C++ Club UK Meeting 131

Gleb Dolgich

2021-06-10

Contents

Status of the Networking TS

A redditor is curious: "Does anyone have any updates on the status of the C++ Networking TS?"

Redditor **14ned** (Niall Douglas), a committee member, replies:

> Last time I looked, it all depends on how covid goes. If normal committee work resumes early 2022, there is a chance Networking makes 23. If it's late 2022, there is very little chance. Currently the C++ 23 standard looks like it will be a fairly light one, but given how severely curtailed the committee's productivity has been, I still think it great anything got in at all. Equally, this probably means 26 will be a correspondingly bumper standard.

Then we have this very emotional and somewhat misguided reply from redditor **Plazmatic**:

> Why do they need to "meet up" to basically just make decisions on things. They are presumably all tech people, we have video chat, or heck, they could do what every other language does and, you know use text platforms to discuss these things. When rust or python makes updates, the people in charge of the language don't all need to be in the same room to make it happen. Same thing for C#, Go, Swift, Kotlin, and heck, even Java. Why is C++ special? Why do they need 5 star hotels and vacation destinations to talk about the future of C++?

Another redditor answers that:

> Basically it all boils down to being a ISO language and having to

> follow ISO processes, which as far as I am aware have already been greatly simplified for C++. Also note that the languages listed by you don't have any presence in many markets where the only option is between C, C++ and Ada, all ISO languages with certified compiler toolchains and language standard compliance.

And more from Niall Douglas:

> Late stage uncontentious proposals can make reasonable progress by teleconference. <...> Early stage proposals all went on pause over a year ago. None have progressed, none can progress, until onsite meetings return. <...> I am unaware of any programming ecosystem upon which billions of dollars of existing infrastructure lie which does not have multiple annual face to face gatherings.

This is what Niall says regarding the hotel "luxuries":

> Unless you go a day or two early, or stay after, you only see your hotel bedroom when sleeping and absolutely none of any facilities nor surrounds. You most definitely do NOT see the location you are in during the meeting itself, which particularly sucks for Kona.

Back on topic, **psyspy2** replies:

> Networking is dependent on executors. That needs to go first.

Redditor **Gracicot** expands on that:

> <...> we need a standard interface to schedule async operations.

Third-party solutions and links

- Seastar (GitHub) from ScyllaDB is an event-driven framework allowing you to write non-blocking, asynchronous code in a relatively straightforward manner. It supports C++17 and C++20, comes with its own native userspace networking stack, and is distributed under the Apache-2.0 licence.
- Asio is a cross-platform C++ library for network and low-level I/O programming that provides developers with a consistent asynchronous model using a modern C++ approach. It can be used either as part of Boost or in standalone mode.
- Video: *The Networking TS from Scratch: I/O Objects - Robert Leahy - CppCon 2020* on YouTube.

Yet another pamphlet about inlining

Thomas Lourseyre writes on his blog Belay the C++:

> I will try here to summarize all you need to remember about inlining, giving pros, cons and situations to use it and to not use it.

The author correctly points out that the `inline` keyword is *a hint* to the compiler, which can ignore it. He then goes through pros and cons, and most of those assume that inlining actually happens, like the pros (avoiding the function call overhead) and cons (code size bloat).

This redditor says:

> In modern C++ `inline` does not mean what you think it means. It currently is only used to be able to define the same function, global variable, or static member variable in multiple compilation units; while satisfying the "one definition rule".

Of course, `inline` being just a hint, the compiler is free to not inline the functions marked `inline`, but even in that case it allows you to keep function definition in the header file, which is useful if you have a header-only library. On the other hand, inline functions pollute headers and increase compiler memory requirements (we experienced this recently as a consequence of developers in one team keeping nearly *all* their functions in headers, which led to MSVC running out of memory on a 64GB machine).

Keep in mind that function templates are implicitly inline, so marking them `inline` is redundant.

There is a compiler-specific keyword `__forceinline` and the corresponding `__attribute__((always_inline))` which indicates that a function *must* be inlined. However, compilers are free to ignore that as well.

There is also a compiler-specific way to never inline a particular function. In GCC and Clang it is `__attribute__((noinline))`. In MSVC it is `__declspec(noinline)`. As I understand, this attribute will always be honoured by the compiler.

With link-time optimization (LTO, LTCG) modern compilers defer inlining until the link stage, when they can inline functions even not explicitly marked `inline`.

The author writes:

> Here is my advice: don't use `inline` unless you are 100% sure it is called inside a bottleneck.

Given all of the above, this advice doesn't make much sense.

For something invented as a 'tactical' solution for a particular problem with interrupt service routines that couldn't have function calls in them, the `inline` keyword turned out to be a language facility that gained additional meaning over the years, and continues to draw attention and fuel discussions in the C++ committee to this day.

Static inline variables

I wanted to remind you that C++17 introduced another very handy use of the `inline` keyword - static inline variables. They allow you to define static variables inside class declaration to avoid having to define them in the implementation file, reducing redundancy.

In C++14 and earlier:

```
// foo.hpp
class Foo
{
    static int bar;
};

// foo.cpp
int Foo::bar = 42;
```

In C++17 and later:

```
// foo.hpp
class Foo
{
    static inline int bar = 42;
    static constexpr int baz = 1; // implicitly inline
};
```

Library: TOML for modern C++

TOML++ v2.4.0

TOML is *Tom's Obvious Minimal Language*, 'a configuration file format for humans' developed by Tom Preston-Werner, the founder of GitHub (he doesn't work there anymore). The language is easy to read and understand, it allows comments (hey, JSON!) and is widely supported - implementations are available in C, C++, C#, Java, JavaScript, Swift, Scala, Python, Ruby and other languages. TOML++ is a modern C++ implementation using C++17 and C++20, released under MIT license. It supports JSON serialization, UTF-8, optional header-only mode, and works with MSVC, Clang and GCC.

The documentation is really nice. And conveniently, the author of TOML++ released a Doxygen wrapper called Poxy that you can use to produce similarly formatted documentation. One nice feature of Poxy is its improved Doxygen configuration file, which uses - you guessed it - TOML. As the author says in the Reddit thread:

> While I can't take credit for the HTML generator itself <...>, I did write an additional front-end layer for it that fixes various Doxygen bugs and oddities, adds features and post-processes the HTML for extra fancy bits.

So, looks like a really useful library. Consider using TOML for your next project's configuration.

Passing smart pointers in C++

Smart pointers as parameters

Another article appeared on this well-discussed topic, this time by Pranay Kumar, who works at Adobe. His article is pretty much a re-hash of Herb Sutter's guidelines in GotW #91 Solution: Smart Pointer Parameters with added illustrations and diagrams.

- Don't pass a smart pointer as a function parameter unless you want to use or manipulate it, such as to share or transfer ownership.
- Express a "sink" function using a by-value `unique_ptr` parameter. Use a non-const `unique_ptr&` parameter only to modify the `unique_ptr`.
- Pass `shared_ptr` by value only when you are sharing ownership. Otherwise use non-owning pointers or references.

In practice I've seen many codebases where everything is a `shared_ptr`, and as a consequence every function takes a `const shared_ptr&` because it has to pass it on to another function. Sometimes you get a circular reference somewhere in a `shared_ptr`, and then - surprise - there is a memory leak. Untangling these is often very time-consuming and involves the entire codebase.

Note that passing `shared_ptr` by value is really expensive, as it involves incrementing the atomic reference count, which is slow.

Returning smart pointers

Pranay says:

> If you really need to return smart pointers from a function, take it easy and always return by value.

This is a good advice, and it works well with factory functions that return newly created objects. If you return a `unique_ptr` from a function, it can be move-assigned to another `unique_ptr` or converted to `shared_ptr` for passing around. This is the samest and most flexible way, and it also works with polymorphism - you can return a unique pointer to a derived class from a function that has return value of unique pointer to base.

Pranay says this about return value optimisation (RVO):

> All modern compilers are able to detect that you are returning an object by value, and they apply a sort of return shortcut to avoid useless copies. Starting from C++17, this is guaranteed by the standard.

C++17 guarantees *copy elision* which means that your class doesn't even have to *have* a copy constructor or a copy assignment operator (or the corresponding move member functions) for RVO to work. Prior to C++17, even if no copying were actually to happen, compiler would require that the class had copy or move member functions available.

I found an article by Jonas Devlieghere that explains guaranteed copy elision in C++17.

> The main problem is that, without guaranteed elision, you cannot get rid of the move and copy constructor, because elision might not take place. This prevents non-movable types from having functions that return by value, such as factories.

Jonas also mentions a subtle issue with it:

> <...> nothing changes for NRVO in C++17 with guaranteed copy elision. This is because, as mentioned before, the change only involves *prvalues*. With NRVO, the named value is a *glvalue*.

You can check out the Reddit thread on the original article for some insightful comments.

C++ Core Guidelines

There are relevant sections in the C++ Core Guidelines:

- R.30: Take smart pointers as parameters only to explicitly express lifetime semantics.
- F.7: For general use, take `T*` or `T&` arguments rather than smart pointers.
- R.32: Take a `unique_ptr<widget>` parameter to express that a function assumes ownership of a `widget`.
- R.33: Take a `unique_ptr<widget>&` parameter to express that a function reseats the `widget`.
- R.34: Take a `shared_ptr<widget>` parameter to express that a function is part owner.
- R.35: Take a `shared_ptr<widget>&` parameter to express that a function might reseat the shared pointer.
- R.36: Take a `const shared_ptr<widget>&` parameter to express that it might retain a reference count to the object.

And again, if you want to write an article that is accessible to your readers free of charge, please don't use Medium, which puts your articles behind a paywall.

Coding font: Cascadia Code 2105.24

I previously mentioned the new Cascadia Code font from Microsoft. It will be included in the Visual Studio 2022 preview this summer. A new version of the

font has been released, and this time it has fancy italics! Go download and try it out with your current text editor or IDE. It's really nice and very readable.

Vcpkg package search

Vcpkg is a C++ package manager from Microsoft. It builds packages from source, and now offers a search function on its official website, which allows you to find the library you need across approximately 1500 available recipes. The nice thing is, vcpkg is cross-platform.

There is another unofficial website, vcpkg.info, which lists even more packages (around 1600).

Good to see C++ package management getting more attention.

Library: Tst - a test framework without macros for C++17 and C++20

Ivan Gagis announced on Reddit his new JTest-style unit testing framework tst that doesn't need macros (well, just a few) and supports C++17. It's cross-platform and comes under MIT licence. There is another testing framework without macros - Boost.UT (which is not actually Boost, ugh) - and Ivan wrote a comparison on the Reddit thread, which shows **tst** favourably vs. Boost.UT (but then that's not really surprising). To me Boost.UT looks more polished and has much better documentation, but it requires C++20 which for many developers is still some time away.

Both frameworks rely on `std::source_location` in C++20, but Fabio Fracassi posted an entire implementation of `source_location` for GCC and Clang using built-in compiler-specific functions:

- `__builtin_LINE()`
- `__builtin_COLUMN()`
- `__builtin_FILE()`
- `__builtin_FUNCTION()`

Sadly, these are not available in MSVC, so `__FILE__` and `__LINE__` macros are still necessary there until C++20.

I use GoogleTest at the moment, and it's full of macros. When C++20 arrives I will likely want to start reducing macro usage in my code, so these test frameworks should come handy.

Library: Flashlight

Flashlight is a fast, flexible machine learning library written entirely in C++ from the Facebook AI Research Speech team and the creators of Torch and

Deep Speech. It is well documented, requires C++17 and is available on GitHub under BSD License.

Flashlight has CUDA and CPU backends for GPU and CPU training. The included projects are:

- Automatic speech recognition
- Image classification
- Object detection
- Language modeling

Flashlight features just-in-time kernel compilation with modern C++, powered by ArrayFire tensor library. The ArrayFire company are consultants for AI & GPU computing projects.

> ArrayFire accelerated computing library is a free, general-purpose, open-source library that simplifies the process of developing software that targets parallel architectures including CPUs, GPUs, and other hardware acceleration devices. ArrayFire is used on devices from low-powered mobile phones to high-powered GPU-enabled supercomputers including CPUs from all major vendors (Intel, AMD, ARM), GPUs from the dominant manufacturers (NVIDIA, AMD, and Qualcomm), as well as a variety of other accelerator devices on Windows, Mac, and Linux.

Flashlight is being actively developed, the last commit was just days ago. It looks like a capable and easy-to-use option for your AI and ML projects.

Library: RmlUI

RmlUi is a C++ user interface package based on the HTML and CSS standards. It is a fork of the libRocket project, introducing new features, bug fixes, and performance improvements. It is also well documented.

The most interesting thing about RmlUI is that it doesn't need a web browser. It has its own layout engine, and it takes the HTML/CSS-like source files and turns them into vertices, indices and draw commands, and then you bring your own renderer to draw them. This is markedly different from all other web-like GUI libraries for C++ which tend to need a web browser to render the UI.

> And of course there is full access to the element hierarchy/DOM, event handling, and all the interactivity and customizability you would expect. All of this directly from C++, or optionally from scripting languages using plugins. The core library compiles down to fractions of the size it takes to integrate a fully fledged web browser as other libraries do in this space.

RmlUI supports Lua for scripting, is cross-platform, and even has a run-time visual debugging facility. The user controls their own update loop, calling into

RmlUi as desired. The library strictly runs as a result of calls to its API, never in the background.

Unfortunately, it seems that you have to handle user input yourself.

Still, RmlUI is a very impressive library. The sample gallery includes games and other interfaces. The library requires C++14 and is available on GitHub under MIT license.

I don't need to read the documentation

Reddit