C++ Club UK Meeting 137

Gleb Dolgich

2021-10-14

Contents

Why are you a C++ enthusiast?

Some feel-good stuff on Reddit:

> Recently I've been struggling with a discouraging feeling about c++. I'm inspired/motivated to learn C++ (as a hobby) by all of the cool projects and games that I like that use it. But there's this discouraging thought in my mind. Would I enjoy the language if it wasn't popular/used in inspiring projects? These thoughts have been making my enjoyment of the language feel fake. Every time I think of the language now, I get negative emotions, whereas before I was really enjoying myself.

We are all people, and we all have emotions. Job satisfaction is a very important issue that affects our lives, so the feeling the OP has when they use C++ is not something to be dismissed.

Redditor pedersenk writes:

> I find it very difficult to get passionate / motivated to write things that won't work in 10-15 years. <...> with C++, the lifespan is there. Almost anything you write with C++ can be tweaked, mangled and #ifdef'ed to work on whatever weird and wonderful platforms of the future.

Redditor Pragmatician writes:

> Because C++ (and C) code is basically the bedrock of every domain software is developed in. It is used for very demanding software, be it games, Web browsers or embedded programs. It talks directly to the OS and hardware.
>
> On top of all that, it provides very powerful abstraction facilities and people are still finding new ways to leverage them. Because

> of this, I believe C++ made me a better programmer and also made it very easy for me to switch to other languages.
>
> All in all, I'm enthusiastic about C++ because there is just so many interesting things to do and areas to explore and I don't feel constrained in any way.

Redditor rhubarbjin says:

> I love how I can encode high-level concepts into the type system, how I can teach the compiler to check my code for correctness, and (of course) how none of that sacrifices run-time performance.
>
> When I write C++ code, I can be somewhat confident that it is bug-free.

(*Wow, I'd like to have that level of confidence at some point.*)

Redditor CxIpp writes:

> I enjoy making things run fast. In that arena, you have basically 3 options: Assembler, C or C++. C++ allows me to write the shortest fastest code with the least effort.

Redditor The_Northern_Light says:

> Gotta go fast.

Redditor WasterDave seconds that:

> It's fast as hell. These days it has usable abstractions - maps, vectors etc - and is still fast as hell.

Redditor ffscc says:

> I take criticism of C++ with a high degree of skepticism, especially on online communities. Many people have not evolved their opinions since the 90s and early 2000s. During those years OOP fever was at its peak, C++ compilers were difficult, the standard was stagnant, and multithreading primitives were primitive to nonexistent. But nowadays the standard is far more comprehensive and responsive, compilers have greatly improved, sanitizers and tooling make development far smoother, and OOP fanaticism has basically died out. Anyway, in my experience, the biggest detractors of modern C++ are usually C programmers who don't want to learn anything new.

The thread has many more good replies, so if you are low on motivation and want to switch to another language go and have a read.

What networking looks like

Vinnie Falco writes:

> Is there some documentation or announcement where it was collectively decided that this is how I/O and asynchrony will look like in standard C++? Why are we pushing this instead of standardizing existing practice - Networking TS has been available for 6 years, based on technology that has stood the test of time for almost 20 years.

Vinnie Falco is a big proponent of Asio and Boost.Beast, his high-level networking library. He seems disappointed with the direction that the Committee took for C++ Networking (*aren't we all*).

He presents a code example as an illustration of his point. Thing is, it's a unit test for pipe, so not exactly user-level code, as Kirk Shoop pointed out. Kirk then presented a user-level code fragment of an echo server, and I must say it doesn't look significantly simpler. It seems to follow the reactive programming pattern, as used in the latest senders/receivers paper, and looks very hard to understand and reason about.

Luckily, Kirk Shoop then rewrites the example code using coroutines, which looks way better.

Answering the OP question, Kirk says:

> The committee has not decided what networking would look like.

About that.

C++ committee polling results for asynchronous programming

Ben Craig posted this on Reddit.

These poll results don't mean much out of context, but are bound to generate lots of pointless discussions and useless hot takes.

Now, let me present my hot take on the polls.

Breaking news: we are not getting networking any time soon!

Ben Craig writes:

> I'd like to share the results of recent C++ committee polling on future asynchrony models for C++. This relates to "Senders and Receivers" / S&R (P2300) and the Networking TS / Asio (P2444).
>
> Each of these proposals represents an enormous amount of effort from the respective authors, and from the committee. "Executors" have been discussed in WG21 (mostly in the concur-

> rency study group) since at least 2012 (N3378), with Christopher Kohlhoff's contributions starting at least as early as 2014 (N4046). Over the last two years, I count 14 meetings in LEWG discussing the NetTS and S&R models. In the Spring of 2020, LEWG did in depth design reviews of P0443 that involved many meetings beyond those 14.
>
> This polling and interpretation was not taken lightly. The chairs know that the NetTS and Asio represent decades of work. However, these polls indicate that there is insufficient consensus for the NetTS to progress beyond LEWG in its current form. Once these issues have been addressed, LEWG can look at the Networking TS again.

The polls look pretty grim to me. (Again, out of context, sorry.) Strap in peeps, here they come!

Poll 1: The Networking TS/Asio async model (P2444) is a good basis for most asynchronous use cases, including networking, parallelism, and GPUs

SF	WF	N	WA	SA
5	10	6	14	18

Weak Consensus Against: LEWG won't be pursuing P2444 as a general async model.

Poll 2: The sender/receiver model (P2300) is a good basis for most asynchronous use cases, including networking, parallelism, and GPUs.

SF	WF	N	WA	SA
24	16	3	6	3

Consensus in favour, but this doesn't mean that P2300 as it stands today will be sent forward to LWG. More work is needed on it.

Poll 3: Stop pursuing the Networking TS/Asio design as the C++ Standard Library's answer for networking.

SF	WF	N	WA	SA
13	13	8	6	10

No consensus. Votes in favour of stopping Asio-based work were those who wanted networking to be based on senders/receivers and those who

thought that Networking TS needed TLS support. This means Networking TS champions will need to do *even more work* for it to be considered for the C++ standard. I don't know how much more patience Chris Kohlhoff has for this.

Poll 4: Networking in the C++ Standard Library should be based on the sender/receiver model (P2300).

SF	WF	N	WA	SA
17	11	10	4	6

Weak consensus in favour: but there is *no paper for this yet*. And still, 10 people opposed!

Translation of the above polls: *"Do we want Networking TS in C++? No. But do we want senders/receivers? Also no."*

Poll 5: It is acceptable to ship socket-based networking in the C++ Standard Library that does not support secure sockets (TLS/DTLS).

SF	WF	N	WA	SA
9	13	5	6	13

Some think a modern networking standard must include support for secure sockets. Others think it must only provide low-level abstractions and allow security to be built on top, the main argument being that security standards evolve much faster than the C++ standard.

There were other telecon polls that I saw posted on GitHub. Again, these are useless when taken out of context, and are likely to be superseded bu subsequent polls. By the way, my impression was that the wording of all these polls was not ideal, with lots of redundancy and weirdly constructed sentences that were bound to cause confusion. Polls are phrased as statements and not as questions, but maybe it's just me who finds it strange.

POLL: Knowing what we know today (*talk about redundancy here*), we should continue considering shipping the Networking TS in C++23, as is.

SF	WF	N	WA	SA
10	6	8	3	3

No consensus. But wait!

POLL: Knowing what we know today, we should continue considering ship-

ping P2300 senders/receivers in C++23.

SF	WF	N	WA	SA
11	6	5	5	4

Also no consensus! And that's not all:

POLL: We must have a single async model for the C++ Standard Library.

SF	WF	N	WA	SA
5	9	10	11	5

No consensus!

The Reddit discussion thread is very active. Some interesting tidbits there:

- You can use Asio today (which is what Networking TS is based on) – it is a well-tested and proven library that is used all across the industry.
- Libunifex (a real-life implementation of P2300) is used at Facebook, which is the driving force behind the new senders/receivers paradigm (Kirk Shoop works there, and Eric Niebler used to work there before moving to NVIDIA).
- Vinnie Falco is, shall we say with a gigantic dose of restraint, not a fan of the senders/receivers paper. He confirmed in the thread he voted strongly against P2300 and in favour of Networking TS. His comments there were incisive and sarcastic. Then he declared that he will be stepping away from his committee involvement (*a beacon of hope enters the chat*).
- People are generally not impressed (to put it mildly) with the current senders/receivers-based code.
- By trashing the original Networking TS in favour of the half-baked P2300 the senders/receivers crowd have squandered good will of the community.

And that's all very very sad, given C++23 is close to the feature cut-off point. I wouldn't even be surprised if C++26 didn't get standard networking. Probably C++29, or, more likely, never. I'm really pessimistic at this point. Sigh.

And now, a rant.

When I look at these polls from the sidelines, it appears to me that the committee is a dysfunctional mess where tribalism and hidden agendas clash, egos are bruised and votes are cast in bad faith to the detriment of C++ and the community. I wonder how much longer this is going to continue before someone decides that radical changes are needed in the way it's run. (*Of course, this is my uninformed opinion, so take it with a huge grain of salt*).

ClangQL

A very interesting development: using SQLite virtual file system on top of clangd in order to enable using SQL to query various aspects of C++ code-bases.

- GitHub (LGPL)
- Reddit

It's a proof of concept and work in progress, but you can already see what is possible. With tool integration this could provide additional introspection capabilities for IDEs, although LGPL licence makes it a bit inconvenient.

BTW, I'm told a similar feature was in development in AT&T Bell Labs in the 90s, but sadly went nowhere. The same developer also worked on Dot and GraphViz, which is very much alive.

Pure virtual functions in C++

Dheeraj Jha gives an example of defining a pure virtual function. Normally you would stop at marking a virtual function as pure and thus turning the class into an abstract class, forcing derived classes to override and implement their version of that function. Sometimes, however, you may want to define some basic behaviour that derived classes could use as a prelude to the overridden version, or as a fallback.

Note that:

- You can define a pure virtual function, but only outside the class declaration.
- Derived function has to call the base version explicitly (except this one weird case — keep reading).

Example code

Reddit expands on the topic and mentions an interesting related trick: to mark a class as abstract you can declare a pure virtual destructor. For example, all virtual functions in your base class have meaningful default implementations, so you don't necessarily want to make derived classes override them, but at the same time you want to make sure the base class cannot be instantiated. In this case you use a pure virtual destructor. You will have to implement it though, as it will be called as part of derived class destruction.

Example code

Herb Sutter discussed these topics in his *Guru of the Week* (GotW) #31 back in 2011. He also mentioned a rare case of "*pure virtual function call*" error that can happen sometimes — and I've seen it manifest in a Windows error message that I'm sure didn't make any sense to an ordinary user. To handle

such an error, Herb Sutter proposed implementing pure virtual functions that should never be called and putting some diagnostics there.

Remember not to call virtual functions from constructors, as they don't behave as virtual functions when called from there. Scott Meyers writes:

> During base class construction, virtual functions never go down into derived classes. Instead, the object behaves as if it were of the base type. Informally speaking, during base class construction, virtual functions aren't.

This article discusses how a call to a pure virtual function can happen, including an obscure case of indirect reference to a dangling pointer:

```
1 AbstractShape* p1 = new Rectangle(width, height,
    valuePerSquareUnit);
2 std::cout << "value = " << p1->value() << '\n';
3 AbstractShape* p2 = p1;  // Need another copy of the
    pointer.
4 delete p1;
5 std::cout << "now value = " << p2->value() << '\n'; // !!!
```

It can happen when the deleted pointer memory is left as is (the standard says it's undefined, so anything can happen). If it's left unchanged, `p2` is an 'dead' instance of the base abstract class, as this is how *vtbl* was left.

The author built several test programs to expose this error using various compilers, which are very old now. Hopefully modern compilers provide better static diagnostics for errors like this.

Some lesser-known powers of `std::optional`

Raymond Chen writes:

> C++17 introduced `std::optional<T>` which lets you augment the values of a type `T` with a bonus value known as `std::nullopt` which semantically represents the absence of a value. A `std::optional` which holds the value `std::nullopt` is known as empty.

He then reminds us of the basic operations on `std::optional`: checking for a value with `has_value()`, retrieving a value with `value()`, assigning a value with the overloaded assignment operator, and clearing the value with `reset()`.

Other less-known powers of `std::optional` are:

- It converts to boolean `true` if it has a value, and `false` if it is empty. Note that if the `std::optional` contains a `bool` it doesn't test the contained value, so to avoid confusion you may want to keep using `has_value()` in that case.

```
if (x.has_value()) { ... } // before
if (x) { ... }             // after
```

- An empty `std::optional<T>` compares unequal to any `T`.

```
if (opt.has_value() && opt.value() == 0) { ... } // before
if (opt == 0) { ... }                            // after
```

- An empty `std::optional` compares less than any non-empty `std::optional`, and also less than any value. This should probably be avoided as it adds confusion.

Lastly, Raymond Chen writes about the failure modes when getting a contained value:

- `x.value()` — this will throw `std::bad_optional_access` if the object is empty;
- `*x` — this will not perform any verification if the object is empty as it is undefined behaviour (UB).

Reddit has some more tidbits.

Recursive Variant

- Reddit
- GitHub (BSL)

Quote:

> Variants are exceedingly useful in C++, but they suffer from a singular and fundamental shortcoming: unlike sum types in many other languages, there's no mechanism to define recursive variants in C++.

In other languages like Haskell you can refer to the type being defined within the type itself:

```
data JsonValue = JsonNull
               | JsonBoolean Bool
               | JsonNumber Double
               | JsonStr String
               | JsonObject (Map String JsonValue)
               | JsonArray [JsonValue]
```

Not in C++ though. The following C++ code doesn't work, because `json_value` is an incomplete type. You could use a pointer, but if you want value semantics you can't use `std::variant`.

```
using json_value = std::variant<
    std::nullptr_t,                    // json null
```

```
    bool,                                 // json boolean
    double,                               // json number
    std::string,                          // json string
    std::map<std::string, json_value>,    // json object
    std::vector<json_value>>;             // json array
```

This kind of works, thanks to the RVA library, which 'hides' the current type in a container, so that it's a pointer in the end:

```
using json_value = rva::variant<
    std::nullptr_t,                        // json null
    bool,                                  // json boolean
    double,                                // json number
    std::string,                           // json string
    std::map<std::string, rva::self_t>,    // json object
 // std::map<std::string, json_value>
    std::vector<rva::self_t>>;             // json array
 // std::vector<json_value>
```

No word on performance though.

Astrée static analyzer

Website

From the website:

> Astrée is a static code analyzer that proves the absence of runtime errors and invalid concurrent behavior in safety-critical software written or generated in C or C++.
>
> Astrée primarily targets embedded applications as found in aeronautics, earth transportation, medical instrumentation, nuclear energy, and space flight. Nevertheless, it can just as well be used to analyze any structured C/C++ programs, hand-written or generated, with complex memory usages, dynamic memory allocation, and recursion.
>
> Astrée is sound — that is, if no errors are signaled, the absence of errors has been proved.

Sounds like a very serious and reliable static analysis tool. Probably costs a ton of money. If you want to try it out, you must fill out a form and email or even *fax* it to them. Talk about being stuck in the 1990s.

BehaviorTree

A library for designing behavior trees for use in robotics, games, or as a replacement for finite state machines.

- GitHub (MIT, C++14)
- Documentation

From the website:

> BehaviorTree.CPP has many interesting features, when compared to other implementations:
>
> - It makes asynchronous Actions, i.e. non-blocking, a first-class citizen.
> - It allows the creation of trees at run-time, using a textual representation (XML).
> - You can link staticaly your custom TreeNodes or convert them into plugins which are loaded at run-time.
> - It includes a logging/profiling infrastructure that allows the user to visualize, record, replay and analyze state transitions.

The library comes with a companion project Groot, a GUI for creating Behavior Trees.

Funding for the project came from the European Union's Horizon 2020 Research and Innovation Programme. Yay EU!

Weggli

GitHub (Apache-2.0)

From the website:

> Weggli is a fast and robust semantic search tool for C and C++ codebases. It is designed to help security researchers identify interesting functionality in large codebases.
>
> Weggli performs pattern matching on Abstract Syntax Trees based on user provided queries. Its query language resembles C and C++ code, making it easy to turn interesting code patterns into queries.

Weggli is programmed in Rust! What do you know, it can be useful to us C++ programmers after all! (*Just kidding, Rust is great.*)

Example: checking for potentially uninitialized pointers

```
weggli '{ _* $p;
    NOT: $p = _;
    $func(&$p);
}' ./target/src
```

Twitter wisdom