C++ Club UK Meeting 132

Gleb Dolgich

2021-06-24

Contents

Trip report: Summer 2021 ISO C++ standards meeting (virtual)

Herb Sutter writes:

> On Monday, the ISO C++ committee held its third full-committee (plenary) meeting of the pandemic and adopted a few more features and improvements for draft C++23.

The following proposals were accepted into C++23.

P1938: `if consteval`, by Barry Revzin, Richard Smith, Andrew Sutton, and Daveed Vandevoorde

In C++17 we had `if constexpr`. In C++20 Standard Library we got `is_constant_evaluated()`. Turns out, if you want to branch on whether or not a particular expression is constant-evaluated, as opposed to runtime-

evaluated, for example, if you want to have both compile-time and run-time branches in a `constexpr` function, you can't just combine `if constexpr` and `is_constant_evaluated()`, as it will always evaluate to `true`. With `if consteval`you don't even have a condition, just curly braces for when it's constant-evaluated.

P1401: Narrowing contextual conversions to `bool`, by Andrzej Krzemieński

This proposal enables testing integers as booleans in `static_cast` and `if constexpr` without having to cast the result to `bool` first (or test against zero).

P1132: `out_ptr` - a scalable output pointer abstraction, by Jean-Heyd Meneide, Todor Buyukliev, and Isabella Muerte

This proposal adds `out_ptr` and `inout_ptr` abstractions to help with potential ownership transfer when passing pointers to functions that have a `T**` parameter.

P1659: `starts_with` and `ends_with`, by Christopher DiBella

This generalizes these function templates for `ranges::starts_with` and `ranges::ends_with`.

P2166: Prohibit `basic_string` and `basic_string_view` construction from `nullptr`, by Yuriy Chernyshov

This improves the situation with construction of `string` and `string_view` from `nullptr`, which is UB, by adding a compile-time check.

Other news

Some progress happened with Concurrency TS, which gained suport for hazard pointers and read-copy-update (RCU).

Herb Sutter says:

> We're going to keep meeting virtually in subgroups, and then have at least one more virtual plenary session to adopt features into the C++23 working draft in October.
>
> The next tentatively planned ISO C++ face-to-face meeting is February 2022 in Portland, OR, USA. (Per our C++23 schedule, this is the "feature freeze" deadline for design-approving new features targeting the C++23 standard, whether the meeting is physical or virtual.)

Visual Studio 2022 Preview 1 now available

Announced on Microsoft Visual Studio blog, the first preview of the next VS release, VS2022, is available for download. Its main purpose is to test and tune the new 64-bit Visual Studio platform. The preview can be installed side-by-side with older versions of VS. The Reddit crowd is excited. Not all developers are lucky to be using a recent version of VS at work. This redditor says:

> We currently use Visual Studio 2013 in work and my manager is planning on upgrading to 2022 when it's out. How much better of an experience am I about to have?

To which the first reply is, "I'm so sorry."

There will be no ABI break this time, but STL says this about ABI:

> It's a limited form of compatibility, from VS 2015 onwards, that allows application developers to upgrade their toolset even when they rely on separately compiled third-party libraries that were built with an earlier toolset. <...> Note that only VS 2015/2017/2019/2022 are binary-compatible; VS 2013 and earlier were from the "break ABI every major version" days, which were glorious for STL maintainers (I remember how awesome it was), and horrible for many customers who had extreme difficulty keeping up with the churn. <...> I'm continually advocating for the vNext ABI-breaking project to happen, but the stars haven't aligned yet.

This version does not contain a new toolchain, which is expected in the next preview.

An amusing snippet from the thread: someone asks, how old STL is, saying:

> You have looked 20 since 2003. I was surprised to see you in an MSDN video from 2013 looking the same.

And STL replies:

> I AM ETERNAL. (Probably because I don't spend a lot of time in the sun.) When pretending to be a human, I claim to be in my late 30s, among the eldest of the millennials.

Dave Abrahams joins Sean Parent at Adobe

Dave Abrahams tweeted:

Dave Abrahams #BLM @DaveAbrahams

Psyched to be be joining @SeanParent at Adobe, where we'll reboot the Software Technology Lab. Sean's contributions to software and the art of programming are legendary; the source of many of my good ideas. I can't think of a better way to empower programmers! #GonnaBeAwesome

2w • 09/06/2021 • 01:52

Dave was a C++ programmer and participated in the committee work before he went to work on Swift at Apple. Now that he is at Adobe, a C++ shop, does that mean he is back in the C++ world? Time will tell...

What is memory safety, or we should all abandon C++

I read this article called *What is memory safety and why does it matter?*, and to say it's biased would be an understatement of the week. But then it was written by the Internet Security Research Group, so they must have some data to back it up.

> Memory safe languages include Rust, Go, C#, Java, Swift, Python, and JavaScript. Languages that are not memory safe include C, C++, and assembly.

The author focuses on two types of memory bugs: out-of-bounds reads and writes, and use-after-free. After presenting some stats on vulnerabilities and exploits they come up with this pearl:

> These vulnerabilities and exploits, and many others, are made possible because C and C++ are not memory safe. Organizations which write large amounts of C and C++ inevitably produce large numbers of vulnerabilities that can be directly attributed to a lack of memory safety. These vulnerabilities are exploited, to the peril of hospitals, human rights dissidents, and health policy experts. Using C and C++ is bad for society, bad for your reputation, and it's bad for your customers.

That's not all. Using C++ apparently also impacts performance negatively, because nowadays performance needs multithreading, and it's hard to do in C++, as Mozilla's example shows us. They couldn't get it right and so they rewrote everything in Rust.

Oh, did I mention the author really likes Rust?

Towards the end they acknowledge that not all C++ programs are unsafe:

> Some practices which can lower the risk of using an unsafe language are: Using some modern C++ idioms which can help produce more safe and reliable code; Using fuzzers and sanitizers to help find bugs before they make it into production; Using exploit mitigations to help increase the difficulty of exploiting vulnerabilities; Privilege separation so that even when a vulnerability is exploited, the blast radius is smaller.

They finish with this:

> We look forward to a time when choosing to use an unsafe language is considered as negligent as not having multi-factor-authentication or not encrypting data in transit.

I had a link to the Reddit thread discussing this article, but I can't find it anymore. The first comment was along the lines of "it looks like Rust programmers cannot praise Rust without piling on C++98". This article certainly reads like a sales pitch. Rust is a fine language, but C++ isn't going anywhere any time soon, and it's not a good look for Rust fans.

`<format>` in Visual Studio 2019 version 16.10

A new post has appeared on the Microsoft Visual Studio blog telling us about the new text formatting facility in C++20 that is available in Visual Studio 16.10 or later when using the switch `/std::c++latest`. It is based on the {fmt} library and has the following major differences:

- Named arguments are not supported.
- None of the miscellaneous formatting functions like `fmt::print` or `fmt::printf` are supported.
- Format strings are not checked at compile time.
- There is no support for automatically formatting types with an `std::ostream& operator<<(std::ostream&, const T&)` overload.

The Reddit thread has many positive comments. One insight is that if your codebase already uses {fmt} there is no real rush to migrate to `std::format` as it's a subset of {fmt} and requires C++20.

Why We Need Build Systems

Martin Bond wrote a long essay on build systems and why we need them. The article is of general interest but has some details targeted at embedded developers. It's a beginning of a series of articles focused on CMake. The author says the most popular build systems for C++ are CMake and Meson, but at the same time he is worried that CMake seems to be a major pain point for C++ developers.

Martin Bond explores such topics as source code organisation, source file dependencies, compilation options (with a focus on GCC). He taklks about include file locations, and which flavour to use when (system include syntax <> vs. user include syntax ""). He warns against using relative paths outside the current file location in user include statements, as this leads to hard-to-track dependencies and unnecessary coupling between project components. He touches on code generation options, preprocessor directives, built-in macros like `__cplusplus`. He finishes by discussing build settings, post-build processing (especially important for embedded programming) and limitations of a build system.

Beware that in MSVC the macro `__cplusplus` is broken and doesn't report the correct standard version used unless you use Visual Studio 2017 version 15.7 or later and specify the flag `/Zc:__cplusplus`. Otherwise you get the value `199711L` which corresponds to C++98.

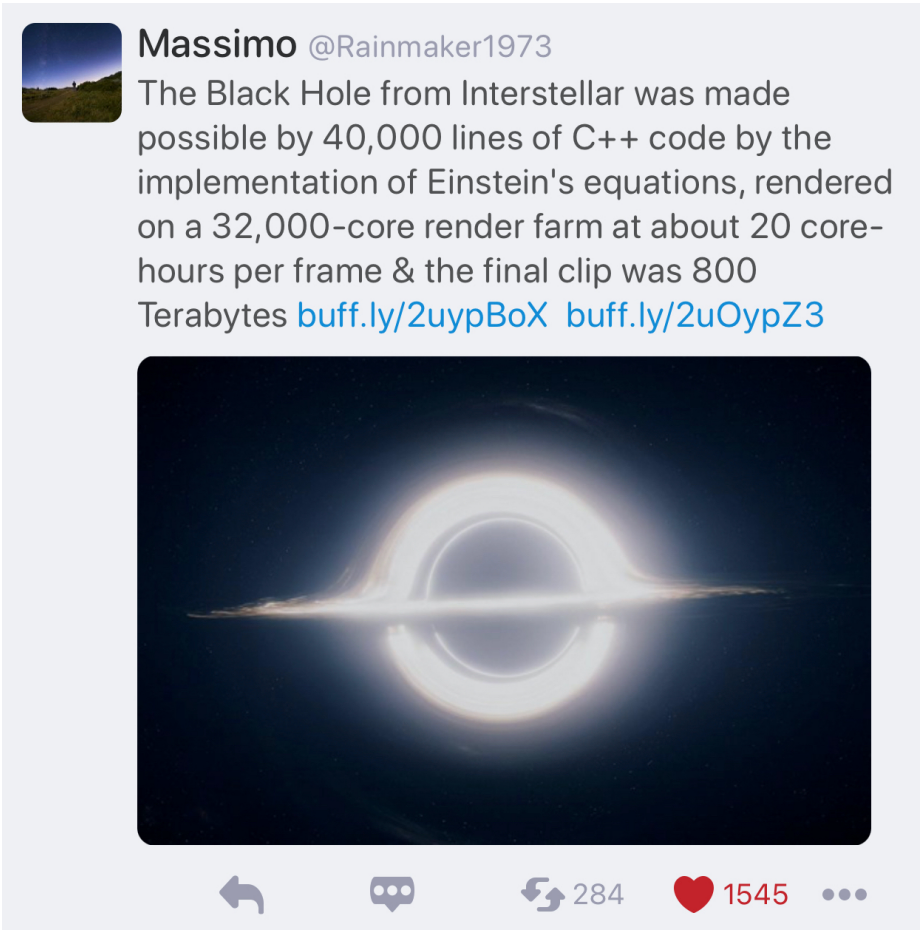
This is a very thorough and useful article, and the series looks very promising.

The top Reddit comment summarises build systems:

> We need them and love them and hate them.

Black hole from Interstellar

In the sci-fi movie Interstellar, we see a black hole in all its glory, and turns out, it is scientifically correct. It was no big susprise to learn that the sequence was rendered using simulation software written in C++.



This is the paper *Gravitational Lensing by Spinning Black Holes in Astrophysics, and in the Movie Interstellar* describing the process. From the abstract:

> *Interstellar* is the first Hollywood movie to attempt depicting a black hole as it would actually be seen by somebody nearby. For this, our team at Double Negative Visual Effects, in collaboration with physicist Kip Thorne, developed a code called DNGR (Double Negative Gravitational Renderer) to solve the equations for ray-bundle (light-beam) propagation through the curved spacetime of a spinning (Kerr) black hole, and to render IMAX-quality, rapidly changing images.

An article in Wired magazine titled *Wrinkles in Spacetime - The Warped Astrophysics of Interstellar* has more details.

> Filmmakers often use a technique called ray tracing to render light and reflections in images. “But ray-tracing software makes the gen-

> erally reasonable assumption that light is traveling along straight paths," says Eugénie von Tunzelmann, a CG supervisor at Double Negative. This was a whole other kind of physics. "We had to write a completely new renderer," she says.

SIMD Vector Classes for C++

Vc is a library of portable, zero-overhead C++ types for explicitly data-parallel programming.

> Current C++ compilers can do automatic transformation of scalar codes to SIMD instructions (auto-vectorization). However, the compiler must reconstruct an intrinsic property of the algorithm that was lost when the developer wrote a purely scalar implementation in C++. Consequently, C++ compilers cannot vectorize any given code to its most efficient data-parallel variant. Especially larger data-parallel loops, spanning over multiple functions or even translation units, will often not be transformed into efficient SIMD code.
>
> The Vc library provides the missing link. Its types enable explicitly stating data-parallel operations on multiple values. The parallelism is therefore added via the type system.

An example using built-in `float` type:

```
using Vec3D = std::array<float, 3>;
float scalar_product(Vec3D a, Vec3D b) {
  return a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
}
```

With **Vc**, this doesn't look much different:

```
using Vc::float_v;
using Vec3D = std::array<float_v, 3>;
float_v scalar_product(Vec3D a, Vec3D b) {
  return a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
}
```

But it will auto-magically scale to 1, 4, 8, 16, etc. scalar products calculated in parallel, depending on the target hardware's capabilities.

The library is cross-platform and comes under BSD-3-Clause license.

The note at the top says:

> You may be interested in switching to std-simd. GCC 11 includes an experimental version of `std::simd` as part of **libstdc++**, which also works with clang. Features present in Vc 1.4 and not present in std-simd will eventually turn into Vc 2.0, which then depends on std-simd.

If we look at the **std-simd** project, or `std::experimental::simd`, it implements a similar concept of portable, zero-overhead C++ types for explicitly data-parallel programming. Currently it only supports GCC 9. Using `std::experimental::simd`, the above example would look like this:

```
using std::experimental::native_simd;
using Vec3D = std::array<native_simd<float>, 3>;
native_simd<float> scalar_product(Vec3D a, Vec3D b) {
  return a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
}
```

It's encouraging to see how C++ makes it simple to use the new advanced programming features like data parallel programming.

C++ for fun ... ctional programmers

I watched this talk by Harald Achitz at Func Prog Sweden. There are slides available for your reference. There is also the corresponding Reddit thread.

Right at the beginning Harald says that he likes to talk about C++, and that C++ evokes many reactions, not all of them rational. Sounds familiar. Then we see a brief history of C++, from 1982 to the current C++20 standard. Harald emphasizes the fact that C++ is a multi-paradigm language. He then presents some example of common functional programming operations and idioms in C++:

- *filter* using `std::copy_if` algorithm, a lambda that checks the condition, and `std::back_inserter` to put the filtered items in another container;
- *map* using `std::transform` with a transformation lambda and `std::back_inserter` again;
- *reduce* using `std::accumulate` and `std::reduce` algorithms (the latter can perform the operation in parallel).
- *transform* and *reduce* using `std::transform_reduce` and two lambdas (one for reduce and the other for transform) – this one also takes execution policy, which can be `std::execution::par` for easy parallelization.

Harald reminds us that the naming is different in C++. There is a handy algorithm selection chart by Conor Hoekstra that can help here.

The observations from code samples for all of the above were that there were no raw loops, no `new` or `delete`, no pointers, and memory was managed automatically. Then a question was raised: how other functional concepts map to C++, namely, higher-order functions, immutability, recursion, function purity, and referential transparency.

Higher-order functions

Aside from lambdas, there are third-party libraries available, like Lift by Björn Fahller, which has a number of time-saving functions for combining other

functions, like `lift::if_then_else()` that takes 3 functions (condition, true-branch, and false-branch) and returns a combined function that can be passed to an algorithm.

Immutability

`const` in C++ is not exactly the same, says Harald. In C++ we have value semantics, mutable-by-default, and copy-by-default, where assignment creates a copy, and so do function value arguments.

There is a library of persistent and immutable data structures called immer by Juanpe Bolívar.

Recursion

Tail call optimization is not part of the standard and is compiler-dependent, so not guaranteed, like in many functional programming languages. Use loops and algorithms.

Pure functions and referential transparency

GCC has `__attribute__((pure))` and `__attribute__((const))` which work in C but rely on the developer to guarantee this behaviour. In C++ we have `constexpr` which makes the compiler check that the function has no side effects.

Pointers and resources

Harald tells his audience of functional programmers that pointers are like a `Maybe` type pointing to a resource. On resource management Harald says:

> Scope is king.

That was easier to say than RAII (which presented some difficulty later, so Harald quipped that C+ doesn't have great marketing). He then says:

> For the objects on the stack, the closing brace is the garbage collector. There is no magic.

He mentions smart pointers, too, and reminds that in today's C++ there's no `new` or `delete`.

> Pointers are nothing to fear.

Ranges

Harald mentions C++20 features like Ranges, which are better composable, can be lazy, and hide details like iterators.

Conclusion

As Harald says to his audience, now that you want to write C++, how to proceed? He lists some books:

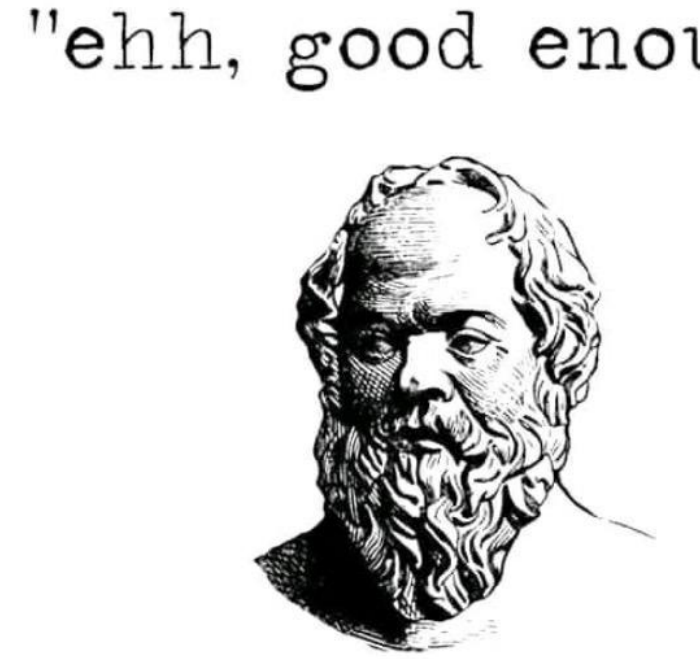
- *A Tour of C++*, 2nd ed., by Bjarne Stroustrup
- *From Mathematics to Generic Programming*, by Alexander Stepanov and Daniel Rose
- *Elements of Programming*, by Alexander Stepanov and Paul McJones
- *Functional Programming in C++*, by Ivan Čukić

Final note from me

This *in-person* meetup took place in Sweden in September 2020, at the height of the coronavirus pandemic, in a closed room presumably full of people, and those who were on camera weren't wearing masks. Videos like this are kind of scary to watch.

Quote for your code review needs

A quote from Reddit that you can use in your code reviews:



Eh, good enough. – Anonymous developer