overload 194

AUGUST 2026 £4.50

On Contracts and Safety

Lucian Radu Teodorescu explores how C++26 contracts help uncover problems.

C++ Reflection: Verifying Compiler-generated Functions

Lieven de Cock shows how reflection can be used to verify special member functions.

Trip Report: ACCU on Sea 2026

Sándor Dargó shares his ACCU on Sea trip report with us.

Implenting std::visit

Quasar Chunawala explores implementing your own version of std::visit.

Afterwood

Chris Oldwood explains why it's a good idea to continue to 'show your workings' after you leave school.

A magazine of ACCU

ISSN: 1354-3172

accu

Monthly journals, printed and online
Local groups run by ACCU members
Discounted rate for the ACCU Conference
Email discussion lists

accu.org

August 2026
ISSN 1354-3172

Editor
Frances Buontempo
overload@accu.org

Advisors
Paul Bennett
t21@angellane.org
Matthew Dodkins
matthew.dodkins@gmail.com
Paul Floyd
pjfloyd@wanadoo.fr
Jason Hearne-McGuiness
coder@hussar.me.uk
Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi
Steve Love
steve@arventech.com
Christian Meyenburg
contact@meyenburg.dev
Barry Nichols
barrydavidnichols@gmail.com
Chris Oldwood
gort@cix.co.uk
Roger Orr
rogero@howzatt.co.uk
Balog Pal
pasa@lib.hu
Honey Sukesan
honey_speaks_cpp@yahoo.com
Jonathan Wakely
accu@kayari.org
Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries
ads@accu.org

Printing and distribution
Parchment (Oxford) Ltd

Cover design
Original design by Pete Goodliffe
pete@goodliffe.net
Cover photo by Tim Peck: the inside of the Mosta Dome, Malta.

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* 195 should be submitted by 1st September 2026 and those for *Overload* 196 by 1st November 2026.

FOMO: Fear of Missing Out

Sometimes hearing updates about things we've missed makes us regret our choice. Frances Buontempo reminds us we can't do everything.

I would use the *ACCU on Sea* conference as an excuse for not writing an editorial; however, my talk was rejected so I didn't go. Nonetheless, watching various social media posts about the conference drove the point home. For many years now, I have attended the ACCU conference, often as a speaker but sometimes paying as a delegate. The change to a combined conference with *C++ on Sea* was understandable – this saved costs for starters. I would love to have gone, but couldn't justify the expense. Many conferences are having a hard time, partly because companies are less willing to pay for staff training and with dwindling recruitment in some areas, sponsors are harder to come by. Conferences usually offer great value for money, in terms of training, so this is a bad decision, in my opinion. If you, like me, missed out you can often volunteer at a conference to get a free ticket. You can even try to persuade your employer, if you have one, to pay. I am sure the ACCU general mailing list could help you write a convincing request and justification if you need one. Or, you could propose a talk and keep your fingers crossed. I need to up my game next time and write a more convincing proposal. On the plus side, I didn't spend hours writing a talk, so have fewer excuses than usual for a lack of editorial. Sometimes a break is a good thing.

I don't know about you, but I often go through a phase of saying 'Yes' to everything, and often regret it afterwards. I volunteered to be *Overload*'s editor when Ric Parkin, the previous editor, asked for someone to take over. Why? Partly, I thought it would make me actually take time to read all the articles. It has, but that wasn't the most time-efficient way to solve that particular problem! Working with the review team is great, though be assured we do sometimes disagree. I have learnt a lot from doing this, even though I never can find a suitable editorial. As ever, if you want to be guest editor do get in touch.

Perhaps being the editor for an edition is too much. You could write an article instead. Or, become an ACCU member and then you have access to the homework puzzles and similar, along with book reviews (yes! Free books) and more besides. I wrote book reviews for ACCU to dip my toe into writing. I then went on to write an article for *Overload* about floating point numbers, inspired by a discussion I started on the ACCU general mailing list [Buontempo09]. The feedback helped me write a better article than I could have managed on my own. The experience also made me braver. Being proud of something you have created, with help, is a very special feeling.

I'd like to see more people writing for us. The members' magazine, *CVu*, is a great place to start. You need to join ACCU to read the members' magazine: if you're not a member yet join us. A new series called 'In conversation with…' might work well. Get in touch – we can provide a few starter questions so new writers can chat with ACCU members and publish these in *CVu*. Then we can move on to finding more well-known people to chat with and write this up for *Overload*. People's background stories are always interesting. For example, go check out the C++ documentary if you haven't yet. It tells the story of C++, starting with "its origins in the corridors of Bell Labs to the global community it shapes today" [CultRepo].

Saying "Yes" to everything is all very well, but as I said it's just a phase I go through sometimes. If I take a moment to reflect, I often realise I have over-committed. After all, you can't do all the things. I knew I'd possibly gone too far when I said OK to playing a small part in a film. (It's an independent comedy horror film – watch this space.) I now have to learn lines; I am not very good at memorizing things. What have I done? Maybe I was driven by fear of missing out (FOMO): a phrase commonly used. However, it's more that I'd like to be supportive and it might be fun. I'm not afraid to miss out; I'm afraid to join in – but it's good to push yourself once in a while and try something new. FOMO itself was coined about twenty years ago [Wikipedia-1], where it was contrasted with FOBO: fear of a better option. FOMO now has overtones of anxiety, stress and further psychological issues. I'm no expert but sometimes you just need to step back and spend times with friends. Even if that does mean learning lines for a film.

Sándor Dargó has shared a trip report (see page 16) about *ACCU on Sea* with us. He mentions Andrei Alexandrescu's opening keynote, 'The Next 20 Years of Systems Engineering' [Alexandrescu26], which promised to look at history, language design and recent developments, including AI. 'AI' (specifically LLMs and agents) might prove to be a fad. We shall see. I have tinkered with various LLM chat apps, and some 'AI' in an IDE. I know about various ways to set up agents, but I have been watching from a distance. Initially, I had FOMO: there is so much to learn if you want to properly understand what is going on. However, I suspect there are still better options. You can let a machine spew forth code, which it can get right some of the time. However, I recall people blindly copying code from the internet years ago, and needing to bring some discipline to the process, including some unit tests. AI hasn't made code craft go away. I do fear there is a better option than sacking all your devs and letting AI generate all the code. That is not to say you can't get some value from AI, or even code you find on the internet. A tool is a tool; just use it sensibly.

When you spot a new technology, it is worth keeping an eye on it while being mindful that many things prove to be fads in the long run. Fax machines, anyone? Or even a mobile phone: how often do you upgrade yours? OK, hardware is one thing. Software has fads too. When Java became a thing, many people insisted that object-oriented programming was the future. It was, for a while, but people have been rediscovering functional approaches of late. Who can say where we will go next?

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 called *Genetic Algorithms and Machine Learning for Programmers*, and one to help you catch up with C++ called *Learn C++ by Example*. 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.

Some new trends to seem to be here to stay. C++ was new once. As the aforementioned documentary points out, it is now one of the world's most widely used programming languages. It has stood the test of time. Pen and paper is an even older technology that is still in use today. Exactly how old pens or paper are is difficult to say. As I understand it, 'writing' probably started with people scratching marks on various surfaces, including stone and clay tablets, which lead to cuneiform. Cuneiform consists of various wedge shaped signs, which are easy to make on such surfaces. It was in use in various forms from several thousand years BC [Wikipedia-2]. Whether C++ lasts as long is another matter. Some tech that just works has staying power.

What makes something stick, or even take hold in the first place? For starters, usage is vital, as we see with C++. Marketing also makes a difference: the choice of VHS (Video Home System: if you ever see a video tape, it's probably one of these) rather than Betamax for videos was partly driven by marketing [Wikipedia-3]. Cuneiform fell out of fashion as pen and paper were adopted; similarly almost no-one owns a video player anymore. DVDs became popular, and now everyone just streams films and the like, right? Nothing lasts forever, but various technologies do stay in use for a long time. That makes it OK to sit back and see what happens when new technology rears its head, rather than FOMO driving you to spend hours learning something you will never use again. However, the act of learning does give you new ways to think about other, seemingly unrelated problems, so you could argue no learning is a waste of time. I was frequently asked "Miss, what's the point of this?" when I taught maths. Sure, most of the pupils would never need to know about Pascal's triangle or Pythagoras' theorem later in life. However, I maintain that any maths can give you ways to think abstractly, and learning how to communicate those abstract ideas is a transferable skill.

Some innovations don't take hold at the time. Babbage's attempts to build early computers called difference and analytical engines springs to mind. From 1821 he tried to build 'Difference Engine Number 1', but ran out of funding, leaving it part done. He then tried to build a 'general purpose programmable computing engine' [CHM], now called an analytical engine. Probably regarded as a slightly odd waste of time and money back then, we can now see how innovative he was. Babbage did actually start building some of his ideas. In contrast, Leonardo de Vinci left us several sketches for helicopters and other flying machines, like gliders. These were not built at the time, but many mock-up versions have been made since For example, see the model at the Museo Leonardiano Vinci [MLV]. Many creations are ahead of their time. Perhaps you could say the same for electric vehicles? The first cars were electric. Starting with an electric motor in 1827, an electric locomotive that could manage a whopping 4 miles per hour was built in 1838 [Wikipedia-4]. By the 1920s, petrol cars had become the norm. FOBO? Well, gasoline power didn't need a hand crank to start, and mass production made them cheaper. Yet today, the better option is seen to be electric vehicles, at least in terms of environmental impact. History often gives examples of ideas being parked and picked up later on. The reappearance of functional programming is an instance. C++20 introduced coroutines, which also have a long history. To quote Ecclesiastes 1:19:

> What has been will be again, what has been done will be done again; there is nothing new under the sun.

If you feel like you are getting left behind and possibly de-skilled as new ideas, frameworks, even programming languages become popular, don't panic. On a surprisingly large number of occasions, I ended up using and modifying some FORTRAN code during my career. I have nothing against FORTRAN: it's a great language but I don't know it very well. It allows you to vectorise calculations, and has done for a long time. You thought C++26's data-parallel types, or SIMD, was new [CppRef]? There's nothing new under the sun… I digress. Being asked to deal with FORTRAN made my heart sink. Other people were playing with newer tech and I was concerned what my CV would look like. I did miss out of some trending tech, but I will argue that I learnt and used various more timeless tools. If you are spotting new trends and haven't had the chance to try them out, it can go one of two ways. Either take it as an opportunity to learn at future some point when you have time: put together a talk proposal or article and see what happens. Or sit back, relax, get the popcorn and see what transpires. If you miss one thing, you might get a chance to do something that's a better option.

References

[Alexandrescu26] Andrei Alexandrescu 'The Next 20 Weeks of Systems Engineering' delivered at *ACCU on Sea* 2026 and available at https://accuonsea.uk/2026/sessions/the-next-20-weeks-of-systems-engineering/

[Buontempo09] Frances Buontempo 'Floating Point Fun and Frolics' in *Overload* 17(91), June 2009, available at https://accu.org/journals/overload/17/91/buontempo_1558/

[CHM] Computer History Museum 'The Engines', available at https://www.computerhistory.org/babbage/engines/

[CppRef] Data-parallel types (SIMD): https://en.cppreference.com/cpp/numeric/simd

[CultRepo26] CultRepo, released 4 June 2026, 'The Story of C++: The World's Most Consequential Programming Language | The Official Story' available at https://www.youtube.com/watch?v=lI7tMxzSJ7w

[MLV] Museo Leonardiano Vinci 'Flying Machine', available at https://museoleonardiano.it/en/opera/flying-machine/

[Wikipedia-1] Fear of Missing Out: https://en.wikipedia.org/wiki/Fear_of_missing_out

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

[Wikipedia-3] Videotape format war: http://en.wikipedia.org/wiki/Videotape_format_war

[Wikipedia-4] History of the electric vehicle: https://en.wikipedia.org/wiki/History_of_the_electric_vehicle

On Contracts and Safety

Composing safe functions may not be safe. Lucian Radu Teodorescu explores how C++26 contracts help uncover problems.

If two operations are safe, is their composition safe too? At first glance, the answer ought to be yes. But a small expression such as `f(g(10))` can appear to contradict that intuition: both functions may be safe under their own contracts, while the nested call still violates memory safety. The missing piece is the caller. Safety does not compose merely because expressions are nested; it composes when obligations and guarantees line up at each boundary.

Building on the previous article [Teodorescu26], this article looks at how contracts help us build safe programs. Contracts give us a structured way to describe obligations and guarantees, both at software boundaries and inside implementations. In that context, we briefly describe C++26 contracts.

We then look more closely at a common source of confusion around the composability of safety. We also consider enforced safe operations as a practical way to build safe programs with respect to selected safety properties. Finally, we look at documentation contracts as a complementary way to express contracts that do not belong in code. Both forms help build safety into applications.

Contracts

'Design by Contract' is the discipline of making the obligations and guarantees of software components explicit [Meyer91]. A routine may impose preconditions on its callers, promise postconditions on return, and preserve invariants associated with the object or module it belongs to. The main point is not just that these predicates may be checked, but that they make the boundary between client and implementation explicit.

C++26 contracts bring part of this discipline into the language [P2900]. The core idea is that users can add *contract assertions* to programs, providing machine-checkable statements about expected program behaviour. These contract assertions come in three forms: preconditions, postconditions, and general-purpose assertion statements.

Listing 1 provides an example of a small function with preconditions and a postcondition. This example is intentionally small. The preconditions, introduced by `pre`, express what the caller must provide. The postcondition, signalled by `post`, expresses the guarantees provided by the function on success.

Because `pre` and `post` appear on the function declaration, they are visible both to callers and to the function implementation. This makes them good candidates for expressing boundary invariants. The `contract_assert` inside the body checks an internal assumption while implementing that boundary contract. In this sense, `contract_assert` is a better form of `assert`, while `pre` and `post` are the language features that put part of the contract on the function declaration.

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

Unlike with the `assert` macro, the expression given to a contract assertion is always checked syntactically and semantically. We also have better controls for turning contract assertions on and off, and for controlling what happens if the predicate evaluates to `false`.

Whether a contract assertion is checked, ignored, or enforced is controlled by its evaluation semantic. The available semantics are *ignore* (do not evaluate the contract assertion), *observe* (evaluate the assertion, report a violation on failure, and continue), *enforce* (evaluate the assertion, report a violation on failure, and stop further execution), and *quick-enforce* (similar to *enforce*, but allowed to terminate immediately, without attempting to report the violation).

The selected evaluation semantic is implementation-defined and is typically configured through compiler flags.

The application can configure a violation handler that is called when an assertion fails in an *observe* or *enforce* evaluation mode. This can be used to perform custom logging or safely shut down critical modules. These options give contracts enough configurability for many real-world applications.

If the function in Listing 1 is called, for example, with an empty `new_name`, and the evaluation mode is configured to *enforce*, then the application will terminate, reporting a precondition violation. Similarly, in that same mode, if the function returns an empty name, the application will terminate with an error.

For a short introduction to C++26 contracts, see Timur Doumler's overview [Doumler24].

Why C++ contracts?

While there may be multiple reasons to use contracts in a codebase, three of them stand out:

- improve reasoning about correctness by replacing implicit assumptions with explicit guarantees;

```
// Changes the name of the user at 'index' to
// 'new_name' and returns the old name.
std::string update_name(std::size_t index,
                        std::string new_name)
  pre(index < users_.size())
  pre(!new_name.empty())     // business rule:
                             // no empty names
  post(r: !r.empty())        // old name must have
                             // been valid
{
  auto& user = users_[index];
  contract_assert(user.is_valid());
  std::string old = std::move(user.name);
  user.name = std::move(new_name);
  return old;
}
```

Listing 1

...contracts do not replace reasoning; they support it with evidence. They help detect mismatches between intended behaviour and observed behaviour...

- allow run-time checking of some invariants, providing empirical validation of the conceptual model;
- improve safety and, as a consequence, improve correctness.

The first benefit is about reasoning at abstraction boundaries. Documentation can express similar obligations and guarantees, but contracts add explicit, machine-checkable structure at the declaration site. When preconditions and postconditions appear in the declaration, both caller and implementer can reason locally: the caller knows what must hold before the call, and the implementer knows what may be assumed inside the routine. This reduces the need to inspect distant call paths and improves modularity with respect to correctness.

The second benefit is empirical validation. Some invariants are hard to verify purely by inspection, especially in large systems, but contracts let us check at run time whether our assumptions hold in real executions. In that sense, contracts do not replace reasoning; they support it with evidence. They help detect mismatches between intended behaviour and observed behaviour, which is valuable both during development and in production configurations that enable checking.

The third benefit is safety, with correctness as a consequence. If more relevant invariants are enforced, fewer invalid states can propagate through the program, and fewer operations execute under violated assumptions. Going back to Lamport's framing, this improves safety, and because safety is part of correctness, it also improves correctness.

If we considered only the last two benefits, we might be tempted to move invariant checks into function bodies as ordinary assertions. That would still improve empirical validation and safety, and therefore correctness. However, it would weaken modular reasoning about correctness: the caller could no longer rely on an explicit contract at the declaration site, because key assumptions and guarantees would be buried in the implementation. Local reasoning would become significantly harder.

Compositional safety

How safety composes

In the previous article [Teodorescu26] we argued that safety always composes if requirements are properly assigned and distributed to implementation units. However, we did not get into enough detail to clarify why this is true for some common examples.

Let us consider the code in Listing 2. For this example, we intentionally wrote the preconditions as documentation. If the preconditions are upheld, `f` is memory safe: it does not lead to invalid memory accesses. `g` is always memory safe. However, the nested call `f(g(10))` at the bottom of the listing leads to a memory violation; we attempt to access `v[-10]`.

How come? We said that the two functions `f` and `g` are memory safe and we also said that safety composes, and yet a simple function composition gives us a memory violation.

```
std::vector<int> v = {1, 2, 3, 4, 5};

// Requires: x >= 0
// Requires: !v.empty()
int f(int x) {
  auto n = std::min(x, (int) v.size()-1);
  return v[n];
}

int g(int x) {
  return -x;
}
...
int r = f(g(10));
...
```

Listing 2

The short answer is that `f(g(10))` is not a valid composition of the contracts we wrote down. Function `f()` is memory safe only under its preconditions. Calling it with a negative argument is outside the domain in which that safety claim applies. The useful question is therefore not whether two arbitrary calls can be nested syntactically, but whether the caller establishes the preconditions required at each call boundary.

To provide an answer to this dilemma, let us start by looking at the functions in our example. How many functions do we have there? If your answer is just two, you are missing one important function: the caller. The code `f(g(10))` needs to be placed in some other function; let us call this `caller()`. We argued that `f()` and `g()` are memory safe, but we have not analysed this third function.

The caller has two operations related to our code. The first one calls `g(10)`; the second one calls `f()` with the result. Now, if we look again at Lamport's paper [Lamport77], safety is about ensuring that all invariants hold, and these invariants are placed between any two adjacent operations. For this call sequence, the relevant invariant is that the value passed to `f()` must be non-negative. This is shown in Listing 3 (next page).

Indeed, if we look closely, before calling `f()` the caller must ensure that it does not pass a negative number. The caller has not established that obligation. Thus, while `f()` and `g()` are memory safe, `caller()` is not. The problem is in `caller()`, not in the way we defined `f()` and `g()`. This does not show that safety fails to compose. We were composing two memory-safe operations with one that does not respect the agreed contracts.

Composability unit

The previous example clarifies how composability works. We might assume that in an expression such as `f(g(10))` composition happens between `f()` and `g()`, but we have shown that this is not the right model. Composability happens through the caller, which establishes obligations and consumes guarantees at each call boundary. This leads to the following realisation:

The composability unit is always caller-callee.

Exposing clear preconditions and postconditions makes reasoning about correctness at function boundaries easier

```
void caller() {
  auto r1 = g(10);
  // contract_assert(r1 >= 0)
  int r = f(r1);
  ...
}
```

Listing 3

Each time we see apparent sibling composability, such as `f(g(10))`, we can decompose it into two caller-callee units. In this example, one unit is `caller()` to `g()`, and the other is `caller()` to `f()` (using the value returned by `g()`).

This applies whenever we compose functions[1]. Expressions like `f(g(10))` are syntactic nesting, but semantically they are two caller-mediated contracts in sequence.

Safety and composability flow through obligations and guarantees at each call boundary: caller establishes callee preconditions, then receives callee postconditions. This is where C++ contracts become especially useful: they make boundary contracts explicit and checkable, even when the source syntax is nested.

Exposing clear preconditions and postconditions makes reasoning about correctness at function boundaries easier. It lets us state invariants at the composability unit, empirically validate our model of correctness, and improve both safety and correctness.

If the caller establishes callee preconditions before each call, and each callee guarantees its postconditions on return, then the sequence preserves the safety property.

Enforcing safety

Safe and enforced safe operations

So far, we have looked at safety from the perspective of invariants and composition across call boundaries. That perspective is useful for reasoning about programs as a whole, but it is not always the most convenient way to decide how to build individual operations.

We now turn to a bottom-up perspective: how to build operations that preserve a chosen safety property. Let us start with some definitions.

We define a ***P*-safe operation** to be an operation that, when used correctly (all preconditions are met), never violates safety property *P*. For example, adding two non-negative `int` values satisfies the property 'arithmetic overflow cannot happen' if the precondition `a <= INT_MAX - b` is met.

We define an **enforced *P*-safe operation** to be an operation that never violates safety property P, even if the preconditions are violated. For example, a hardened implementation [2] of `std::vector` may check the precondition of element access and trap when the index is outside the valid range. In such a configuration, the access operation can be treated as enforced memory-safe: an invalid index does not proceed to an out-of-bounds access.

An enforced *P*-safe operation provides stronger guarantees than a simple *P*-safe operation. This makes reasoning about safety and correctness easier. Therefore, we should prefer an enforced *P*-safe operation if there are no significant extra costs attached to it.

The most common way to transform a *P*-safe operation into an enforced *P*-safe operation is to check all relevant preconditions and fail early if the preconditions are not upheld. The typical way to fail when preconditions are not met is to trap the program, possibly with a useful message pointing to the violated precondition.

This comes with a caveat. The typical way to build an enforced *P*-safe operation does not work if *P* is 'the program shall not trap'. The only way to ensure that enforcement works for all safety properties *P* is to refuse to proceed after detecting a precondition violation, turning a safety problem into a liveness problem. This is not a good strategy for most practical purposes, but the caveat is worth mentioning.

So far, we have discussed only enforcing preconditions. While it is possible to use only this strategy to ensure that operations are enforced *P*-safe, we may want to add checks at different places in our programs. In other words, we may want to check program invariants more often.

Composability

Composing enforced safe operations is much easier than composing simple safe operations. If `A` and `B` are enforced safe operations for property *P*, then calls to these operations cannot violate *P*, even when the caller fails to establish their ordinary preconditions.

In other words, **enforced *P*-safe operations preserve safety property *P*, even for sibling compositions**. The caller may still violate the operation's ordinary preconditions, but such violations cannot proceed into a violation of *P*.

Safe languages such as Rust and Swift improve safety by making many primitive operations enforced safe with respect to selected safety properties. For example, safe Rust prevents memory unsafety in safe code, and Swift checks many operations that would otherwise lead to invalid memory access or undefined behaviour.

There are important qualifications. These languages still have escape hatches: Rust has `unsafe`, Swift has unsafe pointer APIs, and both languages can interact with foreign code. They also do not enforce every desirable safety property. Instead, they enforce a carefully chosen set of properties at language and library boundaries.

Within the checked subset, however, the composability story is much stronger: if the primitive operations enforce property *P*, then ordinary compositions of these primitives preserve *P* without requiring every caller to manually re-establish the low-level preconditions.

1 Coroutines are operationally more involved, but they follow the same boundary principle.

2 C++26 introduced a *hardened* mode in which extra run-time checks are done for common operations in the standard library; the reader can consult [P3471R4] for more details.

Incorrect documentation can produce unmanaged technical debt at a much higher rate than an incorrect implementation.

C++ contracts, if configured properly, can help turn many operations into enforced safe operations. By trapping on failed preconditions, we can ensure that certain safety properties are enforced in configurations where the relevant contract assertions are checked.

There is a caveat here. In typical C++26 implementations, the person configuring the build can turn off assertion checks. There is no way for a library author to specify that the preconditions should be checked by the compiler in all cases. This means that preconditions cannot be fully guaranteed at the code level.

Even with that limitation, checked preconditions are a significant step up in safety.

Beyond C++ contracts

Contracts are useful, but even if contract assertions are always enabled, they have major limitations:

- not all preconditions, or invariants in general, can be expressed with a contract assertion;
- not all invariants should be checked with contract assertions;
- sometimes, encoding all preconditions and postconditions as contract assertions makes it harder to reason about the code.

For the first limitation, let us imagine a sorting algorithm parameterised by a predicate that dictates the order. The precondition is that the predicate implements a strict weak ordering relation, which is not completely checkable in general. We cannot add contract assertions for invariants whose predicates cannot be expressed in code.

Moving on, some invariants can be checked in code, but are not worth checking, at least not in release builds. Consider the contract of a binary search algorithm. This algorithm has as a precondition that the input sequence is sorted. This is something that we can check. But checking whether the elements are sorted is more expensive than the binary search itself.

Another example is appending an element to a vector, where a good postcondition is to ensure that all original elements remain equal to their previous values and in the same order. To enforce that, however, we need to copy all the elements into a separate storage and then compare them with the resulting vector.

Lastly, expressing contracts in code may make the code harder to read and reason about. Take the sort example again. The preconditions need to include that the input sequence is valid (well-formed and accessible) and that the comparison predicate is a strict weak ordering. The postconditions need to state that the output elements are a permutation of the input elements, and that the resulting sequence is sorted. While in human language it is easy to say "the output is a permutation of the input", imagine expressing that in code as a postcondition. Assuming all of these are expressible in code, it would still be a lot of code. That extra code needs to be reasoned about, so there is an extra burden. Instead, the contract of the algorithm can often be summarised with a short comment: Sorts the given sequence in place, using the given predicate.

Documentation as contract

Starting from these shortcomings of machine-checkable contracts, Dave Abrahams argues for the idea that **documentation is a contract** [Parent23]. It is often much easier to express contracts in human language than in a form that machines can understand. Any contract that can be expressed in a machine-checkable way can also be described in human language. When the documentation starts to become too complex, that is a sign that the API may be too complex and may require a redesign.

One counterargument to this viewpoint is that documentation tends to get out of sync with the code. First, this tends to happen more often if the documentation is too large. Large documentation implies a large contract, which implies that the API is too complex. Again, this may be a sign that the API needs to be improved.

Second, we need to analyse this in a broader context to see whether this is really the fault of documentation. When writing a function, we might have contracts at multiple levels:

- an undocumented idea of what the function is required to do;
- what is written in the documentation;
- what is expressed as contracts in the language, for example using C++26 contracts;
- what the tests tell us that the function needs to do, directly or indirectly;
- what the function actually does.

Over time, the contracts expressed at any of these levels can drift away. But there is not always a clear preference for which one is the right contract. Each time we observe a drift, we have a problem that needs to be fixed.

When someone uses the function, they will hopefully read the documentation first. If the documentation is wrong, the function will be used in ways it was not intended to support. At that point, the documented contract and the *de facto* contract have diverged, and the implementation may be wrong relative to how the API is actually used.

Incorrect documentation can produce unmanaged technical debt at a much higher rate than an incorrect implementation.

Let us look at a very simple, yet powerful example. Listing 4 (next page) shows a `push_back` function for a `vector`-like class with compact documentation. Listing 5 (also on next page) expands part of the same contract into more explicit documentation clauses. If we attempted to express all of those clauses as checked contract assertions, the result would be much more complex, and often much less efficient.

The point is not that Listing 4 spells out every detail mechanically. The point is that a clear human-language contract can often carry the

```
// Adds 'e' to the end.
void push_back(E e);
```

Listing 4

```
// Adds 'e' to the end.
//
// - Invariant: '*this' is in a valid state
// - Postcondition: 'this->size() == old.size()
//   + 1'
// - Postcondition: 'std::equal(old.begin(),
//   old.end(), this->begin())'
// - Postcondition: the last element is equal to
//   the value of 'e' on entry
void push_back(E e);
```

Listing 5

intended meaning better than a long list of formal clauses. In ordinary API documentation, "adds `e` to the end" already tells the reader the essential postcondition: the collection has one more element, the previous elements are still there in their previous order, and the new last element corresponds to `e`.

After following Dave's advice for a while, I can report that writing documentation as contracts is one of the best ways to write better code. Having language-checked assertions helps, but it cannot replace good documentation.

Local reasoning

Local reasoning [Parent24] refers to the ability to analyse and verify a defined unit of code, such as a function or class, in isolation, without needing to understand all the contexts in which it is used, or all the details of the code it depends on. These units must have well-defined APIs that separate client-side usage, such as calling a function or instantiating a class, from implementation-side logic, such as the internal details of the function or class.

Local reasoning improves reasoning about programs. Our experience so far suggests that this is one of the hardest parts of software engineering. The complexity of software keeps growing, while human cognitive capacity is limited [3]. Local reasoning ensures that the amount of reasoning we need to perform is closer to linear in the size of the program, because we do not have to reason about the same unit of code multiple times in multiple contexts. Local reasoning offers good composability guarantees.

In that sense, local reasoning is one of the central goals of programming.

Contracts are one of the key requirements for local reasoning. If `f` calls `g`, then reasoning about `f` should not require inspecting the implementation of `g`. Thus, we need the contract of `g` when we reason about `f`.

Not having good contracts implies that local reasoning becomes impossible, which complicates reasoning about the codebase. After a certain threshold, reasoning becomes so hard that it tends to invite the accumulation of unmanaged technical debt.

In this context, both code contracts and documentation contracts are useful. Documentation contracts tend to be more concise and easier to read and reason with, but code contracts benefit from empirical verification.

When to use C++ contracts

There are multiple perspectives on contracts. Different people may have different needs from the C++ contracts feature. Here, I will briefly set out my current perspective on using it.

Reasoning is still the main goal. C++ contracts help local reasoning by making some obligations and guarantees explicit at program boundaries. The help is limited: documentation contracts may do a better job for contracts that are too complex, too expensive, or impossible to encode as contract assertions.

3 Local reasoning also seems important in the age of AI, where LLMs are writing more and more code. But that is a larger topic; we will leave it for another article.

C++ contracts do not replace contracts written in documentation. We still need good documentation to reason about the code. We should make sure that what we express with C++ contracts matches the human-readable documentation.

C++ contracts may provide empirical evidence that our reasoning is correct. In this sense, we use the language to validate our assumptions.

C++ contracts can help provide *enforced P-safe operations*, for various properties. This is especially useful for protecting against undefined behaviour.

C++ contracts are essentially a safety feature. Contract checking is about ensuring that invariants hold; in Lamport's framing, this is a safety concern. The benefits above compound: clearer reasoning, empirical validation, and enforced checks all make it easier to write programs that do not accidentally violate safety properties.

Takeaways

The small `f(g(10))` example is a useful reminder. At first glance, it looks as if two safe functions were composed and safety failed. But the missing piece was the caller. Once we look at the caller-callee boundaries, the problem becomes clearer: one of the contracts was not respected.

That is the pattern contracts help us see. They turn hidden assumptions into visible obligations and guarantees. Safety composes when each caller respects the contracts of the operations it uses, and even syntactically nested expressions can be decomposed into these caller-callee units.

Once those boundaries are visible, we can ask a stronger question. Do we merely rely on the caller to satisfy a contract, or do we stop execution when the contract is violated? This is where enforced safe operations become important. The more we use them, the more we increase safety in the application.

C++26 contracts can support this style of programming through local reasoning, empirical validation, and enforced safety, depending on how checking is configured. Still, documentation contracts remain essential. Some contracts are too complex, too expensive, or impossible to express in code, and they still need to be part of the reasoning structure of the program.

In any case, contracts remain one of the best tools we have to improve the safety of our programs. ■

References

[Doumler24] Timur Doumler, 'C++ Contracts in 5 Minutes' on *TIMUR.AUDIO*, posted 30 January 2025 at https://timur.audio/contracts_explained_in_5_mins.

[Lamport77] Leslie Lamport, 'Proving the Correctness of Multiprocess Programs', *IEEE Transactions on Software Engineering 2*, 1977, https://www.microsoft.com/en-us/research/publication/2016/12/Proving-the-Correctness-of-Multiprocess-Programs.pdf.

[Meyer91] Bertrand Meyer, 'Design by Contract', in *Advances in Object-Oriented Software Engineering*, D. Mandrioli and B. Meyer (eds), Prentice Hall, 1991.

[P2900] Joshua Berne, Timur Doumler, Andrzej Krzemieński, et al., 'P2900R14: Contracts for C++', https://wg21.link/P2900R14.

[P3471R4] Konstantin Varlamov, Louis Dionne, 'P3471R4: Standard library hardening', https://wg21.link/P3471R4.

[Parent23] Sean Parent and Dave Abrahams, 'Better Code: Contracts in C++', *CppCon 2023*, https://www.youtube.com/watch?v=OWsepDEh5lQ.

[Parent24] Sean Parent, 'Local Reasoning in C++', *NDC TechTown 2024*, https://www.youtube.com/watch?v=bhizxAXQlWc.

[Teodorescu26] Lucian Radu Teodorescu, 'Safety, Correctness, and the Shape of Reasoning', *Overload* 193, June 2026, https://accu.org/journals/overload/34/193/overload193.pdf#page=6.

C++ Reflection: Verifying Complier-generated Functions

The compiler generates special member functions as shown in Hinnant's Table. Lieven de Cock shows how reflection can be used to verify them.

The compiler generates special member functions, according to specific rules. These have been described, and have been summarized in a nice table, known as *Howard Hinnants' table* [Hinnant20].

Using reflection, we can verify whether this is indeed correct (which it is), refresh our knowledge of the table, and see some reflection mechanisms at work.

We will use several toy examples, each creating a struct called **`TestX`**, where **`X`** is the number of the row in the above mentioned table.

The struct contains two integer member variable and a method (**`doSomething()`**). What these do doesn't matter, but they are needed for the refleciotn mechanism to be able to point out some key differences.

Used reflection mechanics

This is our starting toy structure, which is the test for row 1 in the table:

```
struct Test1
{
  int x{};
  int y{};

  void doSomething() {};
};
```

The function analyzing every type is templatized based upon the type to inspect (see Listing 1). We will walk through it step by step.

Basics applied:

- reflect on the structure/type, using the lift/reflection operator: **`^^T`**, which gives us a **`std::meta_info`** value that we store for later use.

```
template <typename T>
void printMethods(bool fullDetails = false)
{
  constexpr auto ctx
    = std::meta::access_context::current();
  constexpr auto refl{^^T};
  constexpr auto numberMembers
    = members_of(refl, ctx).size();

  std::println("Number of members for {} : {}",
    identifier_of(refl), numberMembers);

  int methods{};
  int deleted{};
  template for (constexpr auto member :
      // gives shadow warning :-(
      define_static_array(members_of(^^T, ctx)))
    {
    if constexpr (std::meta::is_function(member)
                  && !is_deleted(member))
    {
      constexpr std::string_view name
        = display_string_of(member);
      std::println("  {}", name);
      if constexpr (has_identifier(member))
      {
        std::println("     with identifier {}",
                     identifier_of(member));
      }
      ++methods;

      // do some inspecting
      if(fullDetails)
      {
        std::println("     is_defaulted: {}",
                     is_defaulted(member));
        std::println("     is_user_provided: {}",
                     is_user_provided(member));
        std::println("     is_user_declared: {}",
                     is_user_declared(member));
      }
    }
    // extra: let'salso count deleted functions
    if constexpr (std::meta::is_function(member)
      && is_deleted(member))
    {
      ++deleted;
    }
  }
  std::println("Number of methods for {} : {}",
               identifier_of(refl), methods);
  std::println("Number of deleted methods for {}"
         " : {}", identifier_of(refl), deleted);
  std::println();
}
```

Listing 1 (cont'd)

- next we ask for all the members of the struct/type (**`members_of`**), and ask for the count of those. For the **`access_context`**, see my first article in the series [deCock26].
- we print out this count
- we use **`identifier_of`** to also print out the name of the type/struct we are reflecting on
- next we **`loop`** over all those members (using the **`template for`** construct) which we asked for a second time and we stored those in a static array
- there is NO reflection method which gives you just the member methods (there is one to just get the data members though)

Lieven de Cock Lieven is a software developer, architect, team lead, manager, coach and mentor, with 30 years of experience. He is passionate about C++, software craftsmanship, and clean code. Recently he founded his own consulting company CppDriven, providing services in coaching and workshops for teams on modern C++ and its eco-system of tools. Contact him at lieven.de.cock@telenet.be

- which means during the loop, we need to check if a certain member is actually a function (`std::meta::is_function(xxx)`)
- if it is, it is of interest to us
- however, a method could be deleted, which means it will pop up in this looping, but we should consider it as excluded: `is_deleted(member)`
- we ask for its display string (not everything has an identifier) by means of `display_string_of(member)`
- and in case it would have an identifier (spoiler alert, as for the methods, only our `doSomething` method has one) we print that one out too (`(has_identifier(member), identifier_of(member))`)
- and we increment the counter of methods spotted, which we print out after the loop
- the 'full details' part is covered in the 'We want more, we want more: DesDeMovA' section on page 9
- we also count the deleted functions.

As such, we print out all the methods encountered and we can inspect that outcome to see if it matches what the table tells us (always deduct the number of methods by 1, to exclude our `doSomething` from the equation).

That's our mechanics, our little tool; time to start applying it to our different structs. Buckle up, here we go.

Note that user declaration is sufficient to get the mechanisms rolling; we are not even touching the realm of their implementation.

Row1

```
struct Test1
{
  int x{};
  int y{};

  void doSomething() {};
};
```

This gives the output in Figure 1.

Conclusions:

- default constructor
- `copy` constructor
- `copy` assignment operator
- `move` constructor
- `move` assignment operator
- destructor

We got 6/6.

Row 2

We declare a custom constructor, which means we lose the default constructor.

```
struct Test2
{
  int x{};
  int y{};

  Test2(int); // just declared is enough
  void doSomething() {};
};
```

The output is in Figure 2.

Conclusions:

- the default constructor is indeed gone
- the other 5 are still there

We got 5/6, and added a custom constructor ⇒ 5 + 1 = 6 methods

```
Number of members for Test1 : 9
  void {anonymous}::Test1::doSomething()
          with identifier doSomething
  constexpr {anonymous}::Test1::Test1()
  constexpr {anonymous}::Test1::
    Test1(const {anonymous}::Test1&)
  constexpr {anonymous}::Test1& {anonymous}::
    Test1::operator=(const {anonymous}::Test1&)
  constexpr {anonymous}::Test1::
    Test1({anonymous}::Test1&&)
  constexpr {anonymous}::Test1& {anonymous}::
    Test1::operator=({anonymous}::Test1&&)
  constexpr {anonymous}::Test1::~Test1()
Number of methods for Test1 : 7
Number of deleted methods for Test1 : 0
```

Figure 1

Row 3

We declare the default constructor ourselves.

```
struct Test3
{
  int x{};
  int y{};

  Test3();      // just declared is enough
  void doSomething() {};
};
```

The output is in Figure 3.

Conclusions as for Row 1.

We get 6/6.

Row 4

This is an example of a mistake often made, which typically appears in one of the following forms (remember declaration is sufficient for the repercussions):

- declare a destructor with an empty implementation (`{}`)
- declare a destructor and (`= default`)

```
Number of members for Test2 : 9
  {anonymous}::Test2::Test2(int)
  void {anonymous}::Test2::doSomething()
          with identifier doSomething
  constexpr {anonymous}::Test2::
    Test2(const {anonymous}::Test2&)
  constexpr {anonymous}::Test2& {anonymous}::
    Test2::operator=(const {anonymous}::Test2&)
  constexpr {anonymous}::Test2::
    Test2({anonymous}::Test2&&)
  constexpr {anonymous}::Test2& {anonymous}::
    Test2::operator=({anonymous}::Test2&&)
  constexpr {anonymous}::Test2::~Test2()
Number of methods for Test2 : 7
Number of deleted methods for Test2 : 0
```

Figure 2

```
Number of members for Test3 : 9
  {anonymous}::Test3::Test3()
  void {anonymous}::Test3::doSomething()
          with identifier doSomething
  constexpr {anonymous}::Test3::
    Test3(const {anonymous}::Test3&)
  constexpr {anonymous}::Test3& {anonymous}::
    Test3::operator=(const {anonymous}::Test3&)
  constexpr {anonymous}::Test3::
    Test3({anonymous}::Test3&&)
  constexpr {anonymous}::Test3& {anonymous}::
    Test3::operator=({anonymous}::Test3&&)
  constexpr {anonymous}::Test3::~Test3()
Number of methods for Test3 : 7
Number of deleted methods for Test3 : 0
```

Figure 3

```
struct Test4
{
  int x{};
  int y{};

  ~Test4() = default; // just declared is enough,
  // but here is the typical mistake of
  // setting it to default - not needed at all
  void doSomething() {};
};
```

The output is in Figure 4.

Conclusions:

- no more **move** constructor
- no more **move** assignment operator
- both are not declared, and as such not in the list of members.

We get 4/6, we have lost **move**.

Row 5

We have user-declared the **copy** constructor.

```
struct Test5
{
  int x{};
  int y{};

  Test5(const Test5&); // just declared is enough
  void doSomething() {};
};
```

The output is in Figure 5.

Conclusions:

- we lost the default constructor (remember the moment we declare any other constructor, we lose it)
- we again lost the 2 **move** methods
- both are not declared, and as such not in the list of members.

We got 3/6.

Row 6

We have user-declared the **copy** assignment operator.

```
struct Test6
{
  int x{};
  int y{};

  Test6& operator=(const Test6&); // just
                    // declared is enough
  void doSomething() {};
};
```

The output is in Figure 6.

Conclusions:

- we again lost the 2 **move** methods
- both are not declared, and as such not in the list of members

We got 4/6.

Row 7

We have user-declared the **copy** constructor.

```
struct Test7
{
  int x{};
  int y{};

  Test7(const Test7&&);  // just declared is
                         // enough
  void doSomething() {};
};
```

```
Number of members for Test4 : 7
  constexpr {anonymous}::Test4::~Test4()
  void {anonymous}::Test4::doSomething()
          with identifier doSomething
  constexpr {anonymous}::Test4::Test4()
  constexpr {anonymous}::Test4::
    Test4(const {anonymous}::Test4&)
  constexpr {anonymous}::Test4& {anonymous}::
    Test4::operator=(const {anonymous}::Test4&)
Number of methods for Test4 : 5
Number of deleted methods for Test4 : 0
```

Figure 4

```
Number of members for Test5 : 6
  {anonymous}::Test5::
    Test5(const {anonymous}::Test5&)
  void {anonymous}::Test5::doSomething()
          with identifier doSomething
  constexpr {anonymous}::Test5& {anonymous}::
    Test5::operator=(const {anonymous}::Test5&)
  constexpr {anonymous}::Test5::~Test5()
Number of methods for Test5 : 4
Number of deleted methods for Test5 : 0
```

Figure 5

```
Number of members for Test6 : 7
  {anonymous}::Test6& {anonymous}::
    Test6::operator=(const {anonymous}::Test6&)
  void {anonymous}::Test6::doSomething()
          with identifier doSomething
  constexpr {anonymous}::Test6::Test6()
  constexpr {anonymous}::Test6::
    Test6(const {anonymous}::Test6&)
  constexpr {anonymous}::Test6::~Test6()
Number of methods for Test6 : 5
Number of deleted methods for Test6 : 0
```

Figure 6

```
Number of members for Test7 : 7
  {anonymous}::Test7::
    Test7(const {anonymous}::Test7&&)
  void {anonymous}::Test7::doSomething()
          with identifier doSomething
  constexpr {anonymous}::Test7::~Test7()
Number of methods for Test7 : 3
Number of deleted methods for Test7 : 2
```

Figure 7

The output is in Figure 7.

Conclusions:

- we lost the default constructor (remember: the moment we declare any other constructor, we lose it)
- we lost the 2 **copy** methods, and this time they are 'deleted'
- we also lose the **move** assignment operator (not declared and as such not in the list of members)

We got 2/6.

Row 8

We have user-declared the **move** assignment operator.

```
struct Test8
{
  int x{};
  int y{};

  Test8& operator=(const Test8&&); // just
                      // declared is enough
  void doSomething() {};
};
```

The output is in Figure 8 (next page).

```
Number of members for Test8 : 8
  {anonymous}::Test8& {anonymous}::
    Test8::operator=(const {anonymous}::Test8&&)
  void {anonymous}::Test8::doSomething()
          with identifier doSomething
  constexpr {anonymous}::Test8::Test8()
  constexpr {anonymous}::Test8::~Test8()
Number of methods for Test8 : 4
Number of deleted methods for Test8 : 2
```

Figure 8

Conclusions:

- we again lost the 2 **copy** methods (deleted)
- we also lose the **move** constructor (not declared and as such not in the list of members)

We got 3/6.

Conclusion

The table is correctly verified by using reflection. Sometimes things are according to expectations. ☺

Only in the cases of Row 7 and Row 8 are the lost **copy** methods 'deleted'. In all other cases, when a method was lost it was with a status of 'not declared'.

In all cases, the reflection showed us:

- destructor is always there
- when we lose a **copy** method, it is in the list, with status 'deleted'
- when we lose a **move** method, it is 'not declared', meaning it is not in the members list
- when we lose the default constructor, it is not present in the members list.

Could we add some bonus inspections ? Can we think of other methods the compiler generates for us ...

Spaceship operator: <=>

Time to place your bets, say we default declare it (that's the use case we would like to have), how many new methods would we get:

- 1 (**<=>**)
- 2 (**<=>** and **==**)
- 6 (**<**, **>**, **<=**, **>=**, **==**, **!=**)
- something else

Let's use the following toy example:

```
struct SpaceShip
{
  int x{};
  int y{};

  auto operator<=>(const SpaceShip&)
    const = default;
  void doSomething() {};
};
```

And the output is in Figure 9.

Conclusion:

We get 2 extra methods

- **operator<=>**
- **operator==**

The correct answer was 2. Did you predict it correctly ?

```
Number of members for SpaceShip : 11
  constexpr auto {anonymous}::SpaceShip::
    operator<=>(const {anonymous}::
    SpaceShip&) const
  void {anonymous}::SpaceShip::doSomething()
          with identifier doSomething
  constexpr {anonymous}::SpaceShip::SpaceShip()
  constexpr {anonymous}::SpaceShip::
    SpaceShip(const {anonymous}::SpaceShip&)
  constexpr {anonymous}::
    SpaceShip& {anonymous}::SpaceShip::
    operator=(const {anonymous}::SpaceShip&)
  constexpr {anonymous}::SpaceShip::
    SpaceShip({anonymous}::SpaceShip&&)
  constexpr {anonymous}::SpaceShip& {anonymous}::
    SpaceShip::operator=({anonymous}::
    SpaceShip&&)
  constexpr {anonymous}::SpaceShip::~SpaceShip()
  constexpr bool {anonymous}::SpaceShip::
    operator==(const {anonymous}::SpaceShip&)
    const
Number of methods for SpaceShip : 9
Number of deleted methods for SpaceShip : 0
```

Figure 9

We want more, we want more: DesDeMovA

Ok, let's do it. Many people, if they have a class and they don't want it to be copied or moved, do the following.

```
struct OverDisable1
{
  int x{};
  int y{};

  void doSomething() {};

  OverDisable1(const OverDisable1&) = delete;
  OverDisable1& operator=(const OverDisable1&)
    = delete;
};
struct OverDisable2
{
  int x{};
  int y{};

  void doSomething() {};

  OverDisable2(const OverDisable2&) = delete;
  OverDisable2& operator=(const OverDisable2&)
    = delete;

  OverDisable2(const OverDisable2&&) = delete;
  OverDisable2& operator=(const OverDisable2&&)
    = delete;
};
```

Both are correct, but much more code than actually needed is written, giving the output in Figure 10.

```
Number of members for OverDisable1 : 6
  void {anonymous}::OverDisable1::doSomething()
          with identifier doSomething
  constexpr {anonymous}::OverDisable1::
    ~OverDisable1()
Number of methods for OverDisable1 : 2
Number of deleted methods for OverDisable1 : 2

Number of members for OverDisable2 : 8
  void {anonymous}::OverDisable2::doSomething()
          with identifier doSomething
  constexpr {anonymous}::OverDisable2::
    ~OverDisable2()
Number of methods for OverDisable2 : 2
Number of deleted methods for OverDisable2 : 4
```

Figure 10

```
Number of members for DesDeMovA : 8
  void {anonymous}::DesDeMovA::doSomething()
          with identifier doSomething
  constexpr {anonymous}::DesDeMovA::DesDeMovA()
  constexpr {anonymous}::DesDeMovA::~DesDeMovA()
Number of methods for DesDeMovA : 3
```

Figure 11

In comes the rule *DesDeMovA*, coined by Peter Sommerlad [Sommerlad19]. It is sufficient to only delete the **move** assignment operator, because:

- we lose the **move** constructor (-1)
- we lose the **copy** operations (-2)
- we explicitly deleted the **move** assignment operator (-1)

6 - 1 - 2 - 1 = 2 ⇒ constructor and destructor.

Our inspected output is in Figure 11.

We want even more

Let's use some more inspection methods:

- **is_defaulted**
- **is_user_declared**
- **is_user_provided**

What does this all mean : is_defaulted

We have 2 scenarios to end up in **is_defaulted**

- we didn't mention the special method at all (like in many of our scenarios)
- we did mention the method and said **= default**

What does this all mean : is_user_declared

This is true the moment we declare the method ourselves (again irrelevant of the implementation).

What does this all mean : is_user_provided

This requires it is **user_declared**, since we first need to declare before we can provide it. Note, again this does not look at the implementation (like in our Row examples we did not provide implementations (mostly)).

So what's the difference than between **is_user_declared** and **is_user_provided**, something is user declared, but not user provided if that declaration (or definition in cpp file) mentioned **= default**.

So basically when it is user declared, next to that it is *either* **is_defaulted** or **is_user_provided**.

Inspecting the entire output for our 8 rows is considered homework for the reader.

But let's just pick one, say Row 3. Here's the output of it with full details (shortened to the constructor only).

```
printMethods<Test3>(true);

Number of members for Test3 : 9
  {anonymous}::Test3::Test3()
        is_defaulted: false
        is_user_provided: true
        is_user_declared: true
```

Look at the constructor:

- it is user declared
- it is user provided
- it is NOT defaulted

```
printMethods<Test3Bis>(true);

Number of members for Test3Bis : 9
constexpr {anonymous}::Test3Bis::Test3Bis()
  is_defaulted: true
  is_user_provided: false
    is_user_declared: true
```

Figure 12

Let's adjust the struct to Test3Bis as follows; we declare the constructor, but default it:

```
struct Test3Bis
{
  int x{};
  int y{};

  Test3Bis() = default;
  void doSomething() {};
};
```

which gets the output in Figure 12 (shortened to the constructor only):

Again let's look at the constructor:

- it is user declared
- it is NOT user provided
- it is defaulted

Time to close up our little journey of experimenting with reflection, what did we learn (for some things see the first article for more information):

- reflection/lift operator (**^^**)
- **members_of**
- **access_context**(**current()**)
- **define_static_array**
- **identifier_of**
- **has_identifier**
- **display_string_of**
- **is_function**
- **is_deleted**
- **is_defaulted**
- **is_user_declared**
- **is_user_provided**

You can play with the code online here: https://godbolt.org/z/daeWhed1b

References

[deCock26] Leven de Cock, 'C++ Reflection: a Universal Printer', posted on 5 May 2026 and available at https://www.linkedin.com/pulse/c-reflection-universal-printer-lieven-de-cock-abzze

[Hinnant20] Howard Hinnant, 'How I Declare My class and Why', posted 24 February 2020 on Github and available at https://howardhinnant.github.io/classdecl.html. (The table image is https://howardhinnant.github.io/smf.jpg.)

[Sommerlad19] Peter Sommerlad, 'Introducing the Rule of DesDeMovA', posted 1 July 2019 on *Safe C++ Blog*, available at https://safecpp.com/2019/07/01/initial.html

This article was previously published on LinkedIn on 2 July 2026 at https://www.linkedin.com/pulse/draft/preview/7477750928557670400/


accu
World-class conference
Professional development
Printed journals
Local groups
Email discussion groups
Reviews of technical books

Join us now!

Membership rates on our website.
Visit accu.org for details

Trip Report: ACCU on Sea 2026

This year the ACCU conference and *C++ on Sea* combined. Sándor Dargó shares his ACCU on Sea trip report with us

Once again, I got the chance to come to Folkestone, UK. I think this was my fifth time, but the first one at *ACCU on Sea*. Yes, there is no more *C++ on Sea* [Dargo], there is no more *ACCU* in Bristol – they got merged into *ACCU on Sea*.

The venue is the same as *C++ on Sea*, many organisers are the same, but of course there are also changes. The conference chair is no longer Phil Nash but Guy Davidson. The schedule is more packed – this year, the conference ran over four days (ignoring the two workshop days), including Saturday. Probably due to the fact that these two major C++ conferences were merged, I think there were more people than ever – around 400 C++ enthusiasts. It was sold out.

And yet, in general, familiar faces. This place holds a special place in my heart as it was the first in-person international C++ conference which I attended as a speaker.

In this post I'll share:

- Highlights from talks and ideas that resonated with me.
- Personal impressions, including reflections on my own sessions – both the main talk and the lightning talk.

My favourite talks

Let me share with you my four favourite talks – one per day, in chronological order.

The next 20 weeks of systems engineering (Andrei Alexandrescu)

I think the first time I checked the schedule of *ACCU on Sea* [ACCU], the title of this talk was different. It was about the next 20 years. It turns out, I was not hallucinating as agents do. That's what Andrei originally had in mind. Then he went for 20 months, then 20 weeks, and stopped there.

Frankly, I think this was planned by Andrei, but it clearly demonstrates how rapidly our craft is changing – thanks to, or maybe better to say, because of AI.

Andrei is one of the most entertaining technical speakers and this occasion was no different. But to be fair, about 20 minutes into his keynote, while I was having a great time, I didn't understand where he was going. But hey, he still had 70 minutes left to connect the sometimes biblical, sometimes historical stories with the technical message he wanted to convey.

I must say he did it. By the end, his talk made sense as a whole.

He even got into making predictions – about the future. That's quite a difficult thing to do, not only because, well, it's about the future and therefore unknown, but also because we always try to adapt. That's not a new idea; the book *The Rational Optimist* [Ridley11] also discussed it.

It's worth mentioning that Andrei clearly admitted that he is wrong in 90% of his predictions. Partly because of that and due to the lack of space, I won't go through all of them, but I want to highlight his ideas about abstractions.

Andrei expressed his disagreement with Elon Musk, who claims that the need for abstractions is going away – in the sense that AI will write machine code directly. Andrei thinks Musk is wrong. AI needs abstractions, and moreover, thanks to metacognition, it's going to develop its own without our intervention. Sure, it could write machine code directly, but that's not efficient enough.

> What is metacognition? In short, it's the ability to think about one's own thinking. When applied to AI, it means the system can reflect on its own reasoning process, evaluate its approach, and adjust its strategy on the fly. Instead of blindly following instructions, an AI with metacognitive capabilities can recognise when an abstraction would help and create one – much like a human developer would when noticing repetitive patterns in their code. This is what makes it possible for AI to go beyond simply executing prompts. It can reason about the problem at a higher level and build the right tools for the job.

On a slightly related note, Andrei predicts that prompt engineering is also going to go away. The AI will be able to refine the prompts we feed it on its own.

Where is this all leading? If you ask me, to the haven of project managers. Andrei predicts that projects are going to grow both in size and complexity thanks to better abstractions and because we'll need to hold fewer details about the systems in our heads. Yes, we'll have to keep focusing on the higher level abstractions.

How? We don't exactly see yet, as we're only in the "bubblesort era of AI assisted coding". Maybe in 20 weeks, we'll see more...

Modernizing legacy codebases without stopping the world (Peter Muldoon)

Peter's talks are regularly featured in my trip reports. He is an excellent speaker and our areas of interest often overlap. Code modernization has always been a pet peeve for me, so I was keen to see his brand new talk.

He started with a crucial question when it comes to modernizing legacy code. What is legacy code? Probably the most prevalent answer is code that has no tests, but Peter went further. Legacy code is code that is in production. As simple as that. Understandable.

When it comes to new features – features that we often learn about at conferences – the problem is that it takes years to get them in the compilers we use at work. But let's say you have them. Then what? Will the business pay for the modernization? Or would you take the risk of mass-updating the code? Both answers are probably no.

Sándor Dargó is is a passionate software craftsman focusing on reducing maintenance costs by applying and enforcing clean code standards. He loves knowledge sharing, both oral and written. When not reading or writing, he spends most of his time with his two children and wife in the kitchen or travelling. Feel free to contact him at sandor.dargo@gmail.com

we can explicitly instantiate a template that references private data – and the compiler won't complain about access

Basically you have two paths ahead. A risky, high-impact, difficult-to-reason-about refactor or even rewrite, or the path of small, mechanical fixes with localised reasoning, lower risk and effort.

In the vast majority of cases you'd go with the latter.

But how to do it? Peter walked us through two approaches. Either you use deterministic static analysis tools or non-deterministic AI-based ones. The problem is that based on his experience, neither works out of the box – and on top of that, people don't even trust AI. Who can blame them?

Even a seemingly simple change like replacing `typedef` with `using` doesn't work at scale.

But something that has worked out quite well so far for Peter is combining the two approaches. Fix the shortcomings of static analysis tools through AI-generated scripts, or even add new checks altogether. With that you solve the non-deterministic nature of AI and also the lack of trust in it, while at the same time you can capitalise on its productivity.

The takeaway is Muldoon's Law of Automated Refactoring:

> The more complicated/risky the change, the smaller the scale & scope of the refactoring must be.

Opening the black box: legally testing private members in C++ (Hubert Liberacki)

One of the talks I enjoyed the most was a short one. While in certain rooms 90-minute talks were going on, in two of the rooms three 20-minute talks were stacked back-to-back.

One of them was about how to test the private parts of your classes.

Sounds like a horrible idea. Yet, I absolutely enjoyed this talk.

Hubert approached the question very pragmatically. He said – as one might expect – not to do this, and in fact he's not a heavy user of his own library either.

But if you absolutely need to, because you must test a legacy codebase without clear interfaces or you have to deal with third-party libraries that cannot be modified, there is a way beyond invoking undefined behaviour via the usual tricks that I don't even want to mention.

So how?

There is actually a loophole in the language [CppRef]:

> Explicit instantiation definitions ignore member access specifiers: parameter types and return types may be private.

This means that we can explicitly instantiate a template that references private data – and the compiler won't complain about access. We can exploit this property to store a pointer-to-member for the private data during explicit template instantiation, making it accessible from the outside. There are a couple more issues to solve, but this is the essence of the solution.

You can check the example code [CompExp], and the corresponding library [GitHub].

Contract assertions on virtual functions (Timur Doumler)

I don't want to rank my favourite talks, but if I had to pick a personal favourite from the conference, it would probably be Timur's – delivered right before lunch on a sunny Saturday morning.

As I'm preparing a series on contracts, the topic simplified my choice. When Timur Doumler, Jason Turner, Matt Godbolt, Roth Michaels and Francis Glassborow all give a talk at the same time, it's not an easy decision. But the subject matter made it easy.

Timur is definitely one of the most knowledgeable people to give a talk on this topic. And it's not the first time he's spoken about contracts either.

As he explained, the previous year at *ACCU 2025* he talked about the proposal that was accepted into C++26, but he also covered how different languages such as D or Ada support assertions.

Later, at *C++ on Sea 2025*, he gave a less technical talk on how to make stubborn engineers agree on something.

And now he came back to share how he actually made stubborn engineers agree on how C++ contracts should support pre- and post-conditions on virtual functions.

In the first part of his talk, he focused on his methodology for finding consensus on technical design questions:

1. Establish shared concepts, definitions, terminology
2. Define and motivate the problem
3. List the requirements
4. List the known solutions
5. Arrange the known solutions in an evolution graph
6. Construct conformance table
7. Refine conformance table
8. Re-evaluate solution space
9. Arrange conflicting requirements into a decision graph
10. Approve the consensus solution

Next he presented what contracts are in C++ and what ashtrays on airplanes have to do with contract violation handlers. I'll let you watch the recording to find the answer to that question.

Then the focus shifted to pre- and post-conditions, which are already supported by a few languages – like the already mentioned D and Ada – and we saw why their approaches aren't quite good enough for C++.

There were a couple of deep questions that had to be answered in the design process, such as what should happen if only `Base::f` defines conditions but `Derived::f` does not, or the other way around. After

Two particularly interesting requirements concern multiple inheritance: what happens if there are two mutually incompatible pre-conditions, or mutually exclusive post-conditions?

reviewing all the requirements and the different existing solutions and proposals, we must say that P3097 provides the best option even with the trade-offs that had to be addressed.

Two particularly interesting requirements concern multiple inheritance: what happens if there are two mutually incompatible pre-conditions, or mutually exclusive post-conditions? The solution designed by Timur Doumler, Joshua Berne, and Gašper Ažman handles even this.

And the best news came at the very end: just a week before Timur's presentation, P3097 was approved for C++29 – a moment the audience celebrated with enthusiastic applause.

My favourite ideas

And now my four favourite ideas – again, one per day.

The difference between decorators and adapters (Klaus Iglberger)

'The Design Patterns Guy' gave another excellent talk on design patterns in which he covered 2.5 patterns: adapters, decorators and expression templates. Why don't they add up to 3? For that you'll have to watch his talk. ;)

Now, let's just focus on the difference between decorators and adapters. Both are part of structural patterns – as categorised in the *Design Patterns Book* [GoF] – both are non-intrusive and widely used. But often confused. They are so often confused that even the standard library calls certain decorators adapters.

You can recognise the difference when you think about what they do. A decorator doesn't alter the interface, but it adds functionality or responsibility to the decorated class.

An adapter, on the other hand, doesn't add any functionality. It just alters – adapts – the interface to match the expected one.

When you think about it, it's rather simple. After this talk, I hope I won't mix them up again.

Attack through keys (Kevin Carpenter)

I've wanted to go to one of Kevin Carpenter's talks for a long time and I finally made it to 'O(1) or O(no-no-no): Mastering the unordered_map'. It was an excellent talk on when and why you should choose `unordered_map` over other data structures.

One idea that surprised me and I wanted to share is how the choice of your keys can lead to a security vulnerability. Imagine your server stores user-controlled input – form fields, JSON keys, HTTP headers – in an `unordered_map`.

If an attacker knows how you hash your keys (which is likely if your code is open-source), they can craft keys that are designed to collide into a single bucket. Once all your data ends up in one bucket, every insert and lookup degrades from O(1) to O(n). A few hundred kilobytes of crafted keys can pin a CPU core for minutes. This is known as a HashDoS attack and it's not theoretical – back in 2011, this class of vulnerability hit PHP, Python, Ruby, Java, Node.js, and ASP.NET all at once, resulting in emergency patches across the ecosystem.

Kevin demonstrated the effect live: with 8,000 crafted keys, a strong hash handled them in about 14 milliseconds, while a weak hash under attack took over 500 – roughly 37 times slower.

So what can you do about it? Use randomised, keyed hashing – something like SipHash with a per-process random salt – so that attackers can't precompute collisions. And as Kevin put it: "Don't hand-roll crypto. Use a vetted keyed hash."

The Dusikova / Fahller trick to make [[nodiscard]] transitive (Björn Fahller)

Björn Fahller gave a talk about the composability of callables. I won't go into all the details, but it was an excellent talk – the room was full and I think you need no bigger appreciation than having both Jason Turner and Matt Godbolt in the audience.

Among Björn's design goals, one stood out to me: making `[[nodiscard]]` transitive in his pipe-syntax-based library. The trick he used to achieve this is quite elegant.

The idea starts with a tag-like wrapper for functions that carry the `[[nodiscard]]` attribute:

```
template <typename F>
struct nodiscard : F {}
```

Next, you create a corresponding concept that can detect whether a function type inherits from this tag:

```
template <typename F>
concept nodiscard_function = requires {
  []<typename T>(nodiscard<T>*) {} (
    std::declval<std::remove_cvref_t<F>*>()
  );
};
```

Once you have a tag with the right semantics and a concept to match against, you can use compile-time conditionals to branch on it. The full implementation details are beyond the scope of this article, but the key insight is that you can decide with an `if constexpr` how to handle a given type:

```
if constexpr (nodiscard_function<F>) {
  // return a function wrapped into nodiscard
} else {
  // return a function without nodiscard
}
```

And why do I call this the Dusikova or the Fahller trick? As Björn explained, he saw this technique in Hana Dusikova's CTRE library. But as we discussed during the *hallway track*, techniques and tricks are often not named after the first person to use them, but after someone who rediscovers and popularises them – so this might just end up being the Fahller technique.

The paradox of effort (Yvonne Rogers)

Yvonne Rogers, a professor of Interaction Design at UCL, gave a keynote on what AI is doing to us and for us. Her talk touched on something I've been thinking about a lot lately – something she called the paradox of effort.

The idea is so simple, you probably thought about it: when we make the effort ourselves to perform a task, it can be more rewarding than delegating it. Think of cooking a meal from scratch versus ordering takeaway. Or assembling that IKEA shelf – yes, the result might be slightly crooked, but you feel a certain pride in it. That's sometimes called the IKEA effect, and it applies far beyond furniture.

Now apply this to AI-assisted work. Generative AI is remarkably good at amplifying our cognition – writing, analysing, coding. But by reducing the cognitive effort needed, it may also reduce the satisfaction we get from the work. We no longer tell the computer what to do step by step; we tell it what output we want. That's powerful, but something gets lost in the process.

So should we design AI tools to reduce human effort or to increase it? Rogers argues for the latter. Increasing cognitive effort – involving deeper and more challenging thinking – leads to more engagement, more fulfilment, and ultimately more pride in our work. The goal shouldn't be to eliminate effort, but to redirect it toward systematic reasoning, reflection on goals, and intentional decision-making. Though the results of research are a bit more nuanced.

My talks

And there are more opportunities than before to give lightning talks. As you can imagine, I proposed more than one. But luckily there were so many people interested in giving lightning talks that each person could only go once. That's a great thing!

I talked about clocks and testing. I recently wrote a series on clocks and I wanted to share one concept that fits five minutes. So I talked about why calling `system_clock::now` directly in your production code is not the best idea when it comes to testability and what to do instead. (Read more on it on my blog [Dargo26]. This lightning talk was important to me, because in the last two years I didn't give any new talks on C++ – I was focusing more on higher-level principles or soft skills. Though for this year I also proposed a full technical talk (on clocks), but that's not what was accepted.

My main talk was on code reviews. It was a polished, slightly reworked and updated version of the talk I gave last year at *Meeting C++*. I talked about how code reviews can help build better code and stronger teams at the same time. No code, no processes – just how to write code review comments that won't make people defensive and that won't get ignored.

Personal impressions

It's always great to come here. As I shared last year, I always enjoy Berlin, not to mention Denver. But even though I was born in a big city, deep in my heart, I'm not a big city guy. I enjoy the calm of a small town in the south of France and I don't have the slightest intention to swap it for the hustle and bustle of a big city. Even though from time to time I do enjoy that.

So, in this coastal Kentish town of Folkestone I almost feel at home, even though I'm not a local in any sense. But I already have my favourite places, including a second-hand toy and bookstore – to the great delight of my kids.

I'm really happy that the merged conferences were kept in Folkestone instead of Bristol.

When it comes to organisation, perfection is not what I'm looking for. I mean, that's a great thing to achieve, but I don't think it's expected. What I'm looking for in any event that runs for longer than a day is reactivity and improvements. Do they recognise issues and provide improvements from day to day? They did. I was a bit puzzled on the first day that there

was enough coffee but not enough cups, and water was running low. But by day two, that was better.

And even though lunch was served at fewer places than in earlier years due to the higher number of parallel tracks, somehow it was smoother and I think I queued less than in earlier years.

I cannot express the gratitude I feel towards the people who made it possible for me to be here. That includes my family, the organisers, and also my bosses who let me come.

And even though sometimes a conference is very demanding – especially for introverted people – and I often feel impostor syndrome, it's so nice to get gentle and live feedback from people who follow my work. And when some highly respected people whose shoelaces I couldn't even tie call me by name, that's just humbling and really makes you feel part of the community.

Conclusion

What more to say? As always, the *On Sea* conference was great — ACCU just as much as C++ before it. Four days packed with inspiring talks about topics ranging from AI's future to legacy code modernisation, from design patterns to contract assertions.

I walked away with new insights, fresh ideas, and a stronger sense of belonging in this community. Thanks for having me there. I hope to be back in 2027. ■

References

[ACCU] *ACCU on Sea 2026*: Schedule, available at https://accuonsea.uk/2026/schedule/index.html

[CompExp] Compiler Explorer: https://godbolt.org/z/qrcPnbz1W

[CppRef] cppreference.com, 'Class template', at https://en.cppreference.com/w/cpp/language/class_template

[Dargo] Sandor Dargo, blog posts relating to *C++ on Sea*, available at https://www.sandordargo.com/tags/cpponsea/

[Dargo26] Sandor Dargo, 'Time in C++: Once More About Testing', posted on *Sandor Dargo's Blog* on 27 Jan 2026 (updated 29 June) and available at https://www.sandordargo.com/blog/2026/01/28/clocks-part-9-once-more-about-testing

[GoF] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1995) *Design Patterns: Elements of Reusable Object-Oriented Sofware*, Addison-Wesley Professional.

GitHub] cpp-member-accessor: https://github.com/hliberacki/cpp-member-accessor

[Ridley11] Matt Ridley (2011) *The Rational Optimist: How Prosperity Evolves*, published by Harper Perennial. Online review available on *Sandor Dargo's Blog*, at https://www.sandordargo.com/blog/2020/02/12/the-rational-optimist

Implementing std::visit

Learning how to implement standard library features is illuminating. Quasar Chunawala explores implementing your own version of std::visit.

A `variant` models a **choice between types**. It is a smart union. Variants are **sum types**.

```
struct on { int temperature_; };
struct off { };

using oven_state = std::variant<on, off>;
```

Recursive node-based structures, error handling, state machines etc. are examples of concepts that can be elegantly modelled by variants:

- JSON
- ASTs(Abstract Syntax Trees)
- State Machines
- Error Handling

Variant visitation

What is visitation?

Visitation can be defined as an **abstraction** that allows you to access the currently active variant *alternative* in an **exhaustive** and **expressive** manner. You are forced to provide some behaviour for all the types that the variant supports. Its expressive, because it doesn't work through a chain of if-else; its a very elegant way of saying, for these types perform these actions.

Traditional visitation

Traditional visitation requires a `Callable` object which can be invoked with every possible variant alternative. The traditional way of creating such an object is defining a `struct` (see Listing 1).

If you call `std::visit` with an instance of your `printer` struct and the `variant`, it will automatically figure out, what type is in the variant and call the corresponding overload.

```
// Traditional visitation example - one variant
struct printer{
    void operator()(int){ std::cout <<
      "int" << "\n"; }
    void operator()(float){ std::cout <<
      "float" << "\n"; }
    void operator()(double){ std::cout <<
      "double" << "\n"; }
};
using my_variant =
  std::variant<int, float, double>;
my_variant v0{ 20.0f };

std::visit(printer{}, v0);
```

Listing 1

Quasar Chunawala is a quant-developer. He is deeply passionate about programming in C++, Rust, concurrency and performance-related topics. He enjoys long-distance hiking. He regularly discusses interesting code snippets/language features on his blog and runs a monthly C++ newsletter at https://quantdev.blog/newsletter.

```
struct circle{ double radius_; };
struct box{
    double width_;
    double height_;
    double depth;
};
struct collision_resolver{
    void operator()(circle, circle){ /*...*/ }
    void operator()(circle, box){ /*...*/ }
    void operator()(box, circle){ /*...*/ }
    void operator()(box, box){ /*...*/ }
};
using my_variant = std::variant<circle, box>;
my_variant v0{circle{}};
my_variant v1{box{}};

std::visit(collision_resolver{}, v0, v1);
```

Listing 2

You can extend this logic to multiple variants. (See Listing 2.)

What you get is a form of multiple dispatch, that has to handle all possible combinations. This is also really powerful way of implementing multiple dispatch.

Building helper functions needed for std::visit

Under the hood, `std::visit` must invoke the right candidate for the currently active alternatives from the function overload set. We start with the simplest case of 2 variants. The central idea is to maintain a single array of function pointers. Each entry in the function pointer table has the shape:

```
[](Visitor visitor, Variant0 v0, Variant1 v1)
```

For concreteness, assume that both variants `v0` and `v1` offer a choice between 2 types:

```
using Variant0 = std::variant<int, float>;
using Variant1 = std::variant<int, float>;
```

It is possible to have 4 distinguishable behaviours (overloads), when a visitor visits these variants. So, we build an array of size 4.

The `std::visit` implementation should automatically invoke the correct function overload based on the currently active types in `v0` and `v1`. Before we proceed, let us fix some terminology first. Suppose a variant `v` allows a choice amongst n types:

```
// Pseudo-code
std::variant<T0, T1, ..., Tn> v;
```

In the case of 2 variants `v0` and `v1` offering a choice amongst n_1 and n_2 alternative types respectively, we have a total of $n_1 \times n_2$ distinguishable choices. For example, consider the case of `v0` offer a choice of 3 types and `v1` offering a choice of 4 types. Each choice is a distinct entry in the 2d-table, and we have a total of $\mathbf{3 \times 4 = 12}$ distinct choices. We will have **12** candidates in our function overload set. We can assign a multi-

I strongly recommend creating a simpler version first, maybe hard-coded with two or three variants. So you can get an idea of the actual shape before making it variadic

	j=0	j=1	j=2	j=3
i=0	<0, 0>	<0, 1>	<0, 2>	<0, 3>
i=1	<1, 0>	<1, 1>	<1, 2>	<1, 3>
i=2	<2, 0>	<2, 1>	<2, 2>	<2, 3>

Table 1

index `<i,j>` to the function overload `[](Ti, Tj){ /*...*/ }`. (See Table 1.)

We can convert this 2d-table to a 1d-table (Table 2).

Thus, the pointer to the function `[](Ti, Tj){ /*...*/}` with multi-index `<i,j>` is mapped to `4i + j` 1d-index in a 1d-table.

In the case of 3 variants, `v0`, `v1` and `v2`, offering a choice amongst ,

Multi-index	1D-index
<0, 0>	0
<0, 1>	1
<0, 2>	2
<0, 3>	3
<1, 0>	4
<1, 1>	5
<1, 2>	6
<1, 3>	7
<2, 0>	8
<2, 1>	9
<2, 2>	10
<2, 3>	11

Table 2

$n_1=3$, $n_2=4$ and $n_3=5$ alternative types respectively, we have a total of $3\times4\times5=60$ distinguishable choices. We will have **60** candidates in our function overload set. We can assign a multi-index `<i,j,k>` to the function overload `[](Ti, Tj, Tk){ /*...*/ }` (see Table 3) where each entry `<i,j,k>` expands over $k=0,1,2,3,4$. The full set of multi-indices and their 1D mappings is shown in Table 4.

	j=0	j=1	j=2	j=3
i=0	<0,0,k>	<0,1,k>	<0,2,k>	<0,3,k>
i=1	<1,0,k>	<1,1,k>	<1,2,k>	<1,3,k>
i=2	<2,0,k>	<2,1,k>	<2,2,k>	<2,3,k>

Table 3

Multi-index	1D-index	Multi-index	1D-index	Multi-index	1D-index
<0, 0, 0>	0	<1, 0, 0>	20	<2, 0, 0>	40
<0, 0, 1>	1	<1, 0, 1>	21	<2, 0, 1>	41
<0, 0, 2>	2	<1, 0, 2>	22	<2, 0, 2>	42
<0, 0, 3>	3	<1, 0, 3>	23	<2, 0, 3>	43
<0, 0, 4>	4	<1, 0, 4>	24	<2, 0, 4>	44
<0, 1, 0>	5	<1, 1, 0>	25	<2, 1, 0>	45
<0, 1, 1>	6	<1, 1, 1>	26	<2, 1, 1>	46
<0, 1, 2>	7	<1, 1, 2>	27	<2, 1, 2>	47
<0, 1, 3>	8	<1, 1, 3>	28	<2, 1, 3>	48
<0, 1, 4>	9	<1, 1, 4>	29	<2, 1, 4>	49
<0, 2, 0>	10	<1, 2, 0>	30	<2, 2, 0>	50
<0, 2, 1>	11	<1, 2, 1>	31	<2, 2, 1>	51
<0, 2, 2>	12	<1, 2, 2>	32	<2, 2, 2>	52
<0, 2, 3>	13	<1, 2, 3>	33	<2, 2, 3>	53
<0, 2, 4>	14	<1, 2, 4>	34	<2, 2, 4>	54
<0, 3, 0>	15	<1, 3, 0>	35	<2, 3, 0>	55
<0, 3, 1>	16	<1, 3, 1>	36	<2, 3, 1>	56
<0, 3, 2>	17	<1, 3, 2>	37	<2, 3, 2>	57
<0, 3, 3>	18	<1, 3, 3>	38	<2, 3, 3>	58
<0, 3, 4>	19	<1, 3, 4>	39	<2, 3, 4>	59

Table 4

Thus, the pointer to the function `[](Ti, Tj, Tk){/*...*/}` with multi-index `<i,j,k>` is mapped to the 1D-index $20i+5j+k$.

We shall code up two helper functions to convert a multi-index to a 1d-index and back:

- `to_1d_index<3,4,5>(size_t, size_t, size_t)`
- `from_1d_index<3,4,5>(size_t)`

See Listing 3 (next page).

Simpler case of 1 and 2 variants

I strongly recommend creating a simpler version first, maybe hard-coded with two or three variants. So you can get an idea of the actual shape before making it variadic. Otherwise, you'd be facing both problems at once. See Listing 4 (also next page).

Generalizing to a pack of variants

The shape of a function inside the `vtable_func` array should be something like this:

```
[](Visitor&& vis, Variants&&... var) {/* ... */}
```

```
namespace dev::tools{
  template<std::size_t... Dimensions>
  constexpr auto build_coeffs_array(){
    constexpr std::array<std::size_t, sizeof...
      (Dimensions)> dimensions { Dimensions... };
    constexpr std::size_t coeffs_size = sizeof...
      (Dimensions);
    std::array<std::size_t, coeffs_size> coeffs
      = {};

    for(std::size_t i{0}; i < coeffs_size; ++i)
      coeffs[i] = 1;

    for(std::size_t i{0}; i < coeffs_size - 1;
        ++i){
        // In step i, we need to populate all
        // coeffs[j], j <= i
        for(std::size_t j{0}; j <= i; ++j){
            coeffs[j] *= dimensions[i+1];
        }
    }
    return coeffs;
  }
  /*
  coeff[0] = v1size * ... * v[n-1] size
  coeff[1] = v2size * .... *v[n-1] size
  ...
  coeff[n-2] = v[n-1] size
  coeff[n-1] = 1
  */
  template<std::size_t... Dimensions>
  std::size_t constexpr to_1d_index(auto...
                                    indices){
    constexpr std::array<std::size_t, sizeof...
      (Dimensions)> dimensions { Dimensions... };
    std::size_t coeffs_size = sizeof...
      (Dimensions);
    auto coeffs
      = build_coeffs_array<Dimensions...>();

    const std::array<size_t, sizeof...(indices)>
      indices_arr { indices... };
    //std::println("{}", indices_arr);
    return std::inner_product(coeffs.begin(),
      coeffs.end(), indices_arr.begin(), 0);
  }

  /*
  Suppose we have the dimensions <3, 5, 2> and
  the coordinates (1, 3, 1).
  (1, 3, 1) maps to the linear index 17.
  */

  template<size_t N>
  constexpr auto build_coords_array(
        std::array<size_t,N> coeffs,
        size_t& initState){
    size_t state = initState;
    std::array<size_t, N> coords{};
    for(size_t i{0}; i < N; ++i){
      coords[i] = static_cast<size_t>(
        state / coeffs[i]);
      state -= coords[i] * coeffs[i];
    }
    return coords;
  }

  template<size_t... Dimensions>
  decltype(auto) constexpr
      from_1d_index(std::size_t n){
    constexpr std::size_t coords_arr_size
      = sizeof...(Dimensions);
    constexpr auto coeffs
      = build_coeffs_array<Dimensions...>();
    std::array<size_t, coords_arr_size> coords
      = build_coords_array(coeffs, n);
    return coords;
  }
}
```

Listing 3

Internally, you would invoke `vis` with `std::get<Is>(var)...`, where `Is...` would be an index sequence or constexpr array of the desired indices. See Listing 5 (on next page).

```
// Simple case of 2 variants
namespace dev{

  namespace example{

    struct Dummy{};
    template<typename T>
    using Wrapper = std::
      conditional_t<std::is_void_v<T>,Dummy,T>;

    template<typename Visitor, typename Variant0,
      typename Variant1>
    decltype(auto) visit(Visitor&& visitor,
        Variant0 v0, Variant1 v1){
      constexpr std::array<std::size_t, 2>
        dimensions
        = { std::variant_size_v<Variant0>,
          std::variant_size_v<Variant1> };
      constexpr std::size_t vtable_size
        = (std::variant_size_v<Variant0>
          * std::variant_size_v<Variant1>);
      using cases_t = std::string(*)(Visitor,
        Variant0, Variant1);
      static constexpr auto vtable {
        []<size_t... Indices>(
            std::index_sequence<Indices...>){
          return std::array<cases_t,4>{
            [](Visitor vis, Variant0 v0,
               Variant1 v1){
              constexpr auto multi_idx
                = dev::tools::
                  from_1d_index<2,2>(Indices);
              return vis(
                std::get<multi_idx[0]>(v0),
                std::get<multi_idx[1]>(v1));
            }... };
        }(std::make_index_sequence
          <vtable_size>())
      };

      return vtable[dev::tools::
        to_1d_index<2,2>(v0.index(), v1.index())]
        (visitor, v0, v1);
    }
  };
}

template<typename... Callables>
struct Visitor : Callables...{
    using Callables::operator()...;
};

void test_single_dispatch(){
    std::variant<int, float, double> v{0};
}

void test_double_dispatch(){
    std::variant<int, float> v1{0};
    std::variant<int, float> v2{3.14f};

    auto result = dev::example::visit(
        Visitor{
            [](int, int) -> std::string {
              return "(int, int)"; },
            [](int, float) -> std::string {
              return "(int, float)"; },
            [](float, int) -> std::string {
              return "(float, int)"; },
            [](float, float) -> std::string {
              return "(float, float)"; },
        },
        v1, v2
    );

    std::cout << "result = " << result << "\n";
}
```

Listing 4

More information

A *CppCon 2017* talk by Vittorio Romeo called 'Implementing variant visitiation using lambdas' was an important source of information. The talk is available at https://www.youtube.com/watch?v=3KyW5Ve3LtI

First published on Quasar Chunawala's blog on 25 April 2026, the original is available at https://quantdev.blog/posts/implementing_visit/

```
namespace dev{
  template <typename Visitor,
            typename... Variants>
  decltype(auto) visit(Visitor &&visitor,
      Variants &&...vs) {
    constexpr std::size_t vtable_size
      = (std::variant_size_v<
         std::remove_cvref_t<Variants>> * ...);

    // Each entry in vtable should have the shape
    // [](Visitor visitor, Variants... vs){}
    using result_t
      = decltype(visitor(std::get<0>(vs)...));
    using cases_t = result_t(*)(Visitor,
      Variants...);

    static constexpr auto vtable{
      []<size_t... Indices>
          (std::index_sequence<Indices...>){
        constexpr std::array<std::size_t,
          sizeof...(Variants)> dimensions
          = { std::variant_size_v
            <std::remove_cvref_t<Variants>>... };
        constexpr std::size_t vtable_size
          = (std::variant_size_v
            <std::remove_cvref_t
            <Variants>> * ...);
        return std::array<cases_t, vtable_size>{
          [](Visitor vis, Variants... vs)
              -> result_t{
            constexpr auto multi_idx
              = dev::tools::
                from_1d_index
                <std::variant_size_v
                <std::remove_cvref_t
                <Variants>>...>(Indices);
```

Listing 5

```
            return [&]<size_t... Is>(
                std::index_sequence<Is...>){
              return vis((static_cast
                <std::variant_alternative_t
                <Is, std::remove_cvref_t
                <Variants>>>
                (std::get<multi_idx[Is]>
                (vs)))...);
            }(std::make_index_sequence
              <sizeof...(Variants)>());
          }...
        };
      }(std::make_index_sequence<vtable_size>())
    };

    auto i = dev::tools::
      to_1d_index<std::variant_size_v
                  <std::remove_cvref_t
                  <Variants>>...>(vs.index()...);
    return vtable[i](visitor, vs... );
  }
}

void test_multiple_dispatch(){
  std::variant<int, float, double> v1{0};
  std::variant<int, float, double> v2{3.14f};
  // std::variant<char, int, long, float, double>
  // v3{'H'};

  auto result = dev::visit(
    Visitor{
      [](int, int) -> std::string {
         return "(int, int)"; },
      [](int, float) -> std::string {
         return "(int, float)"; },
      [](int, double) -> std::string {
         return "(int, double)"; },
      [](float, int) -> std::string {
         return "(float, int)"; }
    },
    v1, v2
  );

  std::cout << "result = " << result << "\n";
}
```

Listing 5 (cont'd)

Gor Nishanov (1971–2026)

Guy Davidson pays tribute to a highly respected member of the C++ Standards Committee.

Gor Nishanov will forever be known as the principal architect of coroutines in C++. His passing at the end of June will be keenly felt by all of us on the committee. Not only have we lost a great intellect from our midst, we have also lost a man of great enthusiasm, generosity, diligence and wit.

Besides coroutines, he was the co-author of the Musser-Nishanov sequence-matching algorithm, a very practical improvement on the well-known Boyer-Moore algorithm. His commitment to intellectual honesty was an example to many. During the development of the coroutines proposal, he spent considerable time, often late at night, helping other members of the committee with their proposals which argued *against* his own. This rare demonstration of integrity won the hearts of many.

He worked at Microsoft for many years; his most recent committee work was the proposal on concurrent queues, a proposal that has been under discussion for well over a decade. He jumped in to help things along because he could see it was an important piece of the standard library that needed completing.

On the committee mailing list, the outpouring of reminiscence and fondness for Gor has been considerable. Everyone has a surprising story to tell of his kindness, his love for life, as well as his passion for music and his love for the piano. He leaves behind his wife Anzhela and two daughters, Alisa and Anya. The family have asked that you consider donating blood or plasma, or contribute to esophageal cancer research.

Afterwood

We were told to 'show your workings' when we were at school. Chris Oldwood explains why that's also a good idea in the workplace.

It's a pretty tense time in the Oldwood household at the moment, much like many other families in the UK with children waiting for their exam results. Oldwood Offspring v4.0 (not his real name as we're not that cruel) has had the pleasure of spending the last few months studying hard for his exams – luckily he didn't follow in his father's footsteps and wisely chose to listen to his siblings instead and put the time into revising the content and improving his exam technique.

What I failed to learn at school was that exams are one of life's little games that you sadly have to play, at least until someone comes up with a better assessment strategy that doesn't involve sitting silently in a room and having to remember stuff in a way that bears no resemblance to The Real World™. It's also one reason why I've never been interested in the various certification schemes on offer, but that's in part because I remember the 14-year-old back in the 90's who qualified as a Novell Certified Engineer despite having little to no practical experience. Being able to regurgitate what a particular Windows API does is no guarantee that you can string Windows API functions and procedures together in a useful way to create a working program.

What is different about studying for school exams in the modern era is that you not only have easy access to lots of past papers, but also the marking schemes too. This is a real game changer as you no longer have to merely guess what the examiner might be looking for, you can actually look it up, and have a deeper understanding of how the game is played – 'know thy enemy', as Sun Tzu once advised.

One tip, which I had always assumed was some kind of perverse technique teachers used to make their life easier when marking, or slowing the bright kids down so they could have a bit of peace and quiet, was the idea of showing your workings. It turns out, after recently reading many marking schemes, that this is actually beneficial during exam calculations because you can still be awarded marks for the correct *approach*, even if you accidentally plug in the wrong numbers. In particular, failing to apply the correct units conversion might only cost you a mark, while still receiving the remainder for the body of the calculation. Examiners might be contented to treat unit conversion as a "nice to have" but I'm pretty confident that NASA takes it far more seriously after what happened to the Mars Climate Orbiter, etc.

Unlike eigenvalues and eigenvectors[1], I have found the practice of showing one's workings to be of immense value in the workplace too. Much as I enjoy the occasional game of 'guess the intent' or 'reverse engineer how they achieved this result', I'd rather be like the examiner playing along and validating the approach because, unlike the examiner, I don't already know the right answer.

1 They eventually cropped up 20 years later when working on a finance gig, but luckily by then the only maths I needed for my roles effectively involved ++ and --.

The most obvious manifestation of this is probably the writing of formal tests. When you only have the code to work from, and you discover a problem, without *any* tests it's hard to know whether the author ever considered the scenario in question. There is plenty of code in the world that only appears to work purely by accident. Without any tests to work from you have an uphill battle establishing an initial set of 'likely' invariants before you fix the actual problem. At least if you have some tests to work from, you have the version control history of both the code and tests to consult, along with *some* degree of confidence that the author even considered testing the code. (People writing and pushing code whose correctness is only based on mental simulation is more common than you might think.)

Writing a test plan – either as part of story refinement, or appended to a commit message – is a practice which I have adopted more and more in recent years, especially when not pairing or working as an ensemble. We talk about reviewing the *code*, but rarely do I see people questioning the approach used to test it. The presence of new, or updated unit tests is a good sign, but as a more experienced programmer I find much of my time goes into pondering the Law of Unintended Consequences, i.e. what is not covered by the automated tests that might need special consideration. Sadly, "the unit tests pass" is the new "works on my machine".

Where I find people sharing their workings particularly helpful is in the diagnosis of issues. Posting a screenshot of a log message or chart on a ticket is akin to just writing down the answer in the exam. While it probably is relevant to solving the specific issue, it takes very little effort to also include on the ticket the KQL query or Bash one-liner that led to the answer. If the resolution doesn't seem forthcoming it's then possible for someone to quickly review the approach, just like when testing, to check obvious things haven't been missed. What I've learned from writing (in general), is that sharing my approach has, on many occasions, lead me to spotting my own silly mistakes before sending others off down the wrong path. In environments where issues drag on and handovers are common, your successor will thank you for not forcing them to derive everything from first principles before investigating further.

I know that ultimately I'm an automation junkie, and that I'm also a fan of the concept of marginal gains – my sixth sense is probably 'I see automation opportunities everywhere'. Many of those opportunities start out as crude one-liners (puns not intended) and make their way into wiki pages and runbooks before eventually solidifying as tool. But I feel many other valuable learning opportunities are missed because the approach is locked away in someone's head or on their machine. Marking your own homework might feel like a personal productivity win but you'll never know if you could arrive at the answer quicker or more reliably in future unless you allow your peers to grade your approach too. ■

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

accu.org

accu.org

accu.org

accu.org

accu.org