overload 193

JUNE 2026 £4.50

Safety, Correctness, and the Shape of Reasoning

Lucian Radu Teodorescu shows how safety and progress can break the problem into composable parts.

C++ Reflection: a Universal Printer

Lieven de Cock demonstrates how to print information about class members using this new feature.

A Coalescing Event Queue for High-Frequency Systems

Ajay Pandey shows how a coalescent queue can avoid redundant updates while keeping relevant state.

Afterwood

Chris Oldwood reflects on some milestones he considers to be worth celebrating.

A magazine of ACCU

ISSN: 1354-3172

accu

professionalism in programming

Monthly journals

Annual conference

Discussion lists

To find out more, visit accu.org

June 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 from Adobe Stock Photos.

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* 194 should be submitted by 1st July 2026 and those for *Overload* 195 by 1st September 2026.

Copyrights and trademarks
Some articles and other contributions use terms that are either registered trade marks or claimed as such. The use of such terms is not intended to support nor disparage any trade mark claim. On request, we will withdraw all references to a specific trade mark and its owner.

By default, the copyright of all material published by ACCU is the exclusive property of the author. By submitting material to ACCU for publication, an author is, by default, assumed to have granted ACCU the right to publish and republish that material in any medium as they see fit. An author of an article or column (not a letter or a review of software or a book) may explicitly offer single (first serial) publication rights and thereby retain all other rights.

Except for licences granted to 1) corporate members to copy solely for internal distribution 2) members to copy source code for use on their own computers, no material can be copied from *Overload* without written permission from the copyright holder.

Mission Accomplished

The C++26 standard draft is now complete. Frances Buontempo takes a moment to appreciate this and other jobs well done.

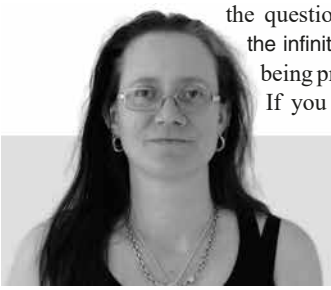
As ever, I haven't written an editorial. Like many, I was watching the Artemis II trip around the far side of the moon with fascination.[1] I just missed out on the first moon landing – which possibly gives away my age. None of this is a proper excuse for no editorial. As ever, guest editors are welcome: do get in touch.

I think the trip to the moon was amazing. The number of things that needed planning was mind blowing. And the new, never before seen perspective of the far side of the moon was exciting. Would there be new craters? Clangers? [Wikipedia-1]. I presume many people had a sense of trepidation too. Previous trips to space have gone wrong. "Houston, we've had a problem…" springs to mind. When things go wrong, we have an opportunity to learn though. So even a failure can be useful.

What have missions to the moon got to do with anything? Well, I find technical things in general interesting. Most programmers have a similar mindset. Whether that means taking toys to pieces as a child, to find out what's inside, or typing in code and tweaking it, maybe to get extra lives in a game, most of us have explored things some other people might just take for granted. Understanding how something works is satisfying. An "Aha, got it" moment can mean you can step back and muse rather than keep digging, enjoying the satisfaction of a job well done Mission accomplished, right?

I note that 'Mission' means send out or 'to send abroad' [etymonline], or release. Think missionary. And, people who write software do usually release software. Rephrasing 'a release coming up' as a 'mission coming up' might give it a sense of excitement, rather than dread! Mind you, excitement might not be what you're after either. Many places avoid 'big bang' (an unfortunate phrase) release, doing small, incremental steps instead. Either way, once you release a new version of software, you no doubt watch it for a long while afterwards to check everything is running as expected. The release isn't the end of the mission.

Some things do have a definite end point, though. For example, a clasp on my favourite necklace broke recently. I managed to acquire a new one and fixed it. Go me: hardware engineering, for a change. Fixing things in software is a different matter. Maybe you fix one bug and another pops up somewhere else. Or someone uses the new feature and notes it isn't precisely what was needed. A woman's work is never done, right [Wiktionary]? I went to the Agile Yorkshire meetup [Agile] last night and the first speaker mentioned the book *The infinite game* by Simon Sinek. I've not read it, but I'm told it has a focus on business strategy, and asks the question "Are you playing the finite game or the infinite game?" I assume this is partly about being prepared to keep innovating and learning. If you are 'stuck' in an infinite process, it's useful to pause and reflect. Give yourself a moment to experience the joy of an achievement unlocked, even if the mission isn't over.



Perhaps you're not playing an infinite game. I mentioned fixing my necklace. That was straightforward. I have 'finished' other things, too. For example, I recently finished my third book. Relax and enjoy, right? I forced myself to do that, because I could see errata arriving. There are several I need to go through. It's astonishing how many typos and similar always sneak through. When you create something, there are aftershocks. Feedback might be a mixture of positive and negative. Books have errata, software has bugs. Rocket launches have issues. Artemis 2 followed Artemis 1, which was a successful 'unmanned' exploration. Artemis 2 had some initial problems. NASA reported a "helium flow issue to the interim cryogenic propulsion stage" [Warner26] during a dry run (described as a wet dress rehearsal – words, eh?). They had to roll back the rocket (literally) and fix the problem. Further missions are planned. Their website lists three more, leading up to building a lunar base for further exploration. Where will it end? In theory, they are trying to get to Mars. We'll see. If that does happen, I am sure that won't be the end of matters.

Things are often never ending. I have a hand written TODO list. Ticking something off is satisfying, but usually when I complete one thing, I end up with two more to do. My mission accomplished point is when the paper is full: it's time to get a new piece and start over. I then get the chance to abandon some tasks. Or realise I finished a few and tick them off on principle too. I do use a Trello board for some projects. The trouble is I can add a LOT of tasks to that. Somehow the physical constraint of a fixed sized piece of paper helps limit my ambitions to something manageable. Once upon a time, a little while ago, I was working as a research assistant. My supervisor booked a high-tech meeting room for the team, which had an electronic whiteboard. We could then save the board and email around the team. Great, right? The trouble was we added to it each time, so a large amount of the meeting was spent trying to scroll far enough to find where we were writing last time. The thing about infinite space is it's very large. I guess the whiteboard wasn't really infinite, but it was very difficult to find anything.

A limited list of things to do is useful. My list encompasses household chores, as well as work related items. I have started sneaking bucket-list ideas on too – experiences I'd like to have before it's too late. I'm going to run out of paper again, real soon now. It might be better to have lists constrained to specific areas. New standards for C++ might seem a bit like a rolling TODO list, but they are grouped in various ways. There are bug reports as well as new ideas or proposals. ISOCpp gives a good overview of the process [ISO]. If you've joined ACCU, you will have seen the C++ standard report in each edition of the member's magazine *CVu*. The most

1 Thanks to the review team for pointing out this is different from the Dark Side of the Moon [Wikipedia-2]

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.

recent *CVu* covered C++26 [McMonagle26]. C++26 has been worked for a while and is now complete. This means we'll see a new International Standard document, once 'editorial' work has been done (fixing typos, I presume, don't @ me!). We'll then watch compilers try to catch up.

So, what's in C++26? Herb Sutter calls out his "Fab Four Features" [Sutter26a]. These are reflection, dealing with some UB, via erroneous behavior for reading unitialised variables and library hardening, contracts and **`std::execution`**. His blog mentions the abstract for a previous *CppCon* keynote, quoting, "We'll need the next decade to discover what this rocket can do." Everyone is distracted by rockets! I suspect C++ won't go to Mars. Yet. There's more to C++26 than just these four features. John McMonagle's *CVu* write up, from just before the Croydon meeting, covers SIMD-friendly changes to random number generation, hazard pointers, and **`constexpr`** virtual inheritance. Which are your fab four features? I'm glad we have a reflection article in this edition. Maybe you will be inspired to write about one of the new features you have been trying out.

Of course, C++26 isn't all about big new features. There are some small, but useful, changes too. I don't have a full list, but one that's crossed my radar is changes to structured bindings. There are several but now structured bindings can introduce a pack. You have probably come across parameter packs before, and this extends the idea. Sandor Dargo wrote a blog giving details [Dargo26], so I'll use some of his examples.

Given a function **`std::tuple<X, Y, Z> f();`**, you could already say **`auto [x,y,z]=f();`**, provided you were using a new-enough version of C++. With the new pack feature, you can say **`auto [x, ...rest] = f();`**

That way, you can just concentrate on the first item. The pack can also refer to the last element and so on. You can also use pack to write neater code. Sandor gives an example for calculating the dot product:

```
template <class P, class Q>
auto dot_product(P p, Q q) {
  // no indirection!
  auto&& [...p_elems] = p;
  auto&& [...q_elems] = q;
  return (... + (p_elems * q_elems));
}
```

Previous C++ standards required nested **`std::apply`** in a couple of loops or use of potentially hard to read **`std::index_sequence`** calls. Reddit suggests Barry Revzin and Jonathan Wakely were involved, but Jonathan claimed Barry did all the work [Reddit-1]! That's just one small feature. Fortunately, *CppReference* is back up and running, so you can check the site to find out more C++26 features [cppreference]. If you missed what happened, the site had been in maintenance mode for a worryingly long time, partly due to sabotage from people allegedly making changes to say many C++ features had been superseded by Rust. Fortunately, the standard C++ foundation stepped in and helped, so it's back up and running [Sutter26b]. I am relieved: *CppReference* had been an invaluable place to look things up.

A new standard is exciting, and many people are trying out new features or giving conferences talks about them. That's great, but in the 'real world', almost no companies will immediately start using newer compilers. Jens Weller has run a few surveys over the years, finding out what people actually use at work. (Other data is available too, that just sprung to mind first.) Though some people are using C++20 (and perhaps C++23), many are stuck on C++11 or C++17. [Weller22]. If you are in that situation, fair enough. Maybe demand some training time at work to try out new features, or ask if your employer will pay for a conference ticket for you. Or try out a feature on your own time, and write it up for us.

I wonder how smooth any upgrades will be. I recall much pain moving to newer versions of boost or more modern compilers. I frequently had to plough through a huge number of errors and warnings, but when the team finally got to the end, we'd often fixed a few bugs or potential security flaws. A recent discussion on accu-general explored what to look out for when porting from C++17 to C++20 (one more data point for Jens). The original posted said they knew that the spaceship operator might cause problems, comparing pointers rather than the strings they point to [Reddit-2], but was there anything else to look out for. Jonathan Wakely replied, "Yes, lots!" Go check out the thread: I'll try to persuade him to write things up for us. GCC does provide a list of things to watch out for [GCC], and Thomas Köppe wrote a more general list in 2020 [Köppe20], which has a section ('New core language features with global applicability') that states "These are features that may happen to you without your knowledge or consent." This includes details on the spaceship operator and default comparisons. No doubt, we'll get lists of things to watch out for in C++26 soon. Herb Sutter's blog did suggest a "whole category of potential vulnerabilities disappears in C++26, just by recompiling your code as C++26". Give it a go and report back.

So, mission accomplished? Maybe not. Each ending is the start of new things. To misquote JFK, we choose to upgrade C++, not because it's easy, but because it includes hardening, and makes C++ safer. Let's accept challenges, and terraform Mars. Or just write an article or two. Over to you.

References

[Agile] Agile Yorkshire: https://agileyorkshire.org/

[cppreference] C++26 available at https://cppreference.com/cpp/26

[Dargo26] Sandor Dargo, 'C++26: Structured Bindings can introduce a Pack', updated on his blog 28 April 2026 and available at http://sandordargo.com/blog/2026/04/22/cpp26-structured-bindings-packs

[etymonline] 'mission': https://www.etymonline.com/word/mission

[GCC] Porting to GCC 16: https://gcc.gnu.org/gcc-16/porting_to.html#cpp

[ISO] C++ Standardization: https://isocpp.org/std]

[Köppe20] Thomas Köppe20, 'Changes between C++17 and C++20 DIS', published 2 March 2020, available at http://open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2131r0.html

[McMonagle26] John McMonagle, 'Standard Report' (regular feature), most recently in *CVu* 38.1 and available to members only at https://accu.org/journals/cvu/38/1/mcmonagle/

[Reddit-1] 'C++26: Structured Bindings can introduce a Pack', discussion available at https://www.reddit.com/r/cpp/comments/1su6ipi/c26_structured_bindings_can_introduce_a_pack/

[Reddit-2] 'The spacesship operator silently broke my code', discussion available at https://www.reddit.com/r/cpp/comments/ra5cpy/the_spacesship_operator_silently_broke_my_code/

[Sutter26a] Herb Sutter, 'C++26 is done! – Trip report: March 2026 ISO C++ standards meeting (London Croydon, UK)', posted on his blog *Sutter's Mill* on 29 March 2026, available at https://herbsutter.com/2026/03/29/c26-is-done-trip-report-march-2026-iso-c-standards-meeting-london-croydon-uk/

[Sutter26b] Herb Sutter, 'Announcement: cppreference.com update' posted 14 April 2026 at https://isocpp.org/blog/2026/04/announcement-cppreference.com-update

[Warner26] Cheryl Warner 'NASA Strengthens Artemis: Adds Mission, Refines Overall Architecture', NASA, posted 3 March 2026, available at https://www.nasa.gov/directorates/esdmd/nasa-strengthens-artemis-adds-mission-refines-overall-architecture/

[Weller22] Jens Weller, 'If your pool runs dry, maybe fix your pipeline?' posted on *Meeting C++ Newsletter*, published 3 November 2022 at https://www.meetingcpp.com/blog/items/If-your-pool-runs-dry--maybe-fix-your-pipeline-.html

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

[Wikipedia-2] Far side of the moon (https://en.wikipedia.org/wiki/Far_side_of_the_Moon) and Dark side of the moon (https://en.wikipedia.org/wiki/The_Dark_Side_of_the_Moon)

[Wiktionary] 'a woman's work is never done': https://en.wiktionary.org/wiki/a_woman%27s_work_is_never_done

Safety, Correctness, and the Shape of Reasoning

Correctness is hard to reason about as a property of a whole program. Lucian Radu Teodorescu shows how safety and progress can break the problem into composable parts.

In recent years, *safety* has become a central theme in the C++ community. This has been driven in part by a series of reports from 2022 and 2023 [NSA22, CR23, WH23, EC22, CISA23a, CISA23b] that strongly criticised memory-unsafe languages. The term *safety* now appears in proposals, talks, and design guidelines, often carrying significant weight (one can find in [McDougall24, Müller25, Doumler25, Bauman26, Vandevoorde26] a small selection of papers that were seen by the C++ standard committee on the subject).

But what do we actually mean by *safety*?

Given the amount of discussion this topic has generated, one might expect us to have a clear understanding of *safety*. In practice, this is not the case. The term is often used with implicit and differing meanings. Sometimes it refers specifically to memory safety. Sometimes it refers to the absence of undefined behaviour. Sometimes it gestures towards broader claims about program correctness, security, reliability, or risk. Instead of clarifying a proposal, an argument, or a design choice, the word itself can become the object of the discussion. This is a sign that we lack a shared model, not merely a shared vocabulary.

This article is an attempt to build such a model. We examine the relationship between *safety* and *correctness*, using Lamport's framework [Lamport77] as the starting point. In that framework, safety is not a vague assurance that a program is good, nor a synonym for the absence of memory errors. It is a class of correctness properties: properties whose violation is already visible after a finite part of an execution.

This perspective is broad enough to cover many discussions that currently take place under the name *safety*: memory safety, arithmetic safety, functional safety, thread safety, and partial correctness. These are not separate meanings of safety. They are different kinds of requirements whose violations are safety violations. The requirements differ, but the shape of the reasoning is the same.

The next step is to connect this model to compositional reasoning. Correctness is difficult to reason about when treated as a single obligation of the whole program. Lamport's distinction gives us a way to separate that obligation. If safety captures the correctness of any produced result, then the remaining liveness obligation becomes progress: does the program eventually produce such a result? This shift does not make correctness easy, but it gives us a more tractable way to reason about it. Correctness can be decomposed into safety obligations and progress obligations, and programs can be structured so that both are easier to reason about locally.

More broadly, the goal is to better understand the theory behind safety and correctness, and to use that understanding as a basis for writing reasonable code and building correct programs.

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

Safety and correctness

Prerequisites: requirements

Before we can discuss the safety or correctness of a program, we need to know what is expected of that program. In software engineering, we typically call these expectations *requirements*. Thus, any discussion of safety and correctness is necessarily relative to some set of requirements.

Some requirements are explicit: a sorting algorithm should return a sorted permutation of its input; a request handler should normally complete within a given latency budget; a component may be required never to perform invalid memory operations. Other requirements are implicit: the program should not halt unexpectedly, execute arbitrary commands, or expose private data to malicious users.

For the rest of the article, we assume that there is a finite list of requirements, explicit or implicit, and that this list is the source of truth when discussing the correctness of the program. In composed systems, these requirements may need to be translated into different local obligations for different program parts.

Lamport's safety and liveness

A *correct program* is a program that fully meets its requirements, whether explicit or implicit. Reasoning about correctness directly is hard. To make it easier, Lamport proposes focusing on two different kinds of properties: *safety* properties and *liveness* properties [Lamport77]. We follow his definitions:

> To prove the correctness of a program, one must prove two essentially different types of properties about it, which we call *safety* and *liveness* properties. A **safety property** is one which states that something will *not* happen. For example, the partial correctness of a single process program is a safety property. It states that if the program is started with the correct input, then it cannot stop if it does not produce the correct output. A **liveness property** is one which states that something *must* happen. An example of a liveness property is the statement that a program will terminate if its input is correct.

The first important consequence is that **safety is part of correctness**. Safety is not opposed to correctness, and it is not a weaker substitute for correctness. It is one class of correctness properties.

The second consequence is that partial correctness is a safety property. Partial correctness says that, if a program produces an answer, that answer is correct. It does not say that the program eventually produces an answer. That missing part is termination, or more generally progress. Thus:

Total correctness = Partial correctness + Termination

At first, this may seem to clash with the statement that safety properties say that something bad does not happen. But partial correctness can be read negatively: *wrong answers are not produced*, and this follows the way we defined safety.

when we talk about safety, we need to qualify what kind of safety we mean. Is it memory safety, thread safety, arithmetic safety, type safety, or partial correctness?

This also shows why the wording of a property is not enough to classify it. A safety property may be written in positive or negative terms. For a property P, we can often move between 'P must not happen' and 'whenever we observe an answer, $\neg P$ holds'.

The better test is not grammatical, but semantic: *If a safety property is violated, it is violated after a finite prefix of the execution.* In simple terms, safety means that no behavioural invariants are violated.

Once a wrong answer has been produced, an invalid memory access has happened, or private data has been exposed, no future continuation can repair that execution. The violation is already present.

Liveness is different: *No finite prefix can prove violation of a liveness property.*

No matter how long the program has failed to produce an answer, the execution may still be extended with an answer later. Liveness violations are therefore not conclusively detectable in finite time.

One safety to rule them all

Often, when we talk about *safety*, we need to qualify what kind of safety we mean. Is it *memory safety*, *thread safety*, *arithmetic safety*, *type safety*, or *partial correctness*? In the C++ standards committee, there is now a tendency to demand such qualification whenever somebody uses the term *safety*. For practical purposes, this is a good thing.

However, we should not mistake these qualifications for separate definitions of safety. That would fragment the reasoning about correctness into many seemingly unrelated topics. The important distinction is not between different meanings of safety, but between different requirements whose violations count as safety violations.

If we reflect on the different kinds of safety, we realise that we do not have multiple definitions of safety. We have the same definition, but the relevant requirements vary. **Different kinds of safety are given by different kinds of properties, not by different definitions of safety.**

The requirements we need for what we call *memory safety* are different from the ones we need for *arithmetic safety*, and so on. These names remain useful, but they describe classes of safety properties, not different concepts of safety.

Having one notion of safety allows us to think holistically about correctness. It also lets us look for common patterns across different domains. We can still solve memory-safety, arithmetic-safety, or thread-safety problems separately, but the correctness of the program depends on all of them. It takes only one safety-property violation for the program to stop being correct. There is only one safety, one safety to rule them all.

Undefined behaviour and safety

If a program reaches *undefined behaviour*, the C++ abstract machine no longer constrains what happens next. The implementation is no longer required to preserve the behaviour encoded in the source code. The practical consequence is that, after undefined behaviour is reached, source-level reasoning no longer applies: anything can happen.

Once undefined behaviour is reachable, we can no longer rely on the source code to preserve any safety property P. Any proof that P holds depends on the execution staying within the rules of the C++ abstract machine.

For this reason, the absence of undefined behaviour is a minimal safety requirement. Other safety requirements can be guaranteed only for executions that remain within the rules of the C++ abstract machine.

Violations of memory safety properties typically lead to undefined behaviour. Thus, it is common to also place memory-safety requirements in the set of minimal safety requirements.

Safe languages

A *safe language* is not a language in which every program is partially correct. Rather, it is a language that guarantees certain classes of safety properties for programs written within a well-defined subset of the language. For example, a memory-safe language prevents certain invalid memory operations by construction: dangling references, out-of-bounds accesses, or use-after-free errors are either rejected, dynamically checked, or made unrepresentable.

In Lamport's terminology, this means that the language enforces some safety properties. It does not enforce all safety properties. A memory-safe program may still compute the wrong answer, violate a protocol, expose private data, or execute arbitrary commands. Language safety therefore reduces part of the proof burden, but it does not replace correctness reasoning.

Functional safety

Functional safety is a related concept from systems engineering. ISO 26262:2018 [ISO26262], for example, defines *safety* as the "absence of unreasonable risk", where risk is determined from the probability of harm and the severity of that harm. The important point is that functional safety is not about eliminating all risk, but about ruling out risks judged to be unreasonable in a given context. More details can be found in Andreas Weis's 'C++Now' presentation [Weis23].

At first glance, functional safety has little to do with Lamport's safety. Looking more closely, the two are compatible. In Lamport's terms, functional safety can be expressed as a set of safety requirements: certain unacceptable-risk states must not be reached. If such a state is reached, the violation is present after a finite prefix of the execution.

The presence of risk factors and probabilities does not change the shape of the property. They help determine which states are unacceptable. Once those requirements are fixed, functional safety fits naturally into Lamport's safety framework.

Proving termination or progress is still a global endeavour. But the scope is now reduced, as we have moved much of the complexity towards safety.

Composability

Composability of safety

We often hear that 'safety composes'. This is true, but the statement hides an important distinction: some safety properties compose almost directly, while others require a non-trivial mapping between local and global requirements.

Let us consider memory-safety properties as a first example. If we have two program parts, *A* and *B*, that are evaluated independently, and we find that they uphold all memory-safety properties, then if we put *A* and *B* together in a way that preserves their assumptions, we obtain a system that upholds those memory-safety properties. In many cases, the proof is relatively direct.

Now let us look at partial correctness as a safety property. If both *A* and *B* are partially correct, it does not follow immediately that the system obtained by putting them together is also partially correct. Composability does not work for partial correctness in the same direct way that it does for memory-safety properties.

The difference is not that memory safety composes while partial correctness does not. The difference is the proof burden. In Lamport's sense, safety is about preserving behavioural invariants. If *A* preserves the invariants required of it, and *B* preserves the invariants required of it, then the composition is safe only if those local invariants are compatible with the requirements of the whole system.

This is where *requirements mapping* enters the picture. The requirements on *A* are not necessarily the same as the requirements on *B*, and neither set is necessarily identical to the requirements of the whole program. To compose partial correctness, we must distribute the requirements of the system to its parts, possibly in different forms, and then show that the local invariants, together with any boundary invariants between *A* and *B*, imply the global requirement.

Safety composes. But the proof may require mapping the requirements of the whole system to the invariants preserved by its parts. This is where most of the work is.

A pivot on the meaning of liveness

Safety is local, structural, and compositional. We can build small systems, prove partial correctness in isolation, and then build more complex systems out of them. Liveness is different: it is global, behavioural, and often scheduler-dependent. Reasoning about liveness is harder, and gets harder as the system becomes more complex.

The difficulty is that liveness often seems to carry two obligations at once. We want the program to eventually produce an answer, and we want that answer to be correct. But these are not the same obligation.

This is the pivot: safety can separate the two. Instead of treating liveness in isolation and proving, for every property *P*, that it eventually holds, we can first prove partial correctness: if an answer is produced, it satisfies *P*. Only after that do we ask whether an answer is eventually produced. Partial correctness is safety, and safety composes. This means that we can reason bottom-up about whether *P* is true when we have an answer.

After the safety-related reasoning is done, we can turn our attention to liveness. But the liveness obligation has changed shape. The important point is not that liveness becomes easy, but that its role changes. We no longer need liveness to prove that the eventual answer is correct; safety already gives us that. The remaining obligation is simpler: termination, or more generally progress.

Proving termination or progress is still a global endeavour. But the scope is now reduced, as we have moved much of the complexity towards safety.

For an arbitrary program, we cannot say that liveness composes. But after safety has done the correctness work, the remaining liveness obligation is progress. This is a narrower obligation than full liveness, because it no longer has to establish the correctness of the produced result. We call this *safety-refined liveness*: liveness after partial correctness has already been established.

If progress composes, so does safety-refined liveness. Thus correctness also composes.

The key is to design programs so that progress composes.

Achieving composable correctness

We can now make the previous pivot concrete. If safety gives us partial correctness, then the remaining question is whether progress can also be made compositional.

In a concurrent program that uses mutexes as the main synchronisation primitive, progress does not compose by default. We can easily get deadlocks and livelocks [1]. Progress does not compose automatically.

By contrast, if we use task queues, also known as *serializers*, and we satisfy a few conditions, progress can be made compositional. A task queue still serialises access to some state, but it does so without requiring work items to block while holding that access.

For example, consider a program structured as a finite chain of work items: one work item parses an input, another validates the parsed representation, and another produces the final output. Each work item may enqueue the next one. If each work item is partially correct, if each work item terminates once scheduled, and if the queue eventually executes queued work, then the whole chain eventually produces a correct result.

Let us separate the assumptions we need.

- For safety:
  - work items shall be partially correct, regardless of the execution order;
  - the execution order of the work items shall not affect their safety properties;

1 Deadlocks are usually considered safety violations because they can be detected from the current execution state: the program has reached a state in which progress is no longer possible. Livelocks, on the other hand, are liveness violations: the program keeps executing, and no finite prefix is enough to prove that progress will never be made.

- For progress:
  - each work item makes progress when it is scheduled and executed (local termination);
  - at least one work item, or a finite chain of work items, produces the final answer;
  - the task queue is fair enough to eventually execute each queued work item.

If the safety assumptions are met, the task queue preserves partial correctness throughout its execution. Partial correctness composes, so we can reason about the partial correctness of the program composed of work items and work queues.

Let us look at progress guarantees. The task queue can ensure that, given a finite set of queued work items, each item will eventually be executed, assuming the queue is fair and each work item terminates. If one of those items, or a finite chain of items, produces the final answer, then the queued system produces the final answer.

Stepping back, we have just argued that progress composes for task queues under certain conditions. If the program is built from components with similar progress guarantees, then progress composes in the same way.

If a program consists only of parts for which progress composes, then the remaining liveness obligation composes. Since safety has already given us partial correctness, correctness composes under these conditions.

This gives us a design principle: **Structure programs around components for which progress composes.**

Key takeaways

Safety and liveness are not alternatives to correctness. They are ways of structuring correctness properties.

Safety is the absence of broken invariants. Once a wrong answer has been produced, undefined behaviour has been reached, or private data has been exposed (to give a few examples), the execution is already invalid. By looking at invariants, we can say that safety is local, structural, and compositional. Partial correctness is one important safety property: it rules out wrong answers.

Liveness properties state that something eventually happens. The important word is "eventually". In general, they are harder to reason about compositionally, because no finite execution is enough to show that the required event will never occur.

The useful move is to separate the two obligations. First, use safety to establish partial correctness: if an answer is produced, it is the right kind of answer. Then the remaining liveness obligation is reduced to progress: whether an answer is eventually produced.

This gives us a practical model:

Correctness = safety + progress, when safety has already captured the correctness obligations of the produced result.

The model does not make correctness trivial. Requirements still need to be identified, mapped to program parts, and preserved across boundaries. But correctness need not be treated as an all-or-nothing property of the whole program. Programs can be structured so that local reasoning is easier, progress obligations are explicit, and the path towards correctness becomes shorter.

This leads to the design principle stated above: structure programs around components for which progress composes. More generally, structure programs so that the reasoning effort is pushed towards local, explicit, and composable obligations.

Closing notes

The original motivation for this article was C++ contracts. But before asking what contracts can guarantee, we need a model of what such guarantees are guarantees of. That is what this article has focused on.

We started from Lamport's taxonomy, which divides correctness properties into safety and liveness properties, and used it to examine safety, correctness, and composability. The main point is that correctness need not be treated as a single all-or-nothing obligation. If safety captures the correctness of any produced result, then the remaining liveness obligation becomes progress. Under the right conditions, both can be reasoned about compositionally.

This does not make correctness easy. Requirements still need to be identified, mapped to program parts, and preserved across boundaries. But it does make the reasoning more tractable: correctness is not a wall that must be climbed all at once, but a path that can be broken into smaller, checkable steps.

In a follow-up article, we turn to contracts. Contracts, whether as a C++ language feature or as a discipline for documenting code, are a natural continuation of this discussion: they make some of the obligations of a program explicit, and therefore easier to reason about. ■

References

[Bauman26] Jon Bauman, Timur Doumler, Nevin Liber, Ryan McDougall, Pablo Halpern, Jeff Garland, Jonathan Müller, 'P3874R1: Should C++ be a memory-safe language?' 2026, https://wg21.link/P3874R1

[CISA23a] Cybersecurity and Infrastructure Security Agency, 'Shifting the Balance of Cybersecurity Risk: Principles and Approaches for Security-by-Design and -Default', Apr 2023, https://www.cisa.gov/sites/default/files/2023-04/principles_approaches_for_security-by-design-default_508_0.pdf

[CISA23b] Cybersecurity and Infrastructure Security Agency, 'Secure by Design', Apr 2023, https://www.cisa.gov/securebydesign

[CR23] Yael Grauer (Consumer Reports), 'Future of Memory Safety: Challenges and Recommendations', Jan 2023, https://advocacy.consumerreports.org/wp-content/uploads/2023/01/Memory-Safety-Convening-Report.pdf

[Doumler25] Timur Doumler, Gašper Ažman, Joshua Berne, P3500R1: Are Contracts "safe"?, 2025, https://wg21.link/P3500R1

[EC22] European Commission, 'Proposal for a REGULATION OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL on horizontal cybersecurity requirements for products with digital elements and amending Regulation (EU) 2019/1020', Document 52022PC0454, Sep 2022, https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A52022PC0454&qid=1703762955224

[ISO26262] ISO, ISO 26262:2018 Road vehicles – Functional safety, 2018, available from https://www.iso.org/standard/68383.html

[Lamport77] Leslie Lamport, 'Proving the Correctness of Multiprocess Programs', *IEEE Transactions on Software Engineering 2*, 1977, https://lamport.azurewebsites.net/pubs/proving.pdf

[McDougall24] Ryan McDougall, 'P3578R1: What is "Safety"?', 2024, https://wg21.link/P3578R1

[Müller25] Jonathan Müller, 'P3649R0: A principled approach to safety profiles', 2025, https://wg21.link/P3649R0

[NSA22] National Security Agency, 'Software Memory Safety', Nov 2022, https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF

[Vandevoorde26], David Vandevoorde, Jeff Garland, Paul E. McKenney, Roger Orr, Bjarne Stroustrup, Michael Wong, ,P3970R0: Profiles and Safety: a call to action,, 2026, https://wg21.link/P3970R0

[Weis23] Andreas Weis, 'Safety-First: Understanding How To Develop Safety-critical Software', *C++Now*, May 2023, https://www.youtube.com/watch?v=mUFRDsgjBrE

[WH23] White House, *National Cybersecurity Strategy*, Mar 2023, https://bidenwhitehouse.archives.gov/wp-content/uploads/2023/03/National-Cybersecurity-Strategy-2023.pdf

C++ Reflection: a Universal Printer

C++26 introduced reflection. Lieven de Cock demonstrates how to print information about class members using this new feature.

Our mission

The challenge of this article is to write a universal printer/formatter, so that we can format (nearly) any type we have. By that we mean print out the values (and names) of all its members, including the members of base classes.

Let's get started.

The Lift operator

The first thing we need to do is move from the code domain to the reflection domain, and that is done by applying the '**lift operator**' (`^^`) to a specific type.

Say we have a type `Coordinate`:

```
struct Coordinate
{
  int x{10};
  int y{20]:
  int z{30};
};
```

Let's reflect on it with `^^Coordinate`. This gives us an object of type `std::meta::info`, and this contains a lot of information we can inspect by calling several **inspection/reflection methods**.

The members of a struct

We can query for all the members of a struct/class by calling the `nonstatic_data_members_of()` method.

This method requires the `std::meta::info` we obtained by reflecting, and also a second argument. We will see later on what we can do with that second argument. For now, let's pass `std::meta::access_context::current()`.

```
constexpr auto ctx =
  std::meta::access_context::current();
nonstatic_data_members_of(^^Coordinate, ctx);
```

Let's store the outcome of that method in an array, so we can loop over it. We'll stuff them in a static array:

```
std::define_static_array(
  nonstatic_data_members_of(^^Coordinate, ctx));
```

And we can loop over it, in a range-based `for`-loop style… only we don't use a regular `for`, but the template `for` (see Listing 1).

`member` is the meta info on each member of our struct. This means we can inspect that using other methods.

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

```
constexpr auto ctx =
  std::meta::access_content::current();
template for (constexpr auto member ;
  std::define_static_array(
   nonstatic_data_members_of(^^Coordinate, ctx)))
{
  //do something with 'member'
}
```

Listing 1

A few things come to mind, which are useful for our goal:

- the **name** of the member: `identifier_of(member)`
- the **value** of the member: hmmmmm…, there is no method for this.

Alright, we cannot inspect the value during reflection (compile time) since the value will be determined at run-time, so we need to **switch back from the reflection domain to the code domain** (using the **splice operator**: `[: :]`) and use our regular specification.

So, on our instance (`co`) of our type (`Coordinate`) we need to identify the correct member – something like `t.x` or `t.y` – so `x` or `y`, but what are we looking at during the template `for` loop, we look at 'member', so let's switch that 'member' back to the code domain with the above mentioned operator: `[:member:]`, giving a statement like: `t.[:member:]`.

Putting these things together we end up with Listing 2, which gives the following output.

```
x: 10
y: 20
z: 30
```

Well, look at that. ☺ See this live at: https://compiler-explorer.com/z/31qhbxMc8

```
#include <meta>
#include <print>
struct Coordinate
{
    int x{10};
    int y{20};
    int z{30};
};
int main()
{
  Coordinate co;
  constexpr auto ctx =
    std::meta::access_context::current();
  template for (constexpr auto member :
    std::define_static_array(
      nonstatic_data_members_of(^^Coordinate,
                                ctx)))
  {
    std::println("{} : {}",
      identifier_of(member), co.[:member:]);
  }
  return 0;
}
```

Listing 2

We will print everything on one line, and members are separated by a comma, but we want to avoid a leading or trailing comma

Different types of access_context

Let's make a little change, and make the `z` coordinate 'private'.

```
struct Coordinate
{
  int x{10};
  int y{20]:

private:
  int z{30};
};
```

Now our output becomes:

```
x: 10
y: 20
```

See this live at: https://compiler-explorer.com/z/Tax1K7YbY

So, we don't get the private members anymore with our `current()` context. In order to get them back, we need to use a different `access_context: unchecked()`.

```
constexpr auto ctx =
  std::meta::access_context::unchecked();
```

And now we have our full output again.

See this live at: https://compiler-explorer.com/z/qK91qP71e

Custom formatter

With the above test program we have put down the basics for implementing a custom formatter for our `Coordinate`, and more generic for any type. So let's implement a custom formatter, which can be used as follows:

```
Coordinate co;
const auto asString = std::format("{}", co);
std::println("{}", co);
```

We need to implement two methods for a custom formatter for our type `T`:

- `parse`
- `format`

The first one is very easy. We will not support any format specifiers and as such our `parse` implementation is:

```
struct UniveralFormatter
{
  constexpr auto parse(auto& ctx) {
    return ctx.begin(); }
};
```

Next up is the `format` method:

```
struct UniversalFormatter
{
  template <typename T>
  auto format(T const& t, auto& ctx) const
  {
    // code to be inserted here
  }
}
```

Two things to observe:

- we go for fully generic `ctx` via `auto` (meaning not restricting to just `char`-based stuff)
- for the first parameter, we use the traditional template way (so we have a name for our type (`T`))

We have to print a lot of things during the format, for that purpose we make a reference to the output iterator of the format context:

```
auto out = ctx.out();
```

And several `std::format_to` calls will use it.

We will print everything on one line, and members are separated by a comma, but we want to avoid a leading or trailing comma. So, in our loop, we will start by inserting a comma, unless it is the very first member.

This can be solved with a lambda that holds the state 'to know if it is the first time or not', and puts the comma (or not) in the `out` (which it captures by reference):

```
auto delim = [first = true, &out]() mutable
{
  if (!first)
  {
    std::format_to(out, ", ");
  )
  first=false;
};
```

Putting some things together gives us Listing 3.

```
templage <typename T>
auto format(T const& t, auto& ctx) const
{
  auto out = ctx.out();
  std::format_to(out, "{}{{",
    identifier_of(^^T));
  auto delim = [first = true, &out]() mutable
  {
    if (!first)
    {
      std::format_to(out, ", ");
    }
    first = false;
  };
  constexpr auto access_ctx =
    std::meta::access_context::unchecked();
  template for (constexpr auto member :
    define_static_array(nonstatic_data_members_of
    (^^T, access_ctx)))
  {
    delim();
    std::format_to(out, ".{} = {}",
      identifier_of(member), t.[:member:]);
  }
  std::format_to(out, "}");
  return out;
 }
```

Listing 3

```
#include <meta>
#include <format>
#include <print>

struct UniversalFormatter
{
  constexpr auto parse(auto& ctx) {
    return ctx.begin(); }

  template<typename T>
  auto format(const T& t, auto& ctx) const
  {
    auto out = ctx.out();
    std::format_to(out, "{}{{",
      identifier_of(^^T));

    auto delim = [first = true, &out]() mutable
    {
      if (!first)
      {
        std::format_to(out, ", ");
      }
      first = false;
    };

    constexpr auto access_ctx
      = std::meta::access_context::unchecked();

    template for (constexpr auto member :
      define_static_array(
      nonstatic_data_members_of(^^T,
      access_ctx)))
    {
      delim();
      std::format_to(out, ".{} = {}",
        identifier_of(member), t.[:member:]);
    }

    std::format_to(out, "}}");
    return out;
  }
};

struct Coordinate
{
  int x{10};
  int y{20};
private:
  int z{30};
};

template <> struct std::formatter<Coordinate>
: UniversalFormatter { };

struct CoordinatePair
{
  Coordinate co1;
  Coordinate co2;
  double dbl{2.42};
};

template <> struct std::formatter<CoordinatePair>
: UniversalFormatter { };

int main()
{
  std::println("{}", Coordinate());
  std::println("{}", CoordinatePair());
}
```

Listing 4

In Listing 4 is the entire code example we have created and, as an extra, we created another struct **`CoordinatePair`**, which contains two **`Coordinate`** members and a **`double`**. We are printing out a **`Coordinate`** struct, and **`CoordinatePair`** struct (the latter of course requires the former to be formattable.

See it live at: https://compiler-explorer.com/z/aGTjseG11

Notice how we need to 'unlock' the formatter for each type we want to use it for:

```
template <> struct std::formatter<Coordinate>
: UniversalFormatter { };

template <> struct std::formatter<CoordinatePair>
: UniversalFormatter { };
```

It gives the following output:

```
Coordinate{.x = 10, .y = 20, .z = 30}
CoordinatePair{.co1 = Coordinate{.x = 10,
.y = 20, .z = 30}, .co2 = Coordinate{.x = 10,
.y = 20, .z = 30}, .dbl = 2.42}
```

Inheritance

When printing a derived class, we want the base class members to be included. So we need to reflect/inspect over the base classes. This can be done by fetching those via the method **`bases_of(^^T, access_ctx)`**.

Again, an **`access_ctx`** comes into play (public versus private inheritance). Re-applying the recipe for looping over the members, but now over the **`bases`**, we have:

```
template for (constexpr auto base
  : define_static_array(bases_of(^^T,
  access_ctx)))
{
  delim();
  std::format_to(out, "{}",
    (typename [: type_of(base) :] const&)(t);
}
```

We are casting our instance **`t`** to the **`base`** class. But that requires we know the type of the **`base`** class (in the code world), which is given to us by: **`type_of(base)`** and next we splice that one back into the code world. Note the keyword **`typename`**, which is needed here.

Don't forget to have custom formatters for the two base types:

```
template <> struct std::formatter<Base1>
  : UniversalFormatter { };
template <> struct std::formatter<Base2>
  : UniversalFormatter { };
```

Our updated code example is in Listing 5.

```
#include <meta>
#include <format>
#include <print>

struct UniversalFormatter
{
  constexpr auto parse(auto& ctx) {
    return ctx.begin(); }

  template<typename T>
  auto format(const T& t, auto& ctx) const
  {
    auto out = ctx.out();
    std::format_to(out, "{}{{",
      identifier_of(^^T));

    auto delim = [first = true, &out]() mutable
    {
      if (!first)
      {
        std::format_to(out, ", ");
      }
      first = false;
    };

    constexpr auto access_ctx
      = std::meta::access_context::unchecked();

    template for (constexpr auto base :
      define_static_array(bases_of(^^T,
      access_ctx)))
```

Listing 5

```
    {
      delim();
      std::format_to(out, "{}",
        (typename [: type_of(base) :] const&)
        (t));
    }

    template for (constexpr auto member :
      define_static_array(
      nonstatic_data_members_of(^^T,
      access_ctx)))
      {
        delim();
        std::format_to(out, ".{} = {}",
          identifier_of(member), t.[:member:]);
      }

      std::format_to(out, "}}");
      return out;
  }
};

struct Base1
{
  int time{100};
};

struct Base2
{
  int fifthDimension{5};
};

struct Coordinate: public Base1, private Base2
{
  int x{10};
    int y{20};
private:
  int z{30};
};

template <> struct std::formatter<Base1>
  : UniversalFormatter { };
template <> struct std::formatter<Base2>
  : UniversalFormatter { };

template <> struct std::formatter<Coordinate>
  : UniversalFormatter { };

struct CoordinatePair
{
    Coordinate co1;
    Coordinate co2;
    double dbl{2.42};
};

template <> struct std::formatter<CoordinatePair>
  : UniversalFormatter { };

int main()
{
    std::println("{}", Coordinate());
    std::println("{}", CoordinatePair());
}
```

Listing 5 (cont'd)

See this live at: https://compiler-explorer.com/z/c9znYKx44

It gives the following output:

```
Coordinate{Base1{.time = 100},
Base2{.fifthDimension = 5}, .x = 10, .y = 20,
.z = 30}

CoordinatePair{.co1 = Coordinate{Base1{.time
= 100}, Base2{.fifthDimension = 5}, .x = 10,
.y = 20, .z = 30}, .co2 = Coordinate{Base1{
.time = 100}, Base2{.fifthDimension = 5},
.x = 10, .y = 20, .z = 30}, .dbl = 2.42}
```

There's more, to be complete

What if a type does not have a name, or member does not have a name…

```
<source>:13:50: error: call to consteval function
'std::meta::identifier_of(^^Foo::<unnamed struct>)' is not a constant expression

   13 |         std::format_to(out, "{}{{", identifier_of(^^T));
      |                                     ~~~~~~~~~~~~~^~~~~

<source>:41:5: error: uncaught exception of type 'std::meta::exception'; 'what()':
'reflection with has_identifier false'

   41 |     }
      |     ^
```

Figure 1

Let's create a type with no name

```
struct Foo
{
  int a{0};
  struct
  {
    int iii{242};
  } b;
}
```

The member `b`, of `Foo`, has a type with no name. How does this print, without making any changes.

Well, it no longer compiles: https://compiler-explorer.com/z/3aYG79hWb, and see Figure 1.

We are asking for an identifier for a type which has no identifier. Let's agree, we will call it `"unnamed"` in this case. All that remains to be solved is, how do we know if there is an identifier or not ?

Maybe, just maybe, there is a method for that. Well, rest assured, there is:

```
has_identifier(...)
```

This allows us to refactor our code at the spot where we write out the name of the type, since when that type is unnamed it does not have an identifier.

```
std::string_view typeName = "unnamed";
if constexpr (has_identifier(^^T))
{
  typeName = identifier_of(^^T);
}
std::format_to(out, "{}{{", typeName);
```

The full program looks like Listing 6.

```
#include <meta>
#include <format>
#include <print>

struct UniversalFormatter
{
  constexpr auto parse(auto& ctx) {
    return ctx.begin(); }

  template<typename T>
  auto format(const T& t, auto& ctx) const
  {
    auto out = ctx.out();

    std::string_view typeName = "unnamed";
    if constexpr (has_identifier(^^T))
    {
      typeName = identifier_of(^^T);
    }
    std::format_to(out, "{}{{", typeName);

    auto delim = [first = true, &out]() mutable
    {
      if (!first)
      {
        std::format_to(out, ", ");
      }
      first = false;
    };
```

Listing 6

```
    constexpr auto access_ctx
      = std::meta::access_context::unchecked();

    template for (constexpr auto base
      : define_static_array(bases_of(^^T,
      access_ctx)))
    {
      delim();
      std::format_to(out, "{}",
        (typename [: type_of(base) :] const&)
        (t));
    }

    template for (constexpr auto member :
      define_static_array
      (nonstatic_data_members_of(^^T,
      access_ctx)))
    {
      delim();
      std::format_to(out, ".{} = {}",
        identifier_of(member), t.[:member:]);
    }

    std::format_to(out, "}}");
    return out;
  }
};

struct Foo
{
  int a{0};
  struct
  {
    int iii{242};
  } b;
};

template <> struct std::formatter<decltype
  (Foo::b)> : UniversalFormatter { };

template <> struct std::formatter<Foo>
  : UniversalFormatter { };

int main()
{
  std::println("{}", Foo());
}
```

Listing 6 (cont'd)

Giving the following output:

```
Foo{.a = 0, .b = unnamed{.iii = 242}}
```

See it live at: https://compiler-explorer.com/z/W1snKMTrd

All that is left is the case where a member does not have a name - it does not have an identifier. This use case seems to be exceptional, and it may be best to avoid it as it involves/comprises anonymous structs (illegal) and anonymous unions (legal). For that reason we are not including that case, which keeps the code simple. If you run into a need for this, it is not hard to add it. Also, watch out for bitfields.

Enumeration

Printing enumeration types is now also easy thanks to reflection.

Let's consider the following **enum** class and **struct**:

```
enum class Color
{
  Green,
  Red
};
struct Colors
{
  int x{0};
  Color col{Color::Green};
};
```

At this moment, our universal printer will again run into a compile error, since the **enum** class type **Color** has no formatter. What if it could just print out '**Green**' ?

```
struct EnumFormatter
{
  constexpr auto parse(auto& ctx) {
    return ctx.begin(); }

  template<typename E>
  requires std::is_enum_v<E>
  auto format(const E& value, auto& ctx) const
  {
    std::string_view label{"unknown"};
    template for(constexpr auto& enu
      : std::define_static_array(
      std::meta::enumerators_of(^^E)))
    {
      if (value == [:enu:])
      {
        label = std::meta::identifier_of(enu);
        break;
      }
    }
    auto out = ctx.out();
    std::format_to(out, "{}", label);
    return out;
  }
};
```

Listing 7

It can. Let's create a (trivial) formatter template for enumerations (see Listing 7.)

Why do I say trivial? Because again we don't allow format specifiers, something we could easily add by inheriting from **std::formatter<Std::string>**.

So, let's analyze the code. Some familiar concepts and a new one pop up:

- **enumerators_of()** gives all the enumerators of an enum
- stuff them into a static array, and loop over them
- one such enumerator has an identifier, obtained by **identifier_of()**
- check the enumerator at hand (enu) with the actual value, so we need to splice that enumerator to bring it back in the code domain
- we protected ourselves for 'integer values not corresponding to any enumerator', which will result in 'unknown'.

The entire code example for this looks like Listing 8 (next page), giving the output:

```
Colors{.x = 0, .y = 0, .z = 0x7fff94787d40,
.col = Green}
Red
unknown
```

See it live at: https://compiler-explorer.com/z/qMWEhsd76

Note, that we also stuffed a reference in our struct, works fine. However, a pointer will not work; it will need some extra care. This is due to the fact that we would have, for example, an **int***, and format only allows (**const**) **void***. Let's consider this a home work assignment.

Recap

We saw how we can achieve things we have wanted for a long time, in a rather trivial way by reflection. Just imagine, what else can be achieved.

So, we learned about:

- lift operator (**^^**)
- splice operator (**[: :]**)
- **define_static_array**
- **nonstatic_data_members_of**
- **bases_of**
- **identifier_of**

- `type_of`
- `access_context` (`current()` and `unchecked()`)
- `has_identifier`
- `enumerators_of`

With that we conclude our first examples of reflection. Enjoy. ■

```
#include <meta>
#include <format>
#include <print>

struct EnumFormatter
{
  constexpr auto parse(auto& ctx) {
    return ctx.begin(); }

  template<typename E>
  requires std::is_enum_v<E>
  auto format(const E& value, auto& ctx) const
  {
    std::string_view label{"unknown"};
    template for(constexpr auto& enu
      : std::define_static_array(
      std::meta::enumerators_of(^^E)))
    {
      if (value == [:enu:])
      {
        label = std::meta::identifier_of(enu);
        break;
      }
    }
    auto out = ctx.out();
    std::format_to(out, "{}", label);
    return out;
  }
};

struct UniversalFormatter
{
  constexpr auto parse(auto& ctx) {
    return ctx.begin(); }

  template<typename T>
  auto format(const T& t, auto& ctx) const
  {
    auto out = ctx.out();

    std::string_view typeName = "unnamed";
    if constexpr (has_identifier(^^T))
    {
      typeName = identifier_of(^^T);
    }
    std::format_to(out, "{}{{", typeName);

    auto delim = [first = true, &out]() mutable
    {
      if (!first)
      {
        std::format_to(out, ", ");
      }
      first = false;
    };
```

Listing 8

```
    constexpr auto access_ctx
      = std::meta::access_context::unchecked();
    template for (constexpr auto base
      : define_static_array(bases_of(^^T,
       access_ctx)))
    {
      delim();
      std::format_to(out, "{}",
        (typename [: type_of(base) :] const&)
        (t));
    }
    template for (constexpr auto member :
      define_static_array(
      nonstatic_data_members_of(^^T,
      access_ctx)))
    {
      delim();
      std::format_to(out, ".{} = {}",
        identifier_of(member), t.[:member:]);
    }

    std::format_to(out, "}}");
    return out;
  }
};

enum class Color
{
  Green,
  Red
};

template <> struct std::formatter<Color>
  : EnumFormatter { };

struct Colors
{
  int x{0};
  int& y{x}; // look a reference --> works,
             // but not for pointer, needs extra
             // care
  //int* z{&x};
  void* z{&x};
  Color col{Color::Green};
};

template <> struct std::formatter<Colors>
  : UniversalFormatter { };

int main()
{
  std::println("{}", Colors());

  std::println("{}", Color(Color::Red));
  std::println("{}", Color(242));
}
```

Listing 8 (cont'd)

Write for us!

CVu and *Overload* rely on article contributions from our members. That's you! Without articles, there are no magazines.

What do you have to contribute?

- What are you doing right now?
- What technology are you using?
- What did you just explain to someone?
- What techniques and idioms are you using?

For further information, contact the editors: cvu@accu.org or overload@accu.org

A Coalescing Event Queue for High-Frequency Systems

Traditional queues collect every event. Ajay Pandey shows how a coalescing queue can avoid redundant updates while keeping relevant state.

High-frequency event-driven systems often ingest more updates than downstream components can usefully process. A conventional FIFO queue preserves every event, but that can be the wrong abstraction when the consumer ultimately needs a current view of state rather than a complete history of every intermediate transition. This article describes a coalescing event queue: a design pattern that buffers events over short, bounded time windows and applies explicit merge policies to reduce redundant updates while preserving the most relevant state. The design uses a multi-producer, single-consumer concurrency model, key-based coalescing, policy-driven merge rules, and a decoupled dispatch layer. A C++ sketch illustrates how the queue can be implemented with clear ownership, bounded synchronization, and extensible merge semantics.

The problem with treating every event equally

Many systems are built around the assumption that every event should be queued, preserved, and processed in arrival order. That is a reasonable default for audit trails, ledgers, command streams, transactional pipelines, and any workflow where the complete sequence is semantically important.

It is not always the right default.

In a large class of high-frequency systems, producers emit repeated updates for the same logical entity. A device reports its current status. A service reports its current health. A distributed component publishes its latest state. A measurement source emits another reading for the same key. If ten updates for the same key arrive in a short interval, the downstream component may not need all ten. It may only need the latest value, or perhaps a small aggregate derived from the interval.

A FIFO queue cannot express that distinction. It treats an obsolete intermediate update and the current update as equally important. Under burst load, that can create several problems:

- queue growth caused by redundant intermediate state;
- rising end-to-end latency;
- wasted serialization, dispatch, and processing work;
- consumers observing stale values because they are still draining old updates;
- unpredictable tail latency during bursts.

Ajay Pandey is a software engineering leader and distributed systems architect specializing in modern C++, concurrency, and high-frequency event-driven systems. He has over two decades of experience developing C++ backend application servers for Fixed Income trading systems and Intelligent Network platforms in the telecom industry. His interests include scalable architectures, low-latency engineering, event coalescing, and resilient real-time platforms. You can reach him at ajaypandey1@gmail.com or through his LinkedIn profile: https://www.linkedin.com/in/ajay-pandey-76158310/

The central observation behind a coalescing queue is simple:

> In many high-frequency systems, state convergence matters more than event completeness.

The goal is not to process less because correctness is unimportant. The goal is to encode the correct notion of importance. If a newer update supersedes an older one, the system should be able to say so explicitly.

Coalescing as queue semantics

An event coalescing queue temporarily stores incoming events for a bounded time window. During that period:

1. coalescing queue groups related events based on their logical key, and
2. a coalescing queue applies a merge policy to combine older updates with newer state transitions, thereby reducing redundant intermediate events.

Design questions to consider are the following:

1. **Key selection**: How is it determined that two events can be coalesced?
2. **Coalescing strategy**: In case there is a common key among events, how should they be coalesced?
3. **Flush strategy**: At which point should buffered events be flushed?
4. **Concurrent publish strategy**: How can several publishers publish messages to a queue to avoid complexity for concurrent coalescing?
5. **Exceptions**: The reasons why messages will not participate in the coalescence process. Basically, events that will bypass coalescing logic.

There are several ways that could be considered here, starting from a simple winner-take-all approach, in which case if there was a message buffered for a key, all incoming messages for this specific key were ignored. Also, other approaches could be considered, such as aggregation, keep last, merge fields, and batching.

This is not just optimization of a regular process; this is a completely new contract for our queue. Whereas a FIFO guarantees only proper message ordering, a coalescing guarantees a certain level of state propagation with limited latency.

System architecture

A typical coalescing queue has four layers: **event producers**, an **ingestion layer**, a **coalescing engine**, and a **dispatch layer**. Event producers are listed as part of the overall architecture, but they are usually outside the queue implementation itself.

Event producers produce events concurrently, and an **ingestion component** receives events. The **coalescing engine** manages the pending state of the coalesced events according to a defined merge policy for the

The concurrency model should simplify the most difficult aspect of the design: merge correctness

events having the same logical key. Finally, once events get processed, the **dispatch component** sends them for processing further downstream.

It is critical to have the separation between ingestion, coalescing, and dispatch components. Producers must not be blocked by expensive processing tasks. The coalescing engine must manage its pending state clearly. Dispatch must be independent of merge logic to prevent transport-specific implementation affecting the coalescing process.

Figure 1 shows the coalescing event queue system architecture.

Choosing the concurrency model

The concurrency model should simplify the most difficult aspect of the design: merge correctness. A practical default approach is a multi-producer, single-consumer model in which multiple producer threads invoke `enqueue()` while a single worker thread owns the pending state map and executes merge policies. Dispatch operations occur outside the producer path so that downstream processing does not interfere with event ingestion. This approach provides two major advantages. First, producer-side operations remain short and predictable, helping maintain stable ingestion latency under burst traffic. Second, merge state ownership remains centralized, reducing the risk of subtle race conditions during policy evaluation.

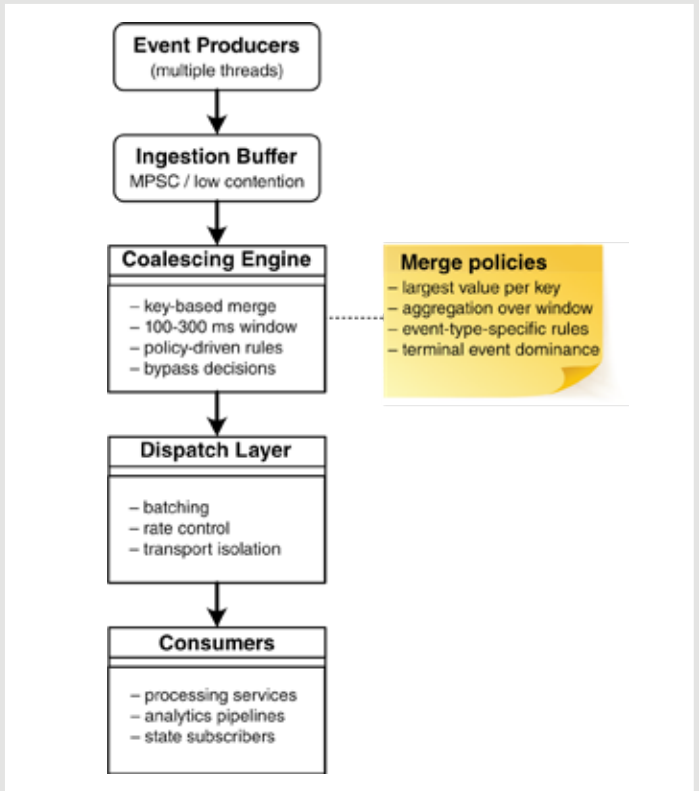


Figure 1

There are different ways to realize such an implementation approach. A system can either protect the pending map with a mutex, protect the vector with a mutex but use indexing by keys, utilize an MPSC queue processed by a dedicated consumer thread, or work with the lockfree ring buffer that supplies the events to a single coalescer. The implementation example provided uses standard C++ constructs just to demonstrate the essence of design rather than get involved in optimizations. Later on, in a production use case, the ingress path might be implemented using lock-free constructs.

A C++ implementation sketch

The next implementation is deliberately designed to serve as a high-level conceptual design of the library in question. The main aim is to outline ownership rules, merge logic, and concurrency strategies.

In order to implement three central ideas, three classes are introduced.

Event structure

The `Event` structure serves as the basic structure that will be processed by the coalescing queue. In practical use cases, the events will have IDs, data, timestamps, statuses, or other information necessary for the coalescing queue to know how the updates need to be coalesced. The simplified version of the event structure utilized within this document was made with the intent of providing just the basic pieces needed to show coalescing based on keys and how to apply merge policies to the update process.

```
struct Event {
  using Key = std::string;

  Key key;
  std::string payload;
  bool bypass_coalescing = false;
};
```

"Latest value wins" merge policy

The `LatestValueWins` structure is an example of a merge policy functor. Ownership transfer is achieved via passing-by-value of the newer `event` because it is intentional that the ownership transfer process should be made efficient for merging processes. This enables the merge policy to perform the necessary payload transfer when needed.

The `MergePolicy` describes a strategy according to which two events with identical keys will be merged together.

```
struct LatestValueWins {
  Event operator()(Event const& older,
                   Event newer) const {
    return newer;
  }
};
```

Coalescing queue implementation

The `CoalescingQueue` handles the processing part, including buffering events and merging them.

The important property is that the queue semantics remain explicit and understandable

The helper function **upsert_locked()** uses the common **upsert** convention: it **inserts** a new pending event when the key is not present, or **updates** the existing pending event when the key is already present.

See Listing 1.

```
#include <chrono>
#include <condition_variable>
#include <functional>
#include <mutex>
#include <string>
#include <thread>
#include <unordered_map>
#include <vector>

template <typename MergePolicy,
          typename Dispatch>
class CoalescingQueue {
public:
  using Clock = std::chrono::steady_clock;
  using Duration = Clock::duration;

  CoalescingQueue(Duration window,
                  MergePolicy merge_policy,
                  (Dispatch dispatch)
    : window_(window),
      merge_policy_(std::move(merge_policy)),
      dispatch_(std::move(dispatch)),
      worker_(&CoalescingQueue::run, this) {}

  // A queue owns a live worker thread and
  // synchronization state shouldn't be
  // accidentally copied or moved.

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

  ~CoalescingQueue() {
    close();
  }
  void enqueue(Event event) {
    std::vector<Event> immediate;
    {
      std::lock_guard<std::mutex> lock(mutex_);

      if (closed_) {
        return;
      }

      if (event.bypass_coalescing) {
        immediate.push_back(std::move(event));
      } else {
        upsert_locked(std::move(event));
      }
    }
```

Listing 1

```
    if (!immediate.empty()) {
      dispatch_(std::move(immediate));
    }

    cv_.notify_one();
  }

  void close() {
    {
      std::lock_guard<std::mutex> lock(mutex_);
      if (closed_) {
        return;
      }
      closed_ = true;
    }
    cv_.notify_one();

    if (worker_.joinable()) {
      worker_.join();
    }
  }

private:
  void upsert_locked(Event event) {
    auto iter = index_.find(event.key);

    if (iter == index_.end()) {
      auto position = pending_.size();
      index_.emplace(event.key, position);
      pending_.push_back(std::move(event));
      return;
    }
    auto& existing = pending_[iter->second];
    existing = merge_policy_(existing,
                             std::move(event));
  }

  std::vector<Event> take_pending_locked() {
    std::vector<Event> result;
    result.swap(pending_);
    index_.clear();
    return result;
  }

  void run() {
    std::unique_lock<std::mutex> lock(mutex_);

    while (true) {
      cv_.wait(lock, [this] {
        return closed_ || !pending_.empty();
      });

      if (closed_) {
        break;
      }
      cv_.wait_for(lock, window_, [this] {
        return closed_;
      });
      auto batch = take_pending_locked();
      lock.unlock();
```

Listing 1 (cont'd)

...coalescing is not simply about dropping events. It is about applying explicit semantics to determine which representation of state should survive...

```
      if (!batch.empty()) {
        dispatch_(std::move(batch));
      }
      lock.lock();
    }
    auto final_batch = take_pending_locked();
    lock.unlock();

    if (!final_batch.empty()) {
      dispatch_(std::move(final_batch));
    }
  }

  Duration window_;
  MergePolicy merge_policy_;
  Dispatch dispatch_;

  std::mutex mutex_;
  std::condition_variable cv_;
  bool closed_ = false;

  std::vector<Event> pending_;
  std::unordered_map<Event::Key, std::size_t>
    index_;
  std::thread worker_;
};
```

Listing 1 (cont'd)

Example usage

Listing 2 is an example of usage. After the coalescing window expires, the dispatched batch contains only the latest event for `component-a`.

Commentary on the design

The most important characteristic of this implementation is not raw performance. The important property is that the queue semantics remain explicit and understandable.

```
auto dispatch = [](std::vector<Event> batch) {
  // Serialize and forward the reduced batch.
};

CoalescingQueue queue{
  std::chrono::milliseconds{200},
  LatestValueWins{},
  dispatch
};

queue.enqueue(Event{.key = "component-a",
  .payload = "state=starting"});

queue.enqueue(Event{.key = "component-a",
  .payload = "state=ready"});

queue.enqueue(Event{.key = "component-b",
  .payload = "load=0.72"});
```

Listing 2

Merge policy injection

The queue itself does not need to understand application semantics. The merge policy decides whether events should be replaced, aggregated, merged field-by-field, or preserved based on lifecycle state.

For example, a terminal-state-preserving policy might look like this:

```
struct TerminalStateDominates {
  Event operator()(Event const& older,
                   Event newer) const {

    if (older.payload == "state=closed") {
      return older;
    }

    if (newer.payload == "state=closed") {
      return newer;
    }
    return newer;
  }
};
```

In production systems, merge policies would typically operate on strongly typed event fields rather than parsing string payloads.

Dispatch outside the lock

The worker thread extracts pending state, clears internal structures, and releases the mutex before dispatching downstream. This separation is essential because dispatch operations may involve serialization, transport I/O, allocation, or external callbacks.

Holding queue locks during dispatch would couple producer latency to downstream performance and significantly reduce scalability.

Explicit bypass semantics

Not every event should be coalesced. Certain events represent commands, audit records, lifecycle transitions, or non-idempotent operations.

The `bypass_coalescing` flag demonstrates one possible mechanism for preserving such events independently from coalesced state updates.

Graceful shutdown

The destructor invokes `close()`, allowing the worker thread to drain pending events before shutdown. Without this behavior, the final coalescing window could be silently lost.

Merge policy comparison

Different merge policies encode different notions of correctness (see Table 1, on next page).

These policies demonstrate that coalescing is not simply about dropping events. It is about applying explicit semantics to determine which representation of state should survive within a bounded window.

Merge Policy	Core Behavior	Preserved Information	Typical Use Case
Latest Value Wins	Replaces older events with newer events for the same key	Most recent state	Intermediate values become obsolete once a newer value arrives
Terminal State Dominates	Preserves terminal lifecycle states	Completion, closure, failure semantics	Certain states must never be overwritten within a window
Aggregate Over Window	Combines multiple events into computed summaries	Aggregate activity within the interval	The interval itself carries semantic meaning

Table 1

Aggregate policy example

An aggregate policy may compute summary statistics such as averages, counts, sums, or minimum and maximum values. (See Listing 3.)

The identical input sequence may result in different output sequences based on the merge policy chosen. For example, consider a series of events corresponding to the same key arriving in the same coalescing window in the following order: `K:10`, `K:20`, `K:30`, followed by `K:closed`. With the `Latest Value Wins` approach, the final state `K:closed` will be kept because it replaces each preceding state with a new one.

The `Terminal State Wins` policy, however, will maintain the `K:closed` state for a different reason, as terminal lifecycle states are purposely shielded from overwriting by future nonterminal events. If the `Aggregate Over Window` approach is adopted, the numeric events will be aggregated into a single summary representation, `K:count=3, avg=20`.

This example clearly illustrates one of the essential guidelines: merge policies must be designed based on event semantics rather than just the shape of the event.

Performance characteristics

The practical impact of coalescing becomes visible under sustained high-frequency workloads.

Table 2 is a theoretical representative rather than absolute, but it illustrates the key property of coalescing: system cost becomes state-driven rather than event-volume-driven.

```
struct NumericEvent {
  std::string key;
  double sum = 0.0;
  std::size_t count = 0;
};

struct AverageOverWindow {
  NumericEvent operator()(NumericEvent older,
      NumericEvent newer) const {
    older.sum += newer.sum;
    older.count += newer.count;
    return older;
  }
};

double average(NumericEvent const& event) {
  return event.count == 0
    ? 0.0
    : event.sum / event.count;
}
```

Listing 3

Metric	FIFO Queue	Coalescing Queue
Events Ingested/sec	100000	100000
Events Dispatched/ sec	100000	4,000 – 8,000
Redundant Events Processed	High	Low
Average Latency	High under burst load	Stable and bounded
Tail Latency	Unpredictable spikes	Predictable
CPU Utilization	Higher	Lower
Network / I/O Overhead	High	Reduced
Memory Growth	Potentially unbounded	Bounded by active keys
Consumer Freshness	Often stale	Near real-time

Table 2

Window size and latency

The coalescing window is the first line of defense when considering latency/throughput tradeoffs. With smaller windows, there will be lower queue delays but fewer opportunities for merge reduction. With large windows, merge efficiency is better during traffic bursts, albeit with higher propagation latency. Hence, the ideal window size is determined by several criteria, such as the degree of input volatility, freshness needs downstream, queue delay tolerance, and efficiency of merge. Moreover, the coalescing queue should not be taken as a black box but be measured appropriately. For instance, useful performance metrics are event acceptance count, emission count, replacement count, bypass count, flush frequency, distribution of batches, dispatch duration, and queue drain characteristics.

Backpressure and capacity management

While coalescing helps reduce pressure downstream, it does not address the issue of capacity management.

The memory usage of a coalescing queue depends mainly on the number of keys that are active in the coalescing window and not on the number of events being processed. Since all events with a common key get coalesced, there cannot be more than one pending item per key.

There is one major benefit that coalescing queues offer over a FIFO approach. However, a workload involving very large numbers of keys may still pose problems with respect to memory pressure.

Some common methods of managing capacity are to reject events and create backpressure, to drop only lossy events, to flush earlier due to increased internal pressure, to distribute keys across several coalescers, or to spill important events to durable storage. The correct method would depend purely on event semantics.

Ordering guarantees

Coalescing disrupts standard semantics of queue ordering.

The example above maintains ordering for the first appearance of the key in the context of a coalescing window. For instance, if there is a sequence of key `A`, followed by key `B`, followed by `A`, then the output order will be `A, B`, and the data of `A` will be updated in place.

Other implementation options would delete the original `A` and enqueue the new version of the event to the end of the batch.

Neither implementation works in all cases. The critical part is to document the guarantee on the queue explicitly.

It should be clear which type of ordering a coalescing queue provides: does it provide the latest value of a key? The initial position of the key during its lifetime in the window? Latest position? Terminal lifecycle state? Aggregations of many events? The whole set of events?

Documenting the semantics explicitly is crucial since the consumer of this queue might get the wrong idea about what is going on with the queue and how it works.

Scaling the design

The single-worker model is often sufficient because expensive downstream work has already been reduced through state convergence.

If a single coalescer starts lagging or becomes too slow, the next step is partitioning by key:

```
partition = hash(key) % number_of_partitions;
```

Each partition maintains its own queue, worker thread, pending map, and dispatch path. Partitioning introduces several trade-offs. Uneven key distribution may create hot partitions, cross-key merge policies become more difficult, ordering guarantees become partition-local, and shutdown and observability become more complex. For this reason, partitioning should generally be introduced only after measured contention.

When not to coalesce

Coalescing should not be done if each and every event is meaningful on its own.

The examples include auditing logs, accounting transactions, legal documentation, control data, event sourcing implementations, cumulative counts, and exact replay processes.

Another design error would be doing coalescing at too low a level in the infrastructure stack where the system cannot ascertain if merging these events can be safely performed.

Advanced variations

The basic design can be extended in several ways.

- **Adaptive Windows**: In case of bursts, the coalescing window could grow, whereas for steadystate operation, the window size could shrink. Such an approach would enhance the efficiency of coalescing but make it harder to predict latency.
- **Priority Flush**: Important high-priority events, or certain terminal events, could result in immediate flushing instead of waiting until the expiration of the whole coalescing window. This would allow for immediate propagation of significant lifecycle changes or events.
- **Structured Merge Policies**: Rather than merging unstructured data payloads, structured event types with explicit properties, such as timestamps, sequence numbers, severity level, or lifecycle state, are used in practice. These structured event formats simplify the implementation of merging policies and preserve semantic details in coalesced data.
- **Lock-Free Ingestion**: To support extremely high throughput of events, producers would publish events directly to a lock-free MPSC queue, and a separate worker would drain that queue and insert events to the private merge map.

This ensures minimal producer interference along with maintaining the single-owner merge semantics.

Testing the coalescing queue

A coalescing queue should be tested at both the semantic and concurrency levels. The first group of tests should verify merge behavior. For example, a `LatestValueWins` policy should emit only the newest event for a key within the window, while a terminal-state policy should preserve terminal states even when later non-terminal updates arrive. These tests should use deterministic inputs and short controlled windows.

The second group of tests should verify flush behavior. Tests should confirm that pending events are emitted after the configured window, that bypass events are dispatched immediately, and that shutdown drains any remaining pending events. These cases are important because most correctness bugs in coalescing queues occur at timing boundaries.

Concurrency tests should run multiple producer threads that call `enqueue()` while a single worker drains the queue. The expected result is not necessarily FIFO ordering of every event, but preservation of the documented coalescing contract. Stress tests should also measure whether events are lost during close, whether duplicate keys are merged correctly under load, and whether dispatch occurs outside the queue lock.

Conclusion

A coalesced event queue is applicable where there are frequent updates being generated by a system but where downstream consumers only care about the current state and not every step in between.

This architecture should not be mistaken as a substitute for FIFO queues, persistent logging, and event sourcing. Rather, it's an alternative approach that works better with state-based operations.

One of the key decisions in implementing this system is making sure that the semantics of the merger are explicitly known. After the merger semantics have been defined, the rest of the implementation remains straightforward: multiple producers generate updates, one merging engine manages the pending state, a time window manages the latency period, and a dispatcher routes reduced events downstream.

With proper implementation, this system will save effort, reduce latency spikes, improve efficiency, and ensure that only important information is passed through. ■

Disclaimer

The technical design patterns that are discussed in this paper are meant to be generic and used as discussion topics. This discussion does not include any copyrighted software or an actual employer's software infrastructure or confidential implementation.

Further reading

1. 'Types of message conflation' in *Diffusion 6.12.0 User Manual*, availabe online at https://docs.diffusiondata.com/docs/latest/manual/html/designguide/data/conflation/conflationtypes.html
2. Marco Terzer, 'Conflation Queues: Protecting High-Freq. Systems During Peak Loads', posted 31 May 2028, available at https://www.linkedin.com/pulse/conflation-queues-protecting-high-frequency-systems-marco-terzer
3. StreamNative, 'Latency Numbers Every Data Streaming Engineer Should Know' posted 24 September 2025 and available online at https://streamnative.io/blog/latency-numbers-every-data-streaming-engineer-should-know
4. Martin Thompson, 'Inter Thread Latency', posted 9 August 2011 on the blog *Mechanical Sympathy*' https://mechanical-sympathy.blogspot.com/2011/08/inter-thread-latency.html
5. Herb Sutter, Lock-free programming and concurrenty articles, posted on *Sutter's Mill* https://herbsutter.com/category/concurrency/
6. Maged M. Michael (2002), 'High performance dynamic lock-free hash tables and list-based sets' in *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82, available at https://dl.acm.org/doi/10.1145/564870.564881
7. Dmitry Vyukov (no date) 'Bounded MPMC Queue' on *1024cores*, available at https://sites.google.com/site/1024cores/home/lock-free-algorithms/queues/bounded-mpmc-queue
8. LMAX Disruptor: High Performance Inter-Thread Messaging Library https://lmax-exchange.github.io/disruptor/

Afterwood

Milestones are important. Chris Oldwood reflects on some of those he considers worth celebrating.

It is a tale, told by an idiot, full of sound and fury, signifying nothing.
~ William Shakespeare, Macbeth

It started with a conversation in the pub. With me, it often starts that way. However, there is some debate between me, and the editor of this journal, about which pub it was. I maintain it was The Chandos in Charing Cross after an ACCU Christmas pizza, whereas she reckons it was The Last Pub Standing in Norwich during the *Norfolk Developers Conference*. At this point, my wife would typically step in and point out that I'm the least reliable witness ever as I can never remember dates, places, faces, etc. And she'd be right; my memory feels more akin to a level 1 CPU cache than an NVMe solid state drive.

No matter, the place is a mere detail, it's more the topic of conversation that's relevant here. Whilst chewing the proverbial fat over a pint, I lamented the death of the printed programming journal and, in particular, the loss of the irreverent column that typically concluded each issue. Ever on the lookout for new content, our illustrious editor suggested I take up this mantel for *Overload* and, clearly on a roll, also proposed the puntastic title 'Afterwood' as a wonderful play on my surname. Even without a King's Shilling dropped into my pint, I still cautiously accepted the challenge.

That all happened a decade ago, and so this is my tenth anniversary article. A milestone such as this felt ripe for some deeper reflection on whether I had achieved what I originally set out to do. And then I read my opening entry again and remembered the only objective I really had was not to write about anything deeply technical and additionally try and write pieces of a more whimsical nature, but with at least a tangential relationship to the world of software development. (The journal might well be titled 'overload' but there must be some limits.)

During my tenure, I've managed to write a fictional story about the Leftpad debacle, invent a game show based on classic programming gotchas, riffed on the terms 'tab', 'thread' and 'get' in popular culture, penned a poem about risks to delivery, and wrote a letter to Santa. Hopefully each piece has been thoughtful or entertaining enough to provide a gentle close to a journal which has a tendency to send your head into a spin with some magical template meta-programming!

Ten years feels like a long time, and it is, but as I write I'm watching Sir David Attenborough celebrate his 100th birthday. Now that is a seriously impressive milestone, even in modern times. For someone who has achieved so much and is such an icon of conservation, a shindig at the Royal Albert Hall is the least we can do to celebrate his amazing life so far.

Curiously that's twice now that I've written about milestones in a positive light, and yet it feels weird saying that. Software Development has this nasty habit of taking positive terms from The Real World™ and putting a negative spin on them – consider 'legacy' and 'inheritance' as two classic examples where you'd normally welcome them with open arms, but not in your codebase.

When I started my professional programming career, I worked for a small software house that produced shrink-wrapped applications. Each release was a big event, with the occasion typically signified with a single digit increase in the 'major' part of the product's version number. The inevitable 'crunch', which preceded this milestone, only served to heighten the relief we felt on the flipside. This was an era where the physical media of floppy disks and manuals had a significant impact on the schedule and approach to quality assurance.

For many of us programmers these days, where delivery comes in the form of bits and bytes passed back-and-forth across the wire, and updates trickle out to consumers, there are less opportunities for any kind of fanfare. The rise of the Internet and Agile movement has allowed us to smooth out those peaks and troughs, and replaced them with a delivery style that focuses on a steady stream of change – continuous integration, continuous testing, continuous delivery, continuous deployment, etc. If you're not doing everything, all at the same time, are you even doing it right? (All the major web browsers are currently floating around the v150 mark, which must be having a detrimental effect on the price of Champagne and canapés in Silicon Valley.)

I think the last *truly* hard deadline I ever faced was the turning of the Millenium as I, like so many others in our field at the time, worked on remediating systems – either by patching, or in my case, largely rewriting – so they would accommodate the new era. Since then, no deadline I've faced has been *that* hard, at least in the same immovable sense, despite what The Management might have tried to suggest, because we all know 'no plan survives contact with the enemy'. Even teams supposedly following the rules of Scrum don't actually abort sprints when it's clear they aren't going to meet their 'commitments'. Instead, we complete the current timebox the best we can, and regroup for the next one – eat, sleep, plan, repeat.

I think it was James Coplien who I initially heard remark: "after the first compile it's all just maintenance". It was at a time when the industry was getting its head around the idea of incremental change, and statements like these were a reaction to the project-oriented mindset of the time. It feels like this overly reductive approach to feature development has had the side-effect of also diminishing our achievements to the point that celebrating them can seem a little churlish. It's nice to work in teams that have a habit of rejoicing in small victories, I just miss the days when we used to celebrate the big ones too, even if they were only big and memorable because of our inability to manage the accidental complexity.

Next year sees *Overload* reach another milestone and I look forward to celebrating its continued existence. We might have missed the boat on booking the Royal Albert Hall but I'm sure we can raise either a physical or metaphorical glass to the achievement. ■

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

professionalism in programming

Monthly journals, available printed and online

Discounted rate for the ACCU Conference

Email discussion lists

Technical book reviews

Local groups run by ACCU members

ACCU is a not-for-profit organisation.

Become a member and support your programming community.

www.ACCU.org

accu

Professionalism in Programming

Professional development
World-class conference

Printed journals
Email discussion groups

Individual membership
Corporate membership

Visit accu.org for details