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

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.

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 “frag-

> ment" 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.

Reddit discussion

TODO

`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.