80% of the stuff you don't need, use the 20% which makes for clean, highly maintainable, efficient code.

Another library mentioned in the thread for its much more friendly interface is POCO.

And *of course* we have this:

> I'm not trolling: use Rust. The grass really is greener.

You are trolling though.

When to use `shared_ptr` for lifetime management in Asio

Prompted by the Reddit thread we just discussed, Vinnie Falco asks when you should use `shared_ptr` for lifetime management in Asio in async programming in general. Except it's not really a question, as he says at the end of his 'question':

> There is no reason to be scared of shared pointers or avoid them just because of some anecdotal evidence that someone in the past sitting next to you in a cubicle wasn't able to use them correctly. Shared pointers make sense when ownership of an object is shared by multiple entites. This is typically the case for network programs.

However, others disagree. Redditor **ExBigBoss** says:

> Technically, Asio programs can be structured so as to avoid all forms of ownership. If one designs their program to have a pool of sockets that lives longer than the the threads that call `io_context::run()`, then no form of ownership is required. Instead, connection objects simply borrow the socket and use it for their own purposes, returning it back to the pool when the connection ends.

Niall Douglas is of the same opinion:

> This really is the right answer. I wish ASIO tutorials taught this instead of the anti-patterns which seem to confuse people. In the past decade of writing ASIO based code I can't think of once that I have used shared_ptr. *Just* structure your ASIO based program to never encounter lifetime issues. Problem solved.

Note to self: beware of answers that start with *'Just'*.

Talking Async

Chris Kolhoff started a series of videos accompanied by a GitHub repository, called Talking Async. From the announcement Reddit post:

> Game-changing new Asio features, C++20 coroutines, and live coding

The first video is called Why C++20 is the Awesomest Language for Network Programming.

The second video is discussing cancellation in depth.

This is a very welcome series from the author of Asio himself.

Intel C/C++ compilers complete adoption of LLVM

Intel's oneAPI C++ compiler now uses LLVM Clang front-end, delivering faster compile times, better optimizations, enhanced standard support, and support for GPU and FPGA offloading. The old ('classic') C/C++ compilers included in Intel Parallel Studio XE product are now legacy technology and won't be updated. For those users who can upgrade to the new compilers this will be a welcome change – gone will be the days of finding cumbersome workarounds for ICC internal compiler errors and C++ standard non-compliance.

Visual Studio 2022 Previews 2 and 3

Preview 2

Visual Studio 2022 Preview 2 has been announced a while ago on the Visual Studio blog. This build comes with the new version of the build tools, v143, which is still binary-compatible with the previous versions back to v140 (VS2015). It also has CMake integration for targeting Windows Subsystem for Linux 2 (WSL2). New productivity features for C++ developers include hot reload, when you can apply changes to a running application. It's like the old Edit and Continue feature, except in many cases you don't have to stop the application to apply the changes.

On the Reddit thread STL posted the STL changelog. Microsoft are beginning to merge C++23 features.

Preview 3.1

New for C++ programmers in Preview 3.1 (released 2021-08-16) are: better CMake support, improved IntelliSense, on-by-default C++ code linter, and improved hot reload feature.

Twitter

Viktor Zverovich of the `libfmt` fame translated a tweet from Russian:

If you didn't get the joke, check out Andrei Alexandrescu and his Wikipedia page.