C++ Club UK Meeting 142

Gleb Dolgich

2022-01-20

Contents

Interview with Bjarne Stroustrup

YouTube channel Context Free interviewed Bjarne Stroustrup: Part 1, Part 2.

James Webb Space Telescope uses C++

During a YouTube Q&A session the JWST team stated that the telescope on-board software is written in C++. See Reddit thread 1 and thread 2.

Looks like the OS is VxWorks, and there is a JavaScript interpreter there which is used to read text files with parameters and issue commands to instruments. The company behind the ScriptEase JavaScript interpreter was called *Nombas* ('No MBAs'?) and it got acquired by OpenWave. There is a history of Nombas and a technology evaluation document for JWST. Here is a document called *Event-Driven James Webb Space Telescope Operations Using On-Board JavaScripts*. This redditor says the CPU that powers JWST is RAD750, which is a modified PowerPC 750 CPU, similar to the CPU in the Nintendo GameCube.

GCC 12 supports mold linker

GCC 12 just added support for the amazingly fast `mold` linker, which leaves other linkers in the dust, including `lld`. Just look at those sweet benchmarks!

2021 C++ Standardization Highlights

Botond Ballo writes:

> The ISO C++ Standards Committee has not met in person since its February 2020 meeting in Prague <...> However, the committee and its subgroups have continued their work through remote collaboration, and a number of notable proposals have been adopted into C++23, with many others in the pipeline. I've been less involved in the committee than before, so this post will not be as comprehensive as my previous trip reports <...>

Botond describes remote collaboration in the committee, where subgroups would meet on Zoom throughout the year, with virtual plenary meetings and votes three times per year. He lists some key procedure documents:

- The C++ International Standard Schedule
- An outline of C++23 priorities
- An outline of remote collaboration model
- Papers/mailings
- GitHub issue tracker for papers

New to me was the GitHub forwarding link for papers: `https://wg21.link/pXXXX/github`.

Botond lists the proposals selected for C++23 and links to the appropriate papers, as well as technical specifications and papers in progress. He pays special attention to the following topics:

- Fixing the range-for loop: the rejected proposal to address the issue of lifetime of temporaries.

  > Unfortunately, the proposal narrowly failed to garner consensus in EWG, with some participants feeling that this was too specialized a solution that would not address similar cases of dangling references in contexts other than the range-based for loop, such as `auto&& range = foo(temp()); /* work with range */`. I think the counterargument here is that in these cases the dangling reference is explicitly visible, whereas in the range-based for loop it's hidden inside the implicit rewrite of the loop.

Perfect is the enemy of good, am I right?

- Named arguments making a comeback

Botond finishes with this:

> Collaboration for the time being continues to be remote. As of this writing, the earliest in-person meeting not to be definitively

> cancelled is the one in July 2022; it remains to be seen whether we will in fact be able to hold this meeting in person.
>
> Whenever the committee does resume in person meetings, they're likely to (at least initially) be hybrid, meaning there will be A/V equipment to allow continued remote participation for those who prefer it.

Almost always unsigned (?)

Dale Weiler wrote an article called Almost Always Unsigned, and I'm not sure what to think. I thought this question has been put to rest and everyone agreed that `unsigned` should only be used for bit manipulation, security and serialization, but not for arithmetics. Seems not everyone think that.

> The need for signed integer arithmetic is often misplaced as most integers never represent negative values within a program. The indexing of arrays and iteration count of a loop reflects this concept as well. There should be a propensity to use unsigned integers more often than signed, yet despite this, most code incorrectly choses to use signed integers almost exclusively.

Dale examines all the usual arguments in favour of signed integers and tries to dismantle them, with varying success, sometimes using questionable C++ code. He addresses the following points:

- The safety argument — dealing with overflow and underflow
- Loop variables ("just use reverse iteration")
- The difference of two numbers can become negative
- Computing indexes using signed integers is safer
- Unsigned multiplication can overflow
- Sentinel values (*come on, nobody uses this to argue in favour of signed integers!*)
- It's the default (*But STL sizes are unsigned*)

There is a section called *Your counter arguments are about pathological inputs*, in which the author dismisses the argument out of hand.

His arguments for unsigned are:

- Most integers in a program never represent negative values (*wow, talk about generalization*)
- Compiler diagnostics are better for unsigned but that's worse overall
- Checking for overflow and underflow is easier and safer

> Since C and C++ make signed integer overflow and underflow

> undefined, it's almost impossible to write safe, correct, and obvious code to check for it.

He presents a code snippet to check for signed overflow/underflow:

```
1 if ((b > 0 && a > INT_MAX - b) || (b < 0 && a < INT_MIN -
    b)) // addition overflows
2 if ((b > 0 && a < INT_MIN + b) || (b < 0 && a > INT_MAX +
    b)) // subtraction underflows
```

- Your code will be simpler and faster
- It actually works (*OK sure I'm convinced*)

Dale concludes:

> You might be wondering how possible it is to actually use unsigned almost always as this title suggests. It's been my personal preference for half a decade now and I haven't actually missed signed integers since switching to it. It's made my code much easier, cleaner, and more robust contrary to popular wisdom.
>
> I would not suggest trying to use it in an existing codebase that is mostly signed integers, in such contexts you're more likely to introduce silent bugs and issues as a result of unsafe type casts. But consider trying it next time you start a new project, you might be pleasantly surprised.

This article unsurprisingly generated a lot of noise on Twitter. What was surprising to me were some very well known C++ people expressing their opinions on this topic.

The conversation started with Arvid Gerstmann tweeting link to the above article and commenting:

> Almost always **signed**!

This prompted Peter Sommerlad to quote his tweet with the comment:

> Mostly correct, with some bad C++ examples, i.e., using new to allocate an array instead of `make_unique`. Could also benefit from being less polemic. And i cannot understand the obsession with using signed integers by some.

Joel Falcou replied:

> Unsigned are not made for arithmetic.

Daniela Engert joined the discussion:

> I don't think this is true.

Joel:

They're rather dangerous. They're not modeling the proper subclass of integers.

Daniela:

What is the "proper subclass of integers" in the first place? I rarely (if ever) need and use signed ints in my code. So do my colleagues.

Joel:

Ones where `a - b` is less than `a` if `b` is positive.

Daniela:

So you never have been working in application domains there this is a required feature? It all depends on the intrinsic nature of the entities you want to model. If you want to model a sunset of Z then go with ints. If you want to model a subset of N0 then go with unsigneds.

Tobias joins in on the signed side:

But N0 does not have well-defined subtraction, i.e. if you do subtraction, you should not use unsigned. The issue I have with unsigned is that 0 is just too close to the edge to be comfortable. With signed, at least you are `INT_MAX` steps away from overflows in both directions.

Dani:

It certainly has a well-defined subtraction if you meet the preconditions. If you can't satisfy the preconditions then you are in a situation where subtraction doesn't make sense in the first place, i. e. where you have to think about the meaning of your model!

Joel:

Yes, therefore using unsigned for stuff like size of containers is a bad idea. Size is not N0, it's Z.

Tony Van Eerd:

Programmers expect numbers to work like numbers. Signed works more like numbers than unsigned.

Peter insists:

How can size be negative? Nobody programming should expect C's built-in types work like numbers. they aren't. eg 42/8

Tobias:

> All operations that are well-defined on numbers work exactly the same way in C: Addition, Subtraction, Multiplication. Division can be sensibly interpreted as returning a pair `(a/b, a%b)`, which fits as well. Except for really large values, this is true for signed. For unsigned, you only need really small values to run into an overflow, e.g. `1u - 2u`. That is much more likely than a signed integer overflow.

Dani:

> I strictly oppose this notion at a fundamental level. Your modelling of container sizes in Z is equivalent to "add 2 more elements to this container to make it 'empty' " a valid operation. UB.

She continues:

> And therein lies the problem. Expectations don't always hold. To me, selecting or creating the most appropriate type to model an entity is the most noble art in software design. If you fail at that in fundamental aspects you're lost.

Joel:

> The container invariant is `size >= 0` so it can't happen. It should not be the invariant of the type containing the size.

Tony:

> Agreed. But it is rare to see anyone make a type that accurately models numbers. Instead 99% of the time everyone just picks a readily available type that is close-ish. Signed is closer to expectations than unsigned. They both suck, but pragmatically signed sucks less.

Peter:

> Unfortunately signed integer arithmetic is very prone to surprising and undefined behavior in C++, especially in cases where operands are of an unsigned flavor... I think we can agree that none of the built-in arithmetic types is without problems, and combining them more.

Dani raises the stakes:

> Home assignment: implement cryptography with an algebra where this property holds true.

Joel counters:

> Home assignment: don't move goal posts between (each) tweets. If you want to assume something is always positive, it expresses your intention much more clearly.

Peter:

> I am sorry, I don't get it. if i have the invariant that values must be zero or positive, why i am supposed to NOT use a type that only has zero or positive integers to represent it?

Joel:

> It's an invariant of the container. Therefore the container expresses it at its level. Invariant of unsigned types are implicit and lead to a false sense of security as soon as people try to use unsigned types as arithmetic types.

So there you have it. This thread shows that signed vs. unsigned argument is far from over. Daniela Engert and Peter Sommerlad are firmly in the unsigned camp, whereas Arvid Gerstmann, Joel Falcou, Tony Van Eerd and others (including myself) think signed integers are better at modelling numbers.

Some quotes from the Reddit thread are below. Looks like redditors think "Almost Always Unsigned" is a bad advice.

> My experience has been the opposite: unsigned arithmetic tends to contain more bugs. Code is written by humans, and humans are really bad at reasoning in unsigned arithmetic. #

> I though I was reading the title wrong for a second, not a good advice at all from my experience. Unsigned sounds like a good idea at the beginning, I learned fast that it has so many quirks, gotchas and weird unpredictable bugs popping way more that I'd like. It turns simple basic operations like subtraction, multiplication with negatives, comparisons, `abs()`, `max()`, `min()` and more into a needless mess for no good reason. Now I use signed exclusively unless I'm working with a lib that needs it, never regretted it once after years of doing it. #

Daniela Engert repeats:

> I like this article as it matches my experiences from decades of software development.

Robert Ramey says:

> This dispute is never, ever going to be resolved. But until it does... use Boost Safe Numerics. *It's his library BTW.*

The Core Guidelines ES.106:

> Don't try to avoid negative values by using `unsigned`. Choosing `unsigned` implies many changes to the usual behavior of integers, including modulo arithmetic, can suppress warnings related to overflow, and opens the door for errors related to

> signed/unsigned mixes. Using `unsigned` doesn't actually eliminate the possibility of negative values.

The Core Guidelines ES.107:

> Don't use `unsigned` for subscripts, prefer `gsl::index` (see Microsoft's GSL).

See also: CppCon 2016: Jon Kalb "unsigned: A Guideline for Better Code"

Little C++ Standard Library Utility: `std::align`

Lesley Lai posted an article about `std::align` in which he describes a use case, then implements a helper function manually, and finally replaces it with `std::align`.

> Arena, also called bump allocator or region-based allocator, is probably the simplest allocation strategy. It is so widely used that even the C++ standard library has an arena implementation called `std::pmr::monotonic_buffer_resource`.
>
> With arena, we first have a large chunk of pre-allocated memory. That chunk of memory itself can either come from the stack or one large allocation of another allocator such as malloc. Afterward, we allocate memory from that chunk by bumping a pointer offset.
>
> Arena allocation has outstanding performance characteristics, especially compared to complicated beasts like `malloc`. Each allocation only needs to bump a pointer, and the deallocation is almost free if the objects allocated are trivially destructible. <...> it is useful in situations where we have a lot of heterogeneous allocations that only need to be freed together, and is widely used in application domains from compilers to video games.

When allocating memory in an arena we need to deal with alignment. Incrementing the 'next' pointer by the size of allocated object is not enough — starting the lifetime of objects on unaligned locations is undefined behavior.

Lesley then explains how to handle alignment, including a full implementation of a helper function that bumps arena pointer to the next allocation address and updates the remaining area space.

Turns out, that's what `std::align` is for:

```
namespace std {
auto align(std::size_t alignment,
           std::size_t size,
           void*& ptr,
           std::size_t& space)
```

```
-> void*;
}
```

From cppreference:

> Given a pointer `ptr` to a buffer of size `space`, returns a pointer aligned by the specified alignment for `size` number of bytes and decreases `space` argument by the number of bytes used for alignment. The first aligned address is returned.

We still need to change `ptr` and `space` according to the actual allocation size, as they are only bumped by the number of alignment bytes.

So here it is, `std::align`, a very useful function for a very limited use case.

`T*` makes for a poor `optional<T&>`

Barry Revzin writes:

> Whenever the idea of an optional reference comes up, inevitably somebody will bring up the point that we don't need to support `optional<T&>` because we already have in the language a perfectly good optional reference: `T*`. <...> The purpose of this post is to point out that, despite these similarities, `T*` is simply not a solution for `optional<T&>`.

As an example Barry chose a function that returns the 1st element of a range or nothing if the range is empty. He develops the function over several iterations and demonstrates that in order to support any input range the return type must be `optional<T&>`. He shows that `T*` doesn't really work, and `optional<T>` is also unsuitable for some input range types, notably `vector<T>` which has reference type of `T&`.

When the return type is `optional<ranges::range_reference_t<R>>` (where `R` is the input range type) it resolves to `optional<int>` for `ranges::iota<int>` but for `vector<int>` it resolves to `optional<int&>`. The author shows that returning `int*` instead doesn't work without a lot of workarounds. If we wanted to have a nice usage where a default value is returned if the input range is empty, it doesn't work with pointers unless more workarounds are provided, and then the call syntax is not as nice:

```
// ideal
int value1 = try_front(r).value_or(-1); // ideal

// workaround
template <typename P, typename U>
constexpr auto value_or(P&& ptrish, U&& dflt) {
  return ptrish ? *FWD(ptrish) : FWD(dflt);
}
```

```
9 int value2 = N::value_or(try_front(r), -1);
```

Barry takes a quick aside to remind us why `vector<booL>` is bad, and then writes:

> If we all agree that `vector<bool>` is bad because of several subtle differences with `vector<T>`, then surely we should all agree that `T*` is a bad `optional<T&>` because it has several very large and completely unavoidable differences with `optional<T>`.
>
> Namely:
>
> - it is spelled differently from `optional<T>` (trivially: it is spelled `T*`)
> - it is differently constructible from `optional<T>` (you need to write `&e` in one case and `e` in the other)
> - it has a different set of supported operations from `optional<T>`

Barry talks about the future too:

> We don't have pattern matching in C++ yet, and we still won't in C++23. But eventually we will, and when we do, we'll want to be able to match on whether our optional reference actually contains a reference, or not. We do not need to match whether we're... holding a derived type or not. This is yet another operation that a `T*` won't do for us.

He concludes:

> The ability to have `optional<T&>` as a type, making optional a total metafunction, means that in algorithms where you want to return an optional value of some computed type `U`, you can just write `optional<U>` without having to worry about whether `U` happens to be a reference type or not. This makes such algorithms easy to write.

Reddit thread has much to say about the topic, and specifically about what should happen when assigning to `optional<T&>`.

Printing tabular data

Some links for when you need to output tables in C++:

tableprinter by Ozan Cansel

A header-only C++17 library, no 3rd-party dependencies, MIT licence.

CppConsoleTable by Denis Samilton

A header-only C++17 library, MIT licence. This one prints table borders using the extended line drawing characters.

variadic_table by Derek Gaston

A header-only C++17 library that uses variadic templates for convenience. This one prints table borders using standard characters (dash and pipe). The licence is LGPL-2.1, so be careful.

Outcome enters sustaining phase, goes ABI stable

Niall Douglas announced on Reddit that his Outcome library has entered sustaining phase and now has a stable ABI. Outcome is an alternative error handling library that may be useful when exceptions are not allowed, or when you need deterministic performance for the 'sad path' (as opposed to 'happy path').

Niall provides a comparison of Outcome with alternative error handling methods and libraries.

Outcome comes as a standalone library and as part of Boost, Boost.Outcome. There is a clarification of the differences between these versions. It has been well reviewed and production-tested, and is available via C++ package managers, such as **conan** and **vcpkg**. The library requires C++14 and comes as a single header under Apache-2.0 license.

Include guards vs. `#pragma once` — again

A new poll on Reddit shows that developers prefer the non-standard but widely supported `#pragma once` over include guard macros. The thread contains anecdotal references for and against each option.

A game developer at Epic says:

> Since people were copying files and renaming them without changing the guard name we have switched to `#pragma once` because all toolchains for targeted platforms have proper support for it.

That sounds like they have a bigger problem...

This redditor reminds about what The Core Guidelines say about it:

> Personally, I tend to follow the cpp core guidelines and use `#ifndef` SF.8: Use #include guards for all .h files

Of course, people argue against that too.

Here is an amusing exchange:

> `#pragma once`. It's 2022.
>
> > In 2022 we use `import`.
> >
> > > I wish.

Overheard: Life lessons

> Life taught me two lessons. The first I don't remember, and the second was write everything down.

Twitter