overload 192

APRIL 2026 £4.50

Let the Compiler Check Your Units

Wu Yongwei takes a quick look at C++ unit libraries that can help keep everything in order.

Elements of Concurrency

Lucian Radu Teodorescu shows how concurrency actually gives a strict partial ordering.

Automatic Release of Resources in C

Alison Chaiken shows how cleanup macros can help.

C++ Standard Library: A Matter of Import

Ian Bruntlett documents what he did to get modules working with an older gcc compiler.

Vulture Culture

Teedy Deigh spends time with us trying to find out how you get companies, products and rockets off the ground.

A magazine of ACCU

ISSN: 1354-3172

CARE about
code?

passionate
about
programming?

Join ACCU

www.accu.org

April 2026
ISSN 1354-3172

Editor
Frances Buontempo
overload@accu.org

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

Advertising enquiries
ads@accu.org

Printing and distribution
Parchment (Oxford) Ltd

Cover design
Original design by Pete Goodliffe
pete@goodliffe.net
Cover photo by Tim Peck: the gardens of Chirk Castle, Wales.

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

4 **Elements of Concurrency**
Lucian Radu Teodorescu shows how concurrency actually gives a strict partial ordering.

10 **Let the Compiler Check Your Units**
Wu Yongwei takes a quick look at C++ unit libraries that can help keep everything in order.

16 **Automatic Release of Resources in C**
Alison Chaiken shows how cleanup macros can help.

18 **C++ Standard Library: A Matter of Import**
Ian Bruntlett documents what he did to get modules working with an older gcc compiler.

23 **Vulture Culture**
Teedy Deigh spends time with us trying to find out how you get companies, products and rockets off the ground.

Copy deadlines

All articles intended for publication in *Overload* 193 should be submitted by 1st May 2026 and those for *Overload* 194 by 1st July 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.

Const Variables and Other Oxymorons

Some things can seem niche or even contradictory. Frances Buontempo explores how you can turn this to your advantage.

Trying to write an editorial might be pointedly foolish, so I'll get to the point: I haven't had time. I'm finishing up my latest book [Buontempo26] and had a last minute acceptance to *nor(DEV):con* [NorDevCon], a relatively small conference in Norwich. I've been several times before, and a handful of ACCU people attend regularly. Go if you get the chance: it's great and not that expensive. I took four trains to get there, but only three to get back. Plenty of thinking time, but I was thinking about my book and various other distractions, like my upcoming talk.

My title was 'Learn how to Learn'. This was based on a cut-down version of my *Meeting C++* keynote, called 'Stoopid Questions' [Buontemp25]. Since I only had 40 minutes, I left out discussion of AI and machine learning and concentrated on how we learn, giving some ProTips for teaching. It seemed to be well received. The keynote that morning had been about imposter syndrome, something which affects many people… including conference speakers, no matter how often you have spoken previously. I was asked to be on a panel about DevOps/Platform in the afternoon. I had to leave part way through for the epic train journey home, but said yes. I managed to have a brief chat with another panelist and the person who had arranged the session just before lunch. It became apparent that I didn't have a strong opinion or angle to share, so we decided I would 'host' for a bit, asking questions. We were all introduced, so I pointed I really *was* an imposter, at least in terms of the panel, and would ask questions so I could learn more. Fortunately, I had been primed with a few pre-canned questions and the audience quickly joined in and asked their own. I have been involved in aspects of DevOps, and was even on a team called 'Platform' once. There's a difference between knowing you know little about the latest state of the art and being very aware that you have gaps in your knowledge or simply feel self-conscious standing in front of a crowd. I was definitely more nervous giving my own talk than being on the panel. Sure, I was on my own on stage during my talk and the panel meant I was part of a group, but it felt like the main difference was imposter syndrome versus being an actual imposter.

Being regarded as knowing your stuff to an extent or even being an expert might cause imposter syndrome, but there are ways to cope. Furthermore, it's a privileged position and can create amazing opportunities. Now, expertise can be quite niche. If you angle yourself to a small corner, you might end up knowing more than others. My brief sojourn into DevOps, both on the panel and during my career, meant I knew where logs were in production, could figure out why a release might have failed or why a build might have broken. Some people on the teams were only interested in the development, so I could be helpful. A small dopamine hit once in a while. Now, that was just a tiny bit of knowledge on my part. I once knew a guy called Richard Freeman. He has a PhD in cryptozoology, and has been described as the UK's leading cryptozoologist [Wikipedia-1]. If you don't know the term, that means he researches non-existent animals, such as investigating bigfoot (Sasquatch) [Wikipedia-2] or even the thylacine (Tasmanian Wolf) [Wikipedia-3]. Bigfoot might only be a folk tale, but the thylacine used to exist. Nonetheless, both might be hard to find on an expedition! Freeman's niche knowledge has given him many opportunity to go on further expeditions. Maybe you know something about a niche topic? Even if you only feel like you know a little, consider writing an article. Get in touch.

I mentioned my new book earlier. Of course, C++ isn't a niche subject, in the grand scheme of things. There are many C++ books out there, but there hasn't been a new one for a while. There are some newer books, but O'Reilly asked me, so here we are. Trying to explain the basics is much more difficult than explaining the deeper topics, which might be a surprise if you've never tried teaching people. You need to introduce terms and syntax. Something that caused discussion was a `const` variable. On the face of it, this is a contradiction or oxymoron. How can something be variable and constant at the same time? As I am sure you know, a variable can change, but if it's marked as `const` in C++ that means your code won't change it (unless you have mutable variables, which is another story). Some other code might, for example if you have a `const` reference, the thing you refer to can be changed elsewhere. These kind of terms cause beginners extra difficulties. Teaching is difficult, and learning isn't a walk in the park either.

Technology is littered with confusing, and sometimes seemingly contradictory terms. I've heard people say "rigid agile" before. I'll just leave that there. Terminals are not usually terminal. What even is static React? Is disaster recovery just wishful thinking? Does serverless really mean there is no server anywhere? As for cloud computing, well, see the article by Teedy Deigh in this edition. You could even ask why a 'computer programmer'? Though not a contradiction, it suggests you program things other than computers. Maybe a calculator, but that's like a computer. It does, after all, compute. You could even argue virtual reality is a misnomer too. It's not really real, and you do actually experience it. Naming is hard, as we know.

I wrote an excuse for no editorial back in 2019, called 'This means War!' [Buontempo19]. I mused on how some computing terms like SNAFU are rooted in the military, and encouraged readers to think about the terms they use. I said, "If you can't follow all of the strange in-house acronyms when you start a new gig, or panic while learning something new as soon as 'foo' or 'widget' gets mentioned, take a breath. You are not alone."

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 Algortithms 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.

When I was at NorDevCon one speaker, GJ Schouten, gave a talk called ‘How a new Calendar API can shape the upcoming settlement on Mars’ about the Lukashian calendar [Lukashian]. I’d not heard about it before, so the talk was very interesting. At a high level, it tries to find way to provide a calendar that works on Earth in any timezone and other planets too. During the talk, Schouten discussed analogue exercises. My brain immediately asked “What?” I usually think analogue versus digital. Of course, he meant running a mission on Earth as an analogue exercise as a counterpart of practice for a mission on Mars. Words cause confusion, but without them communication would be even harder.

As you probably know, I worked as a contractor in finance when I lived in London. Though I had read some basics about yield curves, bonds, swaps and the like, I didn’t know much. In a new environment or subject domain, I tend to make a glossary of terms, even if I only have the term initially. The definition can wait. I kept it to myself initially, but a team mate noticed and thought it was a great idea and encouraged me to share it on the Wiki. I had kept it to myself because I assumed everyone else knew what the mystery words meant and I felt embarrassed. I mentioned imposter syndrome earlier: here’s a data point. You probably aren’t the only person who has lots left to learn. You are not alone. Learning the finance terms was a challenge but fun. There were also unfamiliar computing terms. I had previously worked with embedded devices, so wasn’t used to writing code for servers. I was asked to bounce a server one day. Clearly, this didn’t mean literally, but I wasn’t 100% sure what I was being asked to do. Rather than owning up to being a clueless fraud, I asked “How?” When a colleague show me how to shut down a machine and then bring it back ‘up’ again, the meaning of bounce became clear, and I added a Wiki entry explaining how to bounce a server. Another teammate noticed and whispered that they had always wondered what ‘bounce a server’ meant. Some people might think it’s obvious, but some of us are more literal than others. I bet you have come across terms you don’t understand. Maybe you had to ‘format’ a disk at some point, many years ago. What does that mean? If you find out how, you might discover what. Be brave and ask if you’re not sure. If someone tells you it’s obvious, find someone else to talk to. We all learn by asking questions.

Bouncing servers and formatting disks might make sense to techy people, but sound odd to the uninitiated. The same happens outside tech. We were visiting my father-in-law a little while ago. Someone walked by and said “You’re not Alex.” We both simultaneously said, “Correct.” This elicited the response, “Spooky!” I am not sure what provoked our outburst, but it seemed like the obvious thing to say. What’s obvious depends on context and experience. If you come across something for the first time, your perspective will probably differ to those around you, making some exclamations seem odd or even spooky. As you spend time together, be that at work or with friends, you do build up a shared language. It’s often helpful to do this deliberately, particularly in a technical context. That’s why we have common idioms for various programming languages, such as RAII for C++. We also have patterns, and pattern languages. If I say “Singleton”, you probably know what I mean. I read some of Klaus Iglberger’s *C++ Software Design: Design Principles and Patterns for High-Quality Software*” book [Iglberger22] on my epic train journeys the other week. He covers various design patterns, and introduces the idea by pointing out how verbose it would be if you described a pattern rather than having a short, snappy term for it. Saying ‘singleton’ is much easier than describing a way to provide a single instance of something, ensuring lifetimes and initialisation are correct. I must admit, I sometimes forget which pattern is which. For example, template and strategy are muddled in my mind. If I concentrate, I can work out which is which. Simply being able to say “Let’s use template or strategy – I forget which I mean,” is simpler than trying to spell out the details in full. I guess ultimately, this is the purpose of words. Trying to describe a thing by pointing works sometimes. I can point at a table, for example. However, trying to convey an abstract idea like a colour or number is a challenge. I wonder how we manage to learn such abstract concepts. I guess you try to use new words, and with a bit of encouragement and gentle feedback gradually figure out how to use them properly.

Next time you are confronted by a seeming oxymoron or other confusing term, see it as an opportunity. Maybe you can find a better term or another way to word things. You will learn something and might help others. You might even be inspires to come out with your own conference talk or perhaps an article for *Overload*. At the very least, recognise that you might not be the only person wondering what something means. Don’t be afraid to ask, and if someone laughs at you, or claims it’s obvious, they are obviously wrong, because you didn’t understand the phrase or term. Tech is litter with terms like const variables, which can take a while to get used to, but this gives you an opportunity to learn and explore. Have fun!

References

[Buontempo19] Frances Buontempo ‘This Means War!’ (editorial), in Overload 150, April 2019. Available at https://accu.org/journals/overload/27/150/buontempo_2643/

[Buontempo25] Frances Buontempo ‘Stoopid Questions’, keynote *Meeting C++*, available at https://www.youtube.com/watch?v=zztvhcgXQco

[Buontempo26] Frances Buontempo (2026) *Introducing C++*, O’Reilly Media, Inc., details at and available from https://www.oreilly.com/library/view/introducing-c/9781098178130/

[Iglberger22] Klaus Igleberger (2022) C++ Software Design, O’Reilly Media, Inc., details at and available from https://www.oreilly.com/library/view/c-software-design/9781098113155/

[Lukashian] The Lukashian Calendar: https://www.lukashian.org/?0

[NorDevCon] nor(DEV):con conference, 26th & 27th February 2026, details at https://nordevcon.com/

[Wikipedia-1] Richard Freeman (cryptozoologist): https://en.wikipedia.org/wiki/Richard_Freeman_(cryptozoologist)

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

[Wikipedia-3] Thylacine: https://en.wikipedia.org/wiki/Thylacine

Elements of Concurrency

Concurrency suggests you can’t be sure what order instructions happen in. Lucian Radu Teodorescu shows how concurrency actually gives a strict partial ordering.

I’ve long been attracted to a quote often attributed to Einstein (whether or not he said it, it’s good advice):

> Everything should be made as simple as possible, but no simpler.

I take it as a guiding principle for intellectual work on subjects with high inherent complexity. Software is one such subject; Brooks famously called it *essential complexity* [Brooks95]. Moreover, concurrency is one of the sharpest edges of that complexity.

I have been writing about concurrency in *Overload* for years, but those articles were mostly practical: tutorials, patterns, pitfalls, and implementation techniques. Here I want to step back and talk about the *essence* of concurrency.

I will build on the work of Tony Hoare and collaborators on *the laws of programming with concurrency* [Hoare09, Hoare11, Hoare13, Hoare15] (the talks are particularly enjoyable). I’ll also borrow Leslie Lamport’s viewpoint that concurrency is about the ordering of events as presented in the ‘Time, clocks, and the ordering of events in a distributed system’ article [Lamport78].

The central thesis of this article is simple: **Concurrency is strict partial ordering**.

Once you start seeing programs as collections of work items related by a strict partial order <, much of the apparent complexity becomes a matter of making ordering constraints explicit, manipulating them algebraically, and choosing implementations that respect them.

Some parts are a bit formal, but the payoff is a shift in perspective. Rather than treating concurrency as a bag of mechanisms (threads, locks, async, executors), we treat those mechanisms as different ways of expressing and enforcing the same ordering structure.

TL;DR

- **Concurrency = strict partial ordering between work items.**
- Sequential (`;`) and concurrent (`||`) composition are just ways of adding ordering constraints.
- The exchange law is the key bridge between sequential and parallel structure; it enables modular reasoning.
- Many familiar constructs (fork–join, joins, mutexes, serialisers, schedulers) can be understood as mechanisms that add, remove, or delay ordering constraints.
- Non-local scheduling amounts to repeatedly selecting the set of currently-ready work items.
- Concurrency can be encapsulated by aligning scheduling boundaries with dependency boundaries.

What is concurrency?

A program is non-concurrent if its work items are totally ordered: for any two distinct work items `a` and `b`, either `a < b` or `b < a`.

A program is concurrent if the execution of work items is governed by a *strict partial ordering* relation. In this case, we have the following three execution possibilities:

- `a < b`
- `b < a`
- neither `a < b` nor `b < a`

The third option is specific to concurrent execution. In this case, the execution of `a` *may proceed concurrently* with the execution of `b`.

This is the essence of concurrency. Much of what we call *concurrency* either follows from this ordering view, or belongs to the choice of implementation.

By the definition of strict partial ordering, we have the following properties:

- **irreflexivity**: `¬(a < a)`
- **asymmetry**: if `a < b` then `¬(b < a)`
- **transitivity**: if `a < b` and `b < c` then `a < c`

These properties rule out cycles such as `a < b < a`, which would be an impossible execution constraint.

When we say `a < b`, we mean only that `a` is ordered before `b`. We do not assume a particular memory-visibility or synchronisation model (unlike C++’s *happens-before* [C++Ref]).

Hoare’s laws of concurrency

Hoare’s *laws of programming with concurrency* [Hoare09, Hoare11, Hoare13, Hoare15] provide an algebraic vocabulary for talking about concurrent structure without committing to a particular programming language, runtime, or hardware model. The aim is not to model every low-level effect (such as memory visibility), but to capture the high-level ordering constraints that make a program *more* or *less* concurrent (where *more concurrent* means *fewer ordering constraints*).

We will work with a small set of operators (`;`, `||`, `1`) and a refinement relation `=>` that lets us compare designs, programs, and executions. The central idea is that many familiar concurrency patterns (fork–join, barriers, mutexes, serialisers, schedulers) can be understood as ways of adding or removing ordering constraints.

This approach complements the definition above: from the structure of a term built with `;` and `||` we can read off a dependency graph (a strict partial order) between work items.

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

The aim is not to model every low-level effect (such as memory visibility), but to capture the high-level ordering constraints

Formalism

Let variables `p`, `q`, `r`, … stand for specifications, designs, programs, parts of programs or work items. They describe behaviours of a computer that are desired, planned, or actually executed when the program is executed.

We define *sequential composition*, `p ; q`, as the operation resulting from executing operation `q` immediately after operation `p`. From the definition we imply that `p < q`. A simple example of sequential composition is calling two functions, one after another, in a C-like language: `f(); g();`.

We also define *concurrent composition*, `p || q`, as the operation that executes both `p` and `q` at the same time (concurrently). That is, we express the relation `¬(p < q) ∧ ¬(q < p)` (no ordering constraint between `p` and `q`). As an example, one can imagine starting two threads at the same time, one executing `p` and the other one executing `q`.

Besides these two operations, we also define `1` as the *unit operation*, the operation that does nothing if executed.

With these defined, we have the following axioms:

- `(p ; q) ; r = p ; (q ; r)` – associativity of `;`
- `(p || q) || r = p || (q || r)` – associativity of `||`
- `p ; 1 = 1 ; p = p` – `1` is the unit with respect to `;`
- `p || 1 = 1 || p = p` – `1` is the unit with respect to `||`
- `p || q = q || p` – `||` commutes

The astute reader might have noticed that our exposition (which follows Hoare's exposition) mixes specifications with programs, and expected behaviour with actual behaviour. This is intentional. The laws described here are very powerful and work at different levels, and thus cover a lot of ground. In practice, the intended reading will be clear from context.

Having our variables stand for multiple kinds of entities allows us to make use of refinement. We say that `p` *refines* `q`, noted as `p => q`, if every execution described by `p` is also described by `q`. Equivalently, `p` allows fewer behaviours than `q`. This means that `p` is more determinate, while `q` is more abstract. Another way to view this relation is that `p` adds more constraints than `q` has.

The following are examples of `p => q` refinement:

- `p` is a program that meets specification `q`;
- `p` is a program execution of the program `q`;
- design `p` is more constrained than the design `q`.

Refinement is a weak partial order relation, thus it obeys the following axioms:

- reflexivity – `p => p`;
- anti-symmetry – `if p => q` and `q => p`, then `p = q`;
- transitivity – if `p => q` and `q => r`, then `p => r`.

An operator • is called *monotonic* if `p => q` implies `p • r => q • r` and `r • p => r • q`. Monotonicity is closely related to composability; if a system has two components combined with a monotonic operator, and we improve one component, then the entire system is better.

In our case, both `;` and `||` are monotonic. This means that:

- `p => q` implies `p ; r => q ; r` and `r ; p => r ; q`
- `p => q` implies `p || r => q || r` and `r || p => r || q`

And now, we finally reach the *exchange axiom*, which is (in Hoare's own words) a "remarkably powerful law". This axiom states:

- `(p || q) ; (p' || q') => (p ; p') || (q ; q')`

Intuitively, exchange explains when global phase ordering can be replaced by independent componentwise progress.

Putting this axiom into a graphical form, we get Figure 1 and Figure 2 (corresponding to the left and right sides of the axiom). Figure 1 refines Figure 2: it imposes all constraints of Figure 2, plus additional cross-component constraints. In our case, the ordering constraints shown in Figure 2 are `p < p'` and `q < q'`. In addition to those, Figure 1 also adds the following constraints: `p < q'` and `q < p'`.

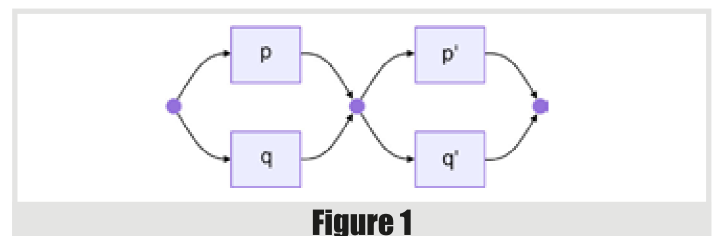

Figure 1

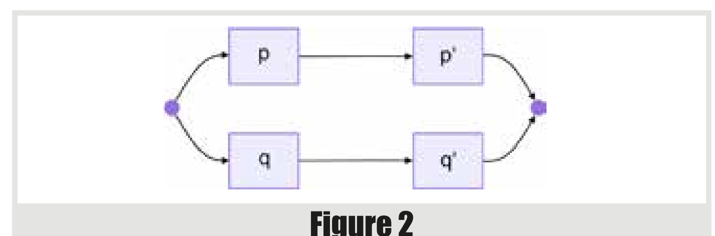

Figure 2

We've arrived at an important intuition: **there is a correspondence between laws of concurrency (as expressed in algebra) and what can be expressed by the strict partial ordering**. We are not going to prove this here.

Derived laws

Substituting `1` into the terms of the exchange law allows us to directly obtain the following laws:

- `(p || q) ; q' => p || (q ; q')`
- `(p || q) ; p' => (p ; p') || q`
- `p ; q => p || q`

Remember: `p => q` means '`p` is more constrained than `q'`. So `p ; q => p || q` says that sequential execution is a special case of concurrent execution with an extra ordering constraint.

For example, to show the first law in the first bullet, we can substitute `p'` with `1` in the exchange law, and thus obtain: `(p || q) ; (1 || q') => (p ; 1) || (q ; q')`. After simplifying `1`, we obtain: `(p || q) ; q' => p || (q ; q')`, which is exactly the law in the first bullet.

The first two bullets express that sequential progress of one component distributes through a concurrent context. The third bullet point states that purely sequential composition is more constrained than unconstrained concurrency.

Hoare also shows that the modularity rule is equivalent to the exchange law. The modularity rule says that if `p ; q => r` and `p' ; q' => r'`, then `(p || p') ; (q || q') => r || r'`. The modularity rule allows separate reasoning about two parts of a concurrent program, and then combining the results.

From `p ; q => r` and `p' ; q' => r'`, monotonicity of `||` gives `(p ; q) || (p' ; q') => r || r'`. Exchange gives `(p || p') ; (q || q') => (p ; q) || (p' ; q')`.By transitivity, we get `(p || p') ; (q || q') => r || r'`, which is the conclusion of the modularity rule.

Benefits of the exchange law

The exchange axiom is the fundamental law relating sequential and concurrent composition. Without it, the operators `;` and `||` would be algebraically independent, and no general reasoning principle could connect sequential and concurrent structure. The axiom therefore provides the essential connective tissue of the algebra.

The exchange law enables compositional reasoning about concurrent systems. We may reason about the evolution of each concurrent component independently and then recombine the results. In refinement terms, the behaviour of the whole system is determined by the componentwise progress `p < p'` and `q < q'`. This captures the core intuition of concurrency: independent components may advance independently. For this reason, the exchange law is often explained as a kind of *distributivity principle* for concurrent progress.

Exchange expresses that ordering constraints propagate locally along each component. The right-hand side constrains only `p < p'` and `q < q'`, while the left-hand side introduces additional cross-component constraints (`p < q'`, `q < p'`). Thus exchange states that a system whose components evolve independently is less constrained (more abstract) than one that enforces global phase ordering.

This supports locality: improving one component can improve the whole without global restructuring.

The law tells us that concurrency may be realised by interleaving (when the platform supports it). Also, because exchange is a refinement law, it licenses correctness-preserving program transformations. For example, parallel blocks may be reordered or fused by replacing a more constrained execution structure with a less constrained one (when the dependency graph permits it). Many optimisation patterns for task graphs and pipelines are instances of exchange.

In summary, the exchange axiom captures the essential property of concurrency: systems composed of independent components evolve componentwise.

Relations to other formalisms

The formalism presented here is a streamlined retelling of Hoare's work on Concurrent Kleene Algebra [Hoare09, Hoare11]. Hoare not only developed the algebraic laws used above, but also established their correspondence with partial-order models of concurrency, such as pomsets (partially ordered multisets of events). In that setting, a program denotes a set of event occurrences equipped with a strict partial order, and sequential and concurrent composition correspond to adding or omitting ordering constraints between them.

Hoare also connects this algebra to the compositional spirit of Hoare logic [Wikipedia-1]. In Hoare logic, reasoning scales by composing proofs along program structure; here, refinement and monotonicity ensure that improvements to subterms lift to improvements of whole terms. The exchange law plays the role that makes sequential and concurrent reasoning interact smoothly.

Hoare further relates this algebra to Milner's Calculus of Communicating Systems (CCS) [Wikipedia-2]. Both frameworks admit operational (transition-based) and partial-order interpretations, and refinement in the algebra aligns with behavioural inclusion in process calculi.

Taken together, these connections show that the same structure reappears across algebraic, logical, and operational accounts of concurrency. That convergence strengthens the claim that strict partial ordering lies close to the essence of concurrency.

Implementation strategies

Now that we've looked at the theoretical foundations, let's reinterpret a few common implementation strategies through the lens of algebra and strict partial ordering.

Fork-join model

Consider the following program [Wikipedia-3]:

```
a()
let f = spawn { b() }
c()
_ = f.await()
d()
```

This is a fork–join pattern: `b()` runs concurrently with the continuation that executes `c()`, and `await()` acts as the join.

In our algebra, this corresponds to: `a ; (b || c) ; d`, and it induces the ordering constraints:

- `a < b` and `a < c`
- `b < d` and `c < d`
- neither `b < c` nor `c < b`

(In this translation to algebra, we did not translate `spawn` and `await`, as we assume implementations can implement them at negligible costs; we only focused on the meaningful work. If, instead, we wanted to model a more accurate execution, the work of `spawn` would need to be captured in `a`, `b` and `c`, while the work for `await` needs to be captured in `b`, `c` and `d`.)

The particular surface syntax is not important. The same structure can be expressed as a library feature, with C++ senders/receivers, or with `async`/`await` [Wikipedia-4].

We assume the join does not require a blocking wait. A common implementation strategy is to ensure that whichever thread finishes last (executing either `b()` or `c()`) continues with `d()`. This may run `d()` on a different thread than `a()`, but the induced ordering constraints are unchanged.

This way of expressing concurrency closely resembles our operators for sequential and concurrent composition. This is the simplest structured-concurrency building block. Glossing over some of the detail, using a principled point of view, all concurrent programs can be built on top of this structure (leaving the proof as an exercise for the reader).

Thread spawning and joining

We can encode the same high-level pattern in C++ using `std::thread`:

```
a();
auto t = std::thread{[]{ b(); }};
c();
t.join();
d();
```

Relative to the fork–join model above, there are two important differences: 1) `join()` may block the calling thread; 2) `d()` is guaranteed to execute on the same thread as `a()`.

The second difference is mostly an implementation detail; it doesn't affect the ordering constraints we care about. The first one matters, because a blocking join introduces an additional *work item*: waiting.

Let `#` denote a blocking wait whose duration depends on the completion of some concurrent work. Then the main thread's behaviour is: `a ; c ; # ; d`, where the execution of `#` depends on the execution of `b`. One convenient algebraic representation is: `a ; (b || (c ; #)) ; d`. The dependency '`#` waits for `b`' is part of the intended interpretation of `#`, not something expressed directly by `;` and `||`.

Introducing the concept of *time/waiting* here is a slight departure from our formalism, in which time was not directly represented. We could have written the representation as `a ; ((b ; i) || (c ; #)) ; d`, where `i=1`, and add the constraints that `¬(# < i)`. This says that `#` cannot finish before the other thread is finishing, without using the notion of time. However, the reader will probably find it easier if we mention time explicitly.

This has the same ordering constraints as fork–join *plus* the fact that the main thread spends time in `#`. Two practical consequences follow:

- **Performance overhead**: the blocked thread is occupied with waiting instead of doing useful work.
- **Weaker locality**: reasoning about the main thread now depends on the termination of `b` (because `#` is coupled to `b`).

If blocking is acceptable (or performance is irrelevant), the extra work item `#` can be ignored in high-level reasoning, and the pattern reduces to the non-blocking fork–join constraints.

Mutexes

A mutex imposes an exclusivity constraint: two work items associated with the same mutex cannot be executed concurrently.

Let `M` be a mutex and let `x`, `y`, `z`, … be work items protected by it. We write `x/M`, `y/M`, `z/M`, … to indicate that these work items are mutually exclusive with respect to `M`. In other words, `x/M` is an operator that expands into `# ; x`, where `#` depends on other work items that may be executing under `M`. This is an implementation-level elaboration, not part of the core algebra. This operator adds extra ordering constraints between work items protected by it.

If `x/M` and `y/M` appear in context as `a ; x/M ; b` and `c ; y/M ; d`, then we have the constraints:

- `a < x` and `x < b`
- `c < y` and `y < d`
- `x < y` or `y < x` (exclusivity under `M`)

With more than one mutex, inconsistent constraints can arise, and thus we have a soundness issue. Consider two mutexes `M1` and `M2` and the concurrent program: `P = (x/M1)/M2 || (y/M2)/M1`.

Intuitively, this is the classic 'lock-order inversion' pattern. One way to expose the conflict in our framework is to insert units to make the phase structure explicit: `P1 = (i ; x/M1)/M2 || (j ; y/M2)/M1`, where `i=j=1` (operations that don't do anything).

From `M2`, we must have `(i ; x) < y` or `y < (i ; x)`. From `M1`, we must have `x < (j ; y)` or `(j ; y) < x`. Now apply the exchange axiom to the units and the protected actions: `P2 = (i || j) ; (x || y) => (i ; x) || (j ; y)`, which corresponds to the intuition that both protected actions can be enabled 'after' both units.

The reader can think of `P2` as a possible interleaving of `P1`, and thus a valid execution of the program. `P2` is more constraint than `P1`, thus it follows all the constraints from `P1`, plus some more:

- `i < x`
- `j < y`
- `(i ; x) < y` or `y < (i ; x)` (from `M2`)
- `x < (j ; y)` or `(j ; y) < x` (from `M1`)
- `i < y` and `j < x` (from `P2`)

Combining the last three bullets, we can reach both `x < y` and `y < x` simultaneously, yielding an impossible strict partial order. The program is unsound. In operational terms, this manifests as deadlock: there is no schedule that can satisfy all required constraints.

Like `join()`, mutex acquisition typically introduces waiting. This is because in expressions `a ; x/M ; b`, a typical implementation wants to execute `x` inline, as early as possible, before executing `b`, on the same thread as `a` and `b`. If another mutex-protected work item is executing under `M`, we need to delay the execution of `x`; the most common strategy is to perform a blocking wait. This corresponds to an execution of `a ; # ; x ; b`, where the length of `#` depends on other operations executed under `M`. This is a second way mutexes can harm both performance and locality: the waiting time is externally determined by other threads.

Serialisers

Serialisers are an alternative to mutual exclusion that avoids blocking by turning exclusivity into *queued execution*. In industry, similar ideas appear under different names (e.g., Intel oneTBB local serializer, Boost.Asio strands, and GCD dispatch queues [Intel] [AsioStrands] [Apple]).

To model a serialiser in our framework, we separate two distinct actions:

- **enqueue**: a local action that requests work to be run under the serialiser;
- **run**: the (possibly later) execution of that work under the serialiser.

Let `S` be a serialiser. For each work item `x` that is submitted to `S`, write `x/>S` for the enqueue action, and keep `x` for the eventual execution of the work. The key properties of the serialiser are:

- enqueue happens before the work can run: `x/>S < x`;
- the serialiser imposes a total order on the executions associated with `S`: for any `x`, `y` submitted to `S`, either `x < y` or `y < x`.

With this notation, we can represent 'submit `x` to serialiser `S`' as `x/>S`. That is, `x/>S` records the local enqueue; the actual execution `x` will occur later, ordered after the enqueue and ordered relative to other work items executed under `S`.

If `x/>S` and `y/>S` appear in context as `a ; x/>S ; b` and `c ; y/>S ; d`, then we have the constraints:

- `a < b` and `a < x/>S`
- `c < d` and `c < y/>S`
- `x/>S < x` and `y/>S < y`
- `x < y` or `y < x` (serialisation under `S`)

Unlike mutexes, we do *not* require `x < b` (or `y < d`). The work item under the serialiser may run later, outside the lexical scope where it was enqueued. This is precisely how serialisers can avoid blocking waits: the producer proceeds, while the serialiser schedules the work item when it becomes eligible.

This non-local execution also avoids the deadlock pattern above: the program `(x/>S1)/>S2 || (y/>S2)/>S1` can enqueue both work items without requiring an immediate consistent ordering between their executions. The local work of this program consists of the enqueue actions `x/>S1` and `y/>S2` (which are not blocking waits), while the executions `x` and y occur later, outside the lexical scope.

Serialisers therefore depart from the purely local patterns above: they introduce *non-local concurrency*, because work items execute

outside the scope in which they were created. Handling such dynamic scopes is outside the scope of this paper; readers can consult C++26's `async_scope`-style facilities (e.g., [P3149R11]) for one approach to scoping non-sequential concurrency.

Non-local concurrency

Non-local concurrent scheduling is the repeated application of a partition function over the set of work items. Let me explain…

Up to the serialiser examples, we have implicitly assumed that the set of work items is fixed and that the concurrency structure is visible in the syntax of the program. In other words, the dependency graph (the strict partial order) can be read directly from the term structure using `;` and `||`. In practice, however, the shape of a concurrent computation is often *data-dependent*: conditionals choose which work items exist, loops create unboundedly many instances, and higher-level schedulers (executors, queues, work-stealing pools) decide where and when enqueued work runs. Once scheduling decisions are driven by run-time state, the concurrency structure can no longer be captured by a single static term; it must be described as a dynamic process.

To reason about these dynamic processes, we can extend our algebra with new operators. That would take too much space. Instead, we will use the strict partial ordering as a main tool to reason about non-local concurrency. At any point at which we look at work items, we can assume that those work items are dynamically created.

Let `W` be the set containing all the work items in a program, for its entire execution. We treat `W` as the set of all work items that will ever exist in a complete execution (a hindsight view); at runtime only a prefix of `W` is known, but our reasoning quantifies over `W` to describe the evolving computation.

Concurrent scheduling is the part of the program that decides which work items in `W` may start executing next. If the work items and their dependencies are directly expressed in the control flow of the program, then the compiler largely determines the schedule, and dependency discovery is implicit in control flow.

Let `s_t` be an operation that schedules work items for execution, at some time `t` (we treat scheduler invocations as work items too). We have more of these operations in the program, but let's focus on one. The scheduler invocation decides to start a set of work items; let's denote that set by `WS_t`. Let's also denote by `WR_t` the set of work items `p` for which `s_t < p` – this is the set of work items that remain to be scheduled. Intuitively, `s_t` is the now boundary: `s_t < p` means `p` is not started before this scheduling step. By construction we have `WS_t ⊆ WR_t`.

In other words, operation `s_t` partitions the set `WR_t` in two parts: `WS_t` and `WR_t - WS_t`. Thus, concurrent scheduling is repeated partitioning of the set of remaining work items.

While this partition can be implemented in many ways, we need to discuss:

- minimal guarantees to be provided by the partition function;
- an efficient implementation of such a function;
- the times in which it makes sense to apply this partition function.

As minimal guarantee, the selected set `WS_t` must only contain elements `p` for which, if `q < p`, then `q < s_t`. In plain English, we cannot schedule for execution a work item for which predecessors haven't yet completed.

When considering efficiency, in practice we often use a greedy scheduler that does not depend on the kinds of work items it schedules: at a given time it tries to start as many work items as possible. To make this precise, we assume that at time `t` every work item in `W` is in exactly one of three states: **remaining** (`WR_t`), **executing** (`WE_t`), or **completed** (`W - (WR_t ∪ WE_t)`). Under this assumption, the minimal guarantee from above ('all predecessors are completed') can be stated as: start only those `p ∈ WR_t` whose predecessors are not in `WR_t ∪ WE_t`. The greedy choice is then to take the largest such set, i.e. the set of all currently-ready work items: `WS_t = { p ∈ WR_t | ∄ q ∈ (WR_t ∪ WE_t), q < p }`. In practice, schedulers might have some other consideration that put additional bounds to the set of work items that can be started; these are typically number of workers available, queue capacity, priorities, affinity, etc.

The only other important question is: how often should we run the scheduler? Practical wisdom tells us that we should do it whenever new work items are created, and whenever they are completed. Indeed, those are the moments when `WR_t ∪ WE_t` from our exposition above might change. Running the scheduling algorithm more often doesn't usually make sense, as we cannot possibly find new work items to schedule (again, in practice, we might have additional concerns like: resources becoming available, priorities changing, etc.).

Modularity

The discussion above on non-local concurrency might lead to the false impression that there must be a single global scheduler for all work items in the program. While such a design is possible, it would concentrate scheduling decisions in one place and make reasoning about independent parts of the program harder. A more modular approach is to allow multiple schedulers, each responsible for a subset of work items.

Recall that the strict partial ordering `<` induces a directed acyclic graph (DAG) of work items. In many realistic programs this graph is sparse: it is composed of loosely connected subgraphs with relatively few ordering relations between them. For example, work items protected by one mutex are often largely independent of work items protected by another mutex. The cases in which all mutexes in a program protect the same region of code are rare.

This observation suggests that we can partition the global set of work items `W` into subsets `W_0`, `W_1`, …, `W_n`, such that the number of ordering relations `p < q` with `p ∈ W_i` and `q ∈ W_j (for i ≠ j)` is relatively small. Each subset can then be assigned its own scheduler, applying the same partition principle locally.

Consider a simple case in which there is a single cross-subset dependency `p < q`, where `p ∈ W_0` and `q ∈ W_1`, and no dependency in the opposite direction. Under this assumption, the scheduler for `W_0` can operate independently: it can schedule all work items in `W_0` as soon as their local predecessors complete. The scheduler for `W_1`, however, must respect the fact that `q` cannot execute before `p`.

One practical strategy is to defer the creation (or, equivalently, the enqueueing) of `q` until `p` completes. In this way, the cross-subset dependency is handled at the boundary: when `p` finishes, it produces or enqueues `q` into the scheduler associated with `W_1`. As a consequence, all work items that (directly or indirectly) depend on `q` are created or enqueued only after `p` completes. With this discipline, the scheduler associated `with W_1` does not need to inspect the internal state of `W_0`; it only reacts to the arrival of new work.

In practice, serialisers and similar abstractions often come with their own scheduler implementations. The user coordinates between multiple serialisers by expressing cross-dependencies through enqueueing at well-defined points (for example, when a predecessor completes). This allows each serialiser to maintain a simple and efficient local scheduling policy, while preserving the global strict partial ordering of work items.

Here (on the following page) is a simple example of communicating between two serialisers, each serialiser performing their own scheduling; the example shows 3 expressions in the program that makes the connection between the serialisers:

```
p/>S0              // enqueue p onto S0
p_done ; (q/>S1)   // when p completes, enqueue q
                   // onto S1
q                  // q eventually runs under S1
                   //(after q/>S1)
```

This realises the cross-dependency `p < q` via `p < (q/>S1) < q`, without requiring `S1` to inspect `S0`.

Modularity, in this view, is achieved by aligning scheduling boundaries with structural boundaries in the dependency graph. As long as cross-subset ordering relations are handled explicitly at those boundaries, each scheduler can reason locally about its own subset of work items without requiring global knowledge of the entire program.

At a deeper level, this modular structure is justified by the exchange law. Exchange tells us that when two components evolve independently, global phase ordering refines independent local progress. This is exactly the situation in which encapsulation is sound: if cross-subset dependencies are handled explicitly at well-defined boundaries, then reasoning about each subset locally is sufficient to reason about the whole. The algebra guarantees that recombining the local progress preserves correctness.

Concurrency can be encapsulated.

Conclusions and practical recommendations

This article argued that concurrency has a simple underlying idea: a program induces a strict partial order between its work items. Hoare's concurrency algebra offers an equivalent, calculational view of the same structure, and it connects naturally to other established formalisms such as Hoare logic and Milner's CCS.

Once this viewpoint is adopted, many common concurrency constructs can be seen as elaborations that add or remove ordering constraints. The practical benefit is a shift in how we design and reason about concurrent software: instead of starting from mechanisms (threads, locks, callbacks), we start from the ordering constraints that must hold, and then choose an implementation strategy that enforces them.

Based on the core ideas in this article, here are a few practical recommendations:

- **Start from relations, not mechanisms.** Identify the work items in the problem and write down the ordering constraints that must hold between them.
- **Make dependencies explicit.** Prefer APIs and designs that surface the relevant `a < b` constraints instead of encoding them implicitly in ad hoc coordination.
- **Prefer static structure when possible.** When dependencies can be expressed directly in control flow (structured concurrency, task graphs), do so; it simplifies both reasoning and scheduling.
- **Keep constraints local.** Aim to express most constraints within a small scope (a module, a component, a serialiser/strand) rather than relying on global coordination.
- **Handle cross-scope dependencies at boundaries.** When non-local concurrency is required, make boundary handoffs explicit (e.g., enqueue-on-completion) so that each local scheduler can remain simple.
- **Avoid manual synchronisation when you can.** Synchronization primitives (mutexes, semaphores, etc.) typically introduce extra work items (waiting) and can harm both performance and locality, and they increase the risk of bugs (race conditions, deadlocks, etc.).

Following these recommendations helps make concurrent programs as simple as possible (but no simpler). ■

Acknowledgements

This article is primarily shaped by the work of Tony Hoare and Leslie Lamport. Hoare's laws of programming with concurrency and Lamport's partial-order view of events provided much of the conceptual vocabulary I use here.

My own approach to concurrency has also been strongly influenced over the years by talks, writing, and conversations with Kevlin Henney, Sean Parent, Eric Niebler, Dave Abrahams, and Dimi Racordon. Even when their contributions are not cited directly, their way of framing problems towards composability, local reasoning, and practical correctness shows up throughout this article.

References

[Apple], Apple, `DispatchQueue`, https://developer.apple.com/documentation/dispatch/dispatchqueue, accessed Feb 2026.

[AsioStrands] Christopher M. Kohlhoff, Strands: Use Threads Without Explicit Locking, https://www.boost.org/doc/libs/latest/doc/html/boost_asio/overview/core/strands.html, accessed Feb 2026.

[Brooks95] Frederick P. Brooks Jr., *The Mythical Man-Month* (anniversary ed.), Addison-Wesley Longman Publishing, 1995.

[C++Ref] cppreference.com, `std::memory_order`, https://en.cppreference.com/w/cpp/atomic/memory_order.html, accessed Feb 2026.

[Hoare09] Tony Hoare, Bernhard Möller, Georg Struth, Ian Wehrman, 'Concurrent Kleene algebra', in: M. Bravetti, G. Zavattaro (Eds.), *Concurrency Theory* (CONCUR 2009), LNCS, vol. 5710, Springer, 2009.

[Hoare11] Tony Hoare, Bernhard Möller, Georg Struth, Ian Wehrman, 'Concurrent Kleene algebra and its foundations', *The Journal of Logic and Algebraic Programming* 80.6, 2011.

[Hoare13] Tony Hoare, 'Verified Concurrent Programmes: Laws of Programming with Concurrency', *Microsoft Research Concurrency Workshop*, 2013, https://www.youtube.com/watch?v=G83nBjXBQCo, accessed Feb 2026.

[Hoare15] Tony Hoare, 'The Laws of Programming with Concurrency', 2015, https://www.youtube.com/watch?v=9kKQ8uLK8mk, accessed Feb 2026.

[Intel] Intel, 'Local Serializer', https://www.intel.com/content/www/us/en/docs/onetbb/developer-guide-api-reference/2021-12/local-serializer.html, accessed Feb 2026.

[Lamport78] Leslie Lamport, 'Time, clocks, and the ordering of events in a distributed system', *Communications of ACM*, July 1978, https://dl.acm.org/doi/pdf/10.1145/359545.359563.

[P3149R11] Ian Petersen, Jessica Wong, Dietmar Kühl, Ján Ondrušek, Kirk Shoop, Lee Howes, Lucian Radu Teodorescu, Ruslan Arutyunyan, 'P3149R11: async_scope – Creating scopes for non-sequential concurrency', https://wg21.link/P3149R11.

[Wikipedia-1] Wikipedia, Hoare logic, https://en.wikipedia.org/wiki/Hoare_logic, accessed Feb 2026.

[Wikipedia-2] Wikipedia, Calculus of communicating systems, https://en.wikipedia.org/wiki/Calculus_of_communicating_systems, accessed Feb 2026.

[Wikipedia-3] Wikipedia, Fork-join model, https://en.wikipedia.org/wiki/Fork-join_model, accessed Feb 2026.

[Wikipedia-4] Wikipedia, 'Async/await', https://en.wikipedia.org/wiki/Async/await, accessed Feb 2026.

Tony Hoare died at the age of 92 on March 5 this year. He gave us quicksort, Hoare logic and much more besides. In an ACM blog, Bertrand Meyer said:

> Style is the defining characteristic of Hoare's work, not just writing style but scientific style, always elegant and focused on what truly matters.

The post ('Tony Hoare and His Imprint on Computer Science') is available at: https://cacm.acm.org/blogcacm/tony-hoare-and-his-imprint-on-computer-science/.

Let the Compiler Check Your Units

Mixing your units can be disastrous. Wu Yongwei takes a quick look at C++ unit libraries that can help keep everything in order.

I recently came across a C++ standard proposal P3045 [P3045R7], which aims to add physical units to C++. Curious, I looked into the existing unit libraries and went down quite a rabbit hole.

Type safety and user-defined literals

Before exploring these libraries, I was already somewhat familiar with the idea of 'type safety'. I was also aware that user-defined literals (UDLs) [CppReference-1] allow creating literals of specific types with ease. Typical uses in the standard library include **string**/**string_view** literals and the chrono library [CppReference-2], which make code both convenient and safe.

Figure 1 shows some simple examples.

```
auto msg = "Hello "s + user_name;
auto t1 = chrono::steady_clock::now();
this_thread::sleep_for(500ms);
auto t2 = chrono::steady_clock::now();
auto duration = t2 - t1;
auto what = t1 + t2;      // Can't compile
cout << duration / 1.0ms; // To double, in ms
```

Figure 1

Unlike a plain **time_t**, where addition, subtraction, multiplication, and division all compile regardless of whether they make sense, the chrono library distinguishes between time points and durations. Only operations that are actually meaningful will compile, such as:

- time point **±** duration → time point
- time point **-** time point → duration
- scalar ***** duration → duration
- duration ***** (or **/** or **%**) scalar → duration
- duration **/** duration → scalar
- duration **%** duration → duration

The duration types defined by the chrono library also encode the underlying data type (which arithmetic type to use) and the ratio relative to the base unit. Implicit conversions must not truncate, e.g.:

- A duration with an integer underlying type can be implicitly converted to one with a floating-point underlying type, but not vice versa.
- When both underlying types are integral, a coarser-precision duration can be implicitly converted to a finer-precision one (e.g. from seconds to milliseconds), but not vice versa.

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

```
class length {
public:
  explicit length(double v) : value_(v) {}
  double value() const { return value_; };

private:
  double value_;
};

length operator+(length lhs, length rhs)
{
  return length(lhs.value() + rhs.value());
}

length operator""_m(long double v)
{
  return length{static_cast<double>(v)};
}

length operator""_cm(long double v)
{
  return length{static_cast<double>(v) * 0.01};
}
```

Listing 1

We can write our own UDLs, and the only difference from the standard library is that UDL suffixes must begin with an underscore (**_**). Listing 1 is a simple example.

With this, we can write '**1.0_m + 10.0_cm**' and get **length{1.1}**. Pretty simple, right?

When I cover UDLs in training sessions, I usually mention the story of the *Mars Climate Orbiter*: software from Lockheed Martin reportedly produced thrust data in pound-force seconds, while NASA's navigation system expected newton-seconds. The unit mismatch caused the *Orbiter* to enter an incorrect orbit and ultimately disintegrate (you can search for more details). This incident is often cited to illustrate the risks of unit mix-ups. If UDLs had existed back then, and NASA had been using C++ with such features, could the accident have been avoided? Of course, 1998 feels like ancient times in the tech world: software tools were not as powerful as they are today, and the first C++ standard had only just appeared.

Back to the code above, we can already see a downside of this approach: when a floating-point number is followed by a literal suffix, the compiler always passes the number as a **long double** to the corresponding UDL operator. Here I unconditionally cast to **double**, but this introduces a potential truncation issue. I will return to this point later.

Unit libraries using UDLs

Unit libraries built on strong types existed even before C++11, and Boost.Units [Boost.Units] is one such example. However, it does not use UDLs and has seen little subsequent development, so I won't cover it here. The code examples in Listing 2 (next page) compile with two unit

The primary problem with UDLs is that the UDL parameter-passing mechanism does not simply forward the literal's original type

```
auto mass = 2.0_kg + 500_g;
auto distance = 9.8_m;
auto acceleration = distance / 1_s / 1_s;
auto force = mass * acceleration;

std::cout << "Mass:         " << mass << '\n';
std::cout << "Distance:     " << distance
          << '\n';
std::cout << "Acceleration: " << acceleration
          << '\n';
std::cout << "Force:        " << force << '\n';
```

Listing 2

libraries that are still maintained (though not under active development): **PhysUnits-CT-Cpp11** [PhysUnits-CT-Cpp11] and **SI** [SI].

The first two lines of output are probably what you'd expect, but the 'Force' line might come as a surprise. With the **PhysUnits-CT-Cpp11** library, the output is:

```
Mass:         2.5 kg
Distance:     9.8 m
Acceleration: 9.8 m s-2
Force:        24.5 N
```

With the **SI** library, the output is (same result, slightly different formatting):

```
Mass:         2.5 kg
Distance:     9.8 m
Acceleration: 9.8 m/s^2
Force:        24.5 N
```

Both libraries support unit composition, and both know that in the International System of Units (abbreviated SI), kg · m/s² is just N (newton), and display the result in newtons directly.

The two libraries are very similar in functionality, both using *dimensions* to constrain the allowed operations (see more details about 'dimensional analysis' in [Wikipedia]). Quantities of the same dimension can be added or subtracted, while multiplication and division can produce new dimensions. From the SI perspective, the dimension of mass is M (with unit kg), the dimension of acceleration is LT^{-2} (length divided by the square of time, with unit m/s²), and multiplying the two yields LMT^{-2} (length times mass divided by the square of time, with unit kg · m/s², or N).

From the user's perspective, the main differences are:

- **PhysUnits-CT-Cpp11** uniformly uses `long double` as the underlying type, while **SI** automatically uses `int64_t` or `long double` depending on whether you write an integer or floating-point literal (`2_kg` vs. `2.0_kg`). Combined with **SI**'s loose implicit conversion rules (the result type of an arithmetic operation is determined by the first argument), this has an unfortunate consequence: writing '`2.0_kg + 500_g`' gives you `2.5_kg` (underlying type `long double`), but writing '`2_kg + 500_g`' gives you `2_kg` (underlying type `int64_t`, with the result truncated to kilograms)!
- **PhysUnits-CT-Cpp11** allows arbitrary combinations of units, while **SI** requires you to include the corresponding header for each valid combination. For instance, if I hadn't included `<SI/acceleration.h>`, the '`auto acceleration =` …' line would have failed to compile.
- **PhysUnits-CT-Cpp11** requires at least the C++11 standard, while **SI** requires at least C++17.

If you only need basic SI unit support (and don't need pounds, inches, etc.), and using `long double` is not a concern, I think **PhysUnits-CT-Cpp11** is a solid choice – it is simple to use and provides basic unit type safety.

Unit libraries without UDLs

Problems with UDLs

You may have already noticed some issues with UDLs, but they haven't been fully illustrated in the code above. The primary problem with UDLs is that the UDL parameter-passing mechanism does not simply forward the literal's original type – it forces conversion through one of a few fixed forms. For numeric literals with UDL suffixes (e.g. `10_m`), there are three ways to define a UDL:

- One approach uses the form '`operator""_m(unsigned long long)`' (for integers) or '`operator""_m(long double)`' (for floating-point numbers). If we want to use a narrower type to represent values (such as `int` or `double`), truncation or precision loss can occur (though this is not generally a problem, and C++20's `consteval` can allow compile-time checks).
- Another approach uses the form '`operator""_m(const char*)`', where the compiler passes the number (e.g. `10`) in as a string. This forces us to decide in advance which concrete arithmetic type to use for the result; if it's not large enough, truncation or precision loss can again occur.
- A third approach uses template parameters ('`template <char...>` … `operator""_m()`'), which is the most flexible – it can produce different result types based on the specific input – but also the most complex to implement.

The consequences of this are:

- UDLs cannot directly represent negative numbers. To represent a negative value, you need to define a unary `operator-` on the result type.
- The way UDLs are defined makes it difficult to flexibly choose the underlying arithmetic type. The standard library handles complex numbers by using different suffixes: `1i` is a `complex<double>`, `1if` is a `complex<float>`, and so on (the underlying types are always floating-point). This is reasonably natural and consistent for complex numbers, but clearly doesn't work for units like the metre (m) or the kilogram (kg) …

we can conveniently compose new physical quantities through arithmetic, but since Au is not SI-centric, results are not automatically converted to SI units

- The value of a quantity can be a complex number, which would be impractical to combine with another UDL suffix representing the unit.
- For compound units (such as the acceleration above, with unit m/s^2), using UDLs means either defining a large number of suffixes (**SI** does define `_m_p_s` and `_km_p_h`, but does not include a UDL suffix for acceleration), or composing with unit literals (such as `1_s`) as in the example code above.

For these reasons, some newer unit libraries have abandoned UDLs in favour of arithmetic composition with unit constants. (In fact, in **PhysUnits-CT-Cpp11** you can write '`9.8_m / second / second`' or even '`9.8 * meter / second / second`'.)

The Au library

In its own words, **Au** [Au] is:

> A C++14-compatible physical units library with no dependencies and a single-file delivery option. Emphasis on safety, accessibility, performance, and developer experience.

The earlier code needs some adjustments to work with **Au**. Since **Au** is a library I'd genuinely recommend, I'll present the full code in Listing 3.

The output is:

```
Mass:               2500 g
Mass (as kg):       2.5 kg
Distance:           9.8 m
Acceleration:       9.8 m / s^2
Force (compound):   24500 (m * g) / s^2
Force (as Newton):  24.5 N
```

Some differences are immediately apparent:

- Units are no longer UDL suffixes but constants. Writing '`2.0 * kg`' may seem slightly more verbose than '`2.0_kg`', but creating the acceleration ('`distance / s / s`', or '`9.8 * (m/s/s)`') is actually more concise. Also, defining new units becomes very easy (**Au** itself only defines base units like `g` and `s`, plus prefixes like `kilo` and `milli`, without defining `kg` or `ms`), and the underlying type of a physical quantity becomes intuitive: it's simply the type of the literal itself. In the expressions above, the underlying type is `double` rather than `long double` (in the mass case it is the result of adding a `double` and an `int`).
- As before, we can conveniently compose new physical quantities through arithmetic, but since **Au** is not SI-centric, results are not automatically converted to SI units. However, we can use `as` to perform conversions, provided that the dimensions match and **Au** does not consider the conversion risky (more on this later); otherwise, the code will not compile. For example, we can write '`(2.0 * kg + 500 * g).as(kg)`', but not '`(2 * kg + 500 * g).as(kg)`'. If we need to allow potentially risky conversions, we can use '`(2 * kg + 500 * g).coerce_as(kg)`' – the result is `2 * kg`, i.e. 2 kilograms (with `int` as the underlying type).

```
#include <au/io.hh>
#include <au/prefix.hh>
#include <au/units/grams.hh>
#include <au/units/meters.hh>
#include <au/units/newtons.hh>
#include <au/units/seconds.hh>
#include <iostream>

using namespace au;
using namespace au::symbols;

constexpr auto kg = symbol_for(kilo(grams));

int main()
{
  // Create quantities
  auto mass = 2.0 * kg + 500 * g;
  auto distance = 9.8 * m;
  auto acceleration = distance / s / s;

  // Calculation
  auto force = mass * acceleration;

  // Output
  std::cout << "Mass:              " << mass
            << '\n';
  std::cout << "Mass (as kg):      "
            << mass.as(kg) << '\n';
  std::cout << "Distance:          " << distance
            << '\n';
  std::cout << "Acceleration:      "
            << acceleration << '\n';
  std::cout << "Force (compound):  " << force
            << '\n';
  std::cout << "Force (as Newton): "
            << force.as(N) << '\n';
}
```

Listing 3

Note that the SI base unit for mass is the kilogram, not the gram. However, **Au** chooses the gram as the base unit at the programming level and composes the kilogram via prefixes, which unifies how prefixes are used.

Au defines not only common base units and prefixes, but also other commonly used units such as the inch used in English-speaking countries (by including `<au/units/inches.hh>`). It encodes in the type system that 1 inch equals 254/100 centimetres, and defines:

- The print label as `"in"`
- The singular and plural quantity maker constants as `inch` and `inches`
- The symbol as `in`
- …

Au has also put significant efforts into error message readability. When you write code with mismatched dimensions, the compiler error messages will contain reasonably readable unit descriptions

Defining custom units is straightforward too. The code below defines a modern Chinese ‘chǐ’ (尺). It uses the C++17 syntax (C++14-style code would be slightly more complicated):

```
struct Chi
  : decltype(au::Meters{} / au::mag<3>()) {
  static constexpr const char label[] = "chǐ";
};
constexpr auto chi = au::QuantityMaker<Chi>{};
```

Its meaning is straightforward:

- 1 metre (the code uses the American spelling `Meters`) equals 3 chǐ.
- The output label is ‘chǐ’.
- The constant representing the unit in code is `chi`, so writing ‘`cout << distance.as(chi)`’ gives us the output ‘29.4 chǐ’.

We’ve already discussed `as`, and `in` is another important member function for an **Au** quantity. `quantity.as(unit)` returns a new `Quantity` object (still a quantity with a unit), while `quantity.in(unit)` returns the underlying numerical value (a scalar without units). For example, `distance.in(m)` returns the `double` value `9.8`. This comes in handy when interfacing with APIs that don’t use the unit library. Similarly, `coerce_as` has a corresponding `coerce_in`.

Au has a few more features worth highlighting:

- All of **Au**’s default unit checks happen at compile time. The generated machine code is essentially identical to hand-written scalar arithmetic – necessary multiplication/division conversions are still there, but there is no extra runtime overhead. This is C++’s zero-overhead abstraction principle in action.
- **Au** uses exact rational arithmetic for integer conversions (e.g. inches to cm is stored as `254/100`). To prevent data loss, the `as` member function forbids conversions that result in truncation or high overflow risk. For example, you cannot strictly convert inches to centimetres (truncation) or `int32_t` inches to nanometres (high overflow risk) without an explicit risk-ignoring parameter (or the `coerce_as` member function).

  Warning: Because **Au** performs the multiplication before the division (e.g. grams to pounds is to multiply by 1,000,000 and then divide by 453,592,370), forced integer conversions can easily overflow intermediate values and cause undefined behaviour! (Floating-point arithmetic has no such problems.)
- **Au** has `QuantityPoint` for ‘absolute values’ and plain `Quantity` for ‘relative quantities’, mirroring the design of chrono’s `time_point` and `duration`. For example, a thermometer reading of 20 °C is a `QuantityPoint` (`celsius_pt(20)`), while a temperature rise of 5 °C is a `Quantity` (`celsius_qty(5)`). The type system enforces physical logic: you can add a `Quantity` to a `QuantityPoint`, but adding two absolute `QuantityPoints` together is a compile-time error.
- **Au** also supports modern formatting – you can use it with C++20’s `std::format` and C++23’s `std::print`, not just traditional IO streams. It also works with the {fmt} [fmt] library.
- If your project only needs a handful of units, **Au** provides a Python tool that can bundle just the units you need into a single header file for direct inclusion in your project.

Au has also put significant efforts into error message readability. When you write code with mismatched dimensions, the compiler error messages will contain reasonably readable unit descriptions, making it easier to track down the problem. In contrast, some earlier unit libraries (such as Boost.Units) were notorious for their extremely verbose and hard-to-read error messages.

The mp-units library

mp-units [mp-units] is currently the most feature-complete and most modern C++ unit library, created and led by Mateusz Pusz, the first author of P3045. It requires C++20 or later and makes heavy use of new features such as concepts and class-type non-type template parameters (NTTPs). It also serves as the foundation for the committee’s ongoing work on adding unit support to the standard library.

Code using **mp-units** is quite similar to **Au** (see Listing 4).

```
#include <iostream>
#include <mp-units/systems/si.h>

using namespace mp_units;
using namespace mp_units::si::unit_symbols;

int main()
{
  // Create quantities
  auto mass = 2.0 * kg + 500 * g;
  auto distance = 9.8 * m;
  auto acceleration = distance / s / s;

  // Calculation
  auto force = mass * acceleration;

  // Output
  std::cout << "Mass:               " << mass
            << '\n';
  std::cout << "Mass (as kg):       "
            << mass.in(kg) << '\n';
  std::cout << "Distance:           " << distance
            << '\n';
  std::cout << "Acceleration:       "
            << acceleration << '\n';
  std::cout << "Force (compound):   " << force
            << '\n';
  std::cout << "Force (as Newton): "
            << force.in(N) << '\n';
}
```

Listing 4

mp-units introduces the concept of 'quantity kind', based on the International System of Quantities (ISQ). It distinguishes not only dimensions but also semantics...

Output:

```
Mass:               2500 g
Mass (as kg):       2.5 kg
Distance:           9.8 m
Acceleration:       9.8 m/s²
Force (compound):   24500 g m/s²
Force (as Newton):  24.5 N
```

Noticeably more modern and polished.

Now let's look at the differences in the code. First, note some naming differences from **Au**: `as` in **Au** becomes `in` in **mp-units**; `coerce_as` becomes `force_in`; and `in` becomes `numerical_value_in`. In particular, both libraries have `in`, but with different semantics – you must be careful if you migrate code between the two libraries.

Another minor difference is that standard prefixed units are fully predefined in **mp-units**, such as `kg` and `ms`. Defining new units is also different, but overall simpler:

```
// Since accented (or Chinese) characters are
// not in the basic character set, we need to
// use symbol_text
inline constexpr struct chi final
  : named_unit<symbol_text{u8"chǐ", "chi"},
               mag_ratio<1, 3> * si::metre> {
} chi;
```

If we use only basic characters (a subset of ASCII) [CppReference-3], the code can be even simpler:

```
// For symbols in the basic character set, we
// don't need to use symbol_text
inline constexpr struct chi final
  : named_unit<"chi",
               mag_ratio<1, 3> * si::metre> {
} chi;
```

This clearly expresses that 1 chǐ equals 1/3 metre. As with **Au**, we can now write '`distance.in(chi)`'.

Broadly speaking, **mp-units** is richer in features and more future-oriented:

- **mp-units** introduces the concept of 'quantity kind', based on the International System of Quantities (ISQ). It distinguishes not only dimensions but also semantics, catching at compile time the mixing of quantities that share the same dimension but have different semantics.
- **mp-units** requires C++20: it makes comprehensive use of concepts to constrain templates (there will be concept constraint examples below), and takes advantage of class-type NTTPs. **Au** defines a `celsius_pt` to help create temperature points, but **mp-units** users can simply write '`point<deg_C>`' (`deg_C` is a `constexpr` class-type object, used as NTTP) to achieve the same purpose, simpler and more consistent.
- **mp-units** is the basis for C++ standard proposals. If you want to align with the future standard as early as possible, **mp-units**' API and conceptual framework will be closer to the final standardized form.

Let me elaborate on 'quantity kind'. A simple example: *width*, *height*, and *radius* are all dimensionally lengths, but in **mp-units** they can be distinct quantity kinds, preventing conceptually meaningless mixing. Another example: *energy* and *moment of force*[1] share the same dimension (both are L^2MT^{-2}, and both quantities can be written in units of $kg \cdot m^2/s^2$), but passing an energy to a function expecting a moment of force could lead to disastrous consequences. **mp-units** will prevent the code in Listing 5 from compiling.

```
// Explicitly require the parameter to be moment
// of force / torque
void foo(
  QuantityOf<isq::moment_of_force> auto moment)
{
  // …
}

// Explicitly mark work as energy
auto work = isq::energy(100 * J);

// work does NOT satisfy foo's constraint
foo(work);
```

Listing 5

Whether you need this level of distinction depends on your requirements, but the option is there if you want it.

By the way, SI considers angles to be dimensionless – but this has always been contentious. You generally can't mix angles, solid angles, and plain dimensionless numbers freely. Both **Au** and **mp-units** sidestep this, and provide automatic conversion between degrees and radians. In **mp-units**, for instance, the function in Listing 6 works regardless of whether you pass in degrees (e.g. `30 * deg`) or radians (e.g. `0.5 * π * rad` – yes, this *works*).

Here, `value_cast` ensures the underlying type is `double`; otherwise, when an integer quantity is passed in, one of the subsequent `in` conversions will fail to compile.

```
void print_angle(
  QuantityOf<isq::angular_measure> auto angle)
{
  auto a = value_cast<double>(angle);
  std::cout << "Angle:   " << a.in(deg) << '\n';
  std::cout << "Radians: " << a.in(rad) << '\n';
  std::cout << "sin:     " << sin(a) << '\n';
}
```

Listing 6

1 Commonly called *torque*, though ISQ and **mp-units** treat torque as a narrower, scalar form of moment of force.

An important design consideration for the standard unit library is integration with the existing chrono library.

If your project is already using C++20 or later, and you need comprehensive unit support or quantity kind distinction, **mp-units** should be a good choice. If your project needs to be compatible with C++14/17 or only requires basic SI unit safety, **Au** is more lightweight and still useful (the example code of **Au** compiles noticeably faster than that of **mp-units**). The core design philosophies of both (compile-time checking, zero runtime overhead, composition by multiplication or division) are the same.

Finally, let me demonstrate that unit checking typically incurs no runtime overhead. At https://godbolt.org/z/aKbsG4P5Y, you can see that `mass * acceleration` is compiled into the following single assembly instruction (line 2):

```
mulsd   xmm0, xmm1
```

Very clean!

A standard unit library for C++29

The C++ standards committee is actively working towards incorporating unit and physical quantity support into the standard library. Several related proposals cover this space, including P1935 [P1935R2] (the initial proposal), P2980 [P2980R1] (motivation, scope, and plan), P2981 [P2981R1] (safety discussion), P2982 [P2982R1] (discussion of `quantity` as a numeric type), and P3045 (mentioned at the beginning of this article). The core content of these proposals is based on **mp-units**' design and implementation experience.

This feature may make it into the C++29 standard (it definitely won't make C++26). The design space is large, and many issues still need discussion and resolution. The current P3045R6 proposal is already very long – it's a hefty read on its own.

An important design consideration for the standard unit library is integration with the existing chrono library. Chrono's `duration` and `time_point` are essentially physical quantities and quantity points in the time dimension. Ideally, the new unit library should be able to subsume or generalize chrono's design. This is a recurring theme across the proposals.

Once a standard unit library lands, it will provide the C++ ecosystem with a unified representation for physical quantities. Scientific computing, engineering software, embedded systems, game engines, and other domains will all be able to interoperate under the same type system, instead of each reinventing the wheel. On a practical note, even though C++29 is still several years away, learning about and using existing libraries like **Au** or **mp-units** is already worthwhile – their core concepts and API designs closely track where the standard is headed, and migration costs should be low.[2]

I hope this gives you a clearer picture of what's available. Zero-overhead abstraction remains one of C++'s greatest strengths, and unit libraries are a perfect showcase for it. ■

References

[Au] https://github.com/aurora-opensource/au

[Boost.Units] https://www.boost.org/doc/libs/latest/doc/html/boost_units.html

[CppReference-1] cppreference.com, 'User-defined literals', https://en.cppreference.com/w/cpp/language/user_literal.html

[CppReference-2] cppreference.com, 'Date and time library', https://en.cppreference.com/w/cpp/chrono.html

[CppReference-3] cppreference.com, 'Character sets and encodings', https://en.cppreference.com/w/cpp/language/charset.html

[fmt] https://github.com/fmtlib/fmt

[mp-units] https://github.com/mpusz/mp-units

[P1935R2] Mateusz Pusz, 'A C++ approach to physical units', https://wg21.link/p1935r2

[P2980R1] Mateusz Pusz *et al*., 'A motivation, scope, and plan for a quantities and units library', https://wg21.link/p2980r1

[P2981R1] Mateusz Pusz *et al*., 'Improving our safety with a physical quantities and units library', https://wg21.link/p2981r1

[P2982R1] Mateusz Pusz and Chip Hogg, '`std::quantity` as a numeric type', https://wg21.link/p2982r1

[P3045R7] Mateusz Pusz *et al*., 'Quantities and units library', https://wg21.link/p3045r7

[PhysUnits-CT-Cpp11] https://github.com/martinmoene/PhysUnits-CT-Cpp11

[SI] https://github.com/bernedom/SI

[Wikipedia] Wikipedia, 'Dimensional analysis', https://en.wikipedia.org/wiki/Dimensional_analysis

2 It is no coincidence that P3045's authors include the lead developers of **mp-units**, **Au**, and **SI**.

Automating Release of Resources in C

Ensuring resources are cleaned up in C can be a challenge. Alison Chaiken shows how cleanup macros can help.

Perhaps the trickiest part of writing correct C code is resource management, particularly in error handlers. A quick scan of Common Vulnerabilities and Exceptions (CVE) lists finds many double-frees and use-after-frees, and bugtrackers are full of lock-contention problems and memory-leak failures caused by sloppy error-handling. If only the C language had scoped locks and destructors!

ISO C still does not support these features, but there's good news: the GCC and clang compiler front-ends do. GCC implemented the `__cleanup__` compiler attribute in 2003 [gcc], and clang added the feature in 2009 [clang]. Several prominent code bases written in C make extensive use of the `__cleanup__` attribute, notably systemd, Linux kernel and glibc, but many large, prominent C-language projects (git, gstreamer, U-Boot for example) do not. The purpose of this article is to encourage more C programmers to take advantage of the `__cleanup__` functionality and remove some of the notorious `goto` labels.

How does `__cleanup__` work and how does one use it? As described in the GCC [gcc-1] and clang documentation [clang-1], a cleanup function is bound to a local variable at the time of its declaration, and runs automatically when the variable goes out of scope, not unlike a C++ destructor. The clang documentation illustrates the point as shown below:

```
static void foo (int *) {... }
static void bar (int *) {... }
void baz (void) {
  int x __attribute__((cleanup(foo)));
  {
    int y __attribute__((cleanup(bar)));
  }
}
```

Here `bar()` will run when the first closing brace is reached and `foo()` upon reaching the second. Note that the compiler's generated code passes the cleanup function a pointer to the variable to which it is bound. The cleanup function need not actually make use of the pointer. The Linux init system systemd's code offers many real-world examples like that below [systemd]:

```
static inline void freep(void *p) {
  free(*(void **) p);
}

#define _cleanup_free_ _cleanup_(freep)
```

The cast inside the cleanup function is necessary since, as mentioned above, the cleanup function receives a pointer to the variable to which it is bound. In the common use case below, `freep()` is invoked with a `char**` parameter [systemd-1]:

```
_cleanup_free_ char *path = strdup(e + 1);
```

As of this writing, systemd's code base has 5852 call-sites for `_cleanup_free_`, compared to 2633 for `goto`. Although C has not gained new features with the speed of C++, it is not in fact entirely static! Other systemd cleanup functions decrement reference counters, close file descriptors and unlock locks, in keeping with expectations from C++. Systemd also defines destructors for static variables [systemd-2].

```
void put_device(struct device *dev);
DEFINE_FREE(put_device, struct device *,
  if (_T) put_device(_T))

static int rzg2l_irqc_common_init(... ) {
  struct device *dev __free(put_device)
  = pdev ? &pdev->dev : NULL;

[...]

  / * On the successful path we don't actually
  want to "put" dev. */
  dev = NULL;
}
```

Listing 1

An obvious question is, suppose the function where a cleanup attribute is declared succeeds, and the release of the resource is therefore undesirable? The Linux kernel simply checks if the pointer parameter is `NULL` before freeing it. Listing 1 shows an excerpt from a kernel interrupt-chip driver in which the sole purpose of the variable `dev` is to signal the runtime whether or not a reference to the device should be released [linux].

Suppose that setting the variable with the cleanup handler to `NULL` is not readily possible? The kernel project has a `no_free_ptr()` macro for that case. In the following example from a GNSS device driver [linux-1], the `ubx_probe()` function upon success passes the pointer to which the cleanup handler is bound to another function. The `no_free_ptr()` wrapper in Listing 2 insures that the pointer is not freed.

The Linux macros which define `no_free_ptr()` and its relatives are nested and a bit difficult to understand. The best course of action for decoding them is to examine the compiler's preprocessor output. Userspace preprocessor output is accessible via the `-E` compiler flag. For

```
DEFINE_FREE(free_serial, struct gnss_serial *,
  if (_T) gnss_serial_free(_T))

static int ubx_probe(struct serdev_device
  *serdev)
{
  struct gnss_serial *gserial __free(free_serial)
    = gnss_serial_allocate(serdev,
    sizeof(*data));

[ ... ]

  ret =
    gnss_serial_register(no_free_ptr(gserial));
}
```

Listing 2

Alison Chaiken is a systems programmer and Linux kernel engineer who has recently relocated from Silicon Valley to Berlin, where she would be delighted to meet over a beer or a coffee to discuss C++/C or compilers. Contact her at alison@she-devel.com

the kernel, set **KCFLAGS** and **KBUILD_CFLAGS** to **--save-temps**. The preprocessed **no_free_ptr()** line above becomes:

```
ret = gnss_serial_register(((typeof(gserial))
__must_check_fn(({ __auto_type __ptr =
&(gserial); __auto_type __val = *__ptr; *__ptr
= ((void *)0); __val; }))));
```

where **__auto_type** is a compiler extension that is essentially equivalent to C++'s **auto** keyword rather than C's [gcc-2]. The actual **no_free_ptr()** function is delimited by curly braces in a way which may remind C++ readers of lambda functions. The construct is a statement expression, about which GCC notes [gcc-3]:

> A compound statement enclosed in parentheses may appear as an expression in GNU C. This allows you to use loops, switches, and local variables within an expression.... Recall that a compound statement is a sequence of statements surrounded by braces... The last thing in the compound statement should be an expression followed by a semicolon; the value of this subexpression serves as the value of the entire construct.

Accordingly, the code makes a copy of the pointer which should not be freed, sets the original pointer to **NULL**, and leaves the copied pointer for the enclosing function to evaluate.

The kernel further provides a **DEFINE_GUARD** macro and **guard()** pseudofunction in analogy to **DEFINE_FREE** and **__free()** above. The guard macros support automatic unlocking, including **_trylock** forms. The nested macros underlying **DEFINE_GUARD** even include a **DEFINE_CLASS** [linux-2].

Nonetheless, it is the Glibc system library which contains the apotheosis of C cleanup methods in its pthread implementation. There we find the split-macro pair **pthread_cleanup_push(ROUTINE, ARG)** and **pthread_cleanup_pop()** [glibc]. As the names suggest, they are used to manage a stack of cleanup handlers which run in order when a thread is canceled, or hard-exits, possibly by throwing an exception. As show below in Listing 3, the push-pop pair of macros must bracket the cleanup code since **push()** starts with **do {** and **pop()** ends with **} while (0)**.

pthread_cleanup_push() and **pthread_cleanup_pop()** are best understood by reading their POSIX manual pages [manpages].

One point of caution concerns functions which contain multiple cleanup handlers. If a particular lock must be held when a particular cleanup action executes, then the declarations of the variables should be ordered accordingly. The implication is (gasp!) that some variable declarations must appear in the middle of the function body at first point of use, just as in C++.

Given the widespread acknowledgement that resource management in C error handlers without destructors is difficult, why isn't the adoption of cleanup handlers universal? For marquee C-language projects like Gstreamer and git, the answer is that they must compile for Windows. MSVC does not support **__cleanup__**, but favors its own **__try**/**__finally** variants [meneide]. The ISO C Committee is in the process of drafting a technical specification for a new **defer** keyword in the hope that all major toolchains will support it. **defer** would presumably be based on the cleanup attribute where it is supported. The draft goes into some detail in an attempt to constrain behavior and discourage anti-patterns. Whether the Committee will adopt the specification or toolchain vendors will implement it is, as always, an open question.

```
# define pthread_cleanup_push(routine, arg)  \
  do {                                       \
    struct __pthread_cleanup_frame __clframe \
      __attribute__ ((__cleanup__            \
        (__pthread_cleanup_routine)))        \
      = {.__cancel_routine = (routine),      \
         .__cancel_arg = (arg),              \
         .__do_it = 1 };

/* Remove a cleanup handler installed by the
matching pthread_cleanup_push. If EXECUTE is non-
zero, the handler function is called. */
# define pthread_cleanup_pop(execute)        \
    __clframe.__do_it = (execute);           \
  } while (0)
```

Listing 3

In the meantime, developers and maintainers of complex C code bases are encouraged to give cleanup macros a try. The three projects highlighted here are replete with helpful examples. Removing the last **goto** statement from a long function is a satisfying feeling, plus it's much less work than reimplementing the whole project in Rust. ■

References

[clang] https://github.com/llvm/llvm-project/commit/d277d790e0f6f23043397ba919619b5c3e157ff3

[clang-1] https://clang.llvm.org/docs/AttributeReference.html#cleanup

[defer-TS] https://thephd.dev/_vendor/future_cxx/papers/C%20-%20Improved%20__attribute__((cleanup))%20Through%20defer.html

[gcc] https://github.com/gcc-mirror/gcc/blob/58735cd75df57ca423dd6189e5f79f4d4686dfa2/gcc/ChangeLog-2003#L20790

[gcc-1] https://gcc.gnu.org/onlinedocs/gcc/Common-Variable-Attributes.html#index-cleanup-variable-attribute

[gcc-2] https://www.gnu.org/software/c-intro-and-ref/manual/html_node/Auto-Type.html

[gcc-3] https://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html

[glibc] https://github.com/bminor/glibc/blob/c9188d333717d3ceb7e3020011651f424f749f93/sysdeps/nptl/pthread.h#L585

[linux] https://github.com/linux4microchip/linux/blob/4d72aeabedfa202d12869e52c40eeabc5401c839/drivers/irqchip/irq-renesas-rzg2l.c#L596

[linux-1] https://github.com/chaiken/linux4microchip/blob/34307c7489f685d16a5a64f89699b4dea2e48fca/drivers/gnss/ubx.c#L1180

[linux-2] https://github.com/torvalds/linux/blob/39d3389331abd712461f50249722f7ed9d815068/include/linux/cleanup.h#L44

[manpages] https://www.man7.org/linux/man-pages/man3/pthread_cleanup_push.3.html

[meneide] https://thephd.dev/c2y-the-defer-technical-specification-its-time-go-go-go

[systemd] https://github.com/systemd/systemd/blob/51190631968f2a69acf5da3e3412b003805538f2/src/boot/util.h#L18

[systemd-1] https://github.com/systemd/systemd/blob/51190631968f2a69acf5da3e3412b003805538f2/src/basic/cgroup-util.c#L564

[systemd-2] https://github.com/systemd/systemd/blob/3e2a5dc2e17aae334d312d63e95159aff9ecaae5/src/basic/static-destruct.h#L9

C++ Standard Library: A Matter of Import

Modules are a relatively new C++ feature. Ian Bruntlett documents what he did to get them working with an older gcc compiler.

Motivation

Bjarne Stroustrup's current C++ primer, *Programming: Principles and Practice Using C++* (PPP3) [Stroustrup24] uses C++ modules and my system's gcc compiler does not support them. Bjarne has a habit of referring to very new facilities in his books – TC++PL2 (exceptions on Sun Workstations), I'm thinking of you! This article documents what I had to do to get 'Hello World' from PPP3 to build successfully on a different system's Ubuntu Linux 24.04.3 (or later) LTS x86_64 with not much additional software installed.

Learners using PPP3 who are using the Gnu Compiler Collection (gcc) either have to resort to using **#include**s or somehow get to grips with using modules. The problem with **#include**s is that they can be really inefficient. This is because when a **#include** is encountered, that file's contents – and the contents of any nested **#include**s – are compiled for each C++ source file (aka translation unit) being built that includes them. Modules provide a way to reduce that workload on the compiler whilst increasing the demands made on compiler authors.

This article relies on a collection of files to work. They will be made available on Github [Bruntlett].

Other build systems

There are all sorts of ways to optimise builds other than **make**. CMake is apparently the most popular build system generator but I am completely unfamiliar with it and my current goal is to work my way through Bjarne Stroustrup's book (PPP3), so I used GNU Make.

Before you start

By the time you read this, your compiler might already support modules and so most of this article will be redundant to you. Try building `greetings.cpp`, included as Listing 5 in this article. If it fails and you get told to install a newer compiler, to misquote Lestat "I'm going to give you the article I never had".

Getting import std; to work

For this article, initially I used Jonathan Wakely's 'Binary packages for GCC snapshots' [Wakely] web page to get a more up to date, if somewhat experimental, compiler. To get access to the compiler, you download the latest snapshot `.deb` file from the web page.

I had to install a package, `make`, as well as the compiler. To install `make`, I used these commands:

```
# get information on current packages
sudo apt update
# update the system with current packages
sudo apt upgrade
sudo apt install make
```

Ian Bruntlett Since getting to grips with Git, Ian has been re-learning C++ using *Programming: Principles and Practice Using C++* and also runs sessions of the sci-fi TTRPG (Mongoose) Traveller.

```
$ make help
makefile to build executables and also a bit of
housework.
make hello-std  - make Hello World programme that
does an import std;
make greetings  - a more modern Hello World
programme.
make hello-ppp3 - make Hello World programme that
uses the PPP module.
make all        - make module files and the
example programmes.
make std.o      - make the standard library
module files.
make PPP.o      - make utility module for the
book PPP3 \(Stroustrup\).
make                from the book PPP3.
make clean      - delete the executable files and
object files, typically done after updating the
compiler.
make look-for-compiler - see what the current
compiler version is
make get-the-compiler  - download the most recent
compiler
module_std = /opt/gcc-latest/include/c++/16.0.0/
bits/std.cc
g++ (GCC) 16.0.0 20260111 (experimental)
```

Figure 1

I created a makefile to automate building programmes and perform other tasks as well. In **make**, these tasks are called *targets*. For the sake of convenience, I added some extra targets in the makefile. If you're not sure what targets are, simply view these extra targets as commands you can request from the makefile at the command line. (See Figure 1 for the usage of **make**.)

The command **make look-for-compiler** looks at the gcc-latest web page and tells you what the latest version is.

```
$ make look-for-compiler
Checking the current version of the compiler
==================================================
wget --spider --content-disposition https://
kayari.org/gcc-latest/gcc-latest.deb 2>&1 | grep
Location
Location: https://kayari.org/gcc-latest/
gcc-latest_16.0.1-20260222git6e35a5d5b297.deb
[following]
```

The command **make get-the-compiler** downloads the latest gcc-latest `.deb` file *only* if it hasn't been downloaded into the current directory already. (See Figure 2, next page.)

The command to install the compiler, where 'name-of-gcc-latest-file.deb' is replaced by the name of the .deb file you downloaded is:

```
sudo apt install ./name-of-gcc-latest-file.deb
```

The makefile used in this article doesn't need `/opt/gcc-latest/bin` to be on the **PATH**. However, I found it convenient. I used the shell script in Listing 1 (next page) to do so and to change to a working directory.

Modules provide a way to reduce [the] workload on the compiler whilst increasing the demands made on compiler authors.

```
$ make get-the-compiler
Only downloading the compiler if we don't already
have it.
==================================================
wget --no-clobber  --content-disposition https://
kayari.org/gcc-latest/gcc-latest.deb
--2026-02-26 09:17:08--  https://kayari.org/gcc-
latest/gcc-latest.deb
Resolving kayari.org (kayari.org)...
176.126.242.131
Connecting to kayari.org (kayari.
org)|176.126.242.131|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://kayari.org/gcc-latest/gcc-
latest_16.0.1-20260222git6e35a5d5b297.deb
[following]
--2026-02-26 09:17:08--  https://kayari.org/gcc-
latest/gcc-latest_16.0.1-20260222git6e35a5d5b297.
deb
Reusing existing connection to kayari.org:443.
HTTP request sent, awaiting response... 200 OK
Length: 463517736 (442M) [application/vnd.debian.
binary-package]
Saving to: 'gcc-latest_16.0.1-
20260222git6e35a5d5b297.deb'

gcc-latest_16.0.1-20260222git6e35a5d5b2 100%[====
==================================================
==================>] 442.04M  11.2MB/s    in 40s

2026-02-26 09:17:48 (11.2 MB/s) - 'gcc-
latest_16.0.1-20260222git6e35a5d5b297.deb' saved
[463517736/463517736]
```

Figure 2

I invoke it on the command line with:

```
source prepare-for-gcc-latest
```

or the briefer:

```
. prepare-for-gcc-latest
```

I do this at the command line rather than in `.bashrc` because I don't want to hide the standard gcc, which is relied upon by vlc and other packages.

You can use 'TAB-completion' for the command line as well. Type in the shortest unique prefix and then press TAB for the command line to be auto-completed.

```
#!/bin/bash
if ! echo $PATH | grep gcc-latest/bin: - ; then
    echo gcc-latest not on path... putting it on
    set -x
    PATH=/opt/gcc-latest/bin:$PATH
    cd ~/gcc-new
    set +x
else
    echo gcc-latest already on path
fi
```

Listing 1

You can compile your own code (e.g. for `another-example.cpp` that uses the PPP module) by editing the makefile (see Listing 2) and adding something like this line:

```
another-example: std.o PPP.o another-example.cpp
```

```
# makefile to use when working on examples for
# Bjarne Stroustrup's book, "Programming
# Principles and Practice Using C++" (3rd
# Edition), working on examples using Ubuntu
# 24.04.4 LTS x86_64.
#
# Ian Bruntlett, January 2026 - 26th February
# 2026 9:50ish
# with help on the accu-general mailing list from
# Jonathan Wakely
# and on the accu-members for people to test
# these example programmes.

# Tell the C++ compiler to include debugging
# information (-g), have # some useful warnings,
# that modules are in use (-fmodules), and
# that we are using a bleeding edge version of
# the C++ language.

# tell make where to find the C++ compiler
CXX = /opt/gcc-latest/bin/g++

# flags for the C++ compiler
CXXFLAGS = -g -Wall -Wshadow -Wconversion
-fmodules -std=c++23

# the names of the programmes built using this
# makefile
executables = hello-std hello-ppp3 greetings

# website address for gcc-latest, so we can check
# its version - $ make look-for-compiler
# or download it - make get-the-compiler
compiler_link = https://kayari.org/gcc-latest/
gcc-latest.deb

module_std=$(shell find /opt/gcc-latest/ -iname
std.cc)

# linker flags, to locate library files without
# having to set:
# export LD_LIBRARY_PATH=/opt/gcc-latest/lib64
LDFLAGS += -Wl,-rpath,/opt/gcc-latest/lib64

# if you want to create a linker map file for
# extra info on your executables then uncomment
# this line below:
# LDFLAGS += -Wl,-Map=$@.map

all : $(executables)

# Note that commands are preceded by a TAB
# character and not a sequence of space
# characters.
```

Listing 2

```
# Build a module for using the Standard C++
# Library
# gcm.cache/std.gcm - the
# 'compiled module interface'
# std.o - contains the modules definitions.
std.o: $(module_std)
   $(CXX) $(CXXFLAGS) -c $(module_std)

# Build a module for using the examples from the
# book PPP
# gcm.cache/ppp.gcm - the ‘compiled module
# interface' for PPP extensions
# PPP.o - contains the PPP extensions
definitions.
PPP.o: PPP.ixx
   $(CXX) $(CXXFLAGS) -c PPP.ixx

# build an executable file that uses the Standard
# Library module
hello-std: std.o hello-std.cpp

# build an executable file that greets the user
# using modern C++ facilities
greetings : std.o greetings.cpp

# build an executable file that uses the PPP
# module from the book PPP3
hello-ppp3: std.o PPP.o hello-ppp3.cpp

.PHONY: clean
clean:
   rm -rfv gcm.cache/
   rm -fv $(executables) *.o *.map

# check to see the version information of the
# current compiler
.PHONY: look-for-compiler
look-for-compiler:
   @echo Checking the current version of the
compiler
   @echo =====================================
   wget --spider --content-disposition $(compiler_
link) 2>&1 | grep Location

# download the current compiler
.PHONY: get-the-compiler
get-the-compiler:
   @echo Only downloading the compiler if we
don\'t already have it.
   @echo ========================================
   wget --no-clobber  --content-disposition
$(compiler_link)

# display help message
.PHONY: help
help:
   @echo makefile to build executables and also a
bit of housework.
   @echo "make hello-std  - make Hello World
programme that does an import std;"
   @echo "make greetings  - a more modern Hello
World programme."
   @echo "make hello-ppp3 - make Hello World
programme that uses the PPP module."
   @echo "make all        - make module files and
the example programmes."
   @echo "make std.o      - make the standard
library module files."
   @echo "make PPP.o      - make utility module
for the book PPP3 \(Stroustrup\)."
   @echo "make             from the book PPP3."
   @echo "make clean      - delete the executable
files and object files"
   @echo "                 typically done after
updating the compiler."
   @echo "make look-for-compiler - see what the
current compiler version is"
   @echo "make get-the-compiler  - download the
most recent compiler"
   @echo module_std = $(module_std)
   @$(CXX) --version | head -1
```

Listing 2 (cont’d)

```
# PPP files from
# https://stroustrup.com/programming.html
# a few programmes by me, a makefile, a
# supporting shell script, and my draft article
module-article.tar.gz : PPP.ixx PPP.h PPP_
support.h PPPheaders.h\
   hello-std.cpp hello-ppp3.cpp greetings.cpp \
   makefile prepare-for-gcc-latest \
   irb-cpp-stdlib-a-matter-of-import.odt \
   irb-delete-elves
   tar -czf $@ $^
```

Listing 2 (cont’d)

Listing 3 contains the code (`hello-std.cpp`) I used to test my environment.

g++ currently implements modules by creating a gcm.cache directory and creates an ELF (Executable and Linker Format) file named after the module it contains – in this case `std.gcm`. To get extra information about the ELF file, you can use the command:

```
$ readelf -a std.gcm | less
```

Getting PPP3 examples to build

This article only covers the building of text mode modules. Those PPP3 examples that use graphics with Qt will have to wait. I went to https://stroustrup.com/programming.html and downloaded `PPP.ixx`. `PPP_support.h`, and `PPP.h` – this gives you the means to build the PPP module. However, I got a compilation failure for **`PPP_support`** as **`size_t`** was not declared. To get this to work, I edited `PPP.ixx` to read as follows:

```
export module PPP;

export import std;

using std::size_t; // added by me to get things
                   // to build
#define PPP_EXPORT export
#include "PPP_support.h"
using namespace PPP;
```

```
/* hello-std.cpp

compile with:
make hello-std

Whilst Bjarne Stroustrup likes to use this in his
examples:
  using namespace std;

I prefer not to do so. This is because in medium-
sized programmes and larger, this can cause name
collisions and headaches so I have to prefix
names from the standard library with std::
*/

import std;

int main()
{
  std::cout << "System  : " ;
  std::flush(std::cout);
  std::system ("hostname");
  std::cout << "FILE    : "  __FILE__ " "
    __DATE__  " " __TIME__ "\n";

  std::cout << "gcc     : "  << __GNUC__ << '.'
            << __GNUC_MINOR__ << '.'
            << __GNUC_PATCHLEVEL__  << '\n';

  std::cout << "standard: " <<  __cplusplus
            << '\n';
}
```

Listing 3

To see if I could access the PPP module, I then created a new file, `hello-ppp3.cpp` (Listing 4).

A more modern Hello World

As recommended by Lieven de Cock on the accu-members email mailing list, Listing 5 (next page) is a more modern take on "Hello, World", using **`std::println`** and **`std::source_location`**.

Compiling your own code

As you write your C++ code, you could modify the makefile to build your code OR, if you are dealing with individual source files (e.g. I wrote a small programme for the exercise on page 50 of PPP3, called: `50-ex-11-how-many-coins.cpp`. To get it to compile and link, all I had to do was:

```
$ make std.io PPP.o 50-ex-11-how-many-coins
```

and **`make`** used its knowledge to execute this command line:

```
/opt/gcc-latest/bin/g++ -g -Wall -Wshadow
-Wconversion -fmodules -std=c++23 -c /opt/gcc-
latest/include/c++/16.0.0/bits/std.cc
/opt/gcc-latest/bin/g++ -g -Wall -Wshadow
-Wconversion -fmodules -std=c++23 -c PPP.ixx
/opt/gcc-latest/bin/g++ -g -Wall -Wshadow
-Wconversion -fmodules -std=c++23  -Wl,-rpath,/opt/
gcc-latest/lib64  50-ex-11-how-many-coins.cpp   -o
50-ex-11-how-many-coins
```

If you have already built `std.o` or `PPP.o`, **`make`** will refrain from re-building those files as they are up to date.

```
/* greetings.cpp

   compile with:
   make greetings

Sources:
https://en.cppreference.com/w/cpp/utility/source_
location.html
https://en.cppreference.com/w/cpp/io/println.html

Whilst Bjarne Stroustrup likes to use this in his
examples:
   using namespace std;

I prefer not to do so. This is because in medium-
sized programmes and larger, this can cause name
collisions and headaches so I have to  prefix
names from the standard library with std::
*/

import std;

int main()
{
  std::source_location here
    = std::source_location::current();
  std::println("Greetings from {},
       function {} @ {}:{}",
       here.file_name(),
       here.function_name(),
       here.line(),
       here.column()
       );
}
```

Listing 5

```
/* hello-ppp3.cpp
  Programme to use some of PPP3's code - in this
  case, a checked string which will complain at
  runtime when abused.

  compile with:
  make hello-ppp3

  Bjarne Stroustrup likes to use this in PPP.h:
   using namespace PPP;
   using namespace std;
  This makes code easier for a novice reader.

  I prefer not to do so. This is because in
  medium-sized programmes and larger, this can
  cause name collisions and headaches.

  However, PPP.h is used by this program which
  means I can write cout instead of std::cout.
*/

#include "PPP.h" // Use Bjarne's PPP3 tutorial
                 // module

int main()
{
  cout << "Hello from : "  __FILE__ "\n";

  // this line use's Bjarne's PPP::Checked_string
  // because PPP.h does this preprocessor
  // trickery:
  // #define string Checked_string

  string astring="Hello\n";

  cout << astring;
  cout << "About to get a deliberate runtime "
         "error\n";
  // get a runtime error, to show that
  // PPP::Checked_string is working.
  astring[100]='!'; // bang! deliberately exceed
                    // the string's text
}
```

Listing 4

Updating your compiler

When your compiler gets updated, I'd advise you to delete your object files and executables and then rebuild the executables, typically done via a 'make clean all' command. At the moment, this requires looking at Jonathan Wakely's webpage, waiting for a new release, and then downloading and installing as previously discussed. ■

GNU Software manuals

- C++ Standards support in GCC: https://gcc.gnu.org/projects/cxx-status.html
- GNU Bash manual: https://www.gnu.org/software/bash/manual/
- GNU Make manual: https://www.gnu.org/software/make/manual/
- GNU Compiler & library manuals: https://www.gnu.org/software/gcc/onlinedocs/
- GNU Linker manual: https://sourceware.org/binutils/docs/

More resources

A fair amount of articles have been written and presentations given on modules; here are a couple.

- 'C++ Modules: A Brief Tour' by Nathan Sidwell, in *Overload* 159, available from the archive at: https://accu.org/journals/overload/28/159/sidwell/
- 'Modules (since C++20), available from *C++ reference* at: https://en.cppreference.com/w/cpp/language/modules.html

References

[Bruntlett] Files associated with this article: https://github.com/ian-bruntlett/studies/tree/main/cpp/PPP3/gcc-latest

[Stroustrup24] Bjarne Stroustrup (2024) *Programming: Principles and Practice Using C++* (3rd edition), Addison-Wesley Professional.

[Wakely] Jonathan Wakely, 'Binary packages for GCC snapshots', at https://jwakely.github.io/pkg-gcc-latest/

accu

Professionalism in Programming

World-class conference

Where:

Leas Cliff Hall

Folkstone, Kent, UK

Jointly with C++ on Sea

Workshops:

15 & 16 June 2026

Conference:

17–20 June 2026

Visit accuconference.org for details

Vulture Culture

How do you get companies, products and rockets off the ground? Teedy Deigh spends time with us trying to find out.

This month in *Vulture Culture*, the corner of *Vibes and Ventures* where we try to plaster a more human face on the shallow greed and Kool-Aid drinking we're often accused of, we talk to innovator, thought-leader and burn-rate-maxxer Teedy Deigh. We like to profile and interview the people trying to make you – and themselves – money, whatever the cost.

Vulture Culture So, Teedy, let's dive straight into the cliches. What is that gets you, a thrusting entrepreneur, out of bed in the morning?

Teedy Deigh Most thrust comes from coffee. On the advice of my doctor I have decaf, but it's important to put one's own spin on any advice, to make it your own, so I mix that with regular coffee – if we're happy calling two double espresso shots 'regular'.

Morning, however, is a relative term, and often not one I approach from bed.

VC Sounds like you have some kind of 3AM exercise or meditation regime. Is it something you could share with our audience? Something that helps offset guilt, or papers over sociopathy, or just sounds a little bit try-hard?

TD What I mean is that I typically burn copious amounts of midnight oil, as well as at least one candle from both ends. If I see dawn, it's only a glimpse receding in the rearview mirror of an ever-growing carbon footprint.

VC That's the kind of sleep-is-for-wimps attitude investors like to hear.

TD Well, it's either that, or I'm fast asleep until noon is little more than yet another a missed milestone in the day's schedule.

VC Sounds like you thrive on chaos.

TD I don't over-plan.

VC Do you find that fosters success?

TD Well, a certain amount of disorder and procrastination can save me from backing the wrong horse. Remember the hype about cryptocurrencies displacing actual currencies rather than becoming speculative assets? And all that web3 FOMO?

VC Yes, many of our subscribers went all in on that.

TD Not me. With so many other things going on, when I finally got round to it there was nothing to get round to.

But sometimes, dodging the bullet is more a matter of consideration rather than one of luck. Remember all the fuss around the metaverse?

VC My goodness, yes, I still have many property holdings there. Are you saying you saw it coming?

TD I'm saying that trying to build on an SF trope from the 1980s while pretending that the 1990s and three decades of the Web hadn't happened might not be a solid business plan.

VC Insider knowledge. Useful.

TD In common with many other technologist founders, I'm a big fan of speculative fiction. But unlike many of them, I actually read the books and watch the films and – here's the real value add – try to *understand* what they're about.

So, for example, I get that cyberpunk is a dystopian critique of technocorporatism, that the societal decimation and decay it portrays is not something we should be striving to create, that a low-life/high-tech aesthetic is only cool when you're looking at it not when you're living it.

This sense and sensibility – coupled with erratic hyperfocus – has helped me steer clear of many mistakes in waiting.

Many, but sadly not all.

VC Which brings us to the question of failure, and how any entrepreneur must inevitably deal with it.

TD Is this the bit where I say how important it is to celebrate failure? It is, isn't it?

VC And, if you don't mind me saying, you have much to celebrate!

TD I'm a serial entrepreneur. Sometimes parallel.

VC Could you spotlight a couple of examples so our audience can pass those off as their own anecdotes, while being surprised when similar circumstances arise again?

TD One thing I've learned – or will at least, for the purposes of this discussion, say I've learned – is not to be too literal about some terms and metaphors.

Did I make it big in cloud computing? No... no, I did not. We were all told having the right cloud strategy was important. Hiring a bunch of meteorologists, however, turned out not to be that strategy. That one was on me. But at least it wasn't my start-up.

As for serverless, yeah, I might have taken that a little too literally. On the plus side, although it was my company, its existence was mercifully short. When that would-be angel investor pointed out the terminology snafu, it was quite the meeting stopper. And company stopper. From start-up to shut-down in less than a month. Quite the record! The complete absence of servers meant our capital expenditure had been quite low, so not much to write-off.

And for all my SFF savvy, it's not always obvious that some fictional technology is not something we should be pursuing.

VC Ah yes, this would be the Torment Nexus incident.

Teedy Deigh When it comes to innovation, Teedy Deigh considers herself second to none. Possibly closer. When it comes to hype, she has seen more cycles than the Tour de France, and has helped to pivot (not always intentionally) many start-ups. When it comes to geeky interests, fan culture and all round nerdery, she has found many hills to die on, but has so far enjoyed the good fortune not to have this tested. All in all, she believes these qualities put her ahead of the usual, run of the hamster mill tech bros.

We're applying the principles of vibe coding to rocketry. The LGTM launch vehicle is going to be our flagship product. That said, we're having some trouble getting it off the ground.

TD Indeed. It was not at all clear that we should not create the Torment Nexus. Many companies, mine included, tried to establish – or at least bubble – a market for it. There was neither clue nor caution in the original *Don't Create the Torment Nexus* novel that we shouldn't be doing this.

VC None at all. Caught a lot of us by surprise, that one.

Are there any products you've decided to shutter early rather than have the market crash them or investors pull the plug first?

TD I recently abandoned a side project to create a platform – social media, AI chatbot, both, I hadn't decided – that doubles as a deepfake porn site. The market is already too crowded. Demagnetised moral compasses are surprisingly common in this space.

VC And, something that will be of particular interest to our audience, is there anything you're looking to get funding for at the moment?

TD A couple of things, as it happens. The one I'm furthest along with is the Moredoor security system. It's like a security system in that, to all intents and purposes, you shall not pass – hence the company name, Balrog – but all of its users will have been deceived as we have intentionally installed multiple backdoors for government – or highest bidder – use. Apart from that, it's completely secure.

It relies heavily on the principle of trust – trust in Balrog, specifically – and we are ramping up marketing to build trust in that trust, hence we're looking for more rounds of funding. We've had some client interest from a number of government bodies, including minipax and miniluv, so this looks like a solid investment.

Another more speculative project involves space. Initially, we want to provide access to low Earth orbit before moving on to our long-term goal of asteroid mining and, in the best traditions of the free market, dominating the global metals market to the point of monopoly. Ideally, we would do so without crashing the economy by flooding the market with a surplus – taking the rare out of rare earth metals! – but it's early days yet.

VC You mentioned low Earth orbit. How are you starting out?

TD We're applying the principles of vibe coding to rocketry. The LGTM launch vehicle is going to be our flagship product. That said, we're having some trouble getting it off the ground.

VC Is that why you need the funding?

TD That too. But literally, we're having trouble getting it off the ground. We 3D print it from vibed blueprints. Those plans and the resulting rocket look pretty plausible. You know, it's got nozzles, tanks, pipes, wiring... all the rockety-looking bits. But when you turn it on, nothing happens. Not even with a performative countdown incantation. Not even when you turn it off then on again.

VC Rockety-looking bits? What is your background in astronautical engineering?

TD Non-existent. But that's not going to stop me. I'm not held back by embedded preconceptions and traditional workflows. That's what it takes to be an entrepreneur. If someone says something can't – or shouldn't – be done, ignore them. It's not like this is rocket science.

VC Umm...

TD Me and the other founders at the Elysium Foundation are inspired by the utopian ideals and cool space habitat from the film *Elysium*.

VC Doesn't that film paint a bleak picture of the 22nd century? Global poverty, supressed employment rights, inaccessible healthcare and so on?

TD Only on Earth. We're not interested in that part of it.

VC As a foundation, are you aiming to be a nonprofit? Something charitable?

TD Oh, don't be misled by the name. We only added 'Foundation' to the name because we like Asimov. We really identify with the Empire.

VC Aren't they the bad guys?

TD Tomayto, tomarto. In one sense, aren't we all? But just 'cos we're the bad guys doesn't mean we're bad guys, right?

VC Right.

That's the kind of positive, feelgood note we like to end on. Hopefully, by exploring Teedy's own story we've managed to pin a more human tale on the money-making ass of tech. ■

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

To connect with like-minded people visit accu.org

accu

“The magazines”

The ACCU's *C Vu* and *Overload* magazines are published every two months, and contain relevant, high quality articles written by programmers for programmers.

“The conferences”

Our respected annual developers' conference is an excellent way to learn from the industry experts, and a great opportunity to meet other programmers who care about writing good code.

“The community”

The ACCU is a unique organisation, run by members for members. There are *many* ways to get involved. Active forums flow with programmer discussion. Mentored developers projects provide a place for you to learn new skills from other programmers.

“The online forums”

Our online forums provide an excellent place for discussion, to ask questions, and to meet like minded programmers. There are job posting forums, and special interest groups.

Members also have online access to the back issue library of ACCU magazines, through the ACCU web site.

ACCU | JOIN: IN

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

Invest in your skills. Improve your code. Share your knowledge.

Join a community of people who care about code. Join the ACCU.

Use our online registration form at **www.accu.org**.

Design: Pete Goodliffe