C++ Club UK Meeting 136

Gleb Dolgich

2021-09-23

Contents

September mailing

The September committee mailing is out. This is what caught my attention.

`move_only_function`

P0288R9

This paper proposes a conservative, move-only equivalent of `std::function`. It is intended to be the same as `std::function`, with the following differences:

1. It is move-only.
2. It does not have the const-correctness bug of `std::function`.
3. It provides support for cv/ref/noexcept qualified function types.
4. It does not have the `target_type` and target accessors (direction requested by users and implementors).
5. Invocation has strong preconditions.

The thing that's unclear to me is that the paper mentions C++20 as the target. That ship has sailed, so presumably it's now targeted for C++23 and the authors just forgot to update the paper accordingly.

`std::hive`

P0447R16

The `std::hive` (or a bucket array) paper reaches revision 16! And there are still so many questions for the committee that I fear it's not going to make it into C++23.

A Plan for C++23 Ranges

P2214R1

> When Ranges was merged into C++20, it was knowingly incomplete. While it was based on the implementation experience in range-v3, only a small part of that library was adopted into C++20. <...> But now that the core of Ranges has been included, later has come and we have to figure out what to do for C++23.

Presentation: Plans for P2300 Revision 2

P2431R0

The slides present a plan for the second revision of the 'final final' `std::execution` paper, P2300. There is a GitHub issue tracker for all the issues related to the proposal, and if you look at it you will see there is no chance of the proposal making in into C++23.

Support for `#warning`

P2437R0

Apparently the preprocessor macro `#warning` that most compilers support is non-standard. This proposal is about fixing that.

The Asio asynchronous model

P2444R0

The C++ networking demigod Christopher Kohlhoff presents a high-level overview of the asynchronous model at the core of the Asio library.

> The asynchronous operations in Asio support callbacks, futures (both eager and lazy), fibers, coroutines, and approaches yet to be imagined.

CppBooks

There is a huge list of C++ books on github, in a repo creatively called CppBooks. Most are paid, but some are freely downloadable. A very useful resource.

Using `constexpr` `std::vector` and `std::string`

B. Filipek writes:

> `constexpr` started small in C++11 but then, with each Standard revision, improved considerably. In C++20, we can say that there's a culmination point as you can even use `std::vector` and `std::string` in constant expressions!

In addition to this, in C++20 there are `constexpr` algorithms to use with vectors and strings.

For this to work, the C++ Committee had to allow the following changes:

- `constexpr` destructors;
- `constexpr` dynamic memory allocation;
- in-place construction using placement new;
- `constexpr` `try/catch` blocks;
- new type traits, like `pointer_traits` and `char_traits`.

The main limitation is that newly constructed vectors and strings cannot leave the `constexpr` function:

> Because vectors and strings use dynamic memory allocations, and currently, compilers don't support so-called "non-transient" memory allocations. That would mean that the memory is allocated at compile-time but then somehow "passed" into runtime and deallocated. For now, we can use memory allocations in one constexpr context, and all of them must be deallocated before we leave the context/function.

Bungie coding guidelines

Bungie published their C++ coding guidelines. These guidelines are presented in the context of Bungie's latest blockbuster game, Destiny.

> There's a lot of teamwork and ingenuity that goes into making a game like Destiny. We have talented people across all disciplines working together to make the best game that we can. However, achieving the level of coordination needed to make Destiny isn't easy.
>
> It's like giving a bunch of people paintbrushes but only one canvas to share between them and expecting a high-quality portrait

> at the end. In order to make something that isn't pure chaos, some ground rules need to be agreed upon. Like deciding on the color palette, what size brushes to use in what situations, or what the heck you're trying to paint in the first place.

And this is a perfect analogy because, as we all know, Mona Lisa was famously painted by a bunch of guys with really good guidelines.

So, what are these guidelines that allowed Bungie to create a great game that is also completely bug-free? (*Just kidding*) It's a mix of formatting rules (*auto-formatting is coming*) and peculiar requirements, which they enforce with code reviews. The codebase is 5.1 million LoC (MLoC?), and some of it is 20 years old. They can't justify modernising the entire code base (*because management, of course*), so the guidelines aim to provide a balance between legacy and modern C++ coding practices. Additionally, they are working on multiple feature and patch branches at the same time, which adds complexity. Of the total 150 engineers, there are more 75 working on C++ code.

At Bungie they use so called *razors* — guidelines that 'shave off' complexity and 'sharpen focus'. Any new guideline is expected to align with one or more of these 'razors' to be adopted.

Favour understandability over time-to-write

- `snake_case` naming (*fair enough*)
- avoid abbreviations (*sure*)
- use helpful inline comments (*agreed*)

Avoid distinction without difference

- use American spelling (*okay*)
- use postincrement in general usage (*now wait a minute — this is not just an arbitrary difference*)
- `*` and `&` go next to the variable name instead of the type name (*this is just wrong — in C++ it's part of the type, unless you are using "C with classes"*)

Leverage visual consistency

- braces should be on their own line (*sure*)
- uppercase preprocessor symbols (*that's pretty common*)
- no space left of the assignment operator, to distinguish from comparisons: `my_number= 42` vs. `my_number == 42` (*I'm sorry, but this is just hilariously stupid — nobody does that, it looks terrible and doesn't make any sense*)

- leverage pointer operators (`*`/`&`/`->`) to advertise memory indirection instead of references (*another weird guideline*)

Favor patterns that make code more robust

- initialize variables at declaration time (*good*)
- follow const correctness principles for class interfaces (*good*)
- leverage asserts to validate state (*good*)
- avoid native arrays and use custom containers (*ok*)
- single `return` statement at the bottom of the function (*wait what? this is bloody stupid and doesn't make sense in C++ code where exceptions are enabled... oh wait, I bet they don't use exceptions too; still, a single return statement is a stupid idea which increases indentation levels unnecessarily and makes the code harder to reason about*)

Centralize lifetime management

- use engine-specific allocation patterns
- do not allocate memory from the OS directly
- avoid using STL for game code (*this is often what game engines do, justified or not; EASTL seems like a good replacement tailored for games*)

Oh, and have you noticed there is not a single mention of testing in the article?

This is from Reddit:

> Based on some of these "standards", and from the kinds of bugs I've seen playing the game, their code must be a real rat's nest. I don't know why they published this, but if I were an engineer looking for a job, this would send me looking elsewhere pretty quick.

Also:

> They notably did not specify whether or not they use C++ exceptions, which is like one of the fundamental aspects of a C++ style guide.

And this:

> Game developers are so consistently in their own bubble, refusing to pick up any advice from the industry around them. It's astounding. The amount of "we do it X way because it's fast" without ever having tested the performance is so large. The industry is full of wisdom based on no data and almost religious dedication to reinventing the wheel. Oh and of course terrible

> programmers. That single return policy is 100% a cover-up for a code base so smelly they have to have air fresheners in the server room.

There are four people in the Coding Guidelines Committee. They use their own experience to come up with new coding guidelines. I'm sure there are stories behind all the guidelines, and some of them must be a real hoot.

Bad practices in industrial projects

The article on the Belay the C++ blog discusses bad practices in C++ projects. The list includes:

- Overly long functions
- Creating classes unnecessarily (fully static classes instead of free functions in a namespace, publicly accessible classes instead of `struct`)
- Implementing undefiuned behaviour (the list of UB is too long)
- Comparing signed and unsigned integers
- Premature optimization
- Premature pessimization
- "Just-in-case" programming (*YAGNI - You Ain't Gonna Need It*)

The Reddit thread has some more discussion on the raised topics. The winner is the following comment:

> I'm a Python user and I have no clue what I just read but it sounds cool

Regarding classes vs. free functions, the user **staletic** replies to the author's point "*Why would you use class at all, where a namespace suffices*?" with this:

> To shut down ADL and not have users of your library "steal" function calls with their functions.

Apparently Bloomberg C++ guidelines prohibit free functions altogether because of this.

Nice but not so well-known features of modern C++

A redditor asks, what are lesser-known features of modern C++. Some of the replies are:

- inheriting constructors

```
class Derived : public Base {
public:
    using Base::Base;
};
```

- structured bindings in `for` loops

```cpp
for (const auto& [key, value] : m) { ... }
```

- `std::exchange` is a handy utility function for implementing move semantics. It's particularly useful for simultaneously transferring ownership and invalidating primitive types, where `std::move` would just copy them #
- The alias constructor of `std::shared_ptr`. You can convert a `std::shared_ptr<T>` into a `std::shared_ptr<U>` for some possibly unrelated `U`. For example, `U` could be a member of some class `T` #

STL says: `reinterpret_pointer_cast()` (the function that performs a `reinterpret_cast` on a `shared_ptr` while respecting the control block) can be implemented with the aliasing constructor, and is in MSVC's implementation. The aliasing constructor is more powerful as previously mentioned — given `struct Point { int x; int y; int z; };` you can get a `std::shared_ptr<int>` to any of its `x`, `y`, or `z`.

`reinterpret_pointer_cast()` will let you arbitrarily change the type, but not adjust the pointer's value.

Use case: For example, you could be implementing a lock free queue, where each node is `std::atomic<std::shared_ptr<Node<T>>>` but then for `front()`, `at()`, etc return `std::shared_ptr<T>` created via the aliasing constructor. This way live handles to the `T` contained in a `Node` prevent the `Node` from being freed until all handles are dead, so mutating changes to the queue don't free references to stored objects out from under the other consumers. #

- fold expressions in C++17
- immediately-invoked lambda expression
- `constexpr if` as a replacement for conditional compilation using macros
- `std::bit_cast` in C++20 allows to convert similar data types used by different libraries, like some imaginary `libA::point` to `libB::point` if they both have the same size. #
- *bit constants:* `0b[bits]` is amazing for writing masks, combined with the `'` separator. `0b1100'0000 == 0xC0`
- raw string literals:

```cpp
auto s = R"json({
  "key": {
    "desc": "raw string literals are really useful for
        this kind of 'string'"
  }
})json";
```

FoundationDB

A seriously impressive open source distributed database from Apple:

> FoundationDB is a distributed database designed to handle large volumes of structured data across clusters of commodity servers. It organizes data as an ordered key-value store and employs ACID transactions for all operations.

FoundationDB links:

- Home page
- Code - Apache licence, portable C++, language bindings
- Documentation
- Downloads
- Forums

This was the reaction on HackerNews to Apple open-sourcing FoundationDB after acquiring it:

> This is INCREDIBLE news! FoundationDB is the greatest piece of software I've ever worked on or used, and an amazing primitive for anybody who's building distributed systems.
>
> The short version is that FDB is a massively scalable and fast transactional distributed database with some of the best testing and fault-tolerance on earth. It's in widespread production use at Apple and several other major companies.
>
> But the really interesting part is that it provides an extremely efficient and low-level interface for any other system that needs to scalably store consistent state. At FoundationDB (the company) our initial push was to use this to write multiple different database frontends with different data models and query languages (a SQL database, a document database, etc.) which all stored their data in the same underlying system. A customer could then pick whichever one they wanted, or even pick a bunch of them and only have to worry about operating one distributed stateful thing.

The thread contains more interesting replies from developers and users of FoundationDB. If you are into databases, give it a read.

Library: Neither

A functional implementation of Either in C++14 as a way to handle errors returned from functions. A bit like `std::expected` or Boost.Outcome with a functional interface.

GitHub — MIT License, C++14, uses Buckaroo build system, which seems to use TOML for its configuration files.

```cpp
// a function that throws, sometimes we can't avoid it...
auto unsafe = [] {
  if (true) {
    throw std::runtime_error("error");
  }
  return 1;
}

// let's lift the exception into the typesystem
Either<std::exception, int> e =
    Try<std::exception>(unsafe);

e.left()
  .map([](auto const& e) {
    return std::cerr << e.what() << '\n';
  }); // print error if available

int result = e
  .leftMap([](auto) { return 42; })
  // ^ do nothing with exception and map to 42
  .rightMap([](auto x) { return x * 2; })
  // ^ do further computation if value available
  .join()
  // ^ join both sides of either

ASSERT_TRUE(result == 42);
```

The motivation section talks about how bad C++ exceptions are (*spoiler alert: they aren't*). If you can't or won't use exceptions in your code, I suppose you could check this library out.

STL Visualizers on GitHub

The Visual Studio debugger has this neat feature where you can show the data in a custom human-readable format, defining the actual formatting rules in XML files called visualizers. Some 3rd-party libraries also have their own visualizers, often supported by the community (Eigen, Boost). And now Microsoft has open-sourced all the STL visualizers on GitHub, to allow everyone to contribute, so that they stay updated. They announced it on the Visual Studio blog.

Using visualizers has become much simpler in VS2017 and VS2019. Previously you had to put them in a certain directory in your Documents folder, of all places. Now you can include the XML files in a dummy container project in your solution, and the debugger will automagically use them to display the appropriate data. Very neat!

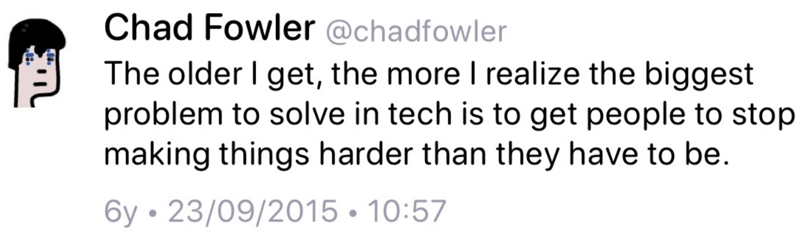
Twitter

Someone saw a poster in a mall that read:

> Lost children will be taught the C programming language

Quote

Chad Fowler via Twitter:



Chad Fowler @chadfowler

The older I get, the more I realize the biggest problem to solve in tech is to get people to stop making things harder than they have to be.

6y • 23/09/2015 • 10:57