C++ Club UK Meeting 138

Gleb Dolgich

2021-10-28

Contents

October mailing

The October committee mailing is out. See also: Reddit.

Pape-e-e-e-e-rs!!

First, let's quickly see some of the papers that have been freshly adopted into C++23.

Monadic operations for `std::optional`

P0798R8

Sy Brand's proposal to add monadic operations to `std::optional` makes it into C++23! These include `map` (or, as we call it in C++ for some reason, `transform`), `and_then`, and `or_else`. Every other language that has an Optional concept also has monadic operations for it, and now so does C++. You may not want to use these operations, but it's nice to have the option. (*See what I did there*?)

Deducing `this`

P0847R7

> We propose a new mechanism for specifying or deducing the value category of the expression that a member-function is invoked on. In other words, a way to tell from within a member function whether the expression it's invoked on is an lvalue or an rvalue; whether it is const or volatile; and the expression's type.

This is very useful when implementing several overloads of the same member function to deal with const and ref qualifiers. The proposed syntax is to 'lift' `this` into an actual member function parameter, so that normal deduction mechanisms could be used:

```
struct X {
    void foo(this X const& self, int i);

    template <typename Self>
    void bar(this Self&& self);
};
```

Add support for preprocessing directives `elifdef` and `elifndef`

P2334R1

Extend *init-statement* to allow *alias-declaration*

P2360R0

In C++20 you can use `typedef` in *init-statement* but not `using` alias declaration, which is a bit inconsistent, and this paper fixes it.

Multidimensional subscript operator

P2128R6

> We propose that user-defined types can define a subscript operator with multiple arguments to better support multi-dimensional containers and views.

```
template<class ElementType, class Extents>
class mdspan {
    template<class... IndexType>
    constexpr reference operator[](IndexType...);
};
int main() {
    int buffer[2*3*4] = { };
    auto s = mdspan<int, extents<2, 3, 4>> (buffer);
    s[1, 1, 1] = 42;
}
```

`move_only_function`

P0288R9

This paper proposes a conservative, move-only equivalent of `std::function`.

`zip`

P2321R2

This paper proposes four new views: `zip`, `zip_transform`, `adjacent`, and `adjacent_transform`, and related functionality. The Ranges feature gains more useful stuff.

Printing `volatile` Pointers

P1147R1

This fixes a quirk of stream output for volatile pointers which were being converted to other types, like `int` or `bool` because they couldn't be printed as normal `const void*` due to qualifier mismatch. Sure, whatever.

Byteswapping for fun&&nuf

P1272R4

This adds an efficient way to flip (swap, reverse) all the bits of an integer type without having to resort to compiler intrinsics.

Heterogeneous erasure overloads for associative containers

P2077R3

> The authors propose heterogeneous erasure overloads for ordered and unordered associative containers, which add an ability to erase values or extract nodes without creating a temporary `key_type` object.

Character sets and encodings

P2314R4

More long-overdue improvements to Unicode string support.

What is a `view`?

P2415R2

Clarifications for the role of `views` in the Ranges feature.

> The paper really stresses two points throughout:
>
> - views are lightweight objects that refer to elements they do not own
> - views are O(1) copyable and assignable

Support `std::generator`-like types in `std::format`

P2418R2

This enables formatting results of coroutine-based generator function results by switching `std::format` to taking parameters by forwarding references. It has been implemented in the {fmt} library since v6 (currently at v8).

Now, some other papers that caught my attention.

Standard Library Modules `std` and `std.compat`

P2465R1

`import std;` imports everything in `namespace std` from C++ headers (e.g. `std::sort` from `<algorithm>`) and C wrapper headers (e.g. `std::fopen` from `<cstdio>`). It also imports `::operator new` etc. from `<new>`.

`import std.compat;` imports all of the above, plus the global namespace counterparts for the C wrapper headers (e.g. `::fopen`).

I think this paper has been adopted into C++23.

Distributing C++ Module Libraries

P2473R0

Daniel Ruoso of Bloomberg proposes a convention for distributing module-based libraries.

> This paper proposes a format for interoperability between build tools, compilers, and static analysis tools that facilitates the adoption of C++ Modules where libraries are distributed as pre-built artifacts as opposed to the build system having access to the entirety of the source code.

I can see how big firms need some sort of convention for their internal library distribution, so this proposal will come handy as long as everyone follows the same convention, just like compilers with their C++ module binary interface files. Oh wait…

Closure-Based Syntax for Contracts

P2461R0

Gašper Ažman and his co-authors think that attribute-based contract syntax is too limiting. They propose a closure-based syntax instead, with three new context-sensitive keywords: `pre`, `post`, and `assert`. The syntax is similar to the `requires` syntax used for concepts:

```cpp
auto plus(auto const x, auto const y) -> decltype(x + y)
    pre { x > 0 }
    pre {
        /* check for overflow - badly */
        (x > 0 && y > 0 ? as_unsigned(x) + as_unsigned(y) > as_unsigned(x) : true) &&
        // since these are conditional-expressions, use '&&' to combine them
        (x < 0 && y < 0 ? as_unsigned(x) + as_unsigned(y) < as_unsigned(x) : true)
    }
    // ret is as-if auto&&
    post (ret) { ret == (x + y) }
{
    assert { x > 0 }; // this is currently "valid" syntax,
                      // but we should reclaim it.
    auto cx = x;
    return cx += y;
}
```

The authors list some of the advantages of this syntax over the currently proposed attribute-based syntax, of which I especially like this one:

> It's consistent with the rest of the language, instead of inventing a yet-another minilanguage.

I can see how this paper has the potential to cause a non-trivial amount of bikeshedding in the contracts study group. Honestly, after all the kerfuffle that happened with contracts, where a good-enough solution was derailed at the last minute by those who wanted a perfect one, I lost all emotion towards contracts and now I'm just watching indifferently from the sidelines.

Let us now look at another area that is totally going super well in the committee.

Networking and Senders/Receivers

- P2464R0 Ruminations on networking and executors
- P2471R1 NetTS, ASIO and Sender Library Design Comparison
- P2469R0 Response to P2464: The Networking TS is baked, P2300 Sender/Receiver is not

The two sides – Networking TS/Asio and Senders/Receivers – are writing letters to each other in the form of proposals. The saddest thing is the fact that there *are* two sides.

Fix the range-for loop

Nico Josuttis wrote a paper P2012R2 proposing to fix a long-standing problem with the range-for loop. As it stands today, it's really easy to hit lifetime bugs that lead to a crash. When the expression iterated over contains more that one function call and one of the functions returns a temporary, its lifetime is not extended, which leads to UB.

Examples:

```
for (auto e : getTmpColl()) // OK
for (auto e : getTmpColl().getRef()) // Runtime error
for (char c : getVectorOfStrings()[0]) // Runtime error
for (auto e : getOptionalVector().value()) // Runtime error
```

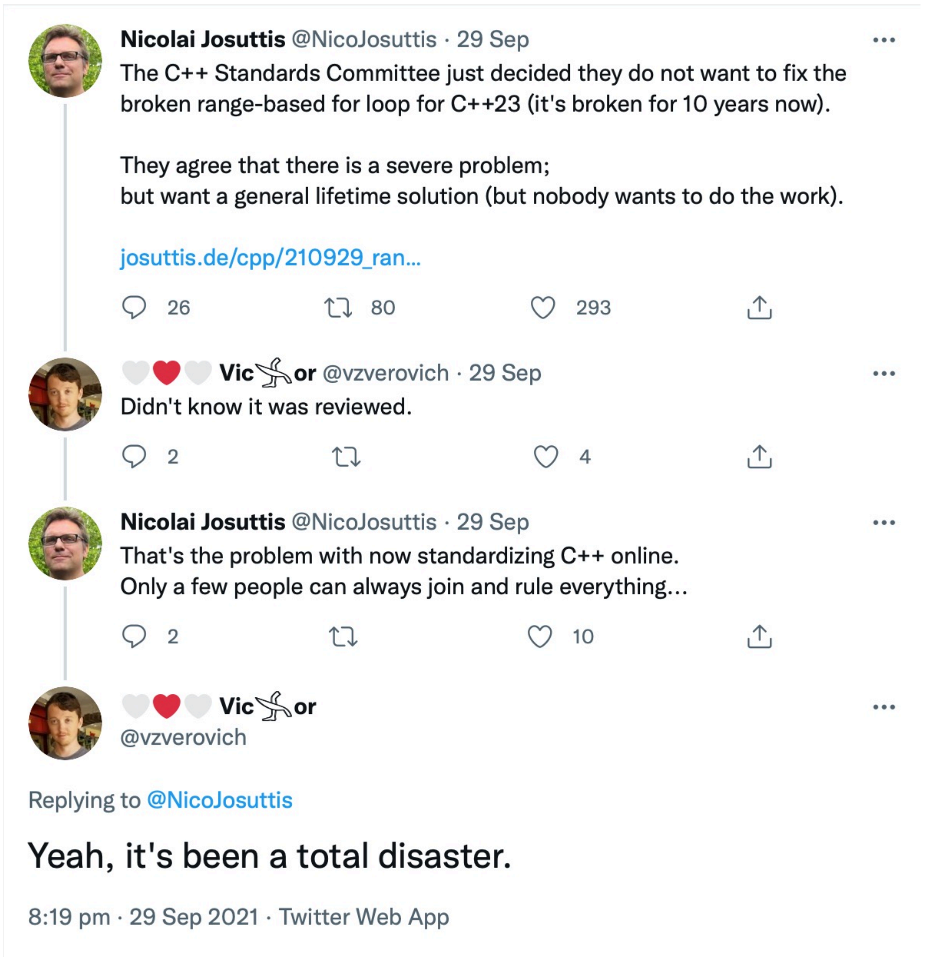
The current solution seems to be warning about the possible UB in code style guides. The MISRA coding standards prohibit more than one function call in *for-range initializer*.

The EWG votes acknowledged the problem strongly, and were in favour of a solution that could break existing code. However, EWG weren't sure about a new kind of "safe" loop.

Nico's proposed solution was to add range-for loop to the list of cases where lifetime of temporary objects is extended:

> When a temporary object is created in the *for-range-initializer* of a range-based `for` statement, such a temporary object persists until the completion of the statement.
>
> The lifetime of temporaries that would be destroyed at the end of the full-expression of the *for-range-initializer* is extended to cover the entire loop.

On 29 September 2021 the paper got rejected by the committee (didn't get enough consensus). Disappointment was expressed on Twitter:


Nicolai Josuttis @NicoJosuttis · 29 Sep

The C++ Standards Committee just decided they do not want to fix the broken range-based for loop for C++23 (it's broken for 10 years now).

They agree that there is a severe problem;
but want a general lifetime solution (but nobody wants to do the work).

josuttis.de/cpp/210929_ran...

26 80 293

Vic⛷or @vzverovich · 29 Sep

Didn't know it was reviewed.

2 4

Nicolai Josuttis @NicoJosuttis · 29 Sep

That's the problem with now standardizing C++ online.
Only a few people can always join and rule everything...

2 10

Vic⛷or
@vzverovich

Replying to @NicoJosuttis

Yeah, it's been a total disaster.

8:19 pm · 29 Sep 2021 · Twitter Web App

So that's it then. There is no general lifetime solution, and the range-`for` loop stays broken in subtle ways.

Stringy Templates

Colby Pike (a.k.a **vector-of-bool**) wrote an article on his blog about using strings as template parameters. This is a feature of C++20 called "Non-Type Template Parameters".

Example: an integer template parameter results in a new type for each value of the parameter. This was supported for a long time.

```cpp
template <int N>
struct foo{};
```

In C++20, we can use a class-type non-type template parameters. Certain requirements have to be met by such class: all its members can only be of type valid as non-type template parameter, or an array of such type. Colby Pike uses it to implement a “fixed string” template:

```cpp
template <size_t Length>
struct fixed_string {
    char _chars[Length+1] = {}; // +1 for null terminator
};
```

Combined with the deduction guide

```cpp
template <size_t N>
fixed_string(const char (&arr)[N])
    -> fixed_string<N-1>;  // Drop the null terminator
```

it allows for an easy initialization that looks like a normal assignment:

```cpp
fixed_string str = "Hello!";
```

Now that we have a string class that can be used as a template parameter, we can do interesting stuff like what Colby Pike calls “stringy templates”:

```cpp
// Declare an empty primary definition of a class template
template <fixed_string>
struct named_type {};

// Declare explicit specializations for different
    fixed_string types
template <>
struct named_type<"integer"> { using type = int; };
template <>
struct named_type<"boolean"> { using type = bool; };

// Create an alias that grabs the nested type from a
    particular specialization of named_type
template <fixed_string S>
using named_type_t = named_type<S>::type;

// Drumroll
named_type_t<"integer"> v = 42;
named_type_t<"boolean"> b = false;
```

This is very interesting, and I’ll be watching for further posts on this topic.

The code is available on GitHub.

The Reddit thread mentions two projects that use similar techniques:

- consteval-huffman — A C++20 utility for compressing string literals at compile-time to save program space.

- CTRE: compile-time regular expressions — Fast compile-time regular expressions with support for matching/searching/capturing during compile-time or runtime.
- Mitama: row polymorphism library

Mitama: Row Polymorphism in C++20

This post is an attempt at implementing Haskell-style extensible records in C++20. The code is on GitHub. The author used the cutting-edge Visual Studio 2022 Preview 5 for development.

> Row polymorphism is a kind of polymorphism that allows one to write programs that are polymorphic on record field types (also known as rows, hence row polymorphism).

This is like prototype-based inheritance in JavaScript, where you inherit objects instead of types, and then extend them by adding fields.

Example code:

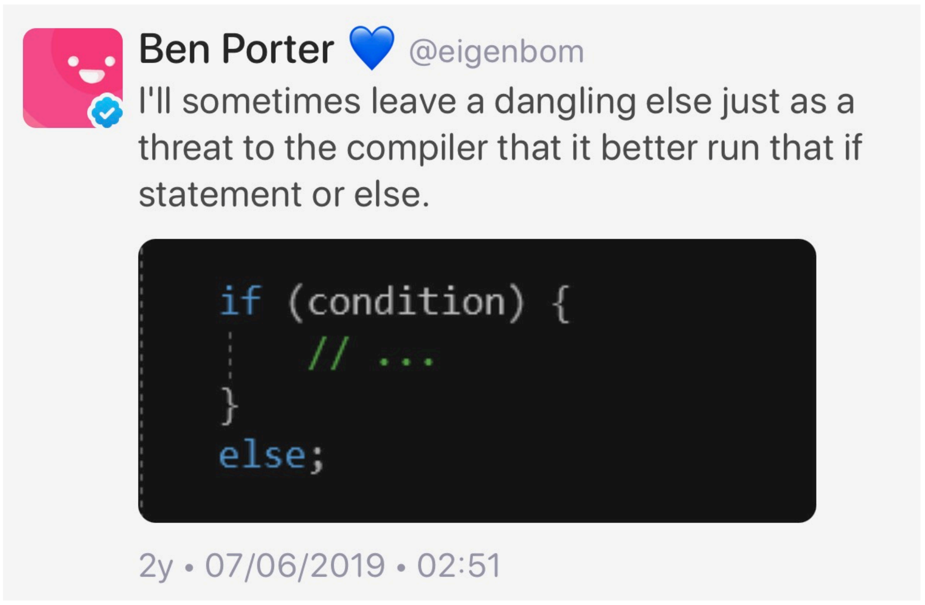
```cpp
// declare record type
using Person = mitama::record
               < mitama::named<"name"_, std::string>
               , mitama::named<"age"_,  int>
               >;

// make record
Person john = mitama::empty
            += "name"_ % "John"s
            += "age"_  % 42
            ;

// access to rows
john["name"_]; // "John"
john["age"_];  // 42
```

The author uses a fixed_string class just like Colby Pike in the previous article.

Twitter



Ben Porter 💙 @eigenbom
I'll sometimes leave a dangling else just as a threat to the compiler that it better run that if statement or else.

```
if (condition) {
    // ...
}
else;
```

2y • 07/06/2019 • 02:51