C++ Club UK Meeting 143

Gleb Dolgich

2022-02-03

Contents

Why do you like C++?

An amusing thread on Reddit.

Some replies:

> I like it because it does what I ask it to do.
>
> And also because it does not do things I didn't ask it to do #
>
> Philosophy: "Programmers should be free to pick their own programming style, and that style should be fully supported by C++." #
>
> I like it because it makes you say "wtf why?" at compile time rather than runtime. #
>
> I don't like C++ except for a singular reason - my employer pays me well for writing C++. #

2021-01 ISO C++ mailing

Mailing, Reddit

Let's look at some pape-e-e-ers!

Direction for C++

P2000

The latest update to this paper moves goalposts for C++23 and C++26, admitting that reflection and pattern matching are now targeted for C++26. Contracts look optimistic for C++26 too.

Request for re-inclusion of `std::hive` proposal in C++23

P2523

> This is a brief rebuttal to the removal of P0447 from the C++23 agenda and a request for its reinstatement after discussion within SG14. In the view of the SG14 committee, this paper has had significant investigation, insight and input over the years from both SG14 and LEWG members and can be considered 'mature' in the sense that the broad strokes will not change.

This paper lists issues raised during LEWG discussion and addresses them (there is a new revision of P0447R18). The author, Matthew Bentley, finishes with this:

> In summary, we request that hive be put back on the agenda for C++23. There are other papers which are much longer but have had considerably less oversight, which are now in progress for C++23, and we consider this egregious.

A quick update: Bryce Lelbach replied to this on GitHub:

> We're a week away from C++23 design freeze. We can't consider adding more things to the plan for C++23.

So `std::hive` is not going to make it to C++23 in the end. Sad.

Scalable Reflection

P1240R2

A new revision of the reflection paper, nice! This revision was harmonized with other papers in this area and adopted syntax proposed in P2320, replacing `reflexpr` with `^`:

```
meta::info r1 = ^int;  // reflects the type-id int
meta::info r2 = ^x;    // reflects the id-expression x
meta::info r2 = ^f(x); // reflects the call f(x)
```

A notable change is replacement of the term *reification* with *splicing* (at the very least it's easier to pronounce, IMHO), which is *turning reflections into ordinary C++ source constructs*.

> Earlier versions of this paper were more exploratory in nature; this version uses experience with implementations based on earlier versions to narrow down a first set of metaprogramming features that are primarily aimed at providing reflection

facilities (with splicing and ordinary template instantiation handling generative programming). However, additional facilities (particularly, for code injection) have been explored along with this proposal and we are not confident that they can be added incrementally on top of this proposal.

The first and most complete is a fork of Clang by Lock3 Software (by, among others, Andrew and Wyatt, authors of this paper). It includes a large portion of the capabilities presented here, albeit not always with the exact syntax or interfaces proposed. In addition to these capabilities, Lock3's implementation supports expansion statements and injection primitives (including "fragment" support). Lock3 is currently not maintaining this implementation, however.

The second is based on the EDG front end (by Faisal and Daveed) and is less complete: It implements the reflection operator and most single splicers (but not the pack splicers; see below), and a few meta-library interfaces. It does not currently implement features in other proposals like expansion statements or injection primitives.

Abbreviated Parameters

P2424

This paper suggests alternative way of declaring parameter lists, one that let us omit parameter types.

WHAAAAAAT?

If a parameter list starts with double parentheses `((`, then all single identifiers are not types, but variables instead

```
// lambda:
[]((a, b)) { return a < b; }
// function:
auto less_than((a, b)) { return a < b; }
```

The paper also talks about omitting parameters altogether and leaving just commas.

Not sure about this one. C++ can be hard to read as is, let's maybe not make it harder?

`std::breakpoint`

P2514

This paper proposed a new library function `std::breakpoint` that stops program execution under debugger. Compiler-specific functions like that exist already: `__debugbreak` in MSVC, `__builtin_trap` in GCC and `__builtin_debugtrap` in Clang. On Windows you can use the Win32 API function `DebugBreakpoint`. It would be a useful function to have in the Standard library for situatons when using IDE or a debugger UI to set a breakpoint is difficult.

`std::is_debugger_present`

P2515

> This paper proposes a new function, `std::is_debugger_present`, that checks if a program is being debugged to aid in software development.

The first thing that comes to mind as a use case is not to allow a protected program to run under debugger, to prevent software copy protection or licensing code from being cracked.

Contract support – working paper

P2521

OK, so this paper doesn't propose anything new. Is its purpose to revive the effort, or to be the driving paper for contracts?

> We propose that there are two modes that a translation unit can be translated in:
>
> - **No_eval**: compiler checks the validity of expressions in contract annotations, but the annotations have no effect on the generated binary.
> - **Eval_and_abort**: each contract annotation is checked at runtime. The check evaluates the corresponding predicate; if the result equals false, the program is stopped an error return value.

I guess we'll see how it goes and if it gets into C++26.

What's up with modules?

From Reddit:

> I know MSVC allegedly has a feature-complete support for C++20 modules, but that doesn't mean much for cross-platform projects. There isn't much related discussion for CMake and GCC (not here, not on stackoverflow, nowhere), and from the little there is, I gather that it's mostly in such an

> early experimental phase that no one encourages using the feature. The MSVC team communicated their plans and status regularly, even joining the discussion on this sub, but other teams are eerily silent. Have users lost interest in modules? Have compiler devs/build system devs lost interest, or did they run into major problems?

Niall Douglas replies:

> <...> the real blocker for most people is cmake build support I think. And they're blocked on the compilers emitting the right Modules build metadata so they can implement cmake build support. It'll come eventually, maybe a year or two more.

Gabriel Dos Reis says:

> Interests in C++ Modules are stronger than ever. We need indeed widespread build system support, e.g. in CMake. The SG15 Tooling Study Group has adopted a format for specifying source level dependencies (useful outside module uses), also implemented in MSVC. I would love to see more progress in GCC and Clang for the benefit of the C++ community at large.

The proposal for the common dependency specification is P1689 Format for describing dependencies of source files

Looks like MSVC is ahead of all the big compilers in module support, but it's not all roses either. This redditor says:

> Visual C++ has huge amount of bugs in module implementation. At least BOOST aware projects couldn't be ported on modules. For example I have 3 module bugs in Visual Studio bug tracker:
>
> - https://developercommunity.visualstudio.com/t/Internal-Compiler-error-In-Project-With/1557697
> - https://developercommunity.visualstudio.com/t/Couldnt-Use-Eric-Niebler-Ranges-V3-Impl/1560676
> - https://developercommunity.visualstudio.com/t/Couldnt-Use-boostphoenix-Library-After/1560697
>
> and they are about 3 month is investigation status. So you can becalm yourself MSVC modules are not OK now.

From the replies:

> Same, I tried porting 4 cpps in real codebase to modules, got ICE in 3 of them

Some more bugs:

> https://developercommunity.visualstudio.com/t/warning-C4005:-Outptr:-macro-redefinit/1546919 I'd add this one. But

> I can confirm - modules are not usable in msvc 2022 :(

And more:

> Yep, from my coding exercises I would add the following issues:
>
> - `_Out` macros and other stuff when Windows headers get imported
> - Precompiled headers cannot be enabled in projects that use modules, lots of coffee time when using C++/WinRT with its pile of template magic generated out of IDL files.
> - Intelisense
> - Pressing F12 on a module symbol might just kill VS
>
> Currently there is no plan how to import stuff that depend on several macros to control their behavior (e.g. `_ITERATOR_DEBUG_LEVEL`) However, it is still the best experience versus GCC or even clang that apparently is still stuck with classical Apple/Google modules (from what I understand from existing docs).

Say you are starting a new cross-platform project. You are choosing a build system. What are your options if you want it to support C++20 modules?

- CMake — in development, waiting for compiler support.
- build2 — full support for modules with GCC announced in February 2021.
- Meson — at the design discussion stage.
- Bazel — at the design discussion stage, some experiments taking place.
- Premake — Supports MSVC modules, there is a proposal for GCC support.
- Xmake — full support for modules with MSVC/GCC/Clang (see examples). *What?!* I haven't tried this, but it just sounds too good to be true. What is Xmake anyway? Watch this space...

`const` all the things?

Arthur O'Dwyer wrote this article on his blog. In it he lists places where he uses `const` and where he doesn't:

Const:

> In function signatures: passing by `const` reference, `const` member functions

No const:

> In function signatures: passing by value Data members: never `const` > <...> the point of making a class with private members

> is to preserve invariants among those members. "This never changes" is just one possible invariant. Some people hear "never changes" and think it sounds a bit like `const`, so they slap const on that data member; but you shouldn't lose sight of the fact that the way we preserve invariants in C++ isn't with `const`, it's with `private`. Return types: never `const`

Rarely `const`:

> Local variables (*I tend to disagree*)

Reddit thread. A few redditors think that locals should be as `const` as possible.

This is another Reddit thread on making local variables `const`, and most commenters there agree with that.

Best CPU vs GPU: 879 GB/s Reductions in C++

This article iterates through various methods to speed up calculations in C++ using vectorization and parallelization on CPU and GPU. The author provides code snippets that add 1GB floating numbers together and their throughput data. Let's see what are the results.

- Plain C++ and STL: around 5.2–5.3 GB/s
- SIMD AVX2 using intrinsics: 17–22 GB/s
- OpenMP: 5.4 GB/s
- Parallel STL based on Intel Threading Building Blocks (TBB): 80–87 GB/s
- SIMD + Threads: 89 GB/s
- CUDA: 817 GB/s
- Thrust: 743 GB/s (nice code!)

```
thrust::reduce(numers.begin(), numbers.end(), float(0),
    thrust::plus<float>());
```

- CUB: 879 GB/s

This is from Thrust home page:

> Thrust is a parallel algorithms library which resembles the C++ Standard Template Library (STL). Thrust's high-level interface greatly enhances programmer productivity while enabling performance portability between GPUs and multicore CPUs. Interoperability with established technologies (such as CUDA, TBB, and OpenMP) facilitates integration with existing software.

Of all the above, based on the presented code snippets, I would choose Thrust.

Swift and C++ interoperability workgroup announcement

We discussed Swift C++ Interoperability Manifesto previously. There is a new development in this area: the creation of a workgroup dedicated to C++ and Swift bi-directional API level interoperability.

> Over the past few years there has been a huge amount of interest in bidirectional interoperability between Swift and C++. <...> the Swift compiler is now able to import and use some C++ APIs, including C++ standard library types like `std::string` and `std::vector`. <...> To advance the interoperability support between Swift and C++, we are announcing the formation of the Swift and C++ interoperability workgroup as part of the Swift project.

This is very welcome news. There are many tasks for which C++ is better suited (like working with memory or system APIs), and to be able to use C++ in a Swift program, especially with two-way access, will be really helpful. It also shows that no matter how focussed Apple is on Swift, they must have realised that C++ isn't going anywhere and there needs to be a way for Swift to use it.

Which standard C++ library elements should I avoid?

A redditor asks:

> I'm aware that due to ABI backward compatibility and historical reasons there are parts of standard library that shouldn't be used. I've seen people complaining and warning about regular expresions/unordered containers since they are (apparently) horrendously slow. What about the other stuff? What else is advised to be ignored?

The most sensible advice seems to be to avoid nothing and measure performance. However, many redditors concur that `std::regex` is very slow and shouldn't be used. Apparently `boost::regex` is about 10 times faster, and there is also much anticipated compile-time regular expressions (CTRE) by Hana Dusíková that you can with C++17 and C++20.

According to a Microsoft STL developer:

> `std::regex` is bad and you should forget it exists.

Regarding `std::map` versus `std::unordered_map`, opinions differ. Some say that `map` is slow and you should use `unordered_map` unless you need ordering. Others point out that `unordered_map` has more requirements for the element type (hashing). I liked this quote by **mark_99**:

> `unordered_map` is never very slow under any of the possible use cases.

A link was posted to a set of benchmarks for the most common hash map implementations, which show that `std::unordered_map` is indeed slow compared to other hash maps.

Anotehr redditor says not to use iostreams, as they are slow and add too much bloat to the binary, which is especially important in the embedded space. The **{fmt}** library is much faster, has very small code size, and is easy to work with.

To speed up maths this redditor says:

> If you don't rely on it disable math-`errno` on your compiler. The C standard mandates that otherwise single instruction operations like `sqrt` return their errors as `errno` value, which can result in half a page of cleanup instructions for every instruction of actual work.

A redditor says:

> I don't use thread anymore, just jthread.

Remember the proposal to make `<random>` usable, which didn't make it into C++23? A redditor writes:

> `<random>` is suboptimal and worth avoiding, because all of the generators provided are slow or have poor statistical qualities, and its generally difficult to use correctly.

There are also discussions of `std::list` vs. `std::vector` which you can read yourself.

I don't know which container to use

A related article by Chloé Lourseyre on the *Belay C++* blog goes into details of container selection given a task and requirements.

> As far as containers go in C++, since `std::vector` is well suited for most cases (with the occasional `std::map` when you need key-value association), it's become easy to forget that there are other types of containers. Each container has its strength and weaknesses <...>.

The author presents two matrices illustrating container properties, one for sequence containers and another for associative containers. She also shows Joe Gibson's data structure selection flowchart.

> Vectors are the most understandable structure because it is quite close to the plain-old arrays. Most C++ users aren't ex-

> perts, and `std::vector` is the container they know how to use best. We shouldn't make mundane code any more difficult to read and understand. Of course, as soon as you have special needs, you should use the most appropriate container, but that doesn't happen very often.

Chloé reminds us that optimization should not be the first consideration. Only after you measure the performance you should start thinking of choosing a faster data structure. She provides her own flowchart that works like a preliminary step before referring to Joe Gibson's flowchart for more granular selection. Her flowchart advises to use `std::vector` and `std::map` by default. There is a footnote clarifying use of unordered containers:

> Unfortunately, the presented flowchart lacks any `unordered_` associative containers. But you can think of it like this: "Values need to be ordered? Yes -> `map/set` ; No -> `unordered_map/unordered_set`".

There is a short Reddit thread discussing the article. The first reply is:

> Almost always vector.

Regarding maps, a redditor says:

> (Almost) Never. Use. `std::map`. If you think you need `std::map`, you really want `std::unordered_map`.

I'm looking forward to `std::hive` in C++26.

Twitter: be positive!

jorin 💤 💝 @YawningJorin

Use programming-positive language!

🚫 DON'T say "arbitrary code execution vulnerability"

✔️ DO say "surprise extension API"

4d • 11/12/2021 • 20:40

Drache @LukArthurion

@YawningJorin

🚫 Don't say "protected Data got leaked"

✔️ Do say "surprise remote backup"

thread

2h • 15/12/2021 • 09:53