

C++ Club UK Meeting 146

Gleb Dolgich

2022-03-31

Contents

March committee mailing	2
P0009 <code>std::mdspan</code>	2
P1684 <code>std::mdarray</code>	2
P2558 Add @, \$, and ' to the basic character set	3
P2565 Supporting User-Defined Attributes	3
It's Time For New C++ Leadership?	3
Xmake package management	4
<code>nft_ptr</code>	5
Comparing Floating-Point Numbers Is Tricky	5
Herbie	7
Binary number representation cheatsheet	7
Qt Creator 7	8
Debugging with GDB	8
Unlib: a lightweight, header-only, dependency-free C++14 library for ISO units	8
Similar work	9
A replacement for <code>std::vector<bool></code>	10
Twitter	11
Exam	11
Quote	12

March committee mailing

The March committee mailing contains some existing papers with new revisions, and some new papers. Let's have a look at some of them.

P0009 `std::mdspan`

Revision 16 of the `std::mdspan` paper contains major wording revisions based on LWG feedback.

This paper proposes adding to the C++ Standard Library a multidimensional array view, `mdspan`, along with classes, class templates, and constants for describing and creating multidimensional array views. It also proposes adding the `submdspan` function that “slices” (returns an `mdspan` that views a subset of) an existing `mdspan`.

At this point why not make a bet on which revision of this paper gets into C++26. I say 21.

P1684 `std::mdarray`

A companion to P0009, this paper proposes an owning equivalent of `mdspan`.

P2558 Add @, \$, and ‘ to the basic character set

Steve Downey wrote this paper, in which he says:

WG14, the C Standardization committee, is adopting N2701 for C23. This will add U+0024 \$ DOLLAR SIGN, U+0040 @ COMMERCIAL AT, and U+0060 ‘ GRAVE ACCENT to the basic source character set. C++ should adopt the same characters for C++26.

Previously, Corentin Jabot discussed the usage of these characters in other programming languages in P2342 *For a Few Punctuators More*.

P2565 Supporting User-Defined Attributes

Bret Brown of Bloomberg proposes to add support for user-defined attributes. At the moment compilers warn you if they encounter an unknown attribute, and at the same time they are allowed to define compiler-specific attributes, like `[[clang::no_sanitize("undefined")]]`. Because of this cross-platform libraries often have to resort to dances with macros to support code that depends on a vendor-specific attribute.

The author proposes to keep the ‘unknown attribute’ warning, but allow a new syntax for user-defined attributes:

```
1 | [[extern gnu::access(...)]];
```

The author chose the `extern` keyword for this because, I guess, this keyword has too few meanings in C++ and could use one more.

Since attributes are built into compilers, it’s not clear how one would define a custom attribute.

It’s Time For New C++ Leadership?

Bryce Leibach writes:

C++ is at an inflection point. The C++ Committee has been successful over the past decade. Yet the landscape has changed, and we must adapt to be successful in the future.

We need engaged new leadership to address the challenges we face, and to build the next generation of C++ leadership. I, Bryce Adelstein Leibach, am seeking the U.S. nomination for the position of ISO/IEC JTC1/SC22/WG21 Convenor to provide such leadership.

Bryce names several problems with how C++ is being developed:

The C++ Committee and the number of proposals we receive has grown 10x over the last decade. For the last 6 years, we have had

a significant backlog of papers waiting for review. In some cases, papers wait years to be reviewed by a subgroup. As Bjarne wrote in [P0977](#), “most people [on the C++ Committee] are working independently towards non-shared goals”. We are 300 individual authors, not 1 team. We lack focus and a shared set of priorities.

Bryce proposes steps to alleviate the situation. Regarding the new ways of working that emerged during the pandemic, he says:

Our heavy reliance on face-to-face meetings seems outdated to the next generation of programmers who are used to modern collaboration tools that have become prevalent over the last two decades.

He wants to ensure that:

remote attendees have an equal seat at all future face-to-face meetings

This is going to be one of the biggest challenges.

Among other things Bryce wants to unify the currently scattered infrastructure under the [isocpp.org](#) domain.

One of his the most important promises is address the lack of diversity and inclusivity:

The first step in fixing a problem is acknowledging it. The C++ Committee and community is not as diverse or inclusive as it should be. This threatens C++’s long term legacy. <...> As Convenor, diversity, inclusion, safety, and growing the next generation of C++ Committee leadership will be my top priorities.

Ambitious goals, and it’ll be interesting to see how it goes for Bryce and the Committee.

Xmake package management

This [article](#) describes package management in CMake using Vcpkg and Conan and compares it to what’s available in [Xmake](#). It also introduces Xmake’s standalone package manager Xrepo. I’m still amazed at the quality and capabilities of Xmake. We lament about how difficult it is to bootstrap a C++ project, we have entire tools that bootstrap CMake projects, but here it is, an easy to use and amazingly capable build system, and nobody seems to know about it. CMake is the de-facto standard, but teaching it to students is akin to starting a modern C++ course by explaining pointers. Xmake could be an ideal student-friendly introduction to build systems at least for their toy projects, to avoid scaring them away before they even start learning C++.

nft_ptr

Non-fungible tokens, or NFTs, are a scam built on the blockchain technology. There are many articles explaining this latest high-tech planet-destroying pyramid scheme, so I'm not going to do that here. Instead let me tell you about this excellent project that highlights the craziness from the C++ point of view. Behold `nft_ptr`: "C++ `std::unique_ptr` that represents each object as an NFT on the Ethereum blockchain."

(This was an April Fools 2021 joke.)

```
1 auto ptr1 = make_nft<Cow>();
2 nft_ptr<Animal> ptr2;
3 ptr2 = std::move(ptr1);
```

This transfers the Non-Fungible Token `0x7faa4bc09c90`, representing the Cow's memory address, from `ptr1` (OpenSea, Etherscan) to `ptr2` (OpenSea, Etherscan).

It works, and is completely bonkers. I especially like the **Why** section:

- C++ memory management is hard to understand, opaque, and not secure.
- As we all know, adding blockchain to a problem *automatically* makes it simple, transparent, and cryptographically secure.
- Thus, we extend `std::unique_ptr`, the most popular C++ smart pointer used for memory management, with blockchain support.
- Written in Rust for the hipster cred.
- Made with *love* by a Blockchain Expert who wrote like 100 lines of Solidity in 2017 (which didn't work).

The **Performance** section doesn't disappoint either:

`nft_ptr` has negligible performance overhead compared to `std::unique_ptr`, as shown by this benchmark on our example program:

- `std::unique_ptr` - 0.005 seconds
- `nft_ptr` - 3 minutes

The project is very thorough and even has a link to a [whitepaper](#)! It's indeed a white paper.

Comparing Floating-Point Numbers Is Tricky

This is an old [article](#) from 2017 but it's still useful and provides a good illustration of the problems with machine representation of floating-point (FP) numbers.

Good things to remember:

- Floats cannot store arbitrary real numbers, or even arbitrary rational numbers.
- Since the equations are exponential, the distance on the number line between adjacent values increases (exponentially!) as you move away from zero.

Over the course of the article the author develops and improves a function to compare two FP numbers. He starts with this code which I've seen many times in our codebases, and explains why it's wrong:

```
1 bool almostEqual(float a, float b)
2 {
3     return fabs(a - b) <= FLT_EPSILON;
4 }
```

We would hope that we're done here, but we would be wrong. A look at the language standards reveals that `FLT_EPSILON` is equal to the difference between 1.0 and the value that follows it. But as we noted before, float values aren't equidistant! For values smaller than 1, `FLT_EPSILON` quickly becomes too large to be useful. For values greater than 2, `FLT_EPSILON` is smaller than the distance between adjacent values, so `fabs(a - b) <= FLT_EPSILON` will always be false.

Boost has FP comparison API but the author explains how it is also not quite correct. He then arrives at ULPs:

It would be nice to define comparisons in terms of something more concrete than arbitrary thresholds. Ideally, we would like to know the number of possible floating-point values—sometimes called *units of least precision*, or ULPs—between inputs. If I have some value `a`, and another value `b` is only two or three ULPs away, we can probably consider them equal, assuming some rounding error. Most importantly, this is true regardless of the distance between `a` and `b` on the number line.

The author emphasises the fact that ULPs don't work for comparing values close to zero, but this can be handled as a special case.

The main takeaways from the article are:

When comparing floating-point values, remember:

- `FLT_EPSILON`... isn't float epsilon, except in the ranges `[-2, -1]` and `[1, 2]`. The distance between adjacent values depends on the values in question.
- When comparing to some known value—especially zero or values near it—use a fixed epsilon value that makes sense for your calculations.
- When comparing non-zero values, some ULPs-based comparison is probably the best choice.

- When values could be anywhere on the number line, some hybrid of the two is needed. Choose epsilons carefully based on expected outputs.

This article was adapted from Bruce Davison's article *Comparing Floating Point Numbers, 2012 Edition*.

The GoogleTest macro `ASSERT_NEAR` uses a combination of ULPs- and epsilon-based comparisons and is the best way to compare FP values in tests against an epsilon difference.

David Goldberg's article *What Every Computer Scientist Should Know About Floating-Point Arithmetic* is a required reading for all programmers. A web-based version is [here](#).

The corresponding Reddit thread is [here](#).

A related article by John D. Cook, *Floating point numbers are a leaky abstraction*, points out a few cases when FP numbers don't behave as expected:

Floating point numbers, the computer representations of real numbers, are leaky abstractions. They work remarkably well: you can usually pretend that a floating point type is a mathematical real number. But sometimes you can't. The abstraction leaks, though not very often.

Herbie

Herbie is a neat tool that simplifies arithmetic expressions to avoid FP issues.

```
sqrt(x+1)-sqrt(x) -> 1/(sqrt(x+1)+sqrt(x))
```

Herbie detects inaccurate expressions and finds more accurate replacements. The [left] expression is inaccurate when $x > 1$; Herbie's replacement, [right], is accurate for all x .

Herbie can be installed locally or used from the [web demo page](#). It is programmed in **Racket** which looks like a Lisp-like language.

Binary number representation cheatsheet

[Source](#)

Common Binary Representations of Numeric Values

h/cpp hackingcpp.com

Unsigned Integers

n bit Positional Binary

value = $d_{n-1} \cdot 2^{n-1} + \dots + d_1 \cdot 2^1 + d_0 \cdot 2^0$

Example: $n = 8$

0000 0000	= 0
0000 0001	= $2^0 = 1$
0000 0010	= $2^1 = 2$
0000 0011	= $2^1 + 2^0 = 3$
0000 0100	= $2^2 = 4$
⋮	⋮
1000 0000	= $2^{n-1} = 128$
⋮	⋮
1111 1111	= $2^n - 1 = 255$

Signed Integers

n bit Two's Complement
($i + 2s \text{Complement}(i) = 2^n$)

Example: $n = 8$

0111 1111	= $2^{n-1} - 1 = +127$
⋮	⋮
0000 0010	Negation: = +2
0000 0001	1. invert = +1
0000 0000	2. add 1 = ±0
1111 1111	= -1
1111 1110	= -2
⋮	⋮
1000 0000	= $-2^{n-1} = -128$

Floating-Point Numbers

IEEE 754 Format and derivatives

sign bit | exponent (x bits) | mantissa (m bits)

$n = (1 + x + m)$ bits

Example: half precision ($n = 16, x = 5, m = 10$)

subnormal

S	00000	M
0	00000	0000000000
0	00000	0000000000
0	00000	0000000001
0	00000	1111111111

normal

S	01	M
0	10	M
0	0001	0000000000
0	0110	0000000000
0	0110	1111111111
0	0111	0000000000
1	0111	0000000000
0	0111	0000000001
0	1110	0000000000
0	1110	1111111111
1	1110	1111111111

special

0	1111	0000000000
0	1111	0000000001
0	1111	1000000001

negative zero
smallest positive subnormal
largest subnormal
exponent bias: $2^{x-1} - 1 = 15$
smallest positive normal
largest less than one
smallest larger than one
largest normal
smallest normal
infinity
"not a number"

Qt Creator 7

Qt Creator 7 has been [released](#) (see also [Reddit thread](#)). Even if you don't use Qt, this is a great general-purpose C++ IDE, which is now based on [Clangd](#), the Clang daemon, using [Language Server Protocol \(LSP\)](#).

Debugging with GDB

The IncrediBuild Blog has another interesting and useful article, [A step-by-step crash course in C++ debugging without IDE – empowering the terminal by using GDB!](#), written by Adam Segoli Schubert. It covers the basics of GDB command-line interface and its Text User Interface (TUI) mode.

The [Reddit thread](#) participants seem to have trouble understanding why in the era of ubiquitous C++ IDEs would anyone want to use a terminal-based debugger. Lucky people, they must never have had to inspect a crash dump over an SSH connection to a Linux server.

Unlib: a lightweight, header-only, dependency-free C++14 library for ISO units

The author [sbi](#) from Berlin wrote [yet another physical units library](#) that supports SI (ISO) units and requires C++14. It's header-only and comes under Boost Licence.

A minimal, header-only, C++14-compatible SI unit library, providing quantities that behave like arithmetic types and feature physical dimensions (e.g. power), scaling (e.g. kilo), and tagging of units. If your code has to deal with physical units, you can use this library so that the compiler checks your usage of dimensions and your formulas at compile-time.

The library features user-defined literals:

```
1 using namespace unlib::literals;
2
3 unlib::newton<double> force = 1._kg * 9.81_m_per_s2;
4
5 unlib::second<int> s = 1_h;
6 std::cout << s; // prints 3600
```

It allows deriving units from the basic set:

```
1 using electrical_charge =
   unlib::mul_dimension_t<unlib::current, unlib::time>;
```

There is also support for ratios and creation of scaled quantities from basic ones:

```
1 using milligram = unlib::milli<unlib::gram>;
2 using kilogram = unlib::kilo<unlib::gram>;
3 using ton = unlib::kilo<kilogram>;
```

The library supports tags for when different quantities which must not be confused are represented by the same physical unit.

Unit conversions are supported by various casts.

There are four different kind of casts available:

- `value_cast` allows casting between units with different value types, e.g., seconds in `int` vs. seconds in `long long`.
- `scale_cast` allows casting between units with different scales, e.g., seconds and minutes.
- `tag_cast` allows casting between units with different tags, e.g., active and reactive power.
- `quantity_cast` allows casting between units where value types, scales, and tags might be different.

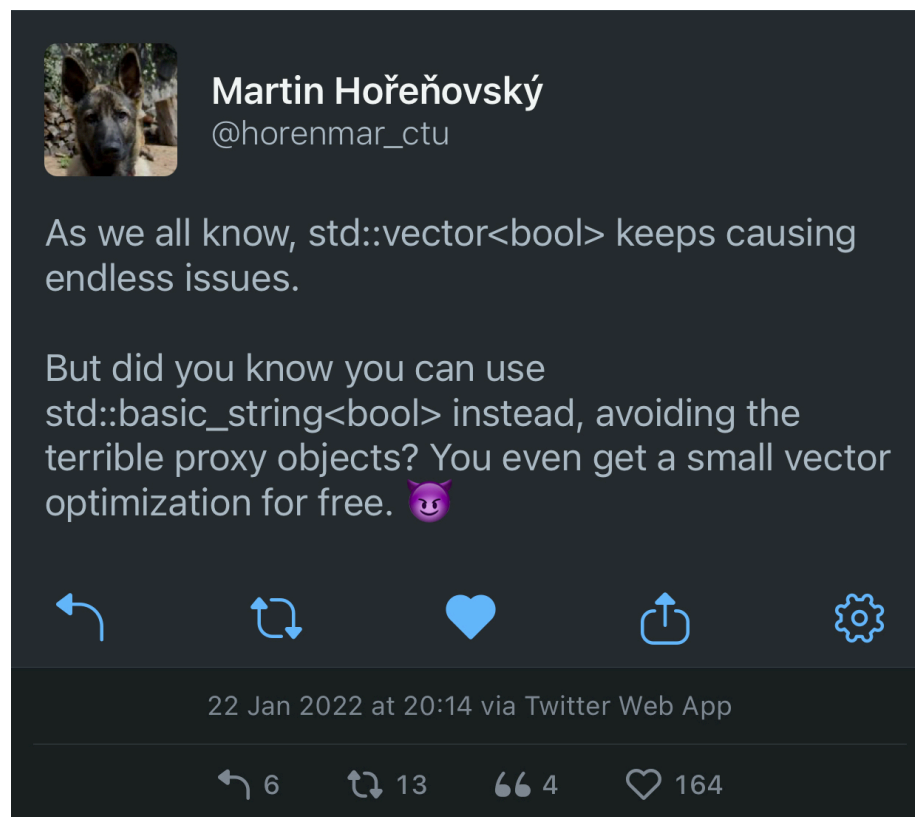
Similar work

- `units` by Mateusz Pusz, which is a subject of standardization effort (see [P1935](#)) — requires C++20, uses Concepts, comes under MIT Licence. This library uses literals but also allows you to specify units as multipliers: `static_assert(1 * h == 3600 * s);`

- [Boost.Units](#) by Matthias C. Schabel and Steven Watanabe, implements dimensional analysis in a general and extensible manner, treating it as a generic compile-time metaprogramming problem. It uses [Boost.MPL](#) and is slow to compile, but allows you to define your own unit systems.
- [units](#) by Nic Holthaus is a compile-time, header-only, dimensional analysis and unit conversion library built on C++14 with no dependencies. It comes under MIT Licence. Each unit has its own type, literals are supported, and unit conversions and manipulations are very fast and efficient.

A replacement for `std::vector<bool>`

Martin Hořeňovský [tweeted](#):

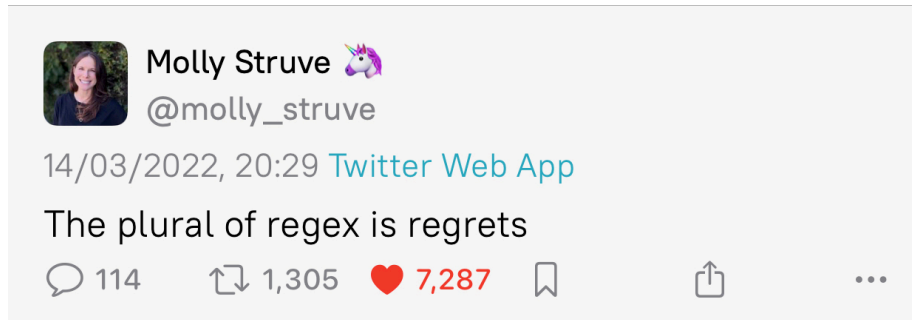


There is also a [StackOverflow question](#) about that. It seems like the idea might work but you shouldn't do it. Some replies:

- *Can somebody get people like this away from the keyboard, before they hurt themselves?* — [Jan Wilmans](#)
- *Somehow I equally love and hate this tweet.* — [Michail Caisse](#)

Twitter

Molly Struve (@molly_struve):



Exam

An incorrect, apparently, exam answer to the question about phases of software development:

2 Short Answer Questions

11. [10 points] Name and describe the five key phases of software development.

1. denial
2. bargaining
3. Anger
4. depression
5. acceptance

@programmerhumour

Quote

