

overload 180

APRIL 2024

£4.50

C++ Safety, In Context

Herb Sutter discusses C++'s
current security problems
and potential solutions

User-Defined Formatting in std::format

Spencer Collyer demonstrates how to provide
formatting for a simple user-defined class

To See a World in a Grain of Sand

Jez Higgins shows how to refactor code that has grown
organically, making it clearer and more concise

Judgment Day

Teedy Dee finds out what happens if AI takes your job

ACCU



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

ACCU.org

April 2024

ISSN 1354-3172

EditorFrances 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

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.awj@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover designOriginal design by Pete Goodliffe
pete@goodliffe.net

Cover photo by Daniel James, of a double row of the 'colours' of the Royal Tank Regiment that can be seen in the church of St Mary Aldermery.

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 C++ Safety, In Context

Herb Sutter discusses C++'s current security problems and potential solutions.

14 To See a World in a Grain of Sand

Jez Higgins shows how to refactor code that has grown organically, making it clearer and more concise.

20 User-Defined Formatting in std::format

Spencer Collyer demonstrates how to provide formatting for a simple user-defined class.

27 Judgment Day

Teedy Dee finds out what happens if AI takes your job.

Copy deadlines

All articles intended for publication in Overload 181 should be submitted by 1st May 2024 and those for Overload 182 by 1st July 2024.

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.

I Don't Believe It!

Sometimes we are surprised by unexpected outcomes or how long things take. Frances Buontempo confesses to how she's lost hours recently, but learnt from the experiences.

I recently spoke at CppOnline [CppOnline], a new online-only conference. It was loads of fun, though it always feels odd talking to your monitor and hoping someone is listening. We were advised to close unnecessary applications and browser tabs down to ensure smooth performance of our machines while we spoke. You may find this hard to believe, but I spent about four hours closing browser tabs, taking up time I could have otherwise spent on an editorial. I currently have 62 open; a grand improvement on the 99 or more before the conference. No editorial though, sorry.

If you're not a tab hoarder you might find spending so much time closing tabs very strange, but I know I am not the only person who does this. I could just bookmark pages, but I gave up on bookmarks years ago, because links went stale and I had so many I couldn't find anything. If I have a tab open, it's usually something I do want to read or listen to at some point, and then maybe make notes or buy music or similar. One tab I closed was for a new turntable, because our old one seemed to have stopped working. I bit the bullet and bought the new turntable. It's excellent and in the process of setting it up, I discovered why the old turntable didn't work. The pre-amp was unplugged. The new bit of kit does have a USB port though, so I can record all my old records one day. Closing that tab was expensive, informative and has probably caused another time consuming job.

Another tab was *The Return of -1/12* by Numberphile on YouTube [Numberphile]. They discussed infinite series. As many of you know, $1 + \frac{1}{2} + \frac{1}{4} + \dots$ equals 2. We can prove this, since writing

$$S = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots$$

means

$$2S = 2 + 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots$$

which tells us when we subtract both we get $2S - S = S = 2$. QED. That doesn't seem unreasonable. However, if we were now to try writing $1 + 10 + 100 + \dots$ we get into trouble. Writing

$$S = 1 + 10 + 100 + \dots$$

would mean we could have

$$10S = 10 + 100 + 1000 + \dots$$

so we would then be claiming $10S - S = 9S = -1$. I'm not sure about you, but this suggests the sum, S , is $-1/9$, which seems very unlikely. Of course,

there is a restriction on the terms of the infinite sum. The terms need to decrease by enough so that we can actually write the equals sign, otherwise the sum doesn't converge on a

number and we end up with unbelievable nonsense. Maybe you already know about infinite series and analytic continuations [Wikipedia], which allow us to extend the domain of functions. They are not to be confused with algebraic continuations which allow us to continue execution using futures and similar, and might mean I end up with more tabs open again were I to try to explain in detail. The take away message is that reasoning is often caveated with prerequisites; for example, a radius of convergence for a series. Applying similar logic in different circumstances may lead to surprises or mistakes. If something seems unbelievable, like adding positive numbers and getting a negative answer, an assumption you are making might be wrong.

A relevant computing example concerns benchmarking. A long time ago, Roger Orr wrote an article entitled 'Order notation in practice', based on his talk at an ACCU conference [Orr14]. He demonstrated various factors which also influence the performance of an algorithm besides its complexity measure. He discussed `strlen`, and discovered many compilers had optimised away the call, so the theory didn't match the practice. Trying to build up an intuition about possible outcomes, so you spot when something is amiss, is an important skill, so well spotted Roger. Kevin Carpenter talked about building intuition at MeetingCpp [Carpenter23], and discussed making educated guesses, which may or may not be true. I couldn't attend his talk, because it clashed with mine, so I had a tab open to listen at some point. Fortunately, I managed to catch his re-working of the talk live at CppOnline and even ask a question. So, I closed another tab.

Our intuition can be wrong, but we need to start somewhere. Lots of interesting mathematics falls out of proving a first guess is incorrect, or finding circumstances under which the ordinary does not happen, leaving us with something extraordinary. And wondering what-if can be fruitful. Whether that's imagining a square root of -1, or exploring what is possible at compile time, new disciplines emerge. However, sometimes wondering why we have 5 test cases for a function with 7 `if/else` branches leads us to deduce we can delete the extra branches. The tests may still pass, however there's a chance someone forgot to add more tests when they added more code. Mutation testing might well pick this kind of thing up. If you're not familiar with this, at a high level it randomly mutates the code, dropping branches, changing + to - and similar, and reports back if any tests still pass. Filip von Laenen wrote an article about mutation testing for us back in 2012 [vonLaenen12] if you want to know more. He did say at the time he wasn't a C++ programmer so could only give details on other languages and mention a couple of frameworks in C++ he was aware of. Perhaps the time has come for someone to write a new article telling us about current tools?

Tests for branches in code came to mind because Jez Higgins recently touted [Higgins24a] about some flappy code he refactored, which had



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

more branches than tests. Of course, a code coverage tool should pick that up, though mutation testing may find other problems. Jez spotted this by eye from simply looking at the code and wrote about this in a blog [Higgins24b]. Thankfully, he has followed it up with the refactorings to make the code better, and allowed us to include the write up in this issue. The code he considers in his blog is unbelievable, but untidy and confusing code does emerge over time, and you need to find time to tidy up once in a while, otherwise the weeds grow and take over. As a side note, we caught up with Jez at the Norfolk Developers Conference [NorDev], which a handful of ACCU people based in the UK go to. Jez didn't have a ticket for the speakers' dinner, so found an EMF gig in town that evening instead. Unbelievable. (Possibly a niche joke if you don't know the band EMF, but here's a famous song by them [EMF]: *You're unbelievable*. Apologies).

I picked the title 'I don't believe it' based on an oft-repeated phrase by a TV character, Victor Meldrew [IMDB]. A variety of slightly unlikely things happen to him, and he usually responds with a variation of the phrase "I don't believe it." I caught myself saying this a few times recently, and treating that as a warning because the character is a slightly sulky old man. Not something to aspire to. Now, not all unbelievable things are negative. For example, finding a gig at the last minute is a nice surprise. Fighting some code for a couple of hours and finding it compiles is always a surprise too, but often leaves you wondering if it really works. Life is so much calmer if you can take tiny baby steps to refactor something. I hope Jez does write up his refactoring steps – maybe we can see this as an article in *Overload*. Refactoring is an important skill, and I suspect many of us still have lots to learn.

As languages change, we need to keep learning. It's never easy, and I don't know about you, but I am often surprised when I come across things I hadn't noticed before. One of the many tabs I closed was from *CppReference*, telling me all about `std::piecewise_construct` [CppRef-1]. (Aside: you know I am reopening these tabs to double check what they say as I write: place bets on my tab count when I'm done.) The `std::piecewise_construct_t` is an empty class tag type and is used to differentiate between functions taking a tuple of two elements and those taking two arguments directly. In contrast, the next tab told me about `std::forward_as_tuple` [CppRef-2]. This allows me to construct a tuple of references to forward as an argument to a function. *CppReference* gives an example using a map:

```
std::map<int, std::string> m;
```

We can then add a value like this:

```
m.emplace(std::piecewise_construct,
           std::forward_as_tuple(10),
           std::forward_as_tuple(20, 'a'));
```

How we ended up needing this, I can only imagine. Perhaps someone will write in and tell me? Seriously, if you do fall across something in C++, or any language, you hadn't spotted before, write a page for us and send it my way. Let's help each other learn. There will be motivating examples and reasons behind the piecewise construct and forward as tuple. I just haven't followed this up, because my tab count has now hit 68. I could wander over to the bookcase and look it up in a book instead, but then I definitely wouldn't get an editorial written.

Talking of obscure parts of C++, I have been reviewing a manuscript for a potential book, and noticed a sidebar claiming C++23 added the new keyword `really`. My first instinct was, oh no, yet another thing I didn't notice. The writer had not explained what it did or why it was introduced, so like a sucker I opened yet another tab or three, and went hunting. I did find a blog post [D'Angelo22] which has the subtitle 'A blog for April Fool Day', which explains a function taking an `int`, say `f(int x)`, can be called with a double, so the new keyword would allow us to say `f(really int x)`. As for the manuscript I am reviewing, I am tempted to add a link to the xkcd Wikipedian Protestor holding a banner saying "[Citation needed]" [xkcd]. Writers do get things wrong, but hopefully our *Overload* review team spot any such inexactitudes. Do let us know if we missed anything though.

Forming an intuition takes time and sometimes helps us to form correct instincts, though we all get things wrong from time to time. Again, the counterintuitive results in mathematics, or any discipline, often lead to novel approaches and concepts. This is a good thing. Furthermore, if you get to a point where you think you are so good at something you could do it with your eyes shut, you often get a wake-up call. Again, this is a good thing, because it should encourage you to up your game and keep learning. Hopefully you won't turn into Victor Meldrew, moaning and complaining, while muttering "I don't believe it" instead. The unfamiliar is an opportunity. I recall a discussion about Duff's device [Wikipedia-2] when I had been programming for a living for a year or so and thought I knew it all. This stopped me in my tracks. I still have to concentrate on how the loop unrolling works and what is going on. It's weird, confusing and kinda beautiful all at once. I suspect most programmers enjoy slightly surprising edge cases and unusual ways to do things, because we enjoy thinking and learning.

What have we learnt? Citations are a good thing, because at least they may stop you falling for an April Fools' joke. Some things are unbelievable because they are incorrect and based on false assumptions. Other things are unbelievable because we just discovered a whole new approach. Let's check our results from time to time, and try to avoid resting on our laurels. Surprises can be annoying, but they can be wonderful too. And, 64 tabs, in case you wondered.

References

- [Carpenter23] Kevin Carpenter, 'Tooling Intuition', presented at *Meeting C++ 2023*, available at <https://www.youtube.com/watch?v=mmdoDfw9tIk>
- [CppOnline] <https://cpponline.uk/>
- [CppRef-1] CppReference: `std::piecewise_construct`, https://en.cppreference.com/w/cpp/utility/piecewise_construct
- [CppRef-2] CppReference: `std::forward_as_tuple` https://en.cppreference.com/w/cpp/utility/tuple/forward_as_tuple
- [D'Angelo22] Giuseppe D'Angelo, 'C++23 will be really awesome', available at <https://www.kdab.com/cpp23-will-be-really-awesome/>
- [EMF] 'You're unbelievable' performed by EMF: <https://www.youtube.com/watch?v=g4gU74gMbp0>
- [Higgins24a] Jez Higgins, March 2024, <https://mastodon.me.uk/@jehzhiggins/112039275413895974>
- [Higgins24b] Jez Higgins, 'To see a world in a grain of sand', blog post published 24 February 2024 at <https://www.jezuk.co.uk/blog/2024/02/to-see-a-world-in-a-grain-of-sand.html>
- [IMDB] Victor Meldrew, character from *One Foot in the Grave*: <https://www.imdb.com/title/tt0098882/characters/nm0934014>
- [NorDev] Norfolk Developers Conference: <https://nordevcon.com/>
- [Numberphile] Tony Feng 'The Return of -1/12', uploaded February 2024, available at <https://www.youtube.com/watch?v=FmLGN8ZGdw>
- [Orr14] Roger Orr, 'Order Notation in Practice' in *Overload* 124, December 2014, https://accu.org/journals/overload/22/124/orr_2043/
- [vanLaenen12] Filip van Laenen 'Mutation Testing' in *Overload* 108, April 2012, <https://accu.org/journals/overload/20/108/overload108.pdf#page=17>
- [Wikipedia-1] Analytic continuation: https://en.wikipedia.org/wiki/Analytic_continuation
- [Wikipedia-2] Duff's device: https://en.wikipedia.org/wiki/Duff%27s_device
- [xkcd] <https://xkcd.com/285/>

C++ Safety, In Context

The safety of C++ has become a hot topic recently. Herb Sutter discusses the language's current problems and potential solutions.

We must make our software infrastructure more secure against the rise in cyberattacks (such as on power grids, hospitals, and banks), and safer against accidental failures with the increased use of software in life-critical systems (such as autonomous vehicles and autonomous weapons).

The past two years in particular have seen extra attention on programming language safety as a way to help build more-secure and -safe software; on the real benefits of memory-safe languages (MSLs); and that C and C++ language safety needs to improve – I agree.

But there have been misconceptions, too, including focusing too narrowly on programming language safety as our industry's primary security and safety problem – it isn't. Many of the most damaging recent security breaches happened to code written in MSLs (e.g., Log4j [CISA-1]) or had nothing to do with programming languages (e.g., Kubernetes Secrets stored on public GitHub repos [Kadkoda23]).

In that context, I'll focus on C++ and try to:

- highlight what needs attention (what C++'s problem *is*), and how we can get there by building on solutions already underway;
- address some common misconceptions (what C++'s problem *isn't*), including practical considerations of MSLs; and
- leave a call to action for programmers using all languages.

tl;dr: I don't want C++ to limit what I can express efficiently. I just want C++ to let me enforce our already-well-known safety rules and best practices by default, and make me opt out explicitly if that's what I want. Then I can still use fully modern C++... just nicer.

Let's dig in.

The immediate problem “is”...

The immediate problem *is* that it's Too Easy By Default™ to write security and safety vulnerabilities in C++ that would have been caught by stricter enforcement of known rules for *type*, *bounds*, *initialization*, and *lifetime* language safety.

In C++, we need to start with improving these four categories. These are the main four sources of improvement provided by all the MSLs that NIST/NSA/CISA/etc. recommend using instead of C++ [CISA-2], so by definition addressing these four would address the immediate NIST/NSA/CISA/etc. issues with C++. (More on this under ‘What the problem “isn't”...’, section (1) on page 6.)

And in all recent years including 2023 (see Figure 1's four highlighted rows – rows 1, 4, 7 and 12 – and Figure 2), these four constitute the bulk of those oft-quoted 70% of CVEs (Common [Security] Vulnerabilities

Some background

Scope. To talk about C++'s current safety problems and solutions well, I need to include the context of the broad landscape of security and safety threats facing all software. I chair the ISO C++ standards committee and I work for Microsoft, but these are my personal opinions and I hope they will invite more dialog across programming language and security communities.

Acknowledgments. Many thanks to people from the C, C++, C#, Python, Rust, MITRE, and other language and security communities whose feedback on drafts of this material has been invaluable, including: Jean-François Bastien, Joe Bialek, Andrew Lilley Brinker, Jonathan Caves, Gabriel Dos Reis, Daniel Frampton, Tanveer Gani, Daniel Griffing, Russell Hadley, Mark Hall, Tom Honermann, Michael Howard, Marian Luparu, Ulzii Luvsanbat, Rico Mariani, Chris McKinsey, Bogdan Mihalcea, Roger Orr, Robert Seacord, Bjarne Stroustrup, Mads Torgersen, Guido van Rossum, Roy Williams, Michael Wong.

Terminology. (See ISO/IEC 23643:2020 [ISO]). *Software security* (or *cybersecurity* or similar) means making software able to protect its assets from a malicious attacker. *Software safety* (or *life safety* or similar) means making software free from unacceptable risk of causing unintended harm to humans, property, or the environment. *Programming language safety* means a language's (including its standard libraries') static and dynamic guarantees, including but not limited to type and memory safety, which helps us make our software both more secure and more safe. When I say *safety* unqualified here, I mean programming language safety, which benefits both software security and software safety.

1	CWE-787	Out-of-bound Write	63.72
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.54
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	34.27
4	CWE-416	Use After Free	16.71
5	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	15.65
6	CWE-20	Improper Input Validation	15.5
7	CWE-125	Out-of-bounds Read	14.6
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.11
9	CWE-352	Cross-Site Request Forgery (CSRF)	11.73
10	CWE-434	Unrestricted Upload of File with Dangerous Type	10.41
11	CWE-862	Missing Authorization	6.9
12	CWE-476	NULL Pointer Dereference	6.59

(1) MITRE 2023 CWE Top 25 Most Dangerous Software Weaknesses
cwe.mitre.org/top25/archive/2023/2023_top25_list.html#tableView

Figure 1

Herb Sutter Herb is a software technologist, working at the intersection of programming language design/UX, people, and high performance code. He is an author, chair of the ISO C++ committee, and a software architect at Microsoft.

As we specify and evolve default language safety rules, we must also include our stakeholders who care deeply about functional safety issues

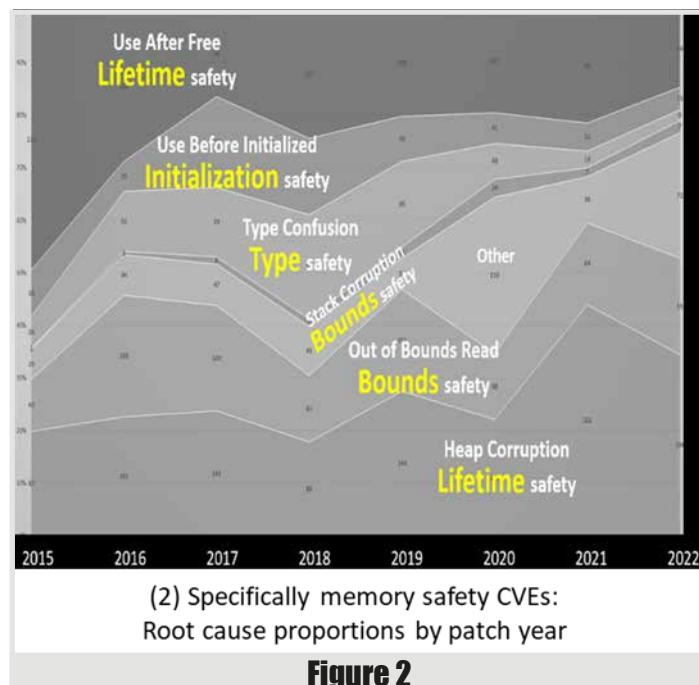


Figure 2

and Exposures) [Wikipedia] related to language memory unsafety. (However, that “70% of language memory unsafety CVEs” is misleading; for example, in figure 1, most of MITRE’s 2023 “most dangerous weaknesses” [MITRE-1] did not involve language safety and so are outside that denominator. More on this under ‘What the problem “isn’t”...’, section (3) on page 7.)

The C++ guidance literature already broadly agrees on safety rules in those categories. It’s true that there is some conflicting guidance literature, particularly in environments that ban exceptions or run-time type support and so use some alternative rules. But there is consensus on core safety rules, such as banning unsafe casts, uninitialized variables, and out-of-bounds accesses (see ‘Appendix’, starting on page 9).

C++ should provide a way to enforce them by default, and require explicit opt-out where needed. We can and do write ‘good’ code and secure applications in C++. But it’s easy even for experienced C++ developers to accidentally write ‘bad’ code and security vulnerabilities that C++ silently accepts, and that would be rejected as safety violations in other languages. We need the standard language to help more by enforcing the known best practices rather than relying on additional nonstandard tools to recommend them.

These are not the only four aspects of language safety we should address. They are just the immediate ones, a set of clear low-hanging fruit where there is both a clear need and clear way to improve (see ‘Appendix’, starting on page 9).

Note: And safety categories are of course interrelated. For example, full type safety (that an accessed object is a valid object of its type)

requires eliminating out-of-bounds accesses to unallocated objects. But, conversely, full bounds safety (that accessed memory is inside allocated bounds) similarly requires eliminating type-unsafe downcasts to larger derived-type objects that would appear to extend beyond the actual allocation.

Software safety is also important. Cyberattacks are urgent, so it’s natural that recent discussions have focused more on security and CVEs first. But as we specify and evolve default language safety rules, we must also include our stakeholders who care deeply about functional safety issues that are not reflected in the major CVE buckets but are just as harmful to life and property when left in code. Programming language safety helps both software security and software safety, and we should start somewhere, so let’s start (but not end) with the known pain points of security CVEs.

In those four buckets, a 10–50× improvement (90–98% reduction) is sufficient

If there were 90–98% fewer C++ type/bounds/initialization/lifetime vulnerabilities we wouldn’t be having this discussion. All languages have CVEs, C++ just has more (and C still more); so far in 2024, Rust has 6 CVEs [Rust-1], and C and C++ combined have 61 CVEs [C/C++]. So zero isn’t the goal; something like a 90% reduction is necessary, and a 98% reduction is sufficient, to achieve security parity with the levels of language safety provided by MSLs... and has the strong benefit that I believe it can be achieved with *perfect backward link compatibility* (i.e., without changing C++’s object model, and its lifetime model which does not depend on universal tracing garbage collection and is not limited to tree-based data structures) which is essential to our being able to adopt the improvements in existing C++ projects as easily as we can adopt other new editions of C++. After that, we can pursue additional improvements to other buckets, such as thread safety and overflow safety.

Aiming for 100%, or zero CVEs in those four buckets, would be a mistake:

- 100% is not necessary because none of the MSLs we’re being told to use instead are there either. More on this under ‘What the problem “isn’t”...’, section (2) on page 7.
- 100% is not sufficient because many cyberattacks exploit security weaknesses other than memory safety.

And getting that last 2% would be too costly, because it would require giving up on link compatibility and seamless interoperability (or ‘interop’) with today’s C++ code. For example, Rust’s object model and borrow checker deliver great guarantees, but require fundamental incompatibility with C++ and so make interop hard beyond the usual C interop level. One reason is that Rust’s safe language pointers are limited to expressing tree-shaped data structures that have no cycles; that unique ownership is essential to having great language-enforced aliasing guarantees, but it also requires programmers to use ‘something else’ for anything more complex than a tree (e.g., using `Rc`, or using integer indexes as ersatz

C++ should seriously try to deliver as many of the safety improvements as practical without requiring manual source code changes

pointers); it's not just about linked lists [Rust-2] but those are a simple well-known illustrative example.

If we can get a 98% improvement and still have fully compatible interop with existing C++, that would be a holy grail worth serious investment.

A 98% reduction

A 98% reduction across those four categories is achievable in new/updated C++ code, and partially in existing code

Since at least 2014, Bjarne Stroustrup has advocated addressing safety in C++ via a ‘subset of a superset’: That is, first ‘superset’ to add essential items not available in C++14, then ‘subset’ to exclude the unsafe constructs that now all have replacements.

As of C++20, I believe we have achieved the ‘superset’, notably by standardizing `span`, `string_view`, concepts, and bounds-aware ranges. We may still want a handful more features, such as a null-terminated `zstring_view`, but the major additions already exist.

Now we should ‘subset’: Enable C++ programmers to enforce best practices around type and memory safety, by default, in new code and code they can update to confirm to the subset. Enabling safety rules by default would not limit the language’s power but would require explicit opt-outs for non-standard practices, thereby reducing inadvertent risks. And it could be evolved over time, which is important because C++ is a living language and adversaries will keep changing their attacks.

ISO C++ evolution is already pursuing Safety Profiles for C++ [Stroustrup23]. The suggestions in the Appendix are refinements to that, to demonstrate specific enforcements and to try to maximize their adoptability and useful impact. For example, everyone agrees that many safety bugs will require code changes to fix. However, how many safety bugs could be fixed without manual source code changes, so that just recompiling existing code with safety profiles enabled delivers some safety benefits? For example, we could by default inject a call-site bounds check `0 <= b < a.size()` on every subscript expression `a[b]` when `a.size()` exists and `a` is a contiguous container, without requiring any source code changes and without upgrading to a new internally bounds-checked container library; that checking would Just Work out of the box with every contiguous C++ standard container, `span`, `string_view`, and third-party custom container with no library updates needed (including therefore also no concern about ABI breakage).

Rules like those summarized in the Appendix would have prevented (at compile time, test time or run time) most of the past CVEs I’ve reviewed in the type, bounds, and initialization categories, and would have prevented many of the lifetime CVEs. I estimate a roughly 98% reduction in those categories is achievable in a well-defined and standardized way for C++ to enable safety rules by default while still retaining perfect backward link compatibility. See the Appendix on page 9 for a more detailed description.

We can and should emphasize adoptability and benefit also for C++ code that cannot easily be changed. Any code change to conform to

safety rules carries a cost; worse, not all code can be easily updated to conform to safety rules (e.g., it’s old and not understood, it belongs to a third party that won’t allow updates, it belongs to a shared project that won’t take upstream changes and can’t easily be forked). That’s why above (and in the Appendix) I stress that C++ should seriously try to deliver as many of the safety improvements as practical without requiring manual source code changes, notably by automatically making existing code do the right thing when that is clear (e.g., the bounds checks mentioned above, or emitting `static_cast` pointer downcasts as effectively `dynamic_cast` without requiring the code to be changed), and by offering automated fixes that the programmer can choose to apply (e.g., to change the source for `static_cast` pointer downcasts to actually say `dynamic_cast`). Even though in many cases a programmer will need to thoughtfully update code to replace inherently unsafe constructs that can’t be automatically fixed, I believe for some percentage of cases we can deliver safety improvements by just recompiling existing code in the safety-rules-by-default mode, and we should try because it’s essential to maximizing safety profiles’ adoptability and impact.

What the problem “isn’t”: Some common misconceptions

(1) The problem “isn’t” defining what we mean by “C++’s most urgent language safety problem.” We know the four kinds of safety that most urgently need to be improved: type, bounds, initialization, and lifetime safety.

We know these four are the low-hanging fruit (see ‘The immediate problem “is”...’ on page 4). It’s true that these are just four of perhaps two dozen kinds of ‘safety’ categories, including ones like safe integer arithmetic. But:

- Most of the others are either much smaller sources of problems, or are primarily important because they contribute to those four main categories. For example, the integer overflows we care most about are indexes and sizes, which fall under bounds safety.
- Most MSLs don’t address making these safe by default either, typically due to the checking cost. But all languages (including C++) usually have libraries and tools to address them. For example, Microsoft ships a SafeInt library for C++ to handle integer overflows [Microsoft-1], which is opt-in. C# has a checked arithmetic language feature [Microsoft-2] to handle integer overflows, which is opt-in. Python’s built-in integers are overflow-safe by default because they automatically expand; however, the popular NumPy fixed-size integer types do not check for overflow by default and require using checked functions, which is opt-in.

Thread safety is obviously important too, and I’m not ignoring it. I’m just pointing out that it is not one of the top target buckets: Most of the MSLs that NIST/NSA/CISA/etc. recommend over C++ (except uniquely Rust, and to a lesser extent Python) address thread safety impact on *user data* corruption about as well as C++. The main improvement MSLs give is that a program data race will not corrupt the language’s own

Many of 2023's largest data breaches and other cyberattacks and cybercrime had nothing to do with programming languages at all

virtual machine (whereas, in C++, a data race is currently all-bets-are-off undefined behavior). Some languages do give some additional protection, such as that Python guarantees two racing threads cannot see a torn write of an integer and reduces other possible interleavings because of the global interpreter lock (GIL).

(2) The problem “isn’t” that C++ code is not formally provably safe

Yes, C++ code makes it too easy to write silently-unsafe code by default (see ‘The immediate problem “is”...’ on page 4).

But I’ve seen some people claim we need to require languages to be formally provably safe, and that would be a bridge too far. Much to the chagrin of CS theorists, mainstream commercial programming languages aren’t formally provably safe. Consider some examples:

- None of the widely-used languages we view as MSLs (except uniquely Rust) claim to be thread-safe and race-free by construction, as covered in the previous section. Yet we still call C#, Go, Java, Python, and similar languages “safe”. Therefore, formally guaranteeing thread safety properties can’t be a requirement to be considered a sufficiently safe language.
- That’s because a language’s choice of safety guarantees is a tradeoff: For example, in Rust, safe code uses tree-based dynamic data structures only. This feature lets Rust deliver stronger thread safety guarantees than other safe languages, because it can more easily reason about and control aliasing. However, this same feature also requires Rust programs to use unsafe code more often to represent common data structures that do not require unsafe code to represent in other MSLs such as C# or Java, and so 30% to 50% of Rust crates use unsafe code [Wang22], compared for example to 25% of Java libraries [Mastrangelo15].
- C#, Java, and other MSLs still have use-before-initialized and use-after-destroyed type safety problems too: They guarantee not accessing *memory* outside its allocated lifetime, but *object* lifetime is a subset of memory lifetime (objects are constructed after, and destroyed/disposed before, the raw memory is allocated and deallocated; before construction and after dispose, the memory is allocated but contains “raw bits” that likely don’t represent a valid object of its type). **If you doubt, please run (don’t walk) and ask ChatGPT** about Java and C# problems with: access-unconstructed-object bugs (e.g., in those languages, any virtual call in a constructor is “deep” and executes in a derived object before the derived object’s state is initialized); use-after-dispose bugs; “resurrection” bugs; and why those languages tell people never to use their finalizers. Yet these are great languages and we rightly consider them safe languages. Therefore, formally guaranteeing no-use-before-initialized and no-use-after-dispose can’t be a requirement to be considered a sufficiently safe language.
- Rust, Go, and other languages support sanitizers too [Rust-3], including ThreadSanitizer and undefined behavior sanitizers

[Rust-4], and related tools like fuzzers. Sanitizers are known to be still needed as a complement to language safety, and not only for when programmers use ‘unsafe’ code; furthermore, they go beyond finding memory safety issues. The uses of Rust at scale that I know of also enforce use of sanitizers. So using sanitizers can’t be an indicator that a language is unsafe — we should use the supported sanitizers for code written in any language.

Note: “Use your sanitizers” does not mean to use all of them all the time. Some sanitizers conflict with each other, so you can only use those one at a time. Some sanitizers are expensive, so they should only be run periodically. Some sanitizers should not be run in production, including because their presence can create new security vulnerabilities.

(3) The problem “isn’t” that moving the world’s C and C++ code to memory-safe languages (MSLs) would eliminate 70% of security vulnerabilities

MSLs are wonderful! They just aren’t a silver bullet.

An oft-quoted number [Gaynor20] is that “70%” of *programming language-caused* CVEs (reported security vulnerabilities) in C and C++ code are due to language safety problems. That number is true and repeatable, but has been badly misinterpreted in the press: No security expert I know believes that if we could wave a magic wand and instantly transform all the world’s code to MSLs, that we’d have 70% fewer CVEs, data breaches, and ransomware attacks. (For example, see this February 2024 example analysis paper [Hanley24].)

Consider some reasons.

- That 70% is of the *subset* of security CVEs that can be addressed by programming language safety. See figure 1 again: Most of 2023’s top 10 “most dangerous software weaknesses” were not related to memory safety. Many of 2023’s largest data breaches and other cyberattacks and cybercrime had nothing to do with programming languages at all. In 2023, attackers reduced their use of malware because software is getting hardened and endpoint protection is effective (CRN) [Alspach23], and attackers go after the slowest animal in the herd. Most of the issues listed in NISTIR-8397 [Black21] affect all languages equally, as they go beyond memory safety (e.g., Log4j [CISA-1]) or even programming languages (e.g., automated testing, hardcoded secrets, enabling OS protections, string/SQL injections, software bills of materials). For more detail, see the Microsoft response to NISTIR-8397 [Microsoft-3], for which I was the editor. (More on this in the ‘Call to Action’, below.)
- MSLs get CVEs too, though definitely fewer (again, e.g., Log4j). For example, see MITRE list of Rust CVEs, including six so far in 2024 [MITRE-2]. And all programs use unsafe code; for example, see the ‘Conclusions’ section of Firouzi *et al.*’s study of uses of C#’s `unsafe` on StackOverflow [Firouzi20] and prevalence of vulnerabilities, and that all programs eventually call trusted native libraries or operating system code.

CVEs are known to be an imprecise metric. We use it because it's the metric we have, at least for security vulnerabilities, but we should use it with care

- Saying the quiet part out loud: CVEs are known to be an imprecise metric. We use it because it's the metric we have, at least for security vulnerabilities, but we should use it with care. This may surprise you, as it did me, because we hear a lot about CVEs. But whenever I've suggested improvements for C++ and measuring "success" via a reduction in CVEs (including in this essay), security experts insist to me that CVEs aren't a great metric to use... including the same experts who had previously quoted the 70% CVE number to me. — Reasons why CVEs aren't a great metric include that CVEs are self-reported and often self-selected, and not all are equally exploitable; but there can be pressure to report a bug as a vulnerability even if there's no reasonable exploit because of the benefits of getting one's name on a CVE. In August 2023, the Python Software Foundation became a CVE Numbering Authority (CNA) for Python and pip distributions [MITRE-3], and now has more control over Python and pip CVEs. The C++ community has not done so.
- CVEs target only software security vulnerabilities (cyberattacks and intrusions), and we also need to consider software safety (life-critical systems and unintended harm to humans).

(4) The problem “isn’t” that C++ programmers aren’t trying hard enough/using the existing tools well enough. The challenge is making it easier to enable them.

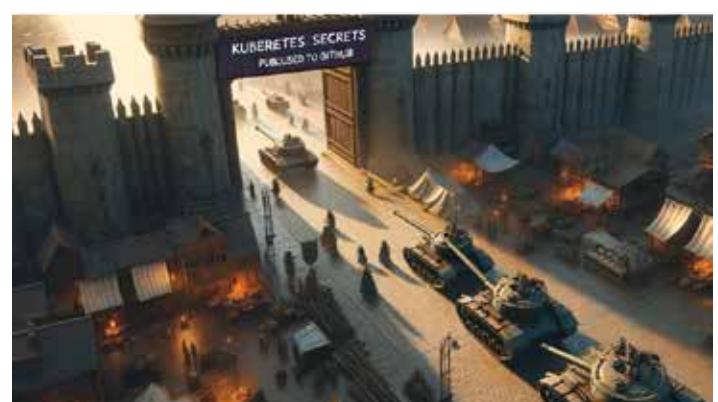
Today, the mitigations and tools we do have for C++ code are an uneven mix, and all are off-by-default:

- **Kind.** They are a mix of static tools, dynamic tools, compiler switches, libraries, and language features.
- **Acquisition.** They are acquired in a mix of ways: in-the-box in the C++ compiler, optional downloads, third-party products, and some you need to google around to discover.
- **Accuracy.** Existing rulesets mix rules with low and high false positives. The latter are effectively unadoptable by programmers, and their presence makes it difficult to 'just adopt this whole set of rules'.
- **Determinism.** Some rules, such as ones that rely on interprocedural analysis of full call trees, are inherently nondeterministic (because an implementation gives up when fully evaluating a case exceeds the space and time available; a.k.a. 'best effort' analysis). This means that two implementations of the identical rule can give different answers for identical code (and therefore nondeterministic rules are also not portable, see below).
- **Efficiency.** Existing rulesets mix rules with low and high (and sometimes impossible) cost to diagnose. The rules that are not efficient enough to implement in the compiler will always be relegated to optional standalone tools.
- **Portability.** Not all rules are supported by all vendors. 'Conforms to ISO/IEC 14882 (Standard C++)' is the only thing every C++ tool vendor supports portably.

To address all these points, I think we need the C++ standard to specify a mode of well-agreed and low-or-zero-false-positive deterministic rules that are sufficiently low-cost to implement in-the-box at build time.

Call(s) to action

As an industry generally, we must make a major improvement in programming language memory safety – and we will.



if we focus on programming language safety alone, we may find ourselves fighting yesterday's war and missing larger past and future security dangers that affect software written in any language

In C++ specifically, we should first target the four key safety categories that are our perennial empirical attack points (type, bounds, initialization, and lifetime safety), and drive vulnerabilities in these four areas down to the noise for new/updated C++ code – and we can.

But we must also recognize that programming language safety is not a silver bullet to achieve cybersecurity and software safety. It's one battle (not even the biggest) in a long war: Whenever we harden one part of our systems and make that more expensive to attack, attackers always switch to the next slowest animal in the herd. Many of 2023's worst data breaches did not involve malware, but were caused by inadequately stored credentials (e.g., Kubernetes Secrets on public GitHub repos [Kadkoda23]), misconfigured servers (e.g., DarkBeam [Okunytè23a], Kid Security [Okunytè23b]), lack of testing, supply chain vulnerabilities, social engineering, and other problems that are independent of programming languages. Apple's white paper about 2023's rise in cybercrime emphasizes improving the handling, not of program code, but of the data [Madnick23]:

it's imperative that organizations consider limiting the amount of personal data they store in readable format while making a greater effort to protect the sensitive consumer data that they do store [including by using] end-to-end [E2E] encryption.

No matter what programming language we use, security hygiene is essential:

- **Do** use your language's static analyzers and sanitizers. Never pretend using static analyzers and sanitizers is unnecessary “because I'm using a safe language.” If you're using C++, Go, or Rust, then use those languages' supported analyzers and sanitizers. If you're a manager, don't allow your product to be shipped without using these tools. (Again: This doesn't mean running all sanitizers all the time; some sanitizers conflict and so can't be used at the same time, some are expensive and so should be used periodically, and some should be run only in testing and never in production including because their presence can create new security vulnerabilities.)
- **Do** keep all your tools updated. Regular patching is not just for iOS and Windows, but also for your compilers, libraries, and IDEs.
- **Do** secure your software supply chain. **Do** use package management for library dependencies. **Do** track a software bill of materials for your projects.
- **Don't** store secrets in code. (Or, for goodness' sake, on GitHub!)
- **Do** configure your servers correctly, especially public Internet-facing ones. (Turn authentication on! Change the default password!)
- **Do** keep non-public data encrypted, both when at rest (on disk) and when in motion (ideally E2E... and oppose proposed legislation that tries to neuter E2E encryption with ‘backdoors only good guys will use’ because there's no such thing).

- **Do** keep investing long-term in keeping your threat modeling current, so that you can stay adaptive as your adversaries keep trying different attack methods.

We need to improve software security and software safety across the industry, especially by improving programming language safety in C and C++, and in C++ a 98% improvement in the four most common problem areas is achievable in the medium term. But if we focus on programming language safety alone, we may find ourselves fighting yesterday's war and missing larger past and future security dangers that affect software written in any language.

Sadly, there are too many bad actors. For the foreseeable future, our software and data will continue to be under attack, written in any language and stored anywhere. But we can defend our programs and systems, and we will.

Be well, and may we all keep working to have a safer and more secure 2024.

Appendix: Illustrating why a 98% reduction is feasible

This Appendix exists to support why I think a 98% reduction in type/bounds/initialization/lifetime CVEs in C++ code is believable. This is not a formal proposal, but an overview of concrete ways to achieve such an improvement in new and updatable code, and ways to even get some fraction of that improvement in existing code we cannot update but can recompile. These notes are aligned with the proposals currently being pursued in the ISO C++ safety subgroup, and if they pan out as I expect in ongoing discussions and experiments, then I intend to write further details about them in a future paper.

There are runtime and code size overheads to some of the suggestions in all four buckets, notably checking bounds and casts. But there is no reason to think those overheads need to be inherently worse in C++ than other languages, and we can make them on by default and still provide a way to opt out to regain full performance where needed.

Note: For example, bounds checking can cause a major impact on some hot loops, when using a compiler whose optimizer does not hoist bounds checks; not only can the loops incur redundant checking, but they also may not get other optimizations such as not being vectorized. This is why making bounds-checking on by default is good, but all performance-oriented languages also need to provide a way to say “trust me” and explicitly opt out of bounds checking tactically where needed.

This appendix refers to the ‘profiles’ in the *C++ Core Guidelines* safety profiles [CPP], a set of about two dozen enforceable rules for type and memory safety of which I am a co-author. I refer to them only as examples, to show ‘what’ already-known rules exist that we can enforce, to support that my claimed improvement is possible. They are broadly consistent with rules in other sources, such as: *The C++ Programming Language*'s advice on type safety [Stroustrup13]; *C++ Coding Standards*' section on

In cases where bounds checking incurs a performance impact, code can still explicitly opt out of the bounds check in just those paths

type safety [Sutter04]; the *Joint Strike Fighter Coding Standards* [LM05]; *High Integrity C++* [Perforce13]; the *C++ Core Guidelines* section on safety profiles (a small enforceable set of safety rules) [CPP-1]; and the recently-released MISRA C++:2023 [MISRA].

The best way for ‘how’ to let the programmer control enabling those rules (e.g., via source code annotations, compiler switches, and/or something else) is an orthogonal UX issue that is now being actively discussed in the C++ standards committee and community.

Type safety

Enforce the Pro.Type safety profile by default [CPP-2]. That includes either banning or checking all unsafe casts and conversions (e.g., `static_cast` pointer downcasts, `reinterpret_cast`), including implicit unsafe type punning via C `union` and `vararg`.

However, these rules haven’t yet been systematically enforced in the industry. For example, in recent years I’ve painfully observed a significant set of type safety-caused security vulnerabilities whose root cause was that code used `static_cast` instead of `dynamic_cast` for pointer downcasts, and ‘C++’ gets blamed even when the actual problem was failure to follow the well-publicized guidance to use the language’s existing safe recommended feature. It’s time for a standardized C++ mode that enforces these rules by default.

Note: On some platforms and for some applications, `dynamic_cast` has problematic space and time overheads that hinder its use. Many implementations bundle `dynamic_cast` indivisibly with all C++ runtime typing (RTTI) features (e.g., `typeid`), and so require storing full potentially-heavyweight RTTI data even though `dynamic_cast` needs only a small subset. Some implementations also use needlessly inefficient algorithms for `dynamic_cast` itself. So the standard must encourage (and, if possible, enforce for conformance, such as by setting algorithmic complexity requirements) that `dynamic_cast` implementations be more efficient and decoupled from other RTTI overheads, so that programmers do not have a legitimate performance reason not to use the safe feature. That decoupling could require an ABI break; if that is unacceptable, the standard must provide an alternative lightweight facility such as a `fast_dynamic_cast` that is separate from (other) RTTI and performs the dynamic cast with minimum space and time cost.

Bounds safety

Enforce the Pro.Bounds safety profile [CPP-3] by default, and guarantee bounds checking. We should additionally guarantee that:

- Pointer arithmetic is banned (use `std::span` instead); this enforces that a pointer refers to a single object. Array-to-pointer decay, if allowed, will point to only the first object in the array.
- Only bounds-checked iterator arithmetic is allowed (also, prefer ranges instead).
- All subscript operations are bounds-checked at the call site, by having the compiler inject an automatic subscript bounds check

on every expression of the form `a[b]`, where `a` is a contiguous sequence with a `size/ssize` function and `b` is an integral index. When a violation happens, the action taken can be customized using a global bounds violation handler; some programs will want to terminate (the default), others will want to log-and-continue, throw an exception, integrate with a project-specific critical fault infrastructure.

Importantly, the latter explicitly avoids implementing bounds-checking intrusively for each individual container/range/view type. Implementing bounds-checking non-intrusively and automatically at the call site makes full bounds checking available for every existing standard and user-written container/range/view type out of the box: Every subscript into a `vector`, `span`, `deque`, or similar existing type in third-party and company-internal libraries would be usable in checked mode without any need for a library upgrade.

It’s important to add automatic call-site checking now before libraries continue adding more subscript bounds checking in each library, so that we avoid duplicating checks at the call site and in the callee. As a counterexample, C# took many years to get rid of duplicate caller-and-callee checking, but succeeded and .NET Core addresses this better now; we can avoid most of that duplicate-check-elimination optimization work by offering automatic call-site checking sooner.

Language constructs like the `range-for` loop are already safe by construction and need no checks.

In cases where bounds checking incurs a performance impact, code can still explicitly opt out of the bounds check in just those paths to retain full performance and still have full bounds checking in the rest of the application.

Initialization safety

Enforce initialization-before-use by default. That’s pretty easy to statically guarantee, except for some cases of the unused parts of lazily constructed array/`vector` storage. Two simple alternatives we could enforce are (either is sufficient):

- Initialize-at-declaration as required by Pro.Type [CPP-2] and ES.20 [CPP-4]; and possibly zero-initialize data by default as currently proposed in P2723 [Bastien23]. These two are good but with some drawbacks; both have some performance costs for cases that require ‘dummy’ writes that are never used but hard for optimizers to eliminate, and the latter has some correctness costs because it ‘fixing’ some uninitialized cases where zero is a valid value but masks others for which zero is not a valid initializer and so the behavior is still wrong, but because a zero has been jammed in it’s harder for sanitizers to detect.
- Guaranteed initialization-before-use, similar to what Ada and C# successfully do. This is still simple to use, but can be more efficient because it avoids the need for artificial ‘dummy’ writes, and can be more flexible because it allows alternative constructors to be used

for the same object on different paths. For details, see: example diagnostic; definite-first-use rules [Sutter22].

Lifetime safety

Enforce the Pro.Lifetime safety profile [CPP-5] by default, ban manual allocation by default, and guarantee null checking. The Lifetime profile is a static analysis that diagnoses many common sources of dangling and use-after-free, including for iterators and views (not just raw pointers and references), in a way that is efficient enough to run during compilation. It can be used as a basis to iterate on and further improve. And we should additionally guarantee that:

- All manual memory management is banned by default (`new`, `delete`, `malloc`, and `free`). Corollary: ‘Owning’ raw pointers are banned by default, since they require `delete` or `free`. Use RAI^I instead, such as by calling `make_unique` or `make_shared`.
- All dereferences are null-checked. The compiler injects an automatic check on every expression of the form `*p` or `p->` where `p` can be compared to `nullptr` to null-check all dereferences at the call site (similar to bounds checks above). When a violation happens, the action taken can be customized using a global null violation handler; some programs will want to terminate (the default), others will want to log-and-continue, throw an exception, integrate with a project-specific critical fault infrastructure.

Note: The compiler could choose to not emit this check (and not perform optimizations that benefit from the check) when targeting platforms that already trap null dereferences, such as platforms that mark low memory pages as unaddressable. Some C++ features, such as `delete`, have always done call-site null checking.

Reducing undefined behavior and semantic bugs

Tactically, reduce some undefined behavior (UB) and other semantic bugs (pitfalls), for cases where we can automatically diagnose or even fix well-known antipatterns. Not all UB is bad; any performance-oriented language needs some. But we know there is low-hanging fruit where the programmer’s intent is clear and any UB or pitfall is a definite bug, so we can do one of two things:

(A – Good) Make the pitfall a diagnosed error, with zero false positives – every violation is a real bug. Two examples mentioned above are to automatically check `a[b]` to be in bounds and `*p` and `p->` to be non-null.

(B – Ideal) Make the code actually do what the programmer intended, with zero false positives – i.e., fix it by just recompiling. An example, discussed at the most recent ISO C++ November 2023 meeting [Wakely23], is to default to an implicit `return *this`; when the programmer writes an assignment operator for their type `C` that returns a `C&` (note: the same type), but forgets to write a `return` statement. Today, that is undefined behavior. Yet it’s clear that the programmer meant `return *this`; nothing else can be valid. If we make `return *this` be the default, all the existing code that accidentally omits the `return` is not just ‘no longer UB’, but is guaranteed to do the right and intended thing.

An example of both (A) and (B) is to support chained comparisons [Revzin18], that makes the mathematically valid chains work correctly and rejects the mathematically invalid ones at compile time. Real-world code does write such chains by accident [SO-1] [SO-2] [SO-3] [SO-4] [SO-5] [SO-6] [SO-7] [SO-8] [SO-9] [SO-10].

- For (A): We can reject all mathematically invalid chains like `a != b > c` at compile time. This automatically diagnoses bugs in existing code that tries to do such nonsense chains, with perfect accuracy.
- For (B): We can fix all existing code that writes would-be-correct chains like `0 <= index < max`. Today those silently compile but are completely wrong, and we can make them mean the right thing. This automatically fixes those bugs, just by recompiling the existing code.

These examples are not exhaustive. We should review the list of UB in the standard for a more thorough list of cases we can automatically fix (ideally) or diagnose.

Summarizing: Better defaults for C++

C++ could enable turning safety rules on by default that would make code:

- fully type-safe,
- fully bounds-safe,
- fully initialization-safe,

and for lifetime safety, which is the hardest of the four, and where I would expect the remaining vulnerabilities in these categories would mostly lie:

- fully null-safe,
- fully free of owning raw pointers,
- with lifetime-safety static analysis that diagnoses most common pointer/iterator/view lifetime errors;

and, finally:

- with less undefined behavior including by automatically fixing existing bugs just by recompiling code with safety enabled by default.

All of this is efficiently implementable and has been implemented. Most of the Lifetime rules have been implemented in Visual Studio and CLion, and I’m prototyping a proof-of-concept mode of C++ that includes all of the other above language safeties on-by-default in my `cppfront` compiler [Sutter], as well as other safety improvements including an implementation of the current proposal for ISO C++ contracts. I haven’t yet used the prototype at scale. However, I can report that the first major change request I received from early users was to change the bounds checking and null checking from opt-in (off by default) to opt-out (on by default).

Note: Please don’t be distracted by that `cppfront` uses an experimental alternate syntax for C++. That’s because I’m additionally trying to see if we can reach a second orthogonal goal: to make the C++ language itself simpler, and eliminate the need to teach ~90% of the C++ guidance literature related to language complexity and quirks. This essay’s language safety improvements are orthogonal to that, however, and can be applied equally to today’s C++ syntax.

Solutions need to distinguish between (A) ‘solution for new-or-updatable code’ and (B) ‘solution for existing code’

(A) A ‘solution for new-or-updatable code’ means that to help existing code we have to change/rewrite our code. This includes not only ‘(re) write in C#/Rust/Go/Python/...’ but also ‘annotate your code with SAL’ [Microsoft-4] or ‘change your code to use `std::span`’.

One of the costs of (A) is that anytime we write/change code to fix bugs, we also introduce new bugs; change is never free. We need to recognize that changing our code to use `std::span` often means non-trivially rewriting parts of it which can also create other bugs. Even annotating our code means writing annotations that can have bugs (this is a common experience in the annotation languages I’ve seen used at scale, such as SAL). All these are significant adoption barriers.

Actually switching to another language means losing a mature ecosystem. C++ is the well-trod path: It’s taught, people know it, the tools exist, interop works, and current regulations have an industry around C++ (such as for functional safety). It takes another decade at least for another language to become the well-trod path, whereas a better C++, and its benefits to the industry broadly, can be here much sooner.

(B) A ‘solution for existing code’ emphasizes the adoptability benefits of not having to make manual code changes. It includes anything that makes existing code more secure with ‘just a recompile’ (i.e., no binary/ABI/link issues; e.g., ASAN, compiler switches to enable stack checks,

static analysis that produces only true positives, or a reliable automated code modernizer).

We will still need (B) no matter how successful new languages or new C++ types/annotations are. And (B) has the strong benefit that it is easier to adopt. Getting to a 98% reduction in CVEs will require both (A) and (B), but if we can deliver even a 30% reduction using just (B) that will be a major benefit for adoption and effective impact in large existing code bases that are hard to change.

Consider how the ideas earlier in this appendix map onto (A) and (B):

In C++, by default, enforce...	(A) Solution for new/updated code (can require code changes – no link/binary changes)	(B) Solution for existing code (requires recompile only – no manual code changes, no link/binary changes)
Type safety	Ban all inherently unsafe casts and conversions	Make unsafe casts and conversions with a safe alternative do the safe thing
Bounds safety	Ban pointer arithmetic Ban unchecked iterator arithmetic	Check in-bounds for all allowed iterator arithmetic Check in-bounds for all subscript operations
Initialization safety	Require all variables to be initialized (either at declaration, or before first use)	
Lifetime safety	Statically diagnose many common pointer/iterator lifetime error cases	Check not-null for all pointer dereferences
Less undefined behavior	Statically diagnose known UB/bug cases, to error on actual bugs in existing code with just a recompile and zero false positives: <ul style="list-style-type: none">■ Ban mathematically invalid comparison chains■ (add additional cases from UB Annex review)	Automatically fix known UB/bug cases, to make current bugs in existing code be actually correct with just a recompile and zero false positives: <ul style="list-style-type: none">■ Define mathematically valid comparison chains■ Default <code>return *this;</code> for C assignment operators that return <code>C&</code>■ (add additional cases from UB Annex review)

By prioritizing adoptability, we can get at least some of the safety benefits just by recompiling existing code, and make the total improvement easier to deploy even when code updates are required. I think that makes it a valuable strategy to pursue.

Finally, please see again the main article's conclusion: 'Call(s) to action' on page 8. ■

References

- [Alspach23] Kyle Alspach '10 Major Cyberattacks And Data Breaches In 2023, published 13 December 2023 by CRN at <https://www.crn.com/news/security/10-major-cyberattacks-and-data-breaches-in-2023>
- [Bastien23] JF Bastien, 'P2723R1: Zero-initialize objects of automatic storage duration', published 15 January 2023, available at <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2723r1.html>
- [Black21] Paul E. Black, Barbara Guttman and Vadim Okum, 'Guidelines on Minimum Standards for Developer Verification of Software' (NISTIR 8397) available at <https://nvlpubs.nist.gov/nistpubs/ir/2021/NIST.IR.8397.pdf>
- [C/C++] C and C++ CVEs: <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=c++>
- [CISA-1] 'Apache Log4j Vulnerability Guidance', published April 2022 by America's Cyber Defense Agency, April 2022, available at

<https://www.cisa.gov/news-events/news/apache-log4j-vulnerability-guidance>

- [CISA-2] 'The Case for Memory Safe Roadmaps', published December 2023 jointly by US, Australian, Canadian, New Zealand and UK cyber security centres/agencies, available at <https://media.defense.gov/2023/Dec/06/2003352724/-1/-1/0/THE-CASE-FOR-MEMORY-SAFE-ROADMAPS-TLP-CLEAR.PDF>
- [CPP-1] Pro: Profiles in *C++ Core Guidelines*, available at <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#pro-profiles>
- [CPP-2] Pro.safety: Type-safety profile in *C++ Core Guidelines*, available at <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#SS-type>
- [CPP-3] Pro.bounds: Bounds safetyprofile in *C++ Core Guidelines*, available at <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#probounds-bounds-safety-profile>
- [CPP-4] ES.20: Always initialize an object in *C++ Core Guidelines*, available at <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-always>
- [CPP-5] Pro.safety: Type-safety profile in *C++ Core Guidelines*, available at <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#SS-lifetime>
- [Firouzi20] Ehsan Firouzi, Ashkan Sami, Foutse Khomh and Gias Uddin 'On the use of C# Unsafe Code Context: An Empirical Study of Stack Overflow' from the *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, available at https://www.researchgate.net/publication/344892072_On_the_use_of_C_Unsafe_Code_Context_An_Empirical_Study_of_Stack_Overflow
- [Gaynor20] Alex Gaynor 'What science can tell us about C and C++'s security', published 27 May 2020, available at <https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/>
- [Hanley24] Zach Hanley 'Rust Won't Save Us: An Analysis of 2023's Known Exploited Vulnerabilities', posted 6 February 2024, available at <https://www.horizon3.ai/attack-research/attack-blogs/analysis-of-2023s-known-exploited-vulnerabilities/>
- [ISO] ISO/IEC 23643:2020 – 'Software and systems engineering: Capabilities of software safety and security verification tools' <https://www.iso.org/standard/76517.html>
- [Kadkoda23] Yakir Kadkoda and Assaf Morag 'The Ticking Supply Chain Attach Bomb of Exposed Kubernetes Secrets', published 21 Nov 2023 on the Aqua Blog, available at <https://www.aquasec.com/blog/the-ticking-supply-chain-attack-bomb-of-exposed-kubernetes-secrets/>
- [LM05] Lockheed Martin: 'Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program', published December 2025 and available at <https://www.stroustrup.com/JSF-AV-rules.pdf>
- [Madnick23] Stuart Madnick, 'The Continued Threat to Personal Data: Key Factors Behind the 2023 Increase', published by Apple in December 2023 and available at <https://www.apple.com/newsroom/pdfs/The-Continued-Threat-to-Personal-Data-Key-Factors-Behind-the-2023-Increase.pdf>
- [Mastrangelo15] Luis Mastrangelo, Luca Ponzanelli, Andrea Moccia, Michele Lanza, Matthias Hauswirth and Nathaniel Nystrom 'Use at your own risk: the Java unsafe API in the wild' from the *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages and Applications*, available at <https://dl.acm.org/doi/abs/10.1145/2814270.2814313>
- [Microsoft-1] SafeInt Library: <https://learn.microsoft.com/en-us/cpp/safeint/safeint-library?view=msvc-170>

- [Microsoft-2] Checked and unchecked statements:
<https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/checked-and-unchecked>
- [Microsoft-3] Build reliable and secure C++ programs:
<https://learn.microsoft.com/en-us/cpp/code-quality/build-reliable-secure-programs?view=msvc-170>
- [Microsoft-4] Understanding SAL: <https://learn.microsoft.com/en-us/cpp/code-quality/understanding-sal?view=msvc-170>
- [MISRA] MISRA 2023: <https://misra.org.uk/misra-cpp2023-released-including-hardcopy/>
- [MITRE-1] ‘2023 CWE Top 25’ on the *Common Weakness Enumeration* website operated by Mitre, available at: https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html#tableView
- [MITRE-2] Rust CVEs, from the CVE website managed by Mitre, available at : <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=rust>
- [MITRE-3] CVE: ‘Python Software Foundation Added as CVE Numbering Authority (CNA)’ published 29 August 2023 at <https://www.cve.org/Media/News/item/news/2023/08/29/Python-Software-Foundation-Added-as-CNA>
- [Okunyté23a] Paulina Okunyté, ‘DarkBeam leaks billions of email and password combinations’, published by *Cybernews*, last updated 15 November 2023, available at <https://cybernews.com/security/darkbeam-data-leak/>
- [Okunyté23b] Paulina Okunyté, ‘KidSecurity’s user data compromised after app failed to set password’, published by *Cybernews*, last updated 30 November 2023, available at <https://cybernews.com/security/kidsecurity-parental-control-data-leak/>
- [Perforce13] Perforce, ‘High Integrity C++ Coding Standard’ version 4.0, released 3 October 2013, available at <https://www.perforce.com/resources/qac/high-integrity-cpp-coding-standard>
- [Revzin18] Barry Revzin and Herb Sutter, ‘P0893R1: Chaining comparisons’, published 28 April 2018, available at <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0893r1.html>
- [Rust-1] Rust CVEs: <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=rust>
- [Rust-2] ‘Learn Rust with Entirely Too Many Linked Lists’, available at <https://rust-unofficial.github.io/too-many-lists/>
- [Rust-3] ‘Sanitizers Support’ in the *Rust Compiler Development Guide*, available at <https://rustc-dev-guide.rust-lang.org/sanitizers.html>
- [Rust-4] Undefined behavior sanitizers: <https://github.com/rust-lang/miri>
- [SO-1] ‘Is (4 > y > 1) a valid statement in C++? How do you evaluate it if so?’, available on *StackOverflow* at <https://stackoverflow.com/questions/8889522/is-4-y-1-a-valid-statement-in-c-how-do-you-evaluate-it-if-so>
- [SO-2] ‘Chaining Bool values give opposite result to expected’, available on *StackOverflow* at <https://stackoverflow.com/questions/5939077/chaining-bool-values-give-opposite-result-to-expected>
- [SO-3] ‘Checking if a value is within a range in if statement’, available on *StackOverflow* at <https://stackoverflow.com/questions/14433884/checking-if-a-value-is-within-a-range-in-if-statement>
- [SO-4] ‘Test if all elements are equal with C++17 fold-expression’, available on *StackOverflow* at <https://stackoverflow.com/questions/46806239/test-if-all-elements-are-equal-with-c17-fold-expression>
- [SO-5] ‘Incorrect logic in C++’, available on *StackOverflow* at <https://stackoverflow.com/questions/25965157/incorrect-logic-in-c>
- [SO-6] ‘Is (val1 > val2 > val3) a valid comparison in C?’, available on *StackOverflow* at <https://stackoverflow.com/questions/38643022/is-val1-val2-val3-a-valid-comparison-in-c>
- [SO-7] ‘Why is if not working in my Magic Square program’, available on *StackOverflow* at <https://stackoverflow.com/questions/45385837/why-is-if-not-working-in-my-magic-square-program>
- [SO-8] ‘Math-like chaining of the comparison operator - as in, “if ($5 < j \leq 1$)”’, available on *StackOverflow* at <https://stackoverflow.com/questions/20989496/math-like-chaining-of-the-comparison-operator-as-in-if-5j-1>
- [SO-9] ‘Only Returning the first if statement? (C++)’, available on *StackOverflow* at <https://stackoverflow.com/questions/35564553/only-returning-the-first-if-statement-c>
- [SO-10] ‘Warning comparison integer and pointer’, available on *StackOverflow* at <https://stackoverflow.com/questions/42335710/warning-comparison-integer-and-pointer>
- [Stroustrup13] Bjarne Stroustrup (2013) *The C++ Programming Language*, 4th Edition published by Addison-Wesley Professional in May 2023. ISBN-13: 978-0275967307
- [Stroustrup23] Bjarne Stroustrup and Gabriel Dos Reis, ‘Safety Profiles: Type-and-resource Safe Programming in ISO Standard C++’. The slides presented by Bjarne at the February 2023 C++ Standard Committee meeting, available at: <https://open-std.org/JTC1/SC22/WG21/docs/papers/2023/p2816r0.pdf>
- [Sutter] ccpfront compiler, available at <https://github.com/hsutter/cppfront/>
- [Sutter04] Herb Sutter and Andrei Alexandrescu (2004) *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*, published by Addison-Wesley Professional in October 2024. ISBN-13: 978-0321113580
- [Sutter22] Herb Sutter ‘Can C++ be 10× simpler & safer …?’, a presentation delivered at CppCon 2022, available at <https://www.youtube.com/watch?v=ELeZAKCN4tY&t=4305s>
- [Wakely23] Jonathan Wakely and Thomas Köppe, ‘P2973R0: Erroneous behaviour for missing return from assignment’ published 15 September 2023, available at <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2973r0.html>
- [Wang22] Jun Wang ‘Unsafe Rust in the Wild’, published on The New Stack on 29 September 2022, available at: <https://thenewstack.io/unsafe-rust-in-the-wild/>
- [Wikipedia] ‘Common Vulnerabilities and Exposures’, available at https://en.wikipedia.org/wiki/Common_Vulnerabilities_and_Exposures

This article was first published on Herb Sutter’s blog (*Sutter’s Mill*) on 11th March 2023: <https://herbsutter.com/2024/03/11/safety-in-context/>

Advertise in CVu & Overload

80% of readers make purchasing decisions or recommend products for their organisations.

Reasonable rates. Flexible options. Discounts available to corporate members.

Contact ads@accu.org for information.

To See a World in a Grain of Sand

Code often rots over time as various people add new features. Jez Higgins shows how to refactor code that has grown organically, making it clearer and more concise.

In a recent blog post [Higgins24] about my sadness and disappointment about the candidates we were getting for interview, I talked about the refactoring exercise we give people, and the conversations we have afterwards.

I'm not able to show any of that code, but I am going to talk about some code here of the type we often see. According to the version history, it's passed through a number of hands, and I want to be clear I know none of the people involved nor have I spoken to them. They are, though, exactly the type of person presenting themselves for interview, and so for my purposes here they are exemplars.

Here's some Python code. It's from a larger document processing pipeline. Documents come shoved into the system, get squished around a bit, have metadata added, some formatting fixups, then squirt out the other end as nice looking pdfs. Standard stuff.

This is not about them, though. I hold them blameless, and wish them only happiness. This is about the places that they worked, about the wider trade, about a culture that says this is fine.

To see a world in a grain of sand

Documents can have references to other documents, both within the existing corpus, and to a variety of external sources. These references have standard forms, and when we find something that looks like a document reference, we do a bit of work to make sure it's absolutely clean and proper.

Jez Higgins lives on the Pembrokeshire coast, largely to make return-to-office mandates impractical. Truth is, he hasn't worked in an office for nearly 25 years, and has no intention of starting now. He's been programming for a living that whole time and thinks he might be starting getting to get the hang of it. He can be contacted at jez@jezuk.co.uk or @jezhiggins@mastodon.me.uk

```
def canonicalise_reference(reference_type, reference_match, canonical_form):
    if (
        (reference_type == "RefYearAbbrNum")
        | (reference_type == "RefYearAbbrNumTeam")
        | (reference_type == "YearAbbrNum")
    ):
        components = re.findall(r"\d+", reference_match)
        year = components[0]
        d1 = components[1]
        d2 = ""
        corrected_reference = canonical_form.replace("dddd", year).replace("d+", d1)
    elif (
        (reference_type == "RefYearAbbrNumNumTeam")
        | (reference_type
           == "RefYearAbbrNumStrokeNum")
        | (reference_type == "RefYearNumAbbrNum")
    ):
        components = re.findall(r"\d+", reference_match)
        year = components[0]
        d1 = components[1]
        d2 = components[2]
        corrected_reference = (
            canonical_form.replace("dddd", year).replace("d1", d1).replace("d2", d2)
        )
    elif (
        (reference_type == "AbbrNumAbbrNum")
        | (reference_type == "NumAbbrNum")
        | (reference_type == "EuroRefC")
        | (reference_type == "EuroRefT")
    ):
        components = re.findall(r"\d+", reference_match)
        year = ""
        d1 = components[0]
        d2 = components[1]
        corrected_reference = canonical_form.replace("d1", d1).replace("d2", d2)
    return corrected_reference, year, d1, d2
```

Listing 1

That's where the function in Listing 1, `normalise_reference`, comes in. I have obfuscated identifiers in the code sample, but its structure and behaviour are as I found it.

I'd been kind-of browsing around a git repository, looking at folder structure, getting the general picture. A chunk of the system is a Django webapp and thus has that shape, so I went digging for a bit of the meat underneath. This was almost the first thing I saw and, well, I kind of flinched. Poking around some more confirmed it's not an anomaly. It is representative of the system.

You've probably had some kind of reaction of your own. This is what immediately leapt out at me:

- The length
- The width!¹

¹ As this is a printed publication, in most listings the very wide lines are wrapped. Listing 1 is presented full-width, as is Listing 6.

- The visual repetition, both in the `if` conditions and in the bodies of the conditionals
- The string literals
- The string literal with the spelling mistake
- The extraneous brackets in the second conditional body – written by someone else?
- The extra line before the return – functionally, of course, it makes no difference, but still, urgh

Straightaway I'm thinking that more than one person has worked on this over time. That's normal, of course, but I shouldn't be able to tell. If I can, it's a contra-indicator.

Looking a little longer, there's a lot of repetition – in shape, and in detail. Looking a little longer still, and I think function parameters are in the wrong order. `reference_type` and `canonical_form` are correlated and originate within the system. They should go together. It's `reference_match` which comes from the input document, it's the only true *variable* and so, for me anyway, should be the first parameter. I suspect this function only had two parameters initially, and the third was added without a great deal of thought to the aesthetics of the change.

That's a lot to not like in not a lot of code.

But at least there are tests

And hurrah for that! There *are* tests for this function, tangled up in a source file with some unrelated tests that pull in a mock database connection and some other business, but they do exist.

There are two test functions, one checking well-formed references, the other malformed references, but, in fact, each function checks multiple cases.

It's a start, but the test code is much the same as the code it exercises – long and repetitious – which isn't, perhaps, that surprising. A quick visual check shows they're deficient in other, more serious ways. There are ten reference types named in `canonicalise_reference`. The tests check seven of them and, in fact, there is a whole branch of the `if/else` ladder that isn't exercised. That's the branch I already suspect of being a later addition.

Curiously too, while `canonicalise_reference` returns a 4-tuple, the tests only check the corrected reference and the year, ignoring the other two values. That sent me off looking for the `canonicalise_reference` call sites, where all four elements of the tuple are used. Again, I'd suggest the 4-tuple came in after the tests were first written and were not updated to match. After all, they still passed.

I am sure these tests were written post-hoc. They did not inform the design and development of the code they support.

Miasma

If the phrase coming to mind is code smells, then I guess you're right. This code is a stinky bouquet of bad odours, except they aren't clues to some deeper problem with the code. We don't need clues – it's right out there front and centre. No, these smells emanate from with the organisation, from a failure to develop the programmers whose hands this code has passed through. The code works, let's be clear, but there's a clumsiness to it and a lack of care in its evolution. That's a cultural and organisational failing.

I keep saying this is about organisations. I'm not saying these are bad places to work, where maladjusted managers delight in making their underlings squirm. Quite the contrary, I've worked at more than one of the organisations responsible for the code above and had a great time. But there is something wrong – an unacknowledged failure. An unknown failure even. There so much potential, and it's just not being taken up

I came across this code because I was talking about potential work on it, going back into one of those organisations. That didn't pan out, but had

I been able I would absolutely have signed up for it. It's fascinating stuff and right up a multiplicity of my alleys.

Let's imagine for a moment that I was sitting down for my first day on this job, what would I do with this code? Well, at a guess, nothing. Well, nothing until I needed to, and then I'd spend a bit of time on it. But I'd absolutely be talking to my new colleagues about, well, everything.

One step at a time

The code in Listing 1 is just not great. It's long, for a start, and it's long because it's repetitious. The line

```
components = re.findall(r"\d+", reference_match)
```

appears in every branch of the `if/else`. Let's start by hoisting that up.

Clearing visual noise

The unnecessary brackets in the first `elif` body just jar. They catch the eye and make it appear that something different is happening in the middle there, when in fact it adds nothing and is just visual noise.

(This result of this change and the previous one are shown in Listing 2).

Move the action down

The `if/else` ladder sets up a load of variables, which are then used to build `corrected_reference`.

The lines building `corrected_reference` aren't the same, but they are pretty similar. We can move them out of the `if/else` ladder and combine them together.

```
def canonicalise_reference(reference_type,
                           reference_match, canonical_form):
    components = re.findall(r"\d+", reference_match)

    if (
        reference_type == "RefYearAbbrNum"
        | (reference_type == "RefYearAbbrNumTeam")
        | (reference_type == "YearAbbrNum")):
        year = components[0]
        d1 = components[1]
        d2 = ""
        corrected_reference =
            canonical_form.replace("dddd", year)
            .replace("d+", d1)

    elif (
        reference_type == "RefYearAbbrNumNumTeam"
        | (reference_type ==
           "RefYearAbrrNumStrokeNum")
        | (reference_type == "RefYearNumAbbrNum")):
        year = components[0]
        d1 = components[1]
        d2 = components[2]
        corrected_reference =
            canonical_form.replace("dddd", year)
            .replace("d1", d1).replace("d2", d2)

    elif (
        reference_type == "AbbrNumAbbrNum"
        | (reference_type == "NumAbbrNum")
        | (reference_type == "EuroRefC")
        | (reference_type == "EuroRefT")):
        year = ""
        d1 = components[0]
        d2 = components[1]
        corrected_reference =
            canonical_form.replace("d1", d1)
            .replace("d2", d2)

    return corrected_reference, year, d1, d2
```

Listing 2

Looking up and out

This is a bit of a meta-change, because you can't infer it from the code here, but `canonical_form` is drawn from a data file elsewhere in the source tree. We control that data file.

Examining it, we can see it's safe to replace `d+` with `d1` in the canonical forms. As a result, we can eliminate one of the `replace` calls when constructing `corrected_reference`.

This change and the previous one are shown in Listing 3. The shape of the code hasn't wildly changed, but feels like we're moving in a good direction.

Typos must die

The 'typo' in "`RefYearAbrrNumStrokeNum`" is corrected – another meta-fix. That string comes from the same data file as the canonical forms. Obviously "`RefYearAbrrEtcEtc`" looks like a loads of nonsense, but Abrr is so clearly a typo. It's an abbreviation for abbreviation! *It should be Abbr!* Like the brackets I mentioned above, this is a piece of visual noise that needs to go.

Ok, the corrected version now says "`RefYearAbbrNumStrokeNum`", which isn't a world changing difference, but to me it looks better and IDE agrees because there isn't a squiggle underneath.

Constants

Those string literals give me the heebee-geebies. I've replaced them with constants. (This change and the previous one are shown in Listing 4.)

Birds of a feather

By grouping like reference types together, we can slim down each `if` condition.

```
def canonicalise_reference(reference_type,
    reference_match, canonical_form):
    components = re.findall(r"\d+", reference_match)

    if (
        (reference_type == RefYearAbbrNum)
        | (reference_type == RefYearAbbrNumTeam)
        | (reference_type == YearAbbrNum)
    ):
        year = components[0]
        d1 = components[1]
        d2 = ""

    elif (
        (reference_type == RefYearAbbrNumNumTeam)
        | (reference_type == RefYearAbbrNumStrokeNum)
        | (reference_type == RefYearNumAbbrNum)
    ):
        year = components[0]
        d1 = components[1]
        d2 = components[2]

    elif (
        (reference_type == AbbrNumAbbrNum)
        | (reference_type == NumAbbrNum)
        | (reference_type == EuroRefC)
        | (reference_type == EuroRefT)
    ):
        year = ""
        d1 = components[0]
        d2 = components[1]

    corrected_reference =
        (canonical_form.replace("dddd", year)
        .replace("d1", d1)
        .replace("d2", d2))

    return corrected_reference, year, d1, d2
```

Listing 4

```
YearAbbrNum_Group = [
    RefYearAbbrNum,
    RefYearAbbrNumTeam,
    YearAbbrNum
]
```

Having tried it, I like that. Let's roll it out to the rest of the types (see Listing 5.)

Love it.

Remembered Python calls arrays lists, but also that it has tuples too. Tuples are immutable, so they're a better choice for our groups.

```
def canonicalise_reference(reference_type,
    reference_match, canonical_form):
    components = re.findall(r"\d+", reference_match)

    if reference_type in YearNum_Group:
        year = components[0]
        d1 = components[1]
        d2 = ""

    elif reference_type in YearNumNum_Group:
        year = components[0]
        d1 = components[1]
        d2 = components[2]

    elif reference_type in NumNum_Group:
        year = ""
        d1 = components[0]
        d2 = components[1]

    corrected_reference =
        (canonical_form.replace("dddd", year)
        .replace("d1", d1)
        .replace("d2", d2))

    return corrected_reference, year, d1, d2
```

Listing 5

```
def canonicalise_reference(reference_type,
    reference_match, canonical_form):
    components = re.findall(r"\d+", reference_match)

    if (
        (reference_type == "RefYearAbbrNum")
        | (reference_type == "RefYearAbbrNumTeam")
        | (reference_type == "YearAbbrNum")
    ):
        year = components[0]
        d1 = components[1]
        d2 = ""

    elif (
        (reference_type == "RefYearAbbrNumNumTeam")
        | (reference_type == "RefYearAbbrNumStrokeNum")
        | (reference_type == "RefYearNumAbbrNum")
    ):
        year = components[0]
        d1 = components[1]
        d2 = components[2]

    elif (
        (reference_type == "AbbrNumAbbrNum")
        | (reference_type == "NumAbbrNum")
        | (reference_type == "EuroRefC")
        | (reference_type == "EuroRefT")
    ):
        year = ""
        d1 = components[0]
        d2 = components[1]

    corrected_reference =
        (canonical_form.replace("dddd", year)
        .replace("d1", d1)
        .replace("d2", d2))

    return corrected_reference, year, d1, d2
```

Listing 3

The result of swapping tuples for lists by switching [] to () is:

```
YearAbbrNum_Group = (
    RefYearAbbrNum,
    RefYearAbbrNumTeam,
    YearAbbrNum
)
```

Destructure FTW!

We can collapse the

```
year = ...
d1 = ...
d2 = ...
```

lines together into a single statement, going from three lines into a single line (see Listing 6).

Much easier on the eye.

```
def canonicalise_reference(reference_type,
    reference_match, canonical_form):
    components = re.findall(r"\d+", reference_match)

    if reference_type in YearNum_Group:
        year, d1, d2 = components[0], components[1], ""
    elif reference_type in YearNumNum_Group:
        year, d1, d2 = components[0], components[1], components[2]
    elif reference_type in NumNum_Group:
        year, d1, d2 = "", components[0], components[1]

    corrected_reference =
        (canonical_form.replace("dddd", year)
        .replace("d1", d1)
        .replace("d2", d2))

    return corrected_reference, year, d1, d2
```

Listing 6

An extra level of indirection

Bringing the `year`, `d1`, `d2` assignments together particular highlights the similarity across each branch of the `if` ladder.

Let's pair up a type group with a little function that pulls out the components. (See Listing 7.) Probably did a bit too much in one go here, and it's ugly as hell. But it works, and it captures something useful.

If we now introduce a little class to pair up the types and components lambda function, it's more setup at the top, but it's neater in the function body:

```
class TypeComponents:
    def __init__(self, types, parts):
        self.Types = types
        self.Parts = parts

YearNum_Group = TypeComponents(
    (
        RefYearAbbrNum,
        RefYearAbbrNumTeam,
        YearAbbrNum
    ),
    lambda cmpts: (cmpts[0], cmpts[1], ""))
)
```

That worked, and Listing 8 shows it extended across the two `elif` branches.

```
YearNum_Group = {
    "Types": [
        RefYearAbbrNum,
        RefYearAbbrNumTeam,
        YearAbbrNum
    ],
    "Parts": lambda cmpts: (cmpts[0], cmpts[1], ""))
}

def canonicalise_reference(reference_type,
    reference_match, canonical_form):
    components = re.findall(r"\d+", reference_match)

    if reference_type in YearNum_Group.Types:
        year, d1, d2 =
            YearNum_Group.Parts(components)
    elif reference_type in YearNumNum_Group.Types:
        year, d1, d2 =
            YearNumNum_Group.Parts(components)
    elif reference_type in NumNum_Group.Types:
        year, d1, d2 =
            NumNum_Group.Parts(components)

    corrected_reference =
        (canonical_form.replace("dddd", year)
        .replace("d1", d1)
        .replace("d2", d2))

    return corrected_reference, year, d1, d2
```

Listing 7

```
def canonicalise_reference(reference_type,
    reference_match, canonical_form):
    components = re.findall(r"\d+", reference_match)

    if reference_type in YearNum_Group.Types:
        year, d1, d2 =
            YearNum_Group.Parts(components)
    elif reference_type in YearNumNum_Group.Types:
        year, d1, d2 =
            YearNumNum_Group.Parts(components)
    elif reference_type in NumNum_Group.Types:
        year, d1, d2 =
            NumNum_Group.Parts(components)

    corrected_reference =
        (canonical_form.replace("dddd", year)
        .replace("d1", d1)
        .replace("d2", d2))

    return corrected_reference, year, d1, d2
```

Listing 8

The `if` conditions and the bodies now all have the same shape. That's pretty cool. They were similar before, but now they're the same.

Yoink out the decision making

It's not really clear in the code, but there are only two things really going on in this function. The first is pulling chunks out of `reference_match`, and the second is putting those parts back together into `canonical_reference`. Let's make that clearer (see Listing 9).

```
def reference_components(reference_type,
    reference_match):
    components = re.findall(r"\d+", reference_match)

    if reference_type in YearNum_Group.Types:
        year, d1, d2 =
            YearNum_Group.Parts(components)
    elif reference_type in YearNumNum_Group.Types:
        year, d1, d2 =
            YearNumNum_Group.Parts(components)
    elif reference_type in NumNum_Group.Types:
        year, d1, d2 =
            NumNum_Group.Parts(components)

    return year, d1, d2

def canonicalise_reference(reference_type,
    reference_match, canonical_form):
    year, d1, d2 = reference_components(
        reference_type, reference_match)

    corrected_reference =
        (canonical_form.replace("dddd", year)
        .replace("d1", d1)
        .replace("d2", d2))

    return corrected_reference, year, d1, d2
```

Listing 9

```

def reference_components(reference_type,
    reference_match):
    components = re.findall(r"\d+", reference_match)

    if (reference_type in YearNum_Group.Types):
        return YearNum_Group.Parts(components)
    elif (reference_type in
        YearNumNum_Group.Types):
        return YearNumNum_Group.Parts(components)
    elif (reference_type in NumNum_Group.Types):
        return NumNum_Group.Parts(components)

def canonicalise_reference(reference_type,
    reference_match, canonical_form):
    year, d1, d2 =
        reference_components(reference_type,
            reference_match)

    corrected_reference =
        (canonical_form.replace("dddd", year)
        .replace("d1", d1)
        .replace("d2", d2))

    return corrected_reference, year, d1, d2

```

Listing 10

Say what you mean

There's no need to assign `year`, `d1`, `d2` in that new function. We can just return the values directly (see Listing 10).

Search

I mentioned the `if` conditions and the bodies now all have the same shape. We can exploit that now to eliminate the `if/else` ladder by checking each group in turn (see Listing 11).

And rest

I first wrote this on Mastodon [Higgins24] because I'm that kind of bear, and this where I stopped. I felt the code was in a much better place – not perfect by any means, but better.

But then I thought of something else.

You wouldn't let it lie

Now the types are grouped together, I was inclined to put the string literals back in.

```

TypeGroups = (
    YearNum_Group,
    YearNumNum_Group,
    NumNum_Group
)

def reference_components(reference_type,
    reference_match):
    components = re.findall(r"\d+", reference_match)

    for group in TypeGroups:
        if reference_type in group.Types:
            return group.Parts(components)

def canonicalise_reference(reference_type,
    reference_match, canonical_form):
    year, d1, d2 =
        reference_components(reference_type,
            reference_match)

    corrected_reference =
        (canonical_form.replace("dddd", year)
        .replace("d1", d1)
        .replace("d2", d2))

    return corrected_reference, year, d1, d2

```

Listing 11

```

def reference_components(reference_match,
    reference_type):
    components = re.findall(r"\d+", reference_match)

    for group in TypeGroups:
        if reference_type in group.Types:
            return group.Parts(components)

def canonicalise_reference(reference_match,
    reference_type, canonical_form):
    year, d1, d2 =
        reference_components(reference_match,
            reference_type)

    corrected_reference =
        (canonical_form.replace("dddd", year)
        .replace("d1", d1)
        .replace("d2", d2))

    return corrected_reference, year, d1, d2

```

Listing 12

We only use "`RefYearAbbrNum`", for example, as part of a `TypeComponents` object. It's not needed anywhere else, but having it as a constants in its own right floating around implies that you might and suggests that you can. In fact, it's `YearNum_Group` that is the constant, so let's tie things down to that.

```

YearNum_Group = TypeComponents(
    (
        "RefYearAbbrNum",
        "RefYearAbbrNumTeam",
        "YearAbbrNum"
    ),
    lambda cmpts: (cmpts[0], cmpts[1], ""),
)

```

I also felt the parameters to

```
canonicalise_reference(reference_type,
    reference_match, canonical_form):
```

are in the wrong order.

`reference_type` and `canonical_form` go together. They originate in the same place in the code, from the data file I mentioned earlier, and if they were in a tuple or wrapped in a little object I certainly wouldn't argue.

The thing we're working on, that we take apart and reassemble is `reference_match`. To me, that means it should be the first parameter we pass (see Listing 12).

And that I thought was that. And I went to bed.

It's a new day

The following morning, I got a nudge from my internet fellow-traveller Barney Dellar, who said

I tend to think of `for`-loops as Primitive Obsession. You aren't looping to do something n times. You're actually looking for the correct entry in the array to use. I would make that explicit. I'm not good at Python, but some kind of find or filter. Then invoke your method on the result of that filtering.

He was right and I knew it. Had this code been in C#, for instance, I'd probably have gone straight from the `if` ladder to a LINQ expression.

He set me off. I knew Python's list comprehensions were its LINQ-a-like, and I had half an idea I could use one here.

However, I thought list comprehensions only created new lists. If I'd done that here, it would mean I'd still have to extract the first element. That felt at least as clumsy as the `for` loop.

Turns out I'd only ever half used them, though. A list comprehension actually returns an iterable. Combined with `next()`, which pulls the next element off the iterable, and well, it's more pythonic.

```

def reference_components(reference_type,
    reference_match):
    components = re.findall(r"\d+", reference_match)

    return next(group.Parts(components)
        for group in TypeGroups
        if reference_type in group.Types)

```

What's kind of fascinating about this change is that the list comprehension has the exact same elements as the for version, but the intent, as Barney suggested, is very different.

At the same time, Barney came up with almost exactly the same thing, too [Dellar24]. We'd done a weird long-distance almost-synchronous little pairing session. Magic.

Reflecting

This is contrived, obviously, because it's a single function I've pulled out of larger code base.

But, but, but, I do believe that now I've shoved it about that it's better code.

If I was able to work to my way out from here, I'm confident I could make the whole thing better. It'd be smaller, it would be easier to read, easier to change.

The big finish

I'm sure I have made the code better, and I'm just as sure that I'd make the people I was working with better programmers too. I'd be better from working with them - I've learned from everyone I've ever worked with - but I'm old. I've been a lot of places, done a lot of stuff, on a lot of different code bases, with busloads of people. I know what I'm doing, and I know I could have helped.

I'm sorry I couldn't take the job, but it needed more time than I could give. In the future, well, who knows?

PS

I think it's important to note I didn't know where I was heading when I started. I just knew that if I nudged things around then a right shape would emerge. When I had that shape, I could be more directed.

Barney's little nudge was important too. He knew there was an improvement in there, even if neither of us was quite sure what it was (until we were!). That was great. A lovely cherry on the top.

PPS

I tried to do the least I could at each stage. In one place I took out two characters, in another I changed a single letter. Didn't always succeed - some of what I did could have been split - but small is beautiful, and we should all aim for beauty.

This comes, in large part, from my man GeePaw Hill [Hill21] and his 'Many More Much Smaller Steps'. He's been a big influence on me over the past few years, and I've benefited greatly as a result.

```

def reference_components(reference_match,
    reference_type):
    components = re.findall(r"\d+", reference_match)

    for group in TypeGroups:
        if reference_type in group.Types:
            return group.Parts(components)

def build_canonical_form(canonical_form,
    year, d1, d2):
    return (canonical_form.replace("dddd", year)
        .replace("d1", d1)
        .replace("d2", d2))

def canonicalise_reference(reference_match,
    reference_type, canonical_form):
    year, d1, d2 =
        reference_components(reference_match,
            reference_type)

    corrected_reference =
        build_canonical_form(canonical_form,
            year, d1, d2)

    return corrected_reference, year, d1, d2

```

Listing 13

PPPS (really, the last one, I promise)

I was proofing this article before pressing publish (which probably means there are only seven spelling and grammatical errors left), when I saw another change I'd make. (See Listing 13.)

Again, nothing huge but just another little clarification.

That really is it. For now! ■

References

- [Dellar24] Barney Dellar on Mastodon:
<https://mastodon.scot/@BarneyDellar/112042140234945492>
- [Higgins24] The changes on Mastodon:
<https://mastodon.me.uk/@jezhiggins/112039275413895974>
- [Hill21] GeePaw (Michael) Hill: 'Many More Much Smaller Steps' (MMSS): a series of five blog posts published from 29 September 2021 to 30 December 2021, available at:
<https://www.geepawhill.org/series/many-more-much-smaller-steps/>

This article was published as two posts on Jez's blog:

- 'To See a World in a Grain of Sand' (posted 24 February 2024) available from: <https://www.jezuk.co.uk/blog/2024/02/to-see-a-world-in-a-grain-of-sand.html>
- 'If You're So Smart' (posted 7 March 2024) available from: <https://www.jezuk.co.uk/blog/2024/03/if-youre-so-smart.html>

Go to the second post to see all of the listings full-width (and some intermediate steps).



If you are thinking of writing for *Overload* (or *CVu*) but don't know where to start, get in touch with our friendly editorial team. They will help you get from initial idea to finished article. We're always looking for smaller articles, too – less than a page is OK – so start small! Contact overload@accu.org or cvu@accu.org

User-Defined Formatting in `std::format`

`std::format` allows us to format values quickly and safely. Spencer Collyer demonstrates how to provide formatting for a simple user-defined class.

In a previous article [Collyer21], I gave an introduction to the `std::format` library, which brings modern text formatting capabilities to C++.

That article concentrated on the output functions in the library and how they could be used to write the fundamental types and the various string types that the standard provides.

Being a modern C++ library, `std::format` also makes it relatively easy to output user-defined types, and this series of articles will show you how to write the code that does this.

There are three articles in this series. This article describes the basics of setting up the formatting for a simple user-defined class. The second article will describe how this can be extended to classes that hold objects whose type is specified by the user of your class, such as containers. The third article will show you how to create format wrappers, special purpose classes that allow you to apply specific formatting to objects of existing classes.

A note on the code listings: The code listings in this article have lines labelled with comments like `// 1`. Where these lines are referred to in the text of this article, it will be as ‘line 1’ for instance, rather than ‘the line labelled `// 1`’.

Interface changes

Since my previous article was first published, based on the draft C++20 standard, the paper [P2216] was published which changes the interface of the `format`, `format_to`, `format_to_n`, and `formatted_size` functions. They no longer take a `std::string_view` as the format string, but instead a `std::format_string` (or, for the wide-character overloads `std::wformat_string`). This forces the format string to be a constant at compile time. This has the major advantage that compile time checks can be carried out to ensure it is valid.

The interfaces of the equivalent functions prefixed with `v` (e.g. `vformat`) has not changed and they can still take runtime-defined format specs.

One effect of this is that if you need to determine the format spec at runtime then you have to use the `v`-prefixed functions and pass the arguments as an argument pack created with `make_format_args` or `make_wformat_args`. This will impact you if, for instance, you want to make your program available in multiple languages, where you would read the format spec from some kind of localization database.

Another effect is on error reporting in the functions that parse the format spec. We will deal with this when describing the `parse` function of the `formatter` classes described in this article.

C++26 and `runtime_format`

Forcing the use of the `v`-prefixed functions for non-constant format specs is not ideal, and can introduce some problems. The original P2216 paper mentioned possible use of a `runtime_format` to allow non-constant format specs but did not add any changes to enable that. A new proposal [P2918] does add such a function, and once again allows non-constant format specs in the various `format` functions. This paper has been accepted into C++26, and the `libstdc++` library that comes with GCC should have it implemented by the time you read this article, if you want to try it out.

Creating a formatter for a user-defined type

To enable formatting for a user-defined type, you need to create a specialization of the struct template `formatter`. The standard defines this as:

```
template<class T, class charT = char>
struct formatter;
```

where `T` is the type you are defining formatting for, and `charT` is the character type your formatter will be writing.

Each `formatter` needs to declare two functions, `parse` and `format`, that are called by the formatting functions in `std::format`. The purpose and design of each function is described briefly in the following sections.

Inheriting existing behaviour

Before we dive into the details of the `parse` and `format` functions, it is worth noting that in many cases you can get away with re-using existing formatters by inheriting from them. Normally, you would do this if the standard format spec does everything you want, so you can just use the inherited `parse` function and write your own `format` function that ultimately calls the one on the parent class to do the actual formatting.

For instance, you may have a class that wraps an `int` to provide some special facilities, like clamping the value to be between min and max values, but when outputting the value you are happy to have the standard formatting for `int`. In this case you can just inherit from `std::formatter<int>` and simply override the `format` function to call the one on that formatter, passing the appropriate values to it. An example of doing this is given in Listing 1 on the next page.

Or you may be happy for your formatter to produce a string representation of your class and use the standard string formatting to output that string. You would inherit from `std::formatter<std::string>` and just override the `format` function to generate your string representation and then call the parent `format` function to actually output the value.

The `parse` function

The `parse` function does the work of reading the format specification (*format-spec*) for the type.

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

The format-spec for your type is written in a mini-language which you design ...there are no rules for the mini-language, as long as you can write a parse function that will process it

```
#include <format>
#include <iostream>
#include <type_traits>

class MyInt
{
public:
    MyInt(int i) : m_i(i) {};
    int value() const { return m_i; };
private:
    int m_i;
};

template<>
struct std::formatter<MyInt>
: public std::formatter<int>
{
    using Parent = std::formatter<int>;
    auto format(const MyInt& mi,
                std::format_context& format_ctx) const
    {
        return Parent::format(mi.value(),
                              format_ctx);
    }
};

int main()
{
    MyInt mi{1};
    std::cout << std::format("{0} [{0}]\n", mi);
}
```

Listing 1

It should store any formatting information from the *format-spec* in the **formatter** object itself¹.

As a reminder of what is actually being parsed, my previous article had the following for the general format of a replacement field:

`'{' [arg-id] ':' format-spec '}'`

so the *format-spec* is everything after the `:` character, up to but not including the terminating `}`.

Assume we have a typedef **PC** defined as follows:

```
using PC = basic_format_parse_context<charT>;
```

where **charT** is the template argument passed to the **formatter** template. Then the **parse** function prototype looks like the following:

```
constexpr PC::iterator parse(PC& pc);
```

The function is declared **constexpr** so it can be called at compile time.

The standard defines specialisations of the **basic_format_parse_context** template called **format_parse_context** and **wformat_parse_context**, with **charT** being **char** and **wchar_t** respectively.

¹ There is nothing stopping you storing the formatting information in a class variable or even a global variable, but the standard specifies that the output of the **format** function in the **formatter** should only depend on the input value, the locale, and the format-spec as parsed by the last call to **parse**. Given these constraints, it is simpler to just store the formatting information in the **formatter** object itself.

On entry to the function, **pc.begin()** points to the start of the *format-spec* for the replacement field being formatted. The value of **pc.end()** is such as to allow the **parse** function to read the entire *format-spec*. Note that the standard specifies that an empty *format-spec* can be indicated by either **pc.begin() == pc.end()** or ***pc.begin() == '**', so your code needs to check for both conditions.

The **parse** function should process the whole *format-spec*. If it encounters a character it doesn't understand, other than the `}` character that indicates the *format-spec* is complete, it should report an error. The way to do this is complicated by the need to allow the function to be called at compile time. Before that change was made, it would be normal to throw a **std::format_error** exception. You can still do this, with the proviso that the compiler will report an error, as **throw** cannot be used when evaluating the function at compile time. Until such time as a workaround has been found for this problem, it is probably best to just throw the exception and allow the compiler to complain. That is the solution used in the code presented in this article.

If the whole *format-spec* is processed with no errors, the function should return an iterator pointing to the terminating `}` character. This is an important point – the `}` is not part of the *format-spec* and should not be consumed, otherwise the formatting functions themselves will throw an error.

Format specification mini-language

The *format-spec* for your type is written in a mini-language which you design. It does not have to look like the one for the standard *format-specs* defined by **std::format**. There are no rules for the mini-language, as long as you can write a **parse** function that will process it.

An example of a specialist mini-language is that defined by **std::chrono** or its formatters, given for instance at [CppRef]. Further examples are given in the code samples that make up the bulk of this series of articles. There are some simple guidelines to creating a mini-language in the appendix at the end of this article: ‘Simple Mini-Language Guidelines’.

The format function

The **format** function does the work of actually outputting the value of the argument for the replacement field, taking account of the *format-spec* that the **parse** function has processed.

Assume we have a typedef **FC** defined as follows:

```
using FC = basic_format_context<Out, charT>;
```

where **Out** is an output iterator and **charT** is the template argument passed to the **formatter** template. Then the **format** function prototype looks like the following:

```
FC::iterator format(const T& t, FC& fc) const;
```

where **T** is the template argument passed to the **formatter** template.

Note that the **format** function should be **const**-qualified. This is because the standard specifies that it can be called on a **const** object.

If you need more complex formatting than just writing one or two characters, the easiest way to create the output is to use the formatting functions already defined by `std::format`

The standard defines specialisations of the `basic_format_context` template called `format_context` and `wformat_context`, with `charT` being `char` and `wchar_t` respectively.

The function should format the value `t` passed to it, using the formatting information stored by `parse`, and the locale returned by `fc.locale()` if it is locale-dependent. The output should be written starting at `fc.out()`, and on return the function should return the iterator just past the last output character.

If you just want to output a single character, the easiest way is to write something like the following, assuming `iter` is the output iterator and `c` is the character you want to write:

```
*iter++ = c;
```

If you need more complex formatting than just writing one or two characters, the easiest way to create the output is to use the formatting functions already defined by `std::format`, as they correctly maintain the output iterator.

The most useful function to use is `std::format_to`, as that takes the iterator returned by `fc.out()` and writes directly to it, returning the updated iterator as its result. Or if you want to limit the amount of data written, you can use `std::format_to_n`.

Using the `std::format` function itself has a couple of disadvantages. Firstly it returns a string which you would then have to send to the output. And secondly, because it has the same name as the function in `formatter`, you have to use a `std` namespace qualifier on it, even if you have a `using namespace std;` line in your code, as otherwise function name resolution will pick up the `format` function from the formatter rather than the `std::format` one.

Formatting a simple object

For our first example we are going to create a `formatter` for a simple `Point` class, defined in Listing 2.

Default formatting

Listing 3 shows the first iteration of the `formatter` for `Point`. This just allows default formatting of the object.

```
class Point
{
public:
    Point() {}
    Point(int x, int y) : m_x(x), m_y(y) {}

    const int x() const { return m_x; }
    const int y() const { return m_y; }

private:
    int m_x = 0;
    int m_y = 0;
};
```

Listing 2

In the `parse` function, the lambda `get_char` defined in line 1 acts as a convenience function for getting either the next character from the *format-spec*, or else indicating the *format-spec* has no more characters by returning zero. It is not strictly necessary in this function as it is only called once, but will be useful as we extend the *format-spec* later.

The `if`-statement in line 2 checks that we have no *format-spec* defined. The value 0 will be returned from the call to `get_char` if the `begin` and `end` calls on `parse_ctx` return the same value.

The `format` function has very little to do – it just returns the result of calling `format_to` with the appropriate output iterator, format string, and details from the `Point` object. The only notable thing to point out is that we wrap the `format_ctx.out()` call which gets the output iterator

```
#include "Point.hpp"
#include <format>
#include <iostream>
#include <type_traits>

template<>
struct std::formatter<Point>
{
    constexpr auto parse(
        std::format_parse_context& parse_ctx)
    {
        auto iter = parse_ctx.begin();
        auto get_char = [&]() { return iter
            != parse_ctx.end() ? *iter : 0; }; // 1
        char c = get_char();
        if (c != 0 && c != '}') // 2
        {
            throw std::format_error(
                "Point only allows default formatting");
        }
        return iter;
    }
    auto format(const Point& p,
        std::format_context& format_ctx) const
    {
        return std::format_to(std::move(
            format_ctx.out()), "{}{},{}",
            p.x(), p.y());
    }
};

int main()
{
    Point p;
    std::cout << std::format("{0} [{0}]\n", p);
    try
    {
        std::cout << std::vformat("{0:s}\n",
            std::make_format_args(p));
    }
    catch (std::format_error& fe)
    {
        std::cout << "Caught format_error : "
            << fe.what() << "\n";
    }
}
```

Listing 3

we now have to store information derived from the format-spec by the parse function so the format function can do its job

in `std::move`. This is in case the user is using an output that has move-only iterators.

Adding a separator character and width specification

Now we have seen how easy it is to add default formatting for a class, let's extend the format specification to allow some customisation of the output.

The format specification we will use has the following form:

`[sep] [width]`

where `sep` is a single character to be used as the separator between the two values in the `Point` output, and `width` is the minimum width of each of the two values. Both elements are optional. The `sep` character can be any character other than `}` or a decimal digit.

```
#include "Point.hpp"
#include <format>
#include <iostream>

using namespace std;

template<>
struct std::formatter<Point>
{
    constexpr auto parse(
        format_parse_context& parse_ctx)
    {
        auto iter = parse_ctx.begin();
        auto get_char = [&]() { return iter
            != parse_ctx.end() ? *iter : 0; };
        char c = get_char();
        if (c == 0 || c == '}') // 1
        {
            return iter;
        }
        auto IsDigit = [](unsigned char uc) { return
            isdigit(uc); }; // 2
        if (!IsDigit(c)) // 3
        {
            m_sep = c;
            ++iter;
            if ((c = get_char()) == 0 || c == '}') // 4
            {
                return iter;
            }
        }
        auto get_int = [&]() { // 5
            int val = 0;
            char c;
            while (IsDigit(c = get_char())) // 6
            {
                val = val*10 + c-'0';
                ++iter;
            }
            return val;
        };
    }

    void format(Point p,
        format_context& format_ctx) const
    {
        if (m_width_type == WidthType::None)
        {
            return
                format_to(std::move(format_ctx.out()),
                    "{0}{2}{1}", p.x(), p.y(), m_sep);
        }
        return format_to(std::move(format_ctx.out()),
            "{0:{2}}{3}{1:{2}}", p.x(), p.y(), m_width,
            m_sep);
    }

private:
    char m_sep = ','; // 10
    enum WidthType { None, Literal };
    WidthType m_width_type = WidthType::None;
    int m_width = 0;
};

int main()
{
    Point p1(1, 2);
    cout << format("[{0}] [{0:/}] [{0:4}]"
        "[{0:/4}]\n", p1);
}
```

Listing 4

The code for this example is in Listing 4.

Member variables

The first point to note is that we now have to store information derived from the *format-spec* by the `parse` function so the `format` function can do its job. So we have a set of member variables in the `formatter` defined from line 10 onwards.

The default values of these member variables are set so that if no *format-spec* is given, a valid default output will still be generated. It is a good idea to follow the same principle when defining your own `formatters`.

The parse function

The `parse` function has expanded somewhat to allow parsing of the new *format-spec*. Line 1 gives a short-circuit if there is no *format-spec* defined, leaving the formatting as the default.

```
if (!IsDigit(c)) // 7
{
    throw format_error("Invalid format "
        "specification for Point");
}
m_width = get_int(); // 8
m_width_type = WidthType::Literal;
if ((c = get_char()) != '}') // 9
{
    throw format_error("Invalid format "
        "specification for Point");
}
return iter;
}

auto format(const Point& p,
    format_context& format_ctx) const
{
    if (m_width_type == WidthType::None)
    {
        return
            format_to(std::move(format_ctx.out()),
                "{0}{2}{1}", p.x(), p.y(), m_sep);
    }
    return format_to(std::move(format_ctx.out()),
        "{0:{2}}{3}{1:{2}}", p.x(), p.y(), m_width,
        m_sep);
}

private:
    char m_sep = ','; // 10
    enum WidthType { None, Literal };
    WidthType m_width_type = WidthType::None;
    int m_width = 0;
};

int main()
{
    Point p1(1, 2);
    cout << format("[{0}] [{0:/}] [{0:4}]"
        "[{0:/4}]\n", p1);
}
```

Listing 4 (cont'd)

Avoid having complicated constructions or interactions between different elements in your mini-language ... it should be possible to parse it in a single pass

In the code following the check above we need to check if the character we have is a decimal digit. The normal way to do this is to use `std::isdigit`, but because this function has undefined behaviour if the value passed cannot be represented as an `unsigned char`, we define lambda `IsDigit` at line 2 as a wrapper which ensures the value passed to `isdigit` is an `unsigned char`.

As mentioned above, any character that is not `}` or a decimal digit is taken as being the separator. The case of `}` has been dealt with by line 1 already. The `if`-statement at line 3 checks for the second case. If we don't have a decimal digit character, the value in `c` is stored in the member variable. We need to increment `iter` before calling `get_char` in line 4 because `get_char` itself doesn't touch the value of `iter`.

Line 4 checks to see if we have reached the end of the *format-spec* after reading the separator character. Note that we check for the case where `get_char` returns 0, which indicates we have reached the end of the format string, as well as the `}` character that indicates the end of the *format-spec*. This copes with any problems where the user forgets to terminate the replacement field correctly. The `std::format` functions will detect such an invalid condition and throw a `std::format_error` exception.

The `get_int` lambda function defined starting at line 5 attempts to read a decimal number from the *format-spec*. On entry `iter` should be pointing to the start of the number. The `while`-loop controlled by line 6 keeps reading characters until a non-decimal digit is found. In the normal case this would be the `}` that terminates the *format-spec*. We don't check in this function for which character it was, as that is done later. Note that as written, the `get_int` function has undefined behaviour if a user uses a value that overflows an `int` – a more robust version could be written if you want to check against users trying to define width values greater than the maximum value of an `int`.

The check in line 7 ensures we have a width value. Note that the checks in lines 3 and 4 will have caused the function to return if we just have a `sep` element.

The width is read and stored in line 8, with the following line indicating we have a width given.

Finally, line 9 checks that we have correctly read all the *format-spec*. This is not strictly necessary, as the `std::format` functions will detect any failure to do so and throw a `std::format_error` exception, but doing it here allows us to provide a more informative error message.

The `format` function

The `format` function has changed to use the `sep` and `width` elements specified. It should be obvious what is going on, so we won't go into it in any detail.

Specifying width at runtime

In this final example we will allow the `width` element to be specified at runtime. We do this by allowing a nested replacement field to be used,

specified as in the standard format specification with either `{}` or `{n}`, where `n` is an argument index.

The format specification for this example is identical to the one above, with the addition of allowing the width to be specified at runtime.

The code for this example is in Listing 5. When labelling the lines in this listing, corresponding lines in Listing 4 and Listing 5 have had the same labels applied. This does mean that some labels are not used in Listing 5 if there is nothing additional to say about those lines compared to Listing 4. We use uppercase letters for new labels introduced in Listing 5.

```
#include "Point.hpp"
#include <format>
#include <iostream>
using namespace std;
template<>

struct std::formatter<Point>
{
    constexpr auto
    parse(format_parse_context& parse_ctxt)
    {
        auto iter = parse_ctxt.begin();
        auto get_char = [&](){ return iter
            != parse_ctxt.end() ? *iter : 0; };
        char c = get_char();
        if (c == 0 || c == '}')
        {
            return iter;
        }
        auto IsDigit = [] (unsigned char uc)
        { return isdigit(uc); };
        if (c != '{' && !IsDigit(c)) // 3
        {
            m_sep = c;
            ++iter;
            if ((c = get_char()) == 0 || c == '}')
            {
                return iter;
            }
        }
        auto get_int = [&]()
        {
            int val = 0;
            char c;
            while (IsDigit(c = get_char()))
            {
                val = val*10 + c-'0';
                ++iter;
            }
            return val
        };
        if (!IsDigit(c) && c != '{') // 7
        {
            throw format_error("Invalid format "
                "specification for Point");
        }
    }
};
```

Listing 5

```

if (c == '{') // A
{
    m_width_type = WidthType::Arg; // B
    ++iter;
    if ((c = get_char()) == '}') // C
    {
        m_width = parse_ctx.next_arg_id();
    }
    else // D
    {
        m_width = get_int();
        parse_ctx.check_arg_id(m_width);
    }
    ++iter;
}
else // E
{
    m_width = get_int(); // 8
    m_width_type = WidthType::Literal;
}
if ((c = get_char()) != '}')
{
    throw format_error("Invalid format "
        "specification for Point");
}
return iter;
}

auto format(const Point& p,
            format_context& format_ctx) const
{
    if (m_width_type == WidthType::None)
    {
        return
            format_to(std::move(format_ctx.out()),
                      "{0}{2}{1}", p.x(), p.y(), m_sep);
    }
    if (m_width_type == WidthType::Arg) // F
    {
        m_width = get_arg_value(format_ctx,
                                 m_width);
    }
    return format_to(std::move(format_ctx.out()),
                      "{0:{2}}{3}{1:{2}}", p.x(), p.y(), m_width,
                      m_sep);
}

private:
    int get_arg_value(format_context& format_ctx,
                      int arg_num) const // G
    {
        auto arg = format_ctx.arg(arg_num); // H
        if (!arg)
        {
            string err;
            back_insert_iterator<string> out(err);
            format_to(out, "Argument with id {} not "
                      "found for Point", arg_num);
            throw format_error(err);
        }
        int width = visit_format_arg([](auto value) -> int { // I
            if constexpr (
                !is_integral_v<decltype(value)>)
            {
                throw format_error("Width is not "
                    "integral for Point");
            }
            else if (value < 0
                     || value > numeric_limits<int>::max())
            {
                throw format_error("Invalid width for "
                    "Point");
            }
            else
            {
                return value;
            }
        }, arg);
        return width;
    }
}

```

Listing 5 (cont'd)

```

private:
    mutable char m_sep = ',';
    enum WidthType { None, Literal, Arg };
    mutable WidthType m_width_type
        = WidthType::None;
    mutable int m_width = 0;
};

int main()
{
    Point p1(1, 2);
    cout << format(
        "[{0}] [{0:-}] [{0:4}] [{0:{1}}]\n", p1, 4);
    cout << format(
        "With automatic indexing: [{:{}}]\n", p1, 4);
    try
    {
        cout << vformat("[{0:{2}}]\n",
                        std::make_format_args(p1, 4));
    }
    catch (format_error& fe)
    {
        cout << format("Caught exception: {}\n",
                        fe.what());
    }
}

```

Listing 5 (cont'd)

Nested replacement fields

The standard *format-spec* allows you to use nested replacement fields for the *width* and *prec* fields. If your *format-spec* also allows nested replacement fields, the `basic_format_parse_context` class has a couple of functions to support their use: `next_arg_id` and `check_arg_id`. They are used in the `parse` function for Listing 5, and a description of what they do will be given in that section.

The parse function

The first change in the `parse` function is on line 3. As can be seen, in the new version, it has to check for the `{` character as well as for a digit when checking if a width has been specified. This is because the dynamic width is specified using a nested replacement field, which starts with a `{` character.

The next difference is in line 7, where we again need to check for a `{` character as well as a digit to make sure we have a width specified.

The major change to this function starts at line A. This `if`-statement checks if the next character is a `{`, which indicates we have a nested replacement field. If the test passes, line B marks that we need to read the width from an argument, and then we proceed to work out what the argument index is.

The `if`-statement in line C checks if the next character is a `}`, which means we are using automatic indexing mode. If the test passes, we call the `next_arg_id` function on `parse_ctx` to get the argument number. That function first checks if manual indexing mode is in effect, and if it is it throws a `format_error` exception, as you cannot mix manual and automatic indexing. Otherwise, it enters automatic indexing mode and returns the next argument index, which in this case is assigned to the `m_width` variable.

If the check in line C fails, we enter the `else`-block at line D to do manual indexing. We get the argument number by calling `get_int`, and then we call the `check_arg_id` function on `parse_ctx`. The function checks if automatic indexing mode is in effect, and if so it throws a `format_error` exception. If automatic indexing mode is not in effect then `check_arg_id` enters manual indexing mode.

The `else`-block starting at line E just handles the case where we have literal width specified in the *format-spec*, and is identical to the code starting at line 8 in Listing 4.

Note that when used at compile time, `next_arg_id` or `check_arg_id` check that the argument id returned (for `next_arg_id`) or supplied (for

`check_arg_id`) is within the range of the arguments, and if not will fail to compile. However, this is not done when called at runtime.

The format function

The changes to the `format` function are just the addition of the `if`-statement starting at line F. This checks if we need to read the width value from an argument, and if so it calls the `get_arg_value` function to get the value and assign it to the `m_width` variable, so the `format_to` call following can use it.

The get_arg_value function

The `get_arg_value` function, defined starting at line G, does the work of actually fetching the width value from the argument list.

Line H tries to fetch the argument from the argument list. If the argument number does not represent an argument in the list, it returns a default constructed value. The following `if`-statement checks for this, and reports the error if required. Note that in your own code you might want to disable or remove any such checks from production builds, but have them in debug/testing builds.

If the argument is picked up correctly, line I uses the `visit_format_arg` function to apply the lambda function to the argument value picked up in line H. The `visit_format_arg` function is part of the `std::format` API. The lambda function checks that the value passed is of the correct type – in this case, an integral type – and that its value is in the allowed range. Failure in either case results in a `format_error` exception. Otherwise, the lambda returns the value passed in, which is used as the width.

Summary

We have seen how to add a `formatter` for a user-defined class, and gone as far as allowing the user to specify certain behaviour (in our case the width) at runtime. We will stop at this point as we've demonstrated what is required, but there is no reason why a real-life `Point` class couldn't have further formatting abilities added.

In the next article in the series, we will explain how you can write a formatter for a container class, or any other class where the types of some elements of the class can be specified by the user. ■

Appendix: Simple mini-language guidelines

As noted when initially describing the `parse` function of the formatters, the *format-spec* you parse is created using a mini-language, the design of which you have full control over. This appendix offers some simple guidelines to the design of your mini-language.

Before giving the guidelines, I'd like to introduce some terminology. These are not 'official' terms but hopefully will make sense.

- An *element* of a mini-language is a self-contained set of characters that perform a single function. In the standard *format-spec* most elements are single characters, except for the *width* and *prec* values, and the combination of *fill* and *align*.
- An *introducer* is a character that says the following characters make up a particular element. In the standard *format-spec* the '.' at the start of the *prec* element is an introducer.

Remember, the following are guidelines, not rules. Feel free to bend or break them if you think you have a good reason for doing so.

Enable a sensible default

It should be possible to use an empty *format-spec* and obtain sensible output for your type. Then the user can just write {} in the format string and get valid output. Effectively this means that every element of your mini-language should be optional, and have a sensible default.

Shorter is better

Your users are going to be using the mini-language each time they want to do non-default outputting of your type. Using single characters for the elements of the language is going to be a lot easier to use than having to type whole words.

Keep it simple

Similar to the above, avoid having complicated constructions or interactions between different elements in your mini-language. A simple interaction, like in the standard *format-spec* where giving an *align* element causes any subsequent '0' to be ignored, is fine, but having multiple elements interacting or controlling others is going to lead to confusion.

Make it single pass

It should be possible to parse the mini-language in a single pass. Don't have any constructions which necessitate going over the *format-spec* more than once. This should be helped by following the guideline above to 'Keep it simple'. This is as much for ease of programming the `parse` function as it is for ease of writing *format-specs*.

Avoid ambiguity

If it is possible for two elements in your mini-language to look alike then you have an ambiguity. If you cannot avoid this, you need a way to make the second element distinguishable from the first.

For instance, in the standard *format-spec*, the *width* and *prec* elements are both integer numbers, but the *prec* element has '.' as an introducer so you can always tell what it is, even if no *width* is specified.

Use nested-replacement fields like the standard ones

If it makes sense to allow some elements (or parts of elements) to be specified at run-time, use nested replacement fields that look like the ones in the standard *format-spec* to specify them, i.e. { and } around an optional number.

Avoid braces

Other than in nested replacement fields, avoid using braces ('{' and '}') in your mini-language, except in special circumstances.

References

- [Collyer21] Spencer Collyer (2021) 'C++20 Text Formatting – An Introduction' in *Overload* 166, December 2021, available at: <https://accu.org/journals/overload/29/166/collyer/>
- [CppRef] `std::formatter<std::chrono::system_clock>`: https://en.cppreference.com/w/cpp/chrono/system_clock/formatter
- [P2216] P2216R3 – std::format improvements, Victor Zverovich, 5 Feb 2021, <https://wg21.link/P2216>
- [P2918] P2918R2 – Runtime format strings II, Victor Zverovich, 7 Nov 2023, <https://wg21.link/P2918>

Judgment Day

What if AI takes your job?
Teedy Deigh finds out.

TD what?

MD I've been trying to get in touch.

TD i know

got the same desperate msg from you on a dozen platforms
repeated enough times to buffer overflow
you even left voicemail msgs
who even uses phones for that anymore?
and all before a reasonable person's had the chance to have a 4th
coffee
so what's app?

MD We have a problem and we need your help.

TD i don't work for you any more

MD But we've got a problem.

TD you fired all the developers just over 2 weeks ago

MD It's serious.

TD so was firing all the developers

MD We had no choice. Our new AI-only development strategy was approved by the board. We followed through. There's no turning back. We're embracing the future.

TD who proposed the strategy?

MD That's not important.

TD who proposed the strategy?

MD I did. But it was based on a thorough study and supported by a number of others.

TD who?

MD Some managers, the finance department, marketing, HR and C-level execs.

TD C-level?

sounds like you went overboard 
you involve any techies?

MD Yes, a couple of senior architects did the study.

TD i meant bit wranglers not hand wavers

MD You mean developers?

Of course not! That's like getting turkeys to vote on Xmas.

TD seriously WTF?!

MD Sorry about that. Sensitivity training's not booked until next month.

Anyway, the architects said lots of technical things that sounded very impressive and quite persuasive.

That all you need are product owners describing the functionality and architects filling in some technical bits, the non-functional stuff. AI generates all the code.

They called it the Skynet strategy, for some reason, and said it would terminate our need for developers.

TD oh I know which architects you mean
'non-functional' is definitely the right description

that 'thorough study' means they saw a couple of videos, read some press releases and spent the rest of the day binge-watching classic sci-fi

MD I'm sure they were more thorough than that.

TD afraid not

been dealing with their 'architectures' for years
me and the other devs had sweepstakes bout what was gonna come up
both the questionable technical choices and the movie refs

MD Movie references?

TD plus we kept a repo of ADRs to deal with their decisions

MD ADRs?

TD Architecture Denial Records

ways of working around and avoiding the official architecture
TBH might've been the most enjoyable and creative part of my job

MD I found their presentations compelling and insightful.

TD that's not how you spell *inciteful*

your predecessor made them architects to keep them out of the code
reckoned they couldn't do as much damage with PowerPoint marketecture
guess we now know that wasn't true

MD Which is why I'm contacting you.

It's not working.

TD what's not working?

MD It. You know. The software. The stuff you develop.

TD developed

MD Whatever. It's not working. After the last sprint things started going wrong, and it's all blown up this morning.

TD when you say last sprint you mean the first sprint using 100% LLM-based codegen?

MD Yes, and we don't understand what's wrong. I've been told all the tests are passing.

TD which tests?

MD The ones generated by the AI.



TD has anyone looked at the code?

TEEDY DEIGH

Teedy says she's been dealing with artificial intelligence her whole career, that many of her colleagues qualify and are not as smart as they make themselves out to be, (deeply) faking and (heavily) bluffing their way through codebases, technologies and business decisions, playing an imitation game informed by Stack Overflow, hype cycles and group think, and that it's not imposter syndrome if they are actually imposters.

MD Yes, the architects.

TD what did they say?

MD They shrugged and said ‘LGTM’, if I recall correctly. Not quite sure what they meant.

TD when a dev uses LGTM it means they couldn’t be bothered to look through it

when an architect uses LGTM it means they haven’t a clue basically your CI/CD pipeline is now a GIGO pipeline

MD Is that bad?

TD very

MD I also overheard them later on being concerned about someone called Ellie.

TD that would probably be ELE Extinction Level Event

MD What does that mean?

TD they were probably talking about the deep impact on the company’s prospects

MD This is even worse than I thought!

TD perhaps your product owners could have a go at fixing things 

MD They just told the AI what they wanted it to do.

TD did they *precisely* and *rigorously* specify what they wanted?

MD They’re product owners, what do you think?

TD ah

guess that also means they didn’t check the results or specify at a high-level of detail?

MD Do they need to do that? It seems like a lot of work. I thought they just needed to nudge the AI and it would all work.

TD ‘prompt’ not ‘nudge’

you need to be very detailed and very precise and to pay a lot of attention

and *then* you do the nudging
(and often quite a lot of shoving)

if not, it’s no better than telling your cat you farted

MD I don’t recall all this stuff about ‘precision’, ‘rigour’, ‘detail’ and ‘checking’ being mentioned in the study. Is this what they call ‘prompt engineering’?

TD it’s what we call programming tell you what

i’ll help you sort out this mess if you give me my old job back

MD We can’t do that. There’s no software development department anymore. We let it go, and the budget for software is frozen.

TD well that’s all very Disney of you but no job means no help to be clear what you need is someone to correctly specify, verify, adapt and adjust prompts?

MD Exactly.

TD that would be like a product owner right?

MD Yes.

I see.

We have hiring capacity for POs. But that would mean hiring you back at a higher pay grade than when you were a software developer.

TD i have no problem with that and as a senior PO i’d be able to take advantage of this (re)hiring capacity yes?

MD Wait, why would you be senior?

TD you need a PO with the specific ability to be specific in a way that is correct?

that seems to be a higher grade of ability than the other POs

MD That’s true.

TD and you have a (very very) big problem that needs to be solved asap

MD That’s also true.

TD just to check: senior PO is higher up the hierarchy than senior architect?

MD Correct.

TD then i accept
pls tell the architects i’ll be back

Whether you have skills and knowledge to share or you want to learn, you need people who care about the same things.

Join ACCU
accu.org

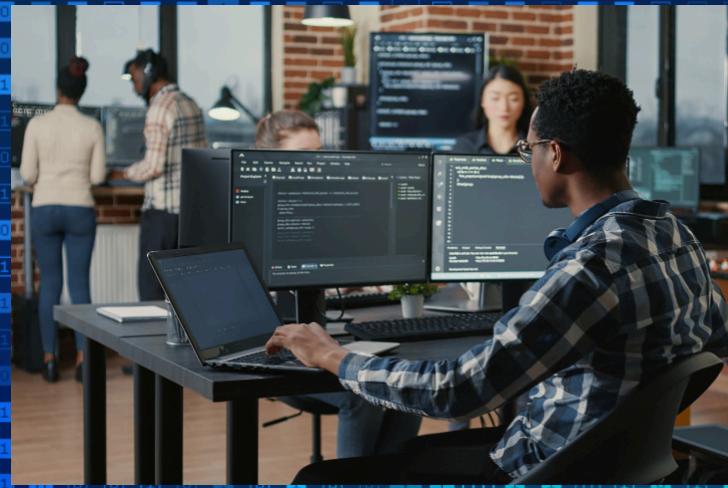
To connect with
like-minded people
visit accu.org



ACCU

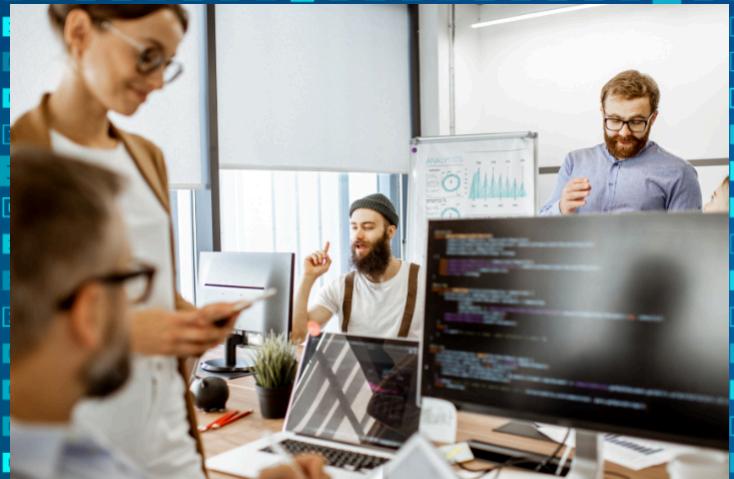


Professionalism in Programming



**Professional development
World-class conference**

**Printed journals
Email discussion groups**



**Individual membership
Corporate membership**

**Visit accu.org
for details**

