overload 184

DECEMBER 2024 £4.50

Static Reflection in C++

Wu Yongwei demonstrates how to achieve reflection now and shows some examples of what C++26 might make possible.

Senders/Receivers: An Introduction

Lucian Radu Teodorescu explains the idea and how to use these in detail.

User Stories and BDD – Part 4, Features Are Not Stories

Seb Rose finishes his BDD series by encouraging us to be mindful of the difference.

Replacing 'bool' values

Spencer Collyer considers when they can cause a world of pain.

Afterwood

Chris Oldwood gives software development a seasonal twist.

A magazine of ACCU

ISSN: 1354-3172

December 2024
ISSN 1354-3172

Printing and distribution
Parchment (Oxford) Ltd

Cover design
Original design by Pete Goodliffe
pete@goodliffe.net
Cover photo by Michael Persson (an Adobe Stock photo) of a gravel road in Dalby Söderkog (a national park), Sweden.

ACCU

ACCU is an organisation of programmers who care about professionalism in programming. We care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

Many of the articles in this magazine have been written by ACCU members – by programmers, for programmers – and all have been contributed free of charge.

Overload is a publication of the ACCU For details of the ACCU, our publications and activities, visit the ACCU website: www.accu.org

Copy deadlines

All articles intended for publication in *Overload* 185 should be submitted by 1st January 2025 and those for *Overload* 186 by 1st March 2025.

Counting Quails

We are taught to count as children. Frances Buontempo wonders: how hard can it be?

I've been somewhat distracted by various goings on recently. I attended the SoCraTes UK Unconference and the inaugural 'AI for the rest of us' conference. I've blogged about these if you're interested [Buontempo24a, Buontempo24b]. We've also been looking after some neighbour's animals while they are on holiday. All of which means I have, of course, not written an editorial.

The neighbours have chickens and guinea pigs, which we have looked after before. They now have several quails too. We were told there are 11, but counting quails turns out to be difficult. For starters, I have never seen a quail in real life before. When we first went round, I could see one and thought it was near three light coloured stones. But the stones then opened their eyes and moved. Quails come in different colours. By the end of the week, I had started to get the hang of counting small animals I don't know much about, and believe I spotted 11. Hard to be sure. Had we spotted more than 11 that would have been a surprise. Counting small birds isn't easy, even when you've got the hang of the shapes and colours to look for. They also move very quickly. Fortunately, we didn't spot any outside the run. We might be invited back.

Counting quails is difficult, but counting anything can be problematic. How many unfixed bugs does a software system have? Hopefully, they are in a bug tracking system, and you can query to get an answer. As with the quails though, are some reported twice, which means you are double counting? Or have some been marked as 'won't fix', so they are no longer open? They are still bugs, surely? Needing to qualify 'how many' might not be your first thought, but it's important. Knowing an absolute number might not be that useful, but a trend might be informative. Does the number of bugs go up over time? On the face of it, an increasing number of bugs sounds dreadful. However, this might mean more people are using the software, rather than new bugs being added to the system on a regular basis. A raw number isn't always helpful.

BBC Radio 4 sometimes runs a programme called *More or Less* by Tim Harford [BBC]. The latest episode unpacked a claim that 50 million leaves will be removed from railway tracks in the South East UK this year. He asked if this was a big number, or a small number, or a silly one. I'm not sure what a silly number is: perhaps I should make up a definition. A biodiversity strategy manager at network rail was interviewed to try to understand where the number had come from. If you're not from the UK, you may not know that we frequently have news in the autumn telling us about too many or even the wrong sort of leaves blocking train lines. There are myriad other excuses for our trains not running on time, so it's a bit of a joke. Anyway, Network Rail counted trees (using LiDAR), 13 million in total on the UK train network (or nearby), with 1.5 million in the South East. So, how many leaves are there per tree? It depends. Apple trees apparently have 50,000. Harford didn't explain how that figure was arrived at, but that's estimates for you. South Eastern Rail used 50,000 leaves per tree to arrive at their total figure of 75 billion leaves. This number was "pruned back to 50 billion for reasons unknown". They then said 99.9% of these might not need to be cleared, but the others might. And there you have 50 million. All of which begs the question, how useful is this number? Harford suggested this is what he thinks of as a silly number. It might be more informative to state how much time or money might be spent clearing leaves.

Estimation is, by definition, usually inaccurate. It can be useful, though. I've written a couple of books, and have started a third recently. The proposal requires an expected number of pages. How do you guess? I prefer shorter books, so I can manage to read them over a few days or weeks without forgetting earlier details. I wrote about book lengths a long time ago, in 'Too Much Information' [Buontempo12]. I weighed K&R (*The C Programming Language*) and discovered it was 375 grammes, which makes it suitable for carrying in a bag on a journey. The exact number of pages varies depending on where you look, but it's around 250. Yes, I know, the weight probably varies too. It's a ball-park figure.

How did I estimate the number of pages I would write? Badly, certainly. But sketching out the potential chapter titles and having in mind 250 pages is ideal, meant I could claim each chapter would be about 20 pages, and bosh, a total page count somewhere around 250. I suspect what is more important than the actual number is using that as a guide, so you can tell if you are going into more detail than you planned, or haven't written as much. An estimate isn't a commitment, but it can be a guide. Sergey Ignatchenko wrote about 'The Importance of Back-of-Envelope Estimates' [Ignatchenko17]. I used a strapline starting "Guestimate questions make many people grumble." I have been asked to guestimate the number of petrol stations in the UK so many times at interviews that I hit a point where I had to make an effort not to groan out loud when asked. The article emphasized finding an order of magnitude, which could show something would be impossible. This could stop you wasting hours trying to code up something that could never work. Both providing a page count and doing a back of the envelope calculation can provide some helpful input to a process from the start, potentially avoiding problems later on. Another simple example might be walking somewhere unfamiliar. If you know approximately how far you need to go, you can track how long you walk for as a hint about whether you are probably going in the wrong direction. If somewhere ought to be a 20 minute walk and you aren't there within half an hour, you may need to retrace your steps.

Estimates are used in various contexts. Agile teams usually come out with story point estimates for code requirements. As I am sure you know, the story points are meant to indicate effort rather than a time commitment.

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD using AI and data mining. She's written a book about machine learning: *Genetic Algortithms and Machine Learning for Programmers*. She has been a programmer since the 90s, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

Whether they are always used like that is another story. Maybe you have seen Lunar Logic's '1/TFB/NFC' estimation cards? [LunarLogic]. The letters on the estimation deck are a bit sweary in full, but your options are 1 point, too big or no chance. I like the idea of deciding if something is plausible, giving it one point, or too big, so able to be broken into smaller chunks, or downright impossible. You may not agree, which is fine. A recent LinkedIn post [Ottinger24] talked about product owners thinking developers tend to seem obstinate and reluctant. Surely they just need to type in the code? The product owners then spent a day pairing up with developers, and saw what they actually needed to do. Imagining how something works and actually doing it can be miles apart. Walking a day in another person's shoes, as the phrase goes, can be illuminating. By the same score, counting quails might sound easy, until you try it yourself. They blend in with the background and move. It's complicated.

To answer questions such as "How many quails?" or "How much effort?", you clearly need a definition of 'quails' or 'effort'. You also need a way to qualify the 'how much/many' as well. Obviously, with numbers? Well, maybe not, as the estimation cards just mentioned show. However, we are usually fundamentally counting when we answer these questions. ChatGPT tells me:

> Counting is a basic mathematical process used to determine the quantity or number of items in a set. It involves incrementally assigning numbers to items, either one-by-one or in groups, until reaching a total count.

It has sneaked the word *set* in there, perhaps to avoid double counting, as I am sure I did with the quails. Ignoring the AI for now, you count by mapping items to the natural numbers, giving you labels 1, 2, 3,… *n* for each quail, or whatever you are counting. This is also called an enumeration and neatly avoids defining numbers, which is a whole other topic. There are several different kinds of numbers. How many, I wonder. The AI listed 7:

1. Natural numbers: 1, 2, 3, …
2. Whole numbers: 0, 1, 2, 3, …
3. Integers: … -3, -2, -1, 0, 1, 2, 3, …
4. Rational numbers like ½ -¾
5. Irrational numbers like √2, π
6. Real numbers (no examples given)
7. Complex numbers like $3 + 4i$.

The rational numbers were not put in order because that's slightly difficult. The AI failed to mention transcendental numbers, and left out Hamilton's quaternions. I could have made the list myself, but hey. If you've not come across quaternions before, go have a read [Wikipedia-1]. They are lots of fun. They extend complex numbers, using a j and k as well, with the property

- $i^2 = j^2 = k^2 = -1$
- ij = k, -ji = k; jk = -kj=i and ki = -ik = j.

I recall a classmate during a mathematics lesson at school going off on a rant when complex nuwmbers were introduced, based on them being obviously made up. They are, but they are interesting in and of themselves. They also have practical uses, including connections with sine and cosine, making them useful for cycles such as sinusoidal currents and voltages [ECStudio]. (Beware that electrical engineers use j for imaginary numbers, rather than i, which is obviously reserved for current.) We didn't cover quaternions at school, which would probably have upset my classmate even more.

I wonder if you can enumerate all the types of numbers. To enumerate, you must be able to order the elements, and I don't know if you can really do that. Some types of numbers are subsets of others. The natural numbers are included in the integers, and so on. This gives you some kind of partial ordering. Weirdly, the set of whole numbers and integers contain the same number of elements. Map 0 to 0, odd whole numbers to 1, 2, 3, … and even whole numbers to -1, -2 , -3, … and you have a one-to-one mapping, so they must be the same size. There are more real numbers though. The proof is left as an exercise for the reader, or go read about Cantor's diagonal argument [Wikipedia-2]. The real numbers are therefore an example of an uncountable set. The quails were almost uncountable, but for different reasons.

Part of the difficulty with the quails was their movement. Counting or, more generally, measuring is difficult when things move. Heisenberg's uncertainty principle immediately springs to mind [Wikipedia-3], which states that we cannot measure the position and momentum of subatomic particles. To be fair, the movement itself might not be the problem, though you don't get much momentum without movement. Trying to measure software does run into similar problems. Instrumenting code for profiling changes the code itself. The numbers will be wrong, but can still be informative.

Now, I'm sure I double-counted some quails, as I mentioned. The approximate figure was close, though. Doing the same thing twice isn't the end of the world, but can be slightly annoying. If you have come across the Lunar estimation cards before, I notice I mentioned them in 'I am not a number' [Buontempo17]. I suspect what I have written now is a slight overlap, rather than a complete clone. Writing an editorial is impossible! Counting is also rather difficult, so don't be too hard on yourself if you have an off-by-one error, or get a number wrong. Spotting the inaccuracy is brilliant, and the order of magnitude approximation might be good enough.

References

[BBC] 'More or Less' on BBC Radio 4: https://www.bbc.co.uk/programmes/b006qshd

[Buontempo12] Frances Buontempo, 'Too Much Information' (editorial), *Overload* 111, published October 2012, available at https://accu.org/journals/overload/20/111/buontempo_1885/

[Buontempo17] Frances Buontempo, 'I Am Not a Number' (editorial), Overload 139, published June 2017, available at https://accu.org/journals/overload/25/139/buontempo_2377/

[Buontempo24a] Frances Buontempo, 'SoCraTesUK 2024' on *BuontempoConsulting*, posted 24 September 2024 at https://buontempoconsulting.blogspot.com/2024/09/socratesuk-2024.html

[Buontempo24b] Frances Buontempo, 'AI for the Rest of Us' on *BuontempoConsulting*, posted 29 October 2024 at https://buontempoconsulting.blogspot.com/2024/10/ai-for-rest-of-us.html

[ECStudio] 'Complex Numbers in Electronics', ECStudio, available at https://ecstudiosystems.com/discover/textbooks/basic-electronics/ac-circuits/complex-numbers-in-electronics/

[Ignatchenko17] Sergey Ignatchenko, 'The Importance of Back-of Envelope Estimates' in *Overload* 137, available at https://accu.org/journals/overload/25/137/ignatchenko_2341/

[LunarLogic] Lunar Logic estimation cards: https://estimation.lunarlogic.io/

[Ottinger24] LinkedIn post (original poster, Tim Ottinger, but many contributors), iniitally posted during November 2024 at https://www.linkedin.com/posts/agileotter_ive-had-pos-tell-me-that-they-didnt-know-activity-7257067028925108224-Stxo

[Wikipedia-1] Quaternion: https://en.wikipedia.org/wiki/Quaternion

[Wikipedia-2] 'Cantor's diagonal argument': https://en.wikipedia.org/wiki/Cantor%27s_diagonal_argument

[Wikipedia-3] 'Uncertainty principle': https://en.wikipedia.org/wiki/Uncertainty_principle

User Stories and BDD – Part 4, Features Are Not Stories

Features and stories serve different purposes in software delivery. Seb Rose finishes off his BDD series by encouraging us to be mindful of the difference.

This is the fourth in a series of articles digging into user stories, what they're used for, and how they interact with a BDD approach to software development. This is the last in this series, but certainly not the last time I'll be talking about user stories. However, since it brings the current narrative arc to a close, it is perhaps the end of the beginning [Churchill42].

Lifecycle of a story – revisited

User Stories start off as placeholders for a conversation [Rose22]. They're ideas, often large, not fully formed. They could be valuable, but they're not ready for development just yet.

As we refine them (through discovery [Rose23]) they become better understood. **Collaboration** ensures that they make sense to the whole team. The business requirements that limit their scope are negotiated and agreed.

Now it's possible to see what's involved in delivering the story, we can split them into smaller chunks. Smaller chunks mean faster feedback, smoother flow, and less waste [Rose24]. They're no longer placeholders. Now they're **detailed small increments**, each one carrying a small payload of valuable functionality (see Figure 1).

As the stories get plucked off the backlog, they deliver enhancements and brand new features. Not in a big bang, but incrementally and iteratively. Many stories contribute to each feature. Some stories contribute to many features. No feature is ever finished – it's just done for now.

No further value

Just because something has been useful doesn't mean that it will continue to be useful. On the contrary, once you have used something its value usually diminishes. Consider a tube of toothpaste or a pack of stickies.

The story never was a requirement. It starts as a placeholder and is then transformed, first into a narrative and then into several detailed small increments of functionality. That process is important, because it enables the team to learn about the problem and the solution. It's important, because it allows us to **discover the requirements**. However, it is the feature files (and ancillary documentation) that capture the requirements, not the stories.

Have you ever tried to make sense of a team's system by reading the completed stories from their issue tracker? It's impossible.

Stories only make sense when seen as a sequence of events, playing out over time. Like one of those flip book animations that you made at school. Useful documentation, on the other hand, describes how the system behaves now – not the history of how it evolved to behave like it does.

Seb Rose Seb has been a consultant, coach, designer, analyst and developer for over 40 years. Co-author of the BDD Books series Discovery and Formulation (Leanpub), lead author of *The Cucumber for Java Book* (Pragmatic Programmers), and contributing author to *97 Things Every Programmer Should Know* (O'Reilly).

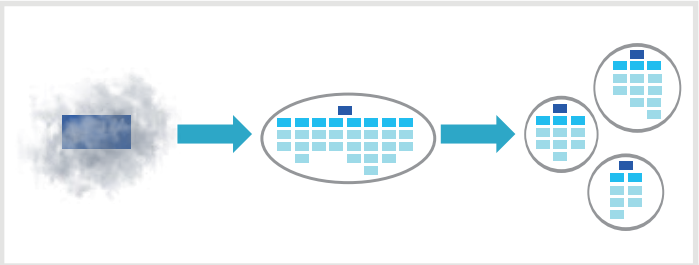

Figure 1

Back in the mists of time, when stories were written on index cards, XP teams used to indulge in a confetti party at the end of each iteration. The story cards in the 'done' column were torn into small pieces and thrown into the air, to rain down as the team danced around, celebrating their success. Most teams have moved on to electronic story tracking systems, which saves paper, but makes the disposal of stories problematic and much less fun.

Stories, once delivered, have no further value. Certainly not as documentation of the system's behaviour.

Souvenirs

So if we don't want to capture stories in our **living documentation**, what do we want in there?

Well, there are many useful things you can capture in a feature file. Written prose that explains the context and need for a feature is still useful for people reading the documentation for the first time. You can add links out to other sources of information like UX wireframes or user research data. But probably the most important thing to document alongside your scenarios are the **rules**.

If you cast your mind back to our earlier discussion of example mapping [Rose23], you'll remember that requirements are also known as acceptance criteria or rules [Keogh11].

Modern versions of Gherkin provide a defined way to capture rules – the `Rule` keyword. All scenarios that follow a `Rule` statement are expected to illustrate that rule. The `Rule` statement is in scope until either the next `Rule` statement is encountered or the end of the feature file.

```
Feature: Hear Shout
  Rule: Shouts have a range of 1000m
    Scenario: In range, shout is heard
    …
    Scenario: Out of range, shout is not heard
    …
 Rule: Shouts must not be empty
   Scenario: Zero length, not valid
   …
   Scenario: Only whitespace, not valid
   …
   Scenario: Single character, valid
  …
```

The bottom line is:

- Stories helped us decide what we want (and how to deliver it).
- Features document what we've got.

By all means keep the story index cards in your drawer as a souvenir (although I guarantee you'll never look at them). Please don't pollute your feature files with them.

Traceability

There are processes and organisations that value historical stories. The word I hear most often in this regard is '**traceability**', so I'd like to write a few words about the challenges.

In regulated industries (health, defence, finance) there is a need to demonstrate a rigorous end-to-end development process, from inception to delivery. Since stories are a visible, tangible artefact, often stored in electronic data management systems, they are easy to include in the web of traceability.

The trouble is that stories are neither definitive nor independent. Their lifecycle makes them no more suitable for traceability purposes than conversations around a watercooler or notes scribbled on your tablet. If you use stories for traceability, one of these days you're sure of a big surprise [Wohlrab20].

Following a link from a story through to the commit(s) that delivered the code and test scripts might give you confidence that the necessary work has been done. And since feature files will be part of those commits, **the resulting behaviour is also documented**. However, since subsequent stories may have been delivered, this means that you cannot infer anything about the current behaviour of the system by traversing links from a story through to commits.

Tools are currently being developed that will make it simpler to trace from a specific version of a scenario through to the stories that caused it to be written, which will help with some compliance needs. Nonetheless, it is important to remember that **stories are neither requirements nor deliverables**. They are **transient artefacts** that facilitate delivery, not persistent artefacts that document behaviour.

Continue the conversation

In this article, I hope that I've demonstrated why there's no place for stories inside your feature files. If you have any feedback or questions, I'd be happy to hear it. ■

References

[Churchill42] Winston Churchill, 1942, uploaded on 15 May 2010 at https://www.youtube.com/watch?v=pdRH5wzCQQw

[Keogh11] Liz Keogh 'Acceptance Criteria vs. Scenarios', posted on 20 June 2011 at https://lizkeogh.com/2011/06/20/acceptance-criteria-vs-scenarios/

[Rose22] Seb Rose 'User Stories and BDD – Part 1' in Overload 171, October 2022, available at https://accu.org/journals/overload/30/171/rose/

[Rose23] Seb Rose 'User Stories and BDD – Part 2, Discovery' in Overload 178, December 2023, available at https://accu.org/journals/overload/31/178/rose/

[Rose24] Seb Rose 'User Stories and BDD – Part 3, Small or Far Away?' in Overload 179, February 2024, available at https://accu.org/journals/overload/32/179/rose/

[Wohlrab20] R Wohlrab, E Knauss, JP Steghöfer *et al*. (2020) 'Collaborative traceability management: a multiple case study from the perspectives of organization, process, and culture', *Requirements Eng* 25, 21–45, available at https://doi.org/10.1007/s00766-018-0306-1

This article was first publishedon Seb Rose's blog on 30 January 2020: https://cucumber.io/blog/bdd/user-stories-and-bdd-features-are-not-stories/ It has been reviewed and updated for *Overload*.

Static Reflection in C++

Static reflection is under consideration for C++26. Wu Yongwei demonstrates how to achieve reflection now and shows some examples of what C++26 might make possible.

Static reflection will be an important part of C++ compile-time programming, as I discussed in the October issue of *Overload* [Wu24]. This time I will discuss static reflection in detail, including how to emulate it right now, before it's been added to the standard.

Background

Many programming languages support reflection (Python and Java, for example). C++ is lagging behind.

While this *is* the case, things are probably going to change in C++26. Also, what will be available in C++ will be very different from what is available in languages like Java or Python. The keyword is 'static'.

Andrew Sutton defined 'static reflection' as follows [Sutton21]:

> Static reflection is the integral ability for a metaprogram to observe its own code and, to a limited extent, generate new code at compile time.

'Compile-time' is the special sauce in C++, and it allows us to do things impossible in other languages:

- **Zero-overhead abstraction.** As Bjarne Stroustrup famously put it, 'What you don't use, you don't pay for. What you do use, you couldn't hand-code any better.' If you do not need static reflection, it will not make your program fatter or slower. But it will be at your hand when you do need it.
- **High performance.** Due to the nature of compile-time reflection, it is possible to achieve unparalleled performance, when compared with languages like Java or Python.
- **Versatility at both compile time and run time.** The information available at compile time *can* be used at run time, but not vice versa. C++ static reflection can do things that are possible in languages like Java, but there are things that C++ can do but are simply impossible in other languages.

What we want from reflection

When we talk about static reflection, what do we really want? We really want to see what a compiler can see, and we want to be able to use the relevant information in the code. The most prominent cases are `enum` and `struct`. We want to be able to iterate over all the enumerators, and know their names and values. We want to be able to iterate over all the data members of a struct, and know their names and types. Obviously, when a data member is an aggregate, we also want to be able to recurse into it during reflection. And so on.

Wu Yongwei Having been a programmer and software architect, Yongwei is currently a consultant and trainer on modern C++. He has nearly 30 years' experience in systems programming and architecture in C and C++. His focus is on the C++ language, software architecture, performance tuning, design patterns, and code reuse. He has a programming page at http://wyw.dcweb.cn/, and he can be reached at wuyongwei@gmail.com.

Regretfully, we cannot do all these things today with 'standard' definitions. Yes, in some implementations it is possible to hack out some of the information with various tricks. I would prefer to use macros and template techniques to achieve the same purpose, as the code is somewhat neater, more portable, and more maintainable – at the cost of using non-standard definition syntaxes. Of course, nothing beats direct support from the future C++ standard.

A few words on macro techniques

I have accumulated some macro code along the years, starting from the work of Netcan [Netcan]. The key facilities are:

- `GET_ARG_COUNT`: Get the count of variadic arguments, so that `GET_ARG_COUNT(a, b, c)` becomes `3`.
- `REPEAT_ON`: Apply the variadic arguments to the main function macro (with a count), so that `REPEAT_ON(func, a, b, c)` becomes `func(0, a) func(1, b) func(2, c)`.
- `PAIR`: Remove the first pair of parentheses from the argument, so that `PAIR((long)v1)` becomes `long v1`.
- `STRIP`: Remove the first part in parentheses, so that `STRIP((long)v1)` becomes `v1`.
- …

Some of the ideas were around at least as early as 2012 [Fultz12a], but Paul Fultz's code was not suitable for real software projects. My current code should be considered production-ready, and its variant has already been used in some large applications. It has also been tested under all mainstream compilers, including the pre-standard MSVC (supporting old MSVC did take some efforts). You can find my definitions in the Mozi open-source project [mozi].

Some consider macros evil, and macros should really be avoided where we can find better alternatives, but I personally find macros easier to understand and maintain than some template hacks.

A taste of enum reflection

Oftentimes we want to know how many enumerators are defined in an enumeration, what their underlying values are, and what their string forms are. The last need is especially important for debugging/logging purposes.

Existing implementations

There are existing libraries that provide such capabilities, like Magic Enum C++ [magic_enum] and Better Enums [better-enums].

Magic Enum C++ requires a recent C++17-conformant compiler, and it works with the standard form of enumeration definition. However, since it uses compile-time counting techniques to find out the values of enumerators, the range of enumerators are limited. Also, it does not live well with enumeration values that are not declared in the enumeration definition (say, something like `Color{100}`) – invoking

We really want to see what a compiler can see, and we want to be able to use the relevant information in the code

`magic_enum::enum_name` on such a value will get an empty `string_view`. This said, I recommend using it, if it satisfies your needs.

Better Enums works with basically any compiler, even old C++98 ones. However, it requires you to use a special form for enumeration definition. That alone is ugly but acceptable. What is uglier is that the result is *not* an `enum`, and it cannot get along with values not declared in the enumeration definition at all – stringifying such a value will cause a segmentation fault…

My handmade implementation

Mainly to understand the problem better, I tried enum reflection myself. Basically, I did the following things:

- Make sure the result of code generation was still an `enum`
- Provide the mapping from enumerators to their string forms via inline constexpr variables
- Support necessary operations using function overloads such as `to_string`

An example of an `enum class` definition:

```
DEFINE_ENUM_CLASS(Color, int,
                  red = 1, green, blue);
```

Then I can use it as follows:

```
cout << to_string(Color::red) << '\n';
cout << to_string(Color{9}) << '\n';
```

And I will get the following output:

```
red
(Color)9
```

Some implementation details

While you can check the implementation details in the Mozi project, I would like to give an overview of what `DEFINE_ENUM_CLASS` does. Its definition is in Listing 1.

You can see clearly that it does three things:

- Define a standard `enum class`
- Define an inline constexpr array that contains pairs of underlying integer values and the string forms of enumerators, which are generated by applying the `ENUM_ITEM` macro on the enumerators
- Declare utility functions for the new `enum` type

With the definition of `Color` above, it will expand to Listing 2 (at first level). The full expansion results in something like Listing 3.

This should be enough for you to see the basic ideas. And you can check out the implementation details in the Mozi project, if interested.

```
#define DEFINE_ENUM_CLASS(e, u, ...)        \
  enum class e : u { __VA_ARGS__ };         \
  inline constexpr std::array<              \
    std::pair<u, std::string_view>,         \
    GET_ARG_COUNT(__VA_ARGS__)>             \
    e##_enum_map_{REPEAT_FIRST_ON(          \
      ENUM_ITEM, e, __VA_ARGS__)};          \
  ENUM_FUNCTIONS(e, u)
```

Listing 1

```
enum class Color : int { red = 1, green, blue };
inline constexpr std::array<
  std::pair<int, std::string_view>, 3>
  Color_enum_map_{
    ENUM_ITEM(0, Color, red = 1),
    ENUM_ITEM(1, Color, green),
    ENUM_ITEM(2, Color, blue),
  };
ENUM_FUNCTIONS(Color, int)
```

Listing 2

```
enum class Color : int { red = 1, green, blue };
inline constexpr std::array<
  std::pair<int, std::string_view>, 3>
  Color_enum_map_{
    std::pair{
      to_underlying(Color(
        (eat_assign<Color>)Color::red = 1)),
      remove_equals("red = 1")},
    std::pair{
      to_underlying(
        Color((eat_assign<Color>)Color::green)),
      remove_equals("green")},
    std::pair{to_underlying(Color((
                eat_assign<Color>)Color::blue)),
              remove_equals("blue")},
  };

inline std::string to_string(Color value)
{
  return enum_to_string(to_underlying(value),
                        "Color",
                        Color_enum_map_.begin(),
                        Color_enum_map_.end());
}
```

Listing 3

Example of enum reflection in C++26

The code in Listing 4 (overleaf) should supposedly work as per P2996 [P2996r7], the current static reflection proposal for C++26. It uses the following reflection features:

- `^E` generates the reflection information for the `enum` type `E`.
- `[:e:]` 'splices' the reflection object back into a source entity, which is an enumerator here.

Some consider macros evil, and macros should really be avoided where we can find better alternatives

```
template <typename E>
  requires std::is_enum_v<E>
std::string to_string(E value)
{
  template for (constexpr auto e :
                std::meta::enumerators_of(^E)) {
    if (value == [:e:]) {
      return std::string(
        std::meta::identifier_of(e));
    }
  }
  return std::string("(") +
         std::meta::identifier_of(^E) + ")" +
         std::to_string(to_underlying(value));
}
```

Listing 4

- The `template for` loop (expansion statement) allows iteration over heterogeneous objects at compile time.
- `std::meta::enumerators_of` gets all enumerators of the enumeration.
- `std::meta::identifier_of` gets the identifier/name of a reflected object. Here we use it once for the name of the enumerator, and once for the name of the enumeration.

It does the same thing as my handmade `to_string` without the manual scaffolding: no macros are needed any more.

The online implementation of an early proposal, P2320 [P2320r0], available in Compiler Explorer, is convenient for demonstration purposes. The obvious differences between P2996r7 and P2320 are function names: `enumerators_of` was `members_of`, and `identifier_of` was `name_of`. There are some other reflection-supporting Godbolt compilers, which are not yet capable enough, mainly due to the lack of support for expansion statements. I have written two different versions of the `enum` reflection code that work under P2320:

- https://cppx.godbolt.org/z/8rWTcf1KP: A simple version that does linear search as shown above
- https://cppx.godbolt.org/z/P5Ycdv3xj: A more complex version that collects the string forms of enumerators and sorts them, so that we can use binary search later on (similar to what I did in Mozi)

As you can see, while it is still not trivial to implement the full logic, the major advantage is that we can use the standard `enum` definition form, without the current limitations of Magic Enum C++. The reflection information can be accessed at compile time, but we can save it so that we can access it later at run time.

Reflection on structs

The need for reflection of `struct`s is even stronger than `enum`s. Reflection is very helpful in debugging/logging, and serialization and deserialization become easy when reflection is available.

Existing implementations

I know two existing implementations for reflection purposes.

Boost.PFR [pfr] is:

> …a C++14 library for very basic reflection that gives you access to structure elements by index and provides other `std::tuple` like methods for user defined types without any macro or boilerplate code.

It is easy to use. It supports common operations like iteration, comparison, and output. However, due to the lack of static reflection, it has no way to access the names of fields.

Struct_pack [struct_pack] is a "very easy to use, high performance serialization library". It requires C++17 and focuses on serialization/deserialization. It is *not* designed for generic reflection purposes, and you cannot really use it for your own serialization scenarios (without some serious hacking).

While not a real implementation, the earliest code I am aware of about struct reflection is from Paul Fultz [Fultz12b]. Modern compile-time techniques were not ready in 2012, so while the basic ideas were similar, Netcan and I did not borrow much code from him.

My handmade implementation

I have my own struct reflection method, which does not have the limitations of Boost.PFR but under the hood requires macro use. However, once static reflection is standardized, much of the code and techniques can be adapted to standard C++.

The basic approach is:

- Use macros to generate code so that the resulting type is really a struct of the supposed size (no fatter!)
- Generate nested types and static constexpr data members which provide the needed information
- Provide stand-alone function templates for the common operations

Here is an example. Suppose we have the following definitions:

```
DEFINE_STRUCT(
  Point,
  (double)x,
  (double)y
);
DEFINE_STRUCT(
  Rect,
  (Point)p1,
  (Point)p2,
  (uint32_t)color
);
```

Then we can initialize such `struct`s as usual:

```
Rect rect{
  {1.2, 3.4},
  {5.6, 7.8},
  12345678
};
```

What is currently possible with macro techniques will be possible with the C++26 static reflection, only that it will be simpler

We can print it easily:

```
print(data);
```

And we will get:

```
{
    p1: {
        x: 1.2,
        y: 3.4
    },
    p2: {
        x: 5.6,
        y: 7.8
    },
    color: 12345678
}
```

Usage scenario: copy same-name fields

The implementation details may not be very interesting, but we *do* have more interesting usage scenarios. One thing I implemented was copying fields of interest.

Suppose the following definitions (please notice that `v2` and `v4` have different types in `S1` and `S2`):

```
DEFINE_STRUCT(S1,
  (uint16_t)v1,
  (uint16_t)v2,
  (uint32_t)v3,
  (uint32_t)v4,
  (string)msg
);

DEFINE_STRUCT(S2,
  (int)v2,
  (long)v4
);

S1 s1{…};
…
S2 s2;
```

Then the following statement will *do the right thing*:

```
copy_same_name_fields(s1, s2);
```

And it is done with the highest possible efficiency, equivalent to `s2.v2 = s1.v2; s2.v4 = s1.v4;`. I have checked its compiler-generated x86-64 assembly code, which is:

```
movzx   eax, WORD PTR s1[rip+2]
mov     DWORD PTR s2[rip], eax
mov     eax, DWORD PTR s1[rip+8]
mov     QWORD PTR s2[rip+8], rax
```

I do not think Java or Python can ever do anything similar!

If this does not look useful, just think about big database records. Imagine we have a container of big `BookInfo` objects, and we want to do something like the SQL `SELECT name, publish_year WHERE author_id =` …. The code would be that in Listing 5.

```
DEFINE_STRUCT(
  BookInfoNameYear,
  (string)name,
  (int)publish_year
);

BookInfoNameYear record{};
vector<BookInfoNameYear> result;
Container<BookInfo> container;
while (…) {
  auto it = container.find(…);
  …
  copy_same_name_fields(*it, record);
  result.push_back(record);
}
```

Listing 5

Isn't the code much simpler than, while as efficient as, manually copying the needed fields? The advantage is especially obvious when there are many such fields.

I have seen copying tens of fields in real code, often followed by serialization (to send the information over the network), which is a topic I will discuss separately.

Under the hood

`DEFINE_STRUCT` is defined as follows:

```
#define DEFINE_STRUCT(st, ...)                  \
  struct st {                                   \
    using is_reflected = void;                  \
    template <typename, size_t>                 \
    struct _field;                              \
    static constexpr size_t _size =             \
      GET_ARG_COUNT(__VA_ARGS__);               \
    REPEAT_ON(FIELD, __VA_ARGS__)               \
  }
```

The `S2` above will first expand to something like:

```
struct S2 {
  using is_reflected = void;
  template <typename, size_t>
  struct _field;
  static constexpr size_t _size = 2;
  FIELD(0, (int)v2)
  FIELD(1, (long)v4)
};
```

And `FIELD(0, (int)v2)` will expand to:

```
int v2;
template <typename T>
struct _field<T, 0> {
  using type = decltype(decay_t<T>::v2);
  static constexpr auto name = CTS_STRING(v2);
  constexpr explicit _field(T&& obj)
    : obj_(std::forward<T>(obj)) {}
  constexpr decltype(auto) value()
  { return (std::forward<T>(obj_).v2); }
  T&& obj_;
};
```

```
template <size_t I, typename T>
constexpr decltype(auto) get(T&& obj)
{
  using DT = decay_t<T>;
  static_assert(I < DT::_size,
                "Index to get is out of range");
  return typename DT::template _field<T, I>(
           std::forward<T>(obj))
    .value();
}

template <typename T, typename F, size_t... Is>
constexpr void
for_each_impl(T&& obj, F&& f,
              std::index_sequence<Is...>)
{
  using DT = decay_t<T>;
  (void(std::forward<F>(f)(
     index_t<Is>{},
     DT::template _field<T, Is>::name,
     get<Is>(std::forward<T>(obj)))),
   ...);
}

template <typename T, typename F>
constexpr void for_each(T&& obj, F&& f)
{
  using DT = decay_t<T>;
  for_each_impl(
    std::forward<T>(obj), std::forward<F>(f),
    std::make_index_sequence<DT::_size>{});
}
```

Listing 6

I leave `CTS_STRING(v2)` unexpanded, as it has two possible definitions, depending on the environment [Wu22]. For now, you can think of it as just `"v2"`, with some additional magic (which `copy_same_name_fields` requires).

When you have an `obj` of type `S2`, you can access its members using their field numbers: `_field<S2&, 0>(obj).value()` is exactly `obj.v2` (with the correct value category), and `S2::_field<S2&,0>::type` is the type of `obj.v2` (which is `int`). With the help of fold expressions, more complex things like compile-time field iteration is now possible, as shown in Listing 6.

Now, a function call like `for_each(obj, f)` will be equivalent to:

```
f(0, S2::_field<S2&, 0>::name, get<0>(obj));
f(1, S2::_field<S2&, 1>::name, get<1>(obj));
```

Facilities like `for_each` is essential in implementing user-visible tools like `print` and serialization.

Example of struct reflection in C++26

As in the case of enum reflection, we will be able to dispense with the macro use when C++26 static reflection arrives. Listing 7 is a demo implementation of `print` (slightly changed from [Wu24] in order to conform to the updated version of P2996).

Given what we have known about `^` and `[:…:]`, the code is pretty straightforward.

We can verify it actually works under P2320 (https://cppx.godbolt.org/z/G3EcvhKxK) and P2996, with an expansion statement workaround (https://godbolt.org/z/77PYjzcW8).

A few more words on Mozi

Mozi is an open-source project I started in late 2023, mostly for the purpose of experimenting with macro-based static reflection. I have implemented

```
template <typename T>
void print(const T& obj, ostream& os = cout,
           std::string_view name = "",
           int depth = 0)
{
  if constexpr (is_class_v<T>) {
    os << indent(depth) << name
       << (name != "" ? ": {\n" : "{\n");
    template for (constexpr meta::info member :
        meta::nonstatic_data_members_of(^T)) {
      print(obj.[:member:], os,
            meta::identifier_of(member),
            depth + 1);
    }
    os << indent(depth) << "}"
       << (depth == 0 ? "\n" : ",\n");
  } else {
    os << indent(depth) << name << ": " << obj
       << ",\n";
  }
}
```

Listing 7

generic comparison, copying, printing, and serialization/deserialization. A serialization scenario called `net_pack` is implemented, which includes fully automatic byte-order swap and is suitable for coping with network datagrams. A special `bit_field` type is provided to provide bit-field support over the network.

I regard it as a demonstration of some interesting things that are possible with static reflection. What is currently possible with macro techniques will be possible with the C++26 static reflection, only it will be simpler, for both the implementer and the user. ■

References

[better-enums] https://github.com/aantron/better-enums

[Fultz12a] Paul Fultz II, 'Is the C preprocessor Turing complete?', May 2012, https://pfultz2.com/blog/2012/05/10/turing

[Fultz12b] Paul. Fultz II, 'C++ Reflection in under 100 lines of code', July 2012, https://pfultz2.com/blog/2012/07/31/reflection-in-under-100-lines

[magic_enum] https://github.com/Neargye/magic_enum

[mozi] https://github.com/adah1972/mozi

[Netcan] https://github.com/netcan/recipes/tree/master/cpp/metaproggramming

[P2320r0] Andrew Sutton *et al.*, 'The Syntax of Static Reflection', 2021, http://wg21.link/p2320r0

[P2996r7] Wyatt Childers *et al.*, 'Reflection for C++26' (revision 7), October 2024, http://wg21.link/p2996r7

[pfr] https://github.com/boostorg/pfr

[struct_pack] https://github.com/alibaba/yalantinglibs

[Sutton21] Andrew Sutton, 'Reflection: Compile-Time Introspection of C++', ACCU 2021, https://www.youtube.com/watch?v=60ECEc-URP8

[Wu22] Yongwei Wu, 'Compile-Time Strings', *Overload*, 30(172):4-7, December 2022, https://accu.org/journals/overload/30/172/wu/

[Wu24] Yongwei Wu, 'C++ Compile-Time Programming', *Overload*, 32(183):7-13, October 2022, https://accu.org/journals/overload/32/183/wu/>

Senders/Receivers: An Introduction

C++26 will introduce a new concurrency feature called std::execution, or senders/receivers. Lucian Radu Teodorescu explains the idea and how to use these in detail.

In June 2024, at the WG21 plenary held in St. Louis, the P2300R10: **`std::execution`** paper [P2300R10], also known as senders/receivers, was formally adopted for inclusion in C++ 26. The content of the paper quickly found its way into the working draft for the C++ standard [WG21]. You can find more about the highlights of the St. Louis meeting in Herb Sutter's trip report [Sutter24].

Senders/receivers represent one of the major additions to C++, as they provide an underlying model for expressing computations, adding support for concurrency, parallelism, and asynchrony. By using senders/receivers, one can write programs that heavily and efficiently exploit concurrenc y, all while maintaining thread safety (no deadlocks, race conditions, etc.). This is applicable not only to a few classes of concurrent problems but, at least in theory, to all types of concurrency problems. Senders/receivers provide a cost-free way of expressing computations that can run on different hardware with different constraints. They support creating computation chains that execute work on the CPU, GPU, and also enable non-blocking I/O.

Although the proposal has many advantages, there are still people who see the addition of this feature to the C++ standard at this point as a mistake. Some of the cited reasons are the complexity of the feature, compilation times, immaturity, and teachability. The last one caught my attention.

In this article, I plan to provide an introduction to senders/receivers as described in P2300 (and some related papers). The goal is not necessarily to showcase the many advantages of this model or delve into the details of complex topics. Rather, it is to offer a gentle introduction for those who have never read the paper or watched a talk on senders/receivers. We want the reader to understand the basic concepts of using senders/receivers without needing to grasp the intricate details of their implementation.

The hope is that, by the end of the article, the reader will be able to write some programs that use senders/receivers. The examples here are written as if the reader is coding with the feature already included in the standard library. Currently, no standard library provider ships senders/receivers; however, the reader can use the reference implementation of the feature [stdexec].

Starting example

Listing 1 shows a simple example that prints *Hello, world!* using senders/receivers. Receivers don't typically appear in the user code (they appear in the implementation of the algorithms that deal with senders), so we can also say that Listing 1 shows an example of using basic senders.

The example is equivalent (up to a point) to the code in Listing 2. We describe the action of printing *Hello, world!* to standard output; this description is stored in the variable computation. Then, we execute the action described by computation, producing the actual printing of the message. The action itself is composed of two parts: one that describes a string value and one that describes an action that takes the string and prints it out.

```
using stdexec = std::execution;

stdexec::sender auto computation
  = stdexec::just("Hello, world!")
  | stdexec::then([](std::string_view s) {
    std::print(s);
  });
std::this_thread::sync_wait(
  std::move(computation));
```

Listing 1

```
std::function<void()> computation = []{
  std::string_view s = "Hello, world!";
  std::print(s);
};
computation();
```

Listing 2

The code **`just(X) | then(f)`** describes work that is equivalent to **`f(X)`**. Adding another **`then`**, we have the work described by **`just(X) | then(f) | then(g)`** as equivalent to **`g(f(X))`**. If **`f`** and **`g`** don't produce any values, then **`just(X) | then(f) | then(g)`** describes work equivalent to **`f(X); g()`**. **Senders are designed with composability in mind**; they allow expressing complex computations in terms of simpler ones.

The actual execution of the work described by **`computation`** occurs when **`sync_wait`** is invoked; if **`sync_wait`** were not present, no work would be executed.

Although simple, Listing 1 demonstrates a few important characteristics of working with senders:

- senders describe computations;
- senders are designed to compose well;
- senders are executed lazily; in our example, nothing happens until **`sync_wait`** is invoked.

In addition to these, there are two more important aspects of senders, both of which will be explored later in this article:

- senders can be used to describe concurrent/asynchronous work;
- senders enable structured concurrency.

Let's look into the first point.

Representing concurrency

The code in Listing 3 shows a simple example of executing code on a different thread. In the senders/receivers world, we don't operate with

Lucian Radu Teodorescu has a PhD in programming languages and is a Staff Engineer at Garmin. He likes challenges; and understanding the essence of things (if there is one) constitutes the biggest challenge of all. You can contact him at lucteo@lucteo.ro

moving between execution contexts is pretty easy, if we arrange the work so that it can be described by a chain of senders

threads; we operate with *schedulers*. Schedulers are handles to execution contexts; that is, schedulers provide access to one or more threads. Schedulers dictate *where* particular work needs to be executed.

In our example, we obtain the system scheduler. This is not part of the original P2300 [P2300R10] proposal, but it has been added as an extension through P2079: System execution context [P2079R5]; the idea of a system scheduler was deemed very important for inclusion in senders/receivers [P3109R0]. The system scheduler describes an execution context intended to be shared by all parts of the application or even across applications.

The call to `schedule(sch)` returns a sender. This sender represents work that starts on a thread belonging to the system execution context. It doesn't send any value to the next sender but ensures that the work described by the next sender occurs on this thread.

The work described by `schedule(sch) | then(f)` is, to a point, equivalent to `std::thread([]{ f() })`, with the difference that the new thread is part of an execution context for which `sch` is a handle.

We use `schedule()` to start new work in an execution context, but sometimes we need to transfer execution from one context to another. For this, we can use the `continue_on()` algorithm. If we have a computation executed in one execution context and another computation that needs to be executed in a different context, we might use `continue_on()` to connect the two computations. For example, this chain describes work that executes `f` on the original thread and executes `g` on a (most likely) different thread represented by the scheduler `sch`:

```
just() | then(f) | continue_on(sch) | then(g)
```

With `schedule()` and `continues_on()` algorithms, one can implement any type of movement of work between threads. To make things easier to express in some cases, the senders/receivers proposal provides another algorithm: `starts_on()`. This can be used when we want to start a chain of work on a specific scheduler, but without specifying the scheduler in the work itself.

Listing 4 gives an example of `starts_on()` and of `continues_on()`. We have a sender that describes the work of reading data from a socket. In this description, we haven't specified on which scheduler this needs to be executed. However, in the overall computation, the expression `starts_on(io_sched, std::move(read_data_snd))` ensures that the work is actually started in the context of the given I/O scheduler.

The example shows also a usage for `continues_on()`. The part that reads data from a socket (i.e., the work represented by `read_data_snd`) will be executed on the I/O scheduler. As we want the processing to happen on a 'work scheduler', we have to specify that the execution should switch threads. This is done by the `continues_on(work_sched)` expression. Similarly, after processing the data on the work scheduler, we want to go back to the I/O scheduler to write back the response. To do this, we call `continues_on()` again, passing the handle to the I/O scheduler.

```
stdexec::scheduler auto sch
  = get_system_scheduler()
stdexec::sender auto computation
  = stdexec::schedule(sch)
  | stdexec::then([] {
    std::print("Hello, from a different thread");
  });
std::this_thread::sync_wait(
  std::move(computation));
```

Listing 3

```
stdexec::sender auto read_data_snd
  = stdexec::just(connection, buffer)
  | stdexec::then(read_data);

stdexec::sender auto process_all_snd
  = stdexec::starts_on(io_sched,
    std::move(read_data_snd))
  | stdexec::continues_on(work_sched)
  | stdexec::then(process_data)
  | stdexec::continues_on(io_sched)
  | stdexec::then(write_result);

std::this_thread::sync_wait(
  std::move(process_all_snd));
```

Listing 4

One can see that moving between execution contexts is pretty easy, if we arrange the work so thatsuch as it can be described by a chain of senders.

Waiting for multiple senders

So far, we've seen examples in which different work items run on different threads, but all the examples assumed a sequenced execution of work items. We did not have an example in which two functions would run concurrently. Let's correct that.

Listing 5 shows an example in which two functions `f` and `g` are run concurrently. To make this possible, we use the `when_all()` algorithm. This receives multiple senders and ensures that the results from all the senders are combined together before printing the results.

Both branches of work that go into the `when_all()` sender are started at the same time, but they are independent. Sometimes, we want to have

```
stdexec::sender auto s1 =
  stdexec::schedule(sch) | stdexec::then(f);
stdexec::sender auto s2 =
  stdexec::schedule(sch) | stdexec::then(g);
stdexec::sender auto both_results =
stdexec::when_all(s1, s2);
stdexec::sender auto print_results
  = std::move(both_results)
  | stdexec::then([](auto... args) {
    std::print("Results: {}, {}", args...);
  });
```

Listing 5

The work of a sender can complete with multiple types of values or multiple types of errors. More precisely, a sender can support any combination of completion signals

```
sender auto common =
  schedule(sch) | then(p) | split();
sender auto s1 = common | then(f);
sender auto s2 = common | then(g);
sender auto both_results = when_all(s1, s2);
sender auto print_results
  = std::move(both_results)
  | then([](auto... args) {
    std::print("Results: {}, {}", args...);
  });
```

Listing 6

some common processing, then execute two (or more) things concurrently, and then join the work chain together. This can be accomplished using the `split()` algorithm. Listing 6 (on the following page) shows an example of this. Here, when the work is started, function `p` is called first, and then `f` and `g` are called concurrently after `p` is finished.

Executing in bulk

The senders we've seen so far can only work on a single item at a given time. But what if we have many items that we need to work on? If one has N elements to process, one can use the `bulk()` algorithm to describe computations that process these elements.

Listing 7 presents an example of implementing the basic linear algebra *axpy* operation (from 'a x plus y') [Wikipedia-1]. For each index `i` in the range [`0, x.size()`), we invoke the given lambda function.

If the sender prior to applying `bulk()` produces a value, that value is passed to the functor given to `bulk()`; naturally, if the previous sender completes with multiple values, they are all passed to the functor. The same example can thus be written as in Listing 8.

```
double a;
std::vector<double> x;
std::vector<double> y;
sender auto process_elements
  = just()
  | bulk(x.size(), [&](size_t i) {
    y[i] = a * x[i] + y[i]
  });
```

Listing 7

```
double some_value;
std::vector<double> x;
std::vector<double> y;
sender auto process_elements
  = just(some_value)
  | bulk(x.size(), [&](size_t i, double a) {
    y[i] = a * x[i] + y[i]
  });
```

Listing 8

Shape of senders and structuredness

One important characteristic of senders that we haven't discussed before is their shape. This allows senders to compose well, be extensible, and achieve structured concurrency.

Similar to a traditional function, the work represented by a sender has one entry point and one exit point, usually called *completion* (or *completion signal*). A function can either complete with a value or throw an exception – there are two ways a function can complete. A sender has a third type of completion indicating cancellation. In the world of senders/receivers, we name them as follows:

- `set_value(auto... values)` – used when the sender's work successfully produces the output values;
- `set_error(auto err)` – used when the sender's work completes with an error `err`;
- `set_stopped()` – used when the work represented by the sender is cancelled.

A traditional function can produce only one value. A sender, on the other hand, can produce multiple values; this is why the signature of `set_value()` allows multiple arguments. A traditional function can signal errors (that are different from return values) only through exceptions; a sender can represent work that can complete with an error of any type – `std::exception_ptr, std::error_code`, or any user-defined error type. When the work of a sender is cancelled, there is no value to produce, and thus, there is no argument to `set_stopped()`.

A regular function has one return type and can additionally produce exceptions. Thus, a function `T f(...)` can either complete with `T` or with an `std::exception_ptr`. There isn't much variance possible with regular functions. The work of a sender, on the other hand, can complete with multiple types of values or multiple types of errors. More precisely, a sender can support any combination of completion signals. Some senders might complete with different sets of value types, while others might complete with different types of errors, and so on.

For example, we can have a sender that has the following completion signals:

- `set_value(int)`,
- `set_value(std::string)`,
- `set_value(int, std::string)`,
- `set_error(std::exception_ptr)`,
- `set_error(std::error_code)`,
- `set_stopped()`.

We can also have senders that complete with just a subset of these types of completion signals. For example, the sender returned by `just()` will only complete with `set_value()`, and the sender returned by `just(2, 3.14)` will only complete with

We can write good concurrent code without the fear of deadlocks and data races, simply by composing senders.

`set_value(int, double)`. Similarly, the sender returned by `just_error("some error string"s)` will only complete with `set_error(std::string)`, and the sender returned by `just_stopped()` will only complete with `set_stopped()`.

These points suggest that senders are generalisations of functions, in the sense that they support multiple types of completion.

The choice of representing the completion signals as function calls is not accidental. This is how the work described by the senders actually calls the receivers. In P2300, a receiver is defined as "a callback that supports more than one channel" [P2300R10]. The end user does not need to be concerned with receivers; they serve merely as glue between senders. This is why, so far, we haven't introduced them and have only discussed senders. We will continue to do so, as senders are the main focus.

There is another important aspect that needs to be addressed for senders. In a regular function, the completion happens on the same thread as the entry point. For the work represented by senders, this is not required. We can start on one thread and complete on another. For example, the `schedule(sch)` algorithm describes work that starts on a thread and moves control to a thread governed by `sch`. Another good example is the `continue_on()` algorithm.

From this perspective too, senders are a generalisation of functions. I can't emphasise enough the importance of this. In non-concurrent code, structured programming taught us to work with functions. This means that with senders we can perform the same type of breakdown we were doing with functions. We can represent all parts of a program with senders, and we can even compose the entire program from senders. I've shown an example in the 'Structured Concurrency' *ACCU* talk [Teodorescu22].

As a consequence of senders describing work that behaves like functions, senders inherit structuredness properties. A sender contained within another sender must complete before its parent completes. We can have senders hide implementation details, thereby providing abstraction points. As mentioned above, we can decompose the program using senders.

In the end, all these structuredness properties make it easier to reason about the code. We can write good concurrent code without the fear of deadlocks and data races, simply by composing senders.

Senders can abstract work, so they can serve as an abstraction for any type of concurrent or asynchronous work. Here are a few examples:

- A sender can encapsulate a concurrent sort algorithm (which may run on the GPU or on the CPU) – an example of using senders to speed up programs.
- A sender can encapsulate the processing of an image; the processing can be done on a single thread, on multiple threads, or on GPUs – an example showing that concurrency concerns are hidden.
- A sender can encapsulate a `sleep` operation; the sender completes when the sleep period ends but doesn't keep any thread busy – an example of asynchrony.
- A sender can encapsulate the wait for the results of a remote procedure call over the network, while not keeping the local threads busy – another example of asynchrony.

Sender algorithms in the standard

The P2300 proposal [P2300R10], which was merged into the working draft for C++ 26, contains a set of algorithms that operate on senders. Because of their structuredness properties, senders compose well, so we should be able to build larger senders from smaller ones.

The C++ 26 standard will include several sender algorithms to be used as primitives for building more complex senders. These are grouped into three categories:

- **Sender factories**: They produce senders without requiring any other senders. Algorithms in the standard: `schedule()`, `just()`, `just_error()`, `just_stopped()`, `read_env()`.
- **Sender adaptors**: Given one or more senders, they return senders based on the provided senders. Algorithms in the standard: `starts_on()`, `continues_on()`, `schedule_from()`, `on()`, `then()`, `upon_error()`, `upon_stopped()`, `let_value()`, `let_error()`, `let_stopped()`, `bulk()`, `split()`, `when_all()`, `into_variant()`, `stopped_as_optional()`, `stopped_as_error()`.
- **Sender consumers**: They consume senders but don't produce any senders. Algorithms in the standard: `sync_wait()`, `sync_wait_with_variant()`.

All the sender factories and adaptors are defined in the `std::execution` namespace. The sender consumer algorithms are defined in the `std::this_thread` namespace.

We will briefly go through each of these algorithms.

Sender factories

We've already seen examples of the `just()` algorithm. This is used to create a sender that completes with the given values. We've also seen the `just_error()` algorithm, which creates a sender that completes with the given error. We've mentioned the `just_stopped()` algorithm as well; this algorithm produces a sender that completes with a `set_stopped()` signal.

The `read_env()` algorithm is more advanced. Given a *tag*, it tries to retrieve the property for that tag from the execution environment. That is, if we have a child sender inside a parent sender, the child sender can use r`ead_env()` to obtain various properties from the parent sender.

Sender adaptors

Before describing the actual sender adaptor algorithms, it's worth highlighting an important aspect of the syntax for most of these adaptors: there are two forms for the algorithm. We have a canonical form and a

pipeable form. The best way to explain this is with an example, and the `then()` algorithm is likely the best choice for illustrating this.

The canonical form of `then()` is: `then(sndr, ftor)`. When this is used, it returns a sender that, when `sndr` completes, applies `ftor` to its produced values and completes with the transformed values (function composition).

The piped form of `then()` is `then(ftor)`. This form should only be used in a piped context. An expression of the form `sndr | then(ftor)` is equivalent to calling `then(sndr, ftor)`. Usually, the piped form is easier to write, so many people prefer it.

Technically, `then(ftor)` is a *sender adaptor closure*, not a sender. The then sender also includes the previous sender, i.e., what comes before the pipe operator. However, colloquially we often refer to it as a sender, for simplicity.

Similar to the `then()` algorithm, we have `upon_error()` and `upon_stopped()`. They function in the same way as `then()`, but are applied to the error or stop completion channels, respectively. `upon_error()` applies the given functor to the incoming error and completes with the result of the function application. `upon_stopped()` calls the given functor and completes with `set_stopped()`.

We've already seen examples of `starts_on()` and `continues_on()`. The `on()` algorithm is a combination of these two: it executes work on a given scheduler (similar to `starts_on()`) but returns to the original scheduler upon completion (resembling `continues_on()`).

The `schedule_from()` algorithm is a foundational operation for `continues_on()`. It's not meant to be called directly by users but can be useful for specialising some of the transitions between schedulers.

We've also briefly described above the algorithms `bulk()` (used to execute the same function multiple times for a range of indices), `split()` (used to ensure that the same sender can be contained in the same chain of computation without executing the same work twice), and `when_all()` (used to combine the results of multiple senders).

The `let_*()` family of algorithms is important, yet they are often misunderstood. The `let_value()` algorithm is similar to the `then()` algorithm, but the given functor is expected to return a sender. This is the monadic bind operation for senders, i.e., a fundamental building block for senders. It is similar to the `optional<T>::and_then()` function (part of the so-called `std::optional` monadic operations).

Instead of this abstract explanation, let's illustrate with an example. Suppose we have a pipeline for performing image transformations (e.g., automatically enhancing an image). We want to abstract this pipeline, so we encapsulate the pipeline building into a function `enhance_image_sndr()` that takes an image as an argument and returns a sender that knows how to enhance the image. Using a pseudo-syntax, we would say that the type of `enhance_image_sndr()` is `Image -> Sender<Image>`. Now, we want to put this pipeline inside another pipeline that first loads the image, enhances it, and then writes it to the destination storage (disk, network, etc.). We cannot inject this function into our flow with `then()`; that would produce a `Sender<Sender<Image>>` instead of `Sender<Image>`. For that, we have `let_value()`. Listing 9 shows how the code may look.

Similar to `let_value()`, the `let_error()` algorithm performs the same job, but applies the given functor to the error produced by the previous sender. Additionally, `let_stopped()` applies the given functor when a stopped signal is received.

The remaining three sender adaptor algorithms (`into_variant()`, `stopped_as_optional()`, and `stopped_as_error()`) are designed to make it easier to work with different types of completion signals.

The first one, `into_variant()`, adapts a sender that might have multiple value completion signatures into a sender with a single completion signature consisting of an `std::variant` of `std::tuples`. It doesn't change any error or stopped completions. For example, if `snd` can complete with `set_value(std::string)` or `set_value(int, double)`, then `into_variant(snd)` is a sender that can complete with:

```
set_value(std::variant<std::tuple<std::string>,
          std::tuple<int, double>>)
```

The `stopped_as_optional()` algorithm removes the need for a stopped completion by transforming it into an empty optional value. Additionally, it transforms the value completion from a type `T` to an `std::optional<T>`. Thus, if `snd` is a sender that completes with either a value of `int` or a stopped signal, then `stopped_as_optional(snd)` will complete only with a value of `std::optional<int>`.

The `stopped_as_error()` algorithm behaves similarly but transforms a stopped completion signal into an error completion. Thus, if `snd` is a sender that completes with either a value of `int` or a stopped signal, then `stopped_as_error(snd, err)` will complete only with a value of type `int` or the error `err`.

```
// Returns a sender that produces 'Image' values
auto enhance_image_sndr(Image img) {...}
Image load();
void save(Image);

sender auto complete_pipeline
  = just()
  | then(load)
  | let_value([](Image img) {
    return enhance_image_sndr(img); })
  | then(save);
```

Listing 9

Sender consumers

The main sender consumer algorithm defined by the proposal is `sync_wait()`. We've seen this in our examples above. This algorithm takes one sender as input and performs the following actions:

- submits the work described by the given sender;
- blocks the current thread until the sender's work is finished;
- returns the result of the sender's work in the appropriate form to the caller:
  - returns an optional tuple of values – those that the given sender completes with – if the sender completes with `set_value()`;
  - throws the received error if the sender completes with `set_error()`;
  - returns an empty optional if the given sender completes with a stopped signal.

For a sender `snd` that completes with `set_value(int, double)`, the resulting type of `sync_wait(snd)` is:

```
std::optional<std::tuple<int, double>>
```

If `snd` completes with a value of type `int`, then `sync_wait(snd)` returns `std::optional<std::tuple<int>>` (not dropping the tuple part). If the given sender doesn't send a stopped completion signal, the return type will still contain the optional part, even if there will always be a value present.

An interesting restriction of this algorithm is that the given sender cannot complete with more than one `set_value()` signal. This is because the return type, as defined, cannot accommodate multiple value completion types.

If we have a sender that completes with multiple types of value signals, we can use the `sync_wait_with_variant()` algorithm. This is similar to `sync_wait()`, but its return type is an `std::optional` of an `std::variant` of `std::tuples`. For example, for a sender `snd` that can complete with `set_value(std::string)` and

`set_value(int, double)`, `sync_wait_with_variant(snd)` returns:

```
std::optional<std::variant<std::tuple
  <std::string>, std::tuple<int, double>>>
```

It may sound a bit complex, but it's straightforward with a bit of practice. After all, this is the most logical conclusion when considering the possible completion types for a sender.

Beyond P2300

The above section may have made it seem like P2300 proposes numerous algorithms to fully cover the needs of concurrency and asynchrony, but this is far from the truth. It simply lays the foundation for building basic senders. In fact, there is a paper, P3109R0: 'A plan for `std::execution` for C++26' [P3109], adopted by the standard committee, which details work we aim to include in the C++ standard and which is not part of P2300. This paper mentions three important facilities that would have a significant impact on end-users:

- system execution context;
- async scope;
- coroutine task type.

The current senders/receiver proposal, as merged into the standard, doesn't define any scheduler, so users may need to write their own schedulers to describe concurrent work. Previous versions of senders/receivers defined a thread pool scheduler, but this was later removed due to numerous issues. The system execution context proposal [P2079R5] introduces a scheduler type that makes use of the system's execution context. On Windows, it should use the Windows Thread Pool [Microsoft] to schedule work, and on macOS, it should use Grand Central Dispatch [GCD]. Aiming to reduce CPU oversubscription [Wikipedia-2], the system scheduler is a good default for spawning CPU-intensive work. We've already seen an example of this in Listing 3.

Until recently, the P2300 proposal, which introduced senders/receivers, included two algorithms called `start_detached()` and `ensure_started()` that would submit the work for a sender eagerly, without a way to join the work. These two algorithms would allow the user to implement unstructured concurrency, as the work spawned by these two algorithms outlives the work that spawned them. (Currently, the only way to submit work is through `sync_wait()`, which is fully structured.) While unstructured concurrency can lead to various issues, it is often useful to have a way to spawn large work from a narrow scope.

The async scope proposal [P3149R6] allows the user to have a weakly-structured way of launching work. It defines an async scope in which we can dynamically launch work that outlives the scope from which it was spawned. The key point is that all work spawned within this async scope must be joined before the scope is destroyed. This means that we allow some unstructuredness, but we contain it within a defined scope.

In addition to enabling some unstructuredness, async scope is also useful for launching a dynamic number of work items and then joining that work within a fully structured context.

The third major feature is a coroutine task type. This would essentially mean writing an `std::execution::task<T>` coroutine that can seamlessly interoperate with senders. Using this, one can `co_await` a sender or consider such a coroutine to be a sender. Thus, this task type can freely interoperate with a sender. This would allow users to write coroutines to handle concurrency and asynchrony instead of using compositions of sender algorithms to build them. While there may be some performance penalties involved with using such a task type, users may prefer it for certain types of programs, as the code is more readable.

Other senders/receivers features that would be highly desirable in C++ but were not part of P3109 include:

- C++ parallel algorithms (synchronous) (P2500)
- C++ asynchronous parallel algorithms (P3300)
- I/O and time-based schedulers
- networking on top of senders/receivers

Conclusions

Senders/Receivers is a new C++ feature that provides a model for expressing computations, supporting concurrency, parallelism, and asynchrony. It allows for structured concurrency, making it easier to reason about concurrent code and avoid common pitfalls. Senders/Receivers has already been voted into C++ and is expected to land in C++ 26.

This article provides an introduction to the subject of senders/receivers so that people can start using it as soon as it's available. Although this feature is used for concurrency, we presented it organically, starting with building computations and touching on the concurrency aspects without needing to explain too much about threading and execution contexts. This is one of the beauties of the model: it abstracts away concurrency concerns without compromising performance or safety.

We've spent a fair amount of time explaining the idea behind senders so that readers can easily grasp the key aspects of the proposal and start writing programs using senders/receivers.

The article didn't go into detail on how to use senders/receivers to implement complex problems. Some of these examples can be found on the Internet, in various talks and examples. And perhaps that's a good topic for a follow-up article. ■

References

[GCD] Apple, Grand Central Dispatch, 2016, https://swiftlang.github.io/swift-corelibs-libdispatch/.

[Microsoft] Microsoft, 'Thread Pools', 2021, https://learn.microsoft.com/en-us/windows/win32/procthread/thread-pools.

[P2300R10] Michał Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, Bryce Adelstein Lelbach, P2300R10: '`std::execution`', 2024, https://wg21.link/P2300R10.

[P2079R5] Lucian Radu Teodorescu, Ruslan Arutyunyan, Lee Howes, Michael Voss, P2079R5: 'System execution context', 2024, https://wg21.link/P2079R5.

[P3109R0] Lewis Baker, Eric Niebler, Kirk Shoop, Lucian Radu Teodorescu, P3109R0: 'A plan for `std::execution` for C++26', 2024, https://wg21.link/P3109R0.

[P3149R6] Ian Petersen, Jessica Wong, Ján Ondrušek, Kirk Shoop, Lee Howes, Lucian Radu Teodorescu, P3149R6: 'async_scope – Creating scopes for non-sequential concurrency', https://wg21.link/P3149R6.

[stdexec] NVIDIA, 'Senders – A Standard Model for Asynchronous Execution in C++', https://github.com/NVIDIA/stdexec.

[Sutter24] Herb Sutter, Trip report: Summer ISO C++ standards meeting (St Louis, MO, USA), 2024, https://herbsutter.com/2024/07/02/trip-report-summer-iso-c-standards-meeting-st-louis-mo-usa/.

[Teodorescu22] Lucian Radu Teodorescu, Structured Concurrency, ACCU Conference, 2022, https://www.youtube.com/watch?v=Xq2IMOPjPs0.

[WG21] WG21, 'Execution control library' in *Working Draft Programming Languages – C++* https://eel.is/c++draft/#exec.

[Wikipedia-1] Wikipedia, Basic Linear Algebra Subprograms, https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms#Level_1.

[Wikipedia-2] Wikipedia, Resource contention, https://en.wikipedia.org/wiki/Resource_contention.

Replacing 'bool' Values

Booleans seem simple to use. Spencer Collyer considers when they can actually cause a world of pain.

When used in the context of programming, the term *Dimensional Analysis* refers to the technique of defining types to represent the kinds of values used in the program. With the appropriate operations between objects of those types defined the compiler can check the expressions in the code to make sure they are valid. This is not generally possible if you rely on using the fundamental types like **`int`** or **`double`**.

For instance, say you have a program that deals with distances, durations, and speeds. It should be obvious that adding or subtracting a distance and a speed are invalid operations, but the compiler would not be able to tell you that this code is incorrect:

```
double distance = 10;
double speed = 2;
double duration = distance - speed;
```

However, if you have types **`Distance`**, **`Duration`**, and **`Speed`**, with only the valid operations between them defined, the compiler can issue an error for this code:

```
Distance distance = 10;
Speed speed = 2;
Duration duration = distance - speed;
```

To be useable, when using this technique most types need to be defined as classes or structures. There are libraries available for many languages that make this task easier – a 2018 survey of them for many languages can be found in [Preussner].

However, if you would normally think of using a **`bool`** variable to hold the value, there are several mechanisms available in the C++ language that can be used instead, with no need for library support. We will outline some of them in this article, as well as try to explain why you might choose to do so.

When reading the problem descriptions and suggested solutions below, and wondering if you want to use them, it is worth applying what I call the TLAMP principle. Pronounced 'tee lamp', it stands for Think Like A Maintenance Programmer. What may seem obvious to you when first writing a piece of code can look completely opaque to someone doing maintenance work on that code in the future. They want the code to be as clear as possible on first reading. That later programmer could be yourself in six months – when you haven't looked at the code for that length of time what seemed obvious when you were writing it may not be so later.

Why bother when bools are so simple?

You might ask why we would bother replacing a bool value with some other mechanism when **`bool`**s are so simple to use. In this section, we will outline some of the problems with using bools that make it worthwhile to at least consider doing so.

Many of these problems arise because programmers decide to use **`bool`** variables or parameters just because the value being represented can only take two values. If you get into the habit of only using **`bool`** for values that are going to be used in boolean expressions, you can avoid them to a large extent.

To illustrate some of the problems we will use the following example[1].

Imagine a water company wants a system written to monitor and control its water network. There is a large amount of equipment on the network, such as sensors for measuring things like flow rate, temperature, chemical concentrations, and also control equipment such as valves and pumps to allow the flows in the network to be controlled. This network has evolved over many years, and the equipment is from different manufacturers and of different ages, with a variety of protocols used to talk to it.

The initial analysis leads to a design in which the connections to this equipment are handled by a **`Connection`** base class which provides a standard interface, with a set of classes derived from **`Connection`** that handle the details of each protocol. There is a factory function, called **`CreateConnection`**, which returns an object of the correct class for each connection. Each class is designed to handle either input or output on the connection. The initial design for the **`CreateConnection`** function interface looks like the following:

```
ConnectionPtr CreateConnection(
  std::string_view id
, bool is_output);
```

The **`is_output`** parameter determines whether an output (**`true`**) or input (**`false`**) connection is being created.

An additional requirement is for some users to have elevated permissions on some connections. This allows for operations like controlling pump speeds to alter flow rates, for instance. To handle this, a second **`bool`** parameter is added to indicate if the user is privileged or not.

During testing of the system, it is found that some parts of the network are so old that they only support 7-bit data. As a result, communications over these connections have to be encoded from binary to ASCII. To indicate this a further **`bool`** parameter is added to the function, called **`is_encoded`**, to indicate if this encoding is required or not.

Finally, a security review of the system raises concerns that some of the connections go over public networks, and a requirement is made that those connections need to be encrypted. A final parameter is added to the function called **`is_encrypted`** which indicates if the connection needs to be encrypted or not.

1 This example may seem contrived, but I once worked on a system that had many functions with three or four bool parameters. A lot of the calls were done using literal values for some or all of the parameters, and only checking the surrounding code could confirm whether the values were correct.

Spencer Collyer Spencer has been programming for more years than he cares to remember, mostly in the financial sector, although in his younger years he worked on projects as diverse as monitoring water treatment works on the one hand, and television programme scheduling on the other.

Unless a programmer knows the function prototype off by heart, it would be easy for them to get the parameter order wrong, and the compiler won't warn about it

```
ConnectionPtr CreateConnection(
std::string_view id
, bool is_output
, bool is_authorised
, bool is_encoded
, bool is_encrypted);
```

Listing 1

The final prototype for the function now looks like Listing 1.

The meaning of true

Or rather, the meaning of `true`. And, indeed, `false`. In many cases where a variable can take just two values, and so at first looks like a good candidate to use a `bool`, it is not obvious which value should map to `true` and which to `false`.

The `is_output` parameter in the `CreateConnection` function is a perfect example of this. The parameter allows the caller of the function to determine if an outgoing or incoming connection is required, but other than the name of the parameter there is nothing that indicates which of those is selected by passing `true` and which by passing `false`.

You could argue that the name of the parameter shows how it is used, but that relies on anyone reading the code either knowing the prototype because they have seen if before, or else are willing to look it up. Neither of which is guaranteed to be done by a maintenance programmer who is under pressure to get a fix out quickly.

All bools look the same

In many cases, the `bool` values do match what we would expect for a given parameter, but they can still be problematic, especially if you have more than one `bool` in the parameter list. This is because all `bool`s look the same to the compiler.

The `CreateConnection` function illustrates this problem. If we ignore the problem with it outlined above, it is reasonable that the `is_output` parameter is the first `bool` in the list, as the direction of the connection is the most important property it has.

Good arguments could be made for any order of the other three `bool` parameters however – the one chosen here has arisen simply because of the order the requirement for them came up in the development process. For instance, it could be argued that the `is_encoded` and `is_encrypted` parameters are the wrong way around for an outbound connection, as encryption occurs before encoding when sending a message.

Unless a programmer knows the function prototype off by heart, it would be easy for them to get the parameter order wrong, and the compiler won't warn about it. Only extensive testing will ensure all calls are correct.

What can be even more confusing for someone reading the code later is if it uses named variables for the parameters, but gets them in the wrong order. For instance, consider the code in Listing 2.

```
bool is_encoded =
     /* code that sets value to true */;
bool is_encrypted =
     /* code that sets value to true */;
...
auto connptr = CreateConnection(id, is_output,
  is_authorised, is_encrypted, is_encoded)
```

Listing 2

This will work, in the sense of giving the expected result, because the `is_encoded` and `is_encrypted` variables have the same value. However, if one of those values needs to change later, or someone copies the code elsewhere and changed one of the values, the result would be incorrect, but it wouldn't be obvious why unless the person reading the code recognises that the last two parameters are in the wrong order.

The compiler cannot report this problem because it just sees the types of parameters passed in. The names of the variables are relevant only to tell it where to read the parameter value from – it doesn't check that they match the names in the function prototype.

Note: This problem doesn't just apply to the `bool` type, of course – lists of parameters all with the same type can be problematic when trying to work out what each parameter means. This article doesn't deal with that situation but it is worth being aware of it.

Conversions to and from bool

The built-in C++ scalar types all implicitly convert to and from the `bool` type. This implicit conversion is useful when writing code that tests that a value is not zero or a null pointer.

Some classes in the standard library also provide an `operator bool` to test that an object is in a valid state – for instance, the `std::basic_ios` class that is the base of many iostreams classes class provides one to check if an error has occurred on the stream.

Another use for this implicit conversion is in the `!!` pseudo-operator, which can be used to return the `bool` equivalent of an expression[2] in any cases where automatic conversion doesn't happen.

However, this implicit conversion can cause problems if it happens when you are not expecting it. For instance when calling a function, if you pass a scalar value in a parameter that expects a `bool`, it will be converted.

Consider the code in Listing 3. The two `PrintArgs` functions simply output their prototype and the values they have been called with. The second one allows the `bool` parameter to be defaulted, hence why the `short` is placed before it in the parameter list.

2 I have seen this pseudo-operator referred to as the 'normalise operator'. The way it works is by relying on the right-to-left binding of the `!` operator. The right-hand `!` applies to the operand, forcing it to the `bool` equivalent and then negating the result. The left-hand `!` then applies to the resulting value and negates it again, giving us back the `bool` equivalent of the original operand.

the overload resolution process is done, and we still have two candidates with no way to pick between them, and hence the call is ambiguous

```
#include <iostream>
#include <string_view>

void PrintArgs(const std::string& s,
  bool to_uc = false)
{
  std::cout
    << "Called PrintArgs(string, bool) with ("
    << s << ", " << to_uc << ")\n";
}
void PrintArgs(const std::string& s, short len,
  bool to_uc = false)
{
  std::cout << "Called PrintArgs(string, short,
    bool) with (" << s << ", " << len << ",
    " << to_uc << ")\n";
}

int main()
{
  std::cout << std::boolalpha;
  PrintArgs("Abc"); // 1
  PrintArgs("Abc", true); // 2
  PrintArgs("Abc", 2); // 3
  PrintArgs("Abc", 2, true); // 4
}
```

Listing 3

Unfortunately, when this program is compiled, the line labelled `// 3` fails to compile. The output in Listing 4 shows the errors when the code is compiled with the GCC on my Linux system.

The problem arises during the overload resolution process to decide which function should be called. The full details of overload resolution are complex (see [CppRef1]) but the case here is relatively simple. An important point is that an integer with no suffix in the code has type **`int`** so the 2 in the problematic call has type **`int`**.

When the compiler sees the call in the line labelled `// 3`, it first finds all the declared functions named PrintArgs and adds them to the overload set. It then checks each one to see if it matches the arguments given. This proceeds as follows:

- For the two-parameter function, the **`"Abc"`** can be converted to a **`std::string`**, so the first argument matches the first parameter. The 2 is an **`int`**, and it can be implicitly converted to the **`bool`** type of the second parameter. Both arguments match the function parameters, so the function is a candidate.
- For the three-parameter function, the **`"Abc"`** is a match as above. The 2 is an **`int`**, and that can be implicitly converted to a **`short`** using a narrowing conversion. The third argument is missing but the parameter has a default value, so it is ignored in the matching. The arguments match the parameter list for this function, so it is also a candidate.

```
#include <iostream>
#include <string_view>

void PrintArgs(const std::string& s,
  bool to_uc = false)
{
  std::cout
    << "Called PrintArgs(string, bool) with
    (" << s << ", " << to_uc << ")\n";
}
void PrintArgs(const std::string& s, bool to_uc,
  short len)
{
  std::cout << "Called PrintArgs(string, bool,
    short) with (" << s << ", " << to_uc << ",
    " << len << ")\n";
}
int main()
{
  std::cout << std::boolalpha;
  PrintArgs("Abc"); // 1
  PrintArgs("Abc", true); // 2
  PrintArgs("Abc", 2); // 3
  PrintArgs("Abc", 2, true); // 4
}
```

Listing 5

At this point, the overload resolution process is done, and we still have two candidates with no way to pick between them, and hence the call is ambiguous.

To solve the ambiguity the programmer changes the second definition so it looks like the one in Listing 5. Unfortunately, the default value for the **`bool`** parameter can no longer be used, but the ambiguity no longer occurs.

```
conversion-1.cpp: In function 'int main()':
conversion-1.cpp:19:23: error: call of overloaded 'PrintArgs(const char [4], int)' is ambiguous
   19 |     PrintArgs("Abc", 2);       // 3
      |                       ^
conversion-1.cpp:4:6: note: candidate: 'void PrintArgs(const string&, bool)'
    4 | void PrintArgs(const std::string& s, bool to_uc = false)
      |      ^~~~~~~~~
conversion-1.cpp:9:6: note: candidate: 'void PrintArgs(const string&, short int, bool)'
    9 | void PrintArgs(const std::string& s, short len, bool to_uc = false)
      |      ^~~~~~~~~
```

Listing 4

each additional parameter replaced doubles the number of new functions required

```
Called PrintArgs(string, bool) with (Abc, false)
Called PrintArgs(string, bool) with (Abc, true)
Called PrintArgs(string, bool) with (Abc, true)
Called PrintArgs(string, bool, short) with (Abc,
  true, 1)
```

Listing 6

The program now compiles without any problems and appears to run fine as well, producing the output in Listing 6. However, looking closely at the output shows that the output from the lines labelled `// 3` and `// 4` do not match the arguments in the code. This is again because of implicit conversions.

In the case of the call in line `// 3`, the 2 is converted from **`int`** to **`bool`**, ending up with the value **`true`**.

In the case of the call in line `// 4`, the 2 in the second argument is again converted from **`int`** to the **`bool`** value **`true`**, and the **`true`** in the third argument is converted from **`bool`** to **`short`**, ending up with the value 1.

This kind of bug can arise if you change the interface of a function and rely on the compiler to catch any calls with incorrect arguments. As can be seen in this example, it does not always issue warnings or errors for calls that you should have changed. A refactoring tool may be able to find them, or you might simply have to check each call by hand.

This kind of problem with implicit conversions can arise in other cases, but the one going to or from a **`bool`** is more insidious because the values of a **`bool`** are fundamentally different from the values of a scalar type, in that they are logical truth values, not numbers. The fact that the C++ spec dictates that **`false`** maps to a value of 0 and **`true`** maps to a value of 1 when converted to a number is just a convention to allow the conversion to occur. Other languages don't allow such conversion, or if they do they use different mappings[3].

It may not matter to you if an **`int`** gets converted to a **`short`** as long as the value doesn't change, but with a **`bool`** you are going from a logical value to a number or from a number to a logical value, which is a more fundamental change, and one that may well make no sense in the context of the code.

More than two values

It might sound trite to say it, but a **`bool`** value can only hold two different values. This may become a problem if you realise that a parameter needs to hold more than two values.

For instance, in our water company example, the binary-to-ASCII encoding on some connections might need doing using UUencoding [Wikipedia-1], while others might use Base64 [Wikipedia-2].

With just two values for **`is_encoded`** and one of those used to indicate no encoding is required, you cannot represent those two different types of encoding in the parameter. You have two options in this case – either add another parameter to give the encoding or else convert the **`bool`** parameter to some other kind that can represent three (or more) values. The first extends the function interface even more, and the second has all the possible problems associated with conversion to/from **`bool`** given above.

Alternatives to bool

We have seen why you might want to avoid using **`bool`** variables and parameters, now we will show some methods that you can use to do so. As mentioned previously, all of these are available from the core language, with no library support required.

Some of these methods are designed primarily for replacing function parameters, while the others are more general and can be used to replace variables as well.

Split one function into two (or more)

Rather than having a single function with different functionality selected by passing a **`bool`** parameter, split the functionality into two different functions, with their names indicating what is being done. Any common functionality can be split off into a third function that the two new functions call.

This is particularly useful for the case where it is not obvious what the mapping from the **`true`** or **`false`** values to the selected functionality is.

In our water company example, rather than passing the **`is_output`** parameter, you would instead create functions called CreateOutboundConnection and **`CreateInboundConnection`**, where the names indicate what type of connection is being created.

This method is fine for replacing one or maybe two parameters. The problem with doing more than that is that each additional parameter replaced doubles the number of new functions required. Also, with descriptive function names, they can get unmanageably long very quickly.

Using a flags variable

This method involves replacing one or more **`bool`** values with a variable holding a collection of single-bit fields. This will generally be an integer value or a **`std::bitset`**.

An example of a flags variable in the standard library is the mode parameter of the **`std::ifstream`** and **`std::ofstream`** constructors, which uses the **`std::ios_base::openmode`** type.

When using this method with an integer, you would normally define a set of constants, one for each flag value. The value of each constant has its particular flag bit set to 1, all other bits set to 0, so the constant represents the flag being turned on. You then use normal binary operations to turn on the flags and to test if they are turned on or not.

3 Anyone old enough to have used one of the microcomputers released during the 1980s home computer boom might remember that the BASIC built into many of them used -1 for the 'true' value, presumably because the representation of that value has all bits set to 1. Sinclair Basic, as used on the ZX81 and Spectrum, went its own way and used 1 for the 'true' value.

You can do the same when using a `std::bitset`, but you also have the option of accessing individual bits using the `[]` operator or the `test()` function, which take the position of the bit in the bitset to check and return true if it is set to 1, else false.

One advantage of using a flag variable is that the user just has to turn on the flags they want, and all the others default to off. On the other hand, it is awkward to explicitly say that a flag is turned off, should you wish to do so.

If you find a flag needs more than two values, you just need to increase the size of the field and adjust the constants appropriately. If you are using a bitset, the direct bit access through `[]` or `test()` could not be used in this case.

A useful trick in case this might happen is to not make bitfields adjacent to each other when they are first defined. For instance with four flags in a four byte integer, set the fields up as the lowest bit in each byte. That way if you do need to increase the number of values represented by a flag, you won't have to change any of the constants that don't relate to that flag.

Using a flags structure

This method uses a structure to hold the flags. The structure members can be either `bool`s or single-bit bitfields.

If using this method, you can directly set the individual fields to turn the flag on or off. For the bitfields version you would usually use `0` for off and `1` for on.

If using the bitfield version you need to define them as unsigned, as they are just one bit wide. If they are defined as signed then setting the value to 1 will end up with it being treated as -1. Listing 7 illustrates this. Checking the output, you can see that structure with `int` fields outputs `-1` for each one, while the structure with `unsigned int` values outputs `1` for them:

```
-1 -1 -1
1 1 1
```

If you don't want to create a variable of the structure type to pass to a function you can use an initializer-list as the parameter and the structure will be created for you. Listing 8 shows examples of both types.

The advantage of setting up a variable before passing it to the function is that someone reading the code later can see exactly which flags are being set, whereas when using an initializer list they have to know what the structure looks like to know which flags are being set.

When using the bitfield version, if you need to extend a field to hold more than two fields you can just extend its width. For the `bool` version, you can just replace the `bool` with a different type.

```
#include <iostream>

struct A
{
  int a1 : 1;
  int a2 : 1;
  int a3 : 1;
};
struct B
{
  unsigned int b1 : 1;
  unsigned int b2 : 1;
  unsigned int b3 : 1;
};
int main()
{
  A a; a.a1 = 1; a.a2 = 1; a.a3 = 1;
  std::cout << a.a1 << " " << a.a2 << " "
            << a.a3 << "\n";
  B b; b.b1 = 1; b.b2 = 1; b.b3 = 1;
  std::cout << b.b1 << " " << b.b2 << " "
            << b.b3 << "\n";
}
```

Listing 7

```
#include <iostream>

struct BitFlags
{
  unsigned int flag1 : 1;
  unsigned int flag2 : 1;
  unsigned int flag3 : 1;
};
struct BoolFlags
{
  bool flag1;
  bool flag2;
  bool flag3;
};
void fbit(BitFlags flags)
{
  std::cout << flags.flag1 << " " << flags.flag2
            << " " << flags.flag3 << "\n";
}
void fbool(BoolFlags flags)
{
  std::cout << std::boolalpha << flags.flag1
            << " " << flags.flag2 << " "
            << flags.flag3 << "\n";
}
int main()
{
  BitFlags bitflags;
  bitflags.flag1 = 0;
  bitflags.flag2 = 1;
  bitflags.flag3 = 0;
  fbit(bitflags);
  fbit({1, 0, 1});

  BoolFlags boolflags;
  boolflags.flag1 = false;
  boolflags.flag2 = true;
  boolflags.flag3 = false;
  fbool(boolflags);
  fbool({true, false, true});
}
```

Listing 8

Using enums

This method simply uses enums with two enumerators defined. Using appropriate names means the values can be self-documenting. Either scoped or unscoped enums can be used.

Unscoped enums have the disadvantage that the enumerators are defined in the scope enclosing the enum, so you cannot have the same enumerator name in two enums that will be used at the same time. On the other hand, it does mean that the enumerators can be used with no qualification.

For scoped enums the enumerators are defined in the scope of the enum, so two enums can have enumerators with the same name if that makes sense. This does mean that they have to be qualified with the enum name when used.

If an unscoped enum is passed as a function parameter that expects an integer, the value in the enum variable will be converted to an integer. This does not happen for a scoped enum – no conversion takes place.

Listing 9 (overleaf) is the scoped enum equivalent of Listing 3. This version compiles with no ambiguous function calls detected, and if you run the resulting program you will see that the `PrintArgs` functions called in each case are the correct ones. The output for the program is shown in Listing 10.

C++20 and using enum

The point was made above that when using scoped enums you need need to precede the enumeration name with the scoped enum name. This has been addressed in C++20 with the addition of the `using enum` construct to pull all the names in the named enum into the current scope.

A brief description of this facility can be found at [CppRef2] – look for *Using-enum-declaration*. The facility was added by P1099r5 [P1099r5], and a fuller description of it can be found by reading that (brief) paper.

```
#include <iostream>
#include <string_view>

enum class RedBlue { Red, Blue };
std::ostream& operator<<(std::ostream& ostr,
  const RedBlue conv)
{
  ostr
     << (conv == RedBlue::Red ? "Red" : "Blue");
    return ostr;
}
void PrintArgs(const std::string& s,
  RedBlue to_uc = RedBlue::Red)
{
  std::cout << "Called PrintArgs(string, RedBlue)
  with (" << s << ", " << to_uc << ")\n";
}
void PrintArgs(const std::string& s, short len,
  RedBlue to_uc = RedBlue::Red)
{
  std::cout
  << "Called PrintArgs(string, short, RedBlue)
  with (" << s << ", " << len << ",
  " << to_uc << ")\n";
}
int main()
{
  std::cout << std::boolalpha;
  PrintArgs("Abc"); // 1
  PrintArgs("Abc", RedBlue::Blue); // 2
  PrintArgs("Abc", 2); // 3
  PrintArgs("Abc", 2, RedBlue::Blue); // 4
}
```

Listing 9

As of the time of writing (April 2021), the C++20 language features pages for GCC (at version 11) and MSVC (at VS 2019 16.4) show this feature as being implemented. The equivalent Clang page shows this feature has not yet implemented.

Problems versus suggested alternatives

In this section, we will check if the suggested alternatives solve any of the problems outlined.

The meaning of true

Splitting into two functions works, as long as you use sensible names for the new functions.

Using a flags variable mostly works, as long as you use sensible names for the constants representing the flags. As noted in the description it is not as simple to explicitly indicate the flag is turned off.

Using a flags structure works as long as the structure members have sensible names. Unlike the case above, it is also simple to set the correct member to indicate the flag is turned off.

Using enums works as long as the enumerators have sensible names.

All bools look alike

Splitting into two functions can work if you only have two **`bool`** parameters, but any more than that and it becomes impractical.

Using a flags variable or a flags structure works as we no longer have multiple variables.

Using enums works because all enums are distinct from each other.

Conversions to and from bool

Splitting into two functions works for the parameter that has been removed, although any remaining **`bool`** parameters being passed could still suffer from conversion.

Using a flags variable held in an integer can undergo all the normal integer conversions, so it does not solve this problem.

```
Called PrintArgs(string, RedBlue) with (Abc, Red)
Called PrintArgs(string, RedBlue) with
  (Abc, Blue)
Called PrintArgs(string, short, RedBlue) with
  (Abc, 2, Red)
Called PrintArgs(string, short, RedBlue) with
  (Abc, 2, Blue)
```

Listing 10

Using a flags variable held in a **`std::bitset`** is better because you cannot assign an integer to a **`bitset`** or vice versa. Note however that you can initialize a **`bitset`** with an integer, so passing an integer to a function when it expects a **`bitset`** will use the integer to initialize the **`bitset`**.

Using a flags structure works as structs do not implicitly convert to anything else.

Using unscoped enums partially solves the conversion problem. An integer or floating-point type cannot be converted to the enum type implicitly[4]. On the other hand, values of the enum type are implicitly convertible to integral types.

Using scoped enums solves the implicit conversion problem completely [5].

More than two values

Splitting into two functions could solve this problem as you just need to add a function for each new value. If your functions are handling two conditions then you'll need a new function for each possible new combination, so it may be worth redesigning at this point to stop the number of functions from exploding.

Using a flags variable works as you can just increase the number of bits each flag uses to represent its value. You do have to be careful that the constants for different flags don't overlap each other.

Using a flags structure works by allowing you to easily determine the size of each member of the structure. Unlike for the flags variable above you do not need to keep fields separated manually.

Using enum types works as you just need to add new enumerators for the new values. If using unscoped enums you have to be careful not to create any name clashes with enumerators belonging to other unscoped enum types.

Potential disadvantages with suggested alternatives

This section will discuss some potential disadvantages with the suggested alternatives, and hopefully show that they are either not a problem or else the pros outweigh the cons.

More verbose code

All of the alternatives suggested make the code more verbose. For most of them this is simply a case of replacing code like

```
if (x) { ... }
```

with an explicit test like

```
if (x == value) { ... }
```

It could be argued that making the test explicit does make the code more self-documenting, so should not be seen as a disadvantage.

The alternative using constants to define flag bits, either in an integer or a **`std::bitset`**, does have code that looks more complicated, as you have to use a binary 'and' to isolate the flag bit and test if it is set, like

```
if ((x & flagbit) == flagbit)
```

4 Although you can use an explicit cast, such as a **`static_cast`**, to convert integer, floating-point, or enumeration values to an enum type, whether unscoped or scoped.

5 Scoped enum values can be converted to integer values using a **`static_cast`**, though.

or if you are happy to rely on the implicit conversion to **bool** you can use

```
if (x & flagbit)
```

instead. Neither is as clear as the simple test against a value. On the other hand, with the **std::bitset** you can use the **[]** operator or **test** function to check a **bit** at a position.

Namespace pollution

All of the suggested alternatives insert new entities into the current namespace, whether functions, constants, or types. All of those entities introduce new names into the current namespace which wouldn't need to exist if you just used **bool** values. This will cause problems if they clash with any names already in that namespace.

Of course, this isn't specific to this case – it occurs whenever you add new entities to a scope, so do whatever you normally would to get around it.

An easy solution is to add the new entities in their own namespace. This does mean that the names need the namespace as an extra qualifier, but you can use a **using** declaration to bring the name into the current namespace. If the new entities are only used in a single `\*.cpp` file you can put them in an anonymous namespace in that file and you won't even need the extra qualifier.

Size and speed of compiled programs

A common concern when using the alternatives is that the code will be larger and/or slower than when using **bool**s. This should not be a concern as modern compilers are intelligent enough to recognise what the code is doing and optimizing it appropriately.

Sample code to show this can be found on [BitBucket], [GitHub], or [GitLab], depending on your preferred supplier. The various `\*.cpp` files each demonstrate one alternative, except the `bools.cpp` one which shows the original form with **bool** variables.

The `find-medians.sh` shell script in that directory runs all the programs and captures the runtimes, then works out the median and mode runtimes for each one. Running this script on my main machine gives the runtimes shown in Table 1 for code optimized with **-O3**.

As can be seen, the runtimes for the optimized programs are virtually identical for all the programs. This shows that you don't lose much if any speed when using the alternatives.

As far as code size is concerned, for the optimized code the program sizes range from 17160 bytes for `functions.opt` to 17320 for `bitsetconsts.opt`. The `bools.cpp` file is 17272 bytes. So there is little difference in code size either.

Test Case	Median	Mode
Bools	387	387
Bitset with constants	387	388
Bitset with indexing	387	388
Scoped enum	387	387
Unscoped enum	387	387
Functions	382	382
Integer flags	387	387
Struct with bitfields	387	386
Struct with bools	387	387

Table 1

So no more bools then?

It might seem that this article is saying that you shouldn't use **bool** values in your programs at all. This is not the intention.

One target is the use of **bool**s in what might be termed long-range code. What do we mean by long-range code?

Calling a function is long-range, as you are leaving the current function's scope and entering the called one. You should think carefully before using **bool**s as parameters of functions. As this article has tried to show, there are alternatives which can be both safer and clearer, with little or no loss of program speed.

Code in a single function could also be considered long-range if the whole usage cannot be seen on a single screenful of code[6]. Using a bool to store the result of a logical operation which is used in the immediately following code is fine, as it's obvious what is going on. Even if the value is only used once, if it simplifies a condition expression it can still be valid to do so.

Another target is the use of bool for class member variables. This is an ideal case for using one of the alternatives, especially enums. Classes provide their own scope, so the potential for namespace pollution is immediately reduced. And if the member variable is private (as they should normally be), all the code using it will be written by the class maintainer, so the users of the class won't have to handle it at all.

So in summary, if the use of the bool would be obvious from the immediate context of the code, it is fine to use it. In all other cases, consider using an alternative. This article provides several such alternatives as a starting point. ■

References

[BitBucket] https://bitbucket.org/dustycorner/articles/src/master/replacing-bool-values/testcode

[CppRef1] https://en.cppreference.com/w/cpp/language/overload_resolution

[CppRef2] https://en.cppreference.com/w/cpp/language/enum

[GitHub] https://github.com/dustycorner/articles/tree/master/replacing-bool-values/testcode

[GitLab] https://gitlab.com/dustycorner/articles/-/tree/master/replacingbool-values/testcode

[P1099r5] Gašper Ažman and Jonathan Müller, 'Using Enum', http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1099r5.html

[Preussner] 'Dimensional Analysis in Programming Languages', https://gmpreussner.com/research/dimensional-analysis-in-programming-languages

[Wikipedia-1] UUencode: https://en.wikipedia.org/wiki/Uuencoding

[Wikipedia-2] Base64: https://en.wikipedia.org/wiki/Base64

This article was first published in *Overload* 163 in June 2021.

6 And by a single screenful of code I don't mean using huge monitors and small fonts to get 150+ lines of code on a screen at a time. Think more like 40 to 50 lines maximum, so a quick scan up and down is easy to do.

Afterwood

Bar. Hmmmm. Bug?! Chris Oldwood gives software development a seasonal twist.

Quality was dead: to begin with.

Okay, that's not the real opening to Charles Dicken's popular yuletide novella *A Christmas Carol*, set in 19th century England long before the appearance of software development as an industry, but Marley's warning can easily be seen as an allegory for technical debt. Despite what George Box once said about all metaphors being wrong, but some being useful – see what I did there – I reckon we can look to Marley for inspiration about how we should treat our code, and the ramifications of not giving it enough TLC. Marley confesses that he now wears the chain he forged in life, made link-by-link, and yard-by-yard of his own free will. He could equally be talking about not bothering to refactor, with every link being a missed opportunity to rename a variable or function, extract logic to a separate method, write an automated test, etc.

Back in 2008, Thom Holwerda proposed that the only real measurement of code quality was WTFs per minute. The fallout from Marley's decision to continually cut corners emerges verbally, later, as his successor tries to make sense of that code. This being a family friendly publication though I can't spell out WTF in full and propose instead that just for the festive season we switch to the far more old-fashioned form of WTDs (What the Dickens!) per minute.

I should note though that the 'Dickens' in that expression of surprise is entirely unrelated to the author in question, having been used by Shakespeare a few hundred years before Charles was even born. The etymology suggests it's a euphemism for the Devil, also known as Old Nick – not to be confused with the more lovable Saint Nick, who also enters our consciousness this time of year. It's an easy mistake to make, especially when you consider that Satan and Santa are anagrams of each other. Marley was also trying to tell us that naming is hard, and typos can lead to a lot of confusion if left unchecked. (I once ran across a variable named 'NoErrors' where 'no' was actually an abbreviation of 'number', in a programming language that allowed an implicit conversion from an integer to boolean – convince me that's not the Devil at work.)

In the Oldwood household, the favoured adaption is *A Muppet Christmas Carol*, with Albert Finney's *Scrooge* coming in a close second, at least for the parents. Jim Henson's decision to cast both Statler and Waldorf as the Marley brothers was genius. Their modus operandi is to sit on the sidelines and make snarky comments about the various goings-on, but never actually make any sensible suggestions on how to genuinely improve the state of affairs. If you've never had to work with a Statler or a Waldorf, then I envy you. Code reviews often feel like an interview with those grumpy old men as I've found it quite rare for people to point out the positive aspects of a code change and only focus on the bits we disagree with. We should all strive to 'be more Kermit'.

Despite being one of the more faithful adaptions, *A Muppet Christmas Carol* glosses over the same time paradox as many others. In the book, Scrooge is told that he will be visited on three consecutive nights, and yet the tale starts on Christmas Eve but he still wakes up on Christmas Day after the three visits, exclaiming "The Spirits have done it all in one night!" Clearly this is a classic case of management not liking the estimate that Marley proposed. Knowing how poor the codebase had become, Marley estimates three days but somebody upstairs decides Christmas Day is a hard deadline and the ghosts need to work overtime and get redemption delivered in one night. Releasing on Christmas Day is fraught with danger unless you're part of a well-oiled machine, mostly because pretty much everyone else apart from the skeleton support crew will be on holiday.

What of the three ghosts though? Even though our industry is still in its infancy in comparison to many others, we still have plenty to reflect on. Also, the ghosts are with us permanently now in the guise of blog posts, journals, books, videos, talks, etc. We only have to remember to learn from the past to avoid repeating it. How hard can that be?

As I write, the legendary Fred Brooks passed away exactly two years ago to the day. (This also gives you an insight into my inability to meet publishing deadlines and turns the irony level of writing about learning from the past right up to eleven and beyond.) Of his most famous works the 'no silver bullet' statement – about there being no single development in technology or management that can provide even an order of magnitude improvement to productivity within a decade – is probably the one which many would love to prove wrong.

There have definitely been some excellent advances over the years, like the introduction of structured programming and the continued efforts to avoid so many of the traps and pitfalls of our forefathers. Incremental software delivery, the one technique which Brooks conceded in the mid-90s might come close, has also paid dividends and helped us to focus more on the essential complexity. Likewise automated testing and refactoring help us tackle the accidental complexity.

My current fear is the Ghost of Christmas Future showing us a world where we have put all our efforts into AGI in the mistaken belief that writing code is the hard part of software development and we end up repeating the foolish promises of 4GLs and UML. In this picture, Tiny Tim – the sick child of Scrooge's bookkeeper Bob Cratchit – is not just a single codebase or company but the entire software industry as we fail to comprehend what 'describing a solution in unambiguous detail' really means.

Hopefully, this charade will be exposed for the pantomime that it currently is and those working on LLM based tools which provide valuable, direct assistance to Bob Cratchits at every level of their career can get on with improving their tools, undistracted by the Scrooge's of this world who see many programmers as 'the surplus population' that just want to 'pick their pocket every ~~December 25th~~ day'. Seriously, what the actual Dickens!

Blimey, that took a bleak turn, I reckon my text editor must have enabled dark mode. It's Christmas, a time for festive cheer, and we should remember that the book ends on a high note with Scrooge achieving redemption and Dickens revealing that "Tiny Tim, who did not die", providing us with hope for the future. So, in the immortal words of Tiny Tim: "$Deity bless us, every one!"

Chris Oldwood is a freelance programmer who started out as a bedroom coder in the 80s writing assembler on 8-bit micros. These days it's enterprise grade technology from ~~plush corporate offices~~ the comfort of his breakfast bar. He also commentates on the Godmanchester duck race and is easily distracted by emails and DMs to gort@cix.co.uk and @chrisoldwood