C++ safety, in context

Herb Sutter 2024-03-112024-04-07 28 Minutes

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 (https://www.iso.org/standard/76517.html)). *"**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.*

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.

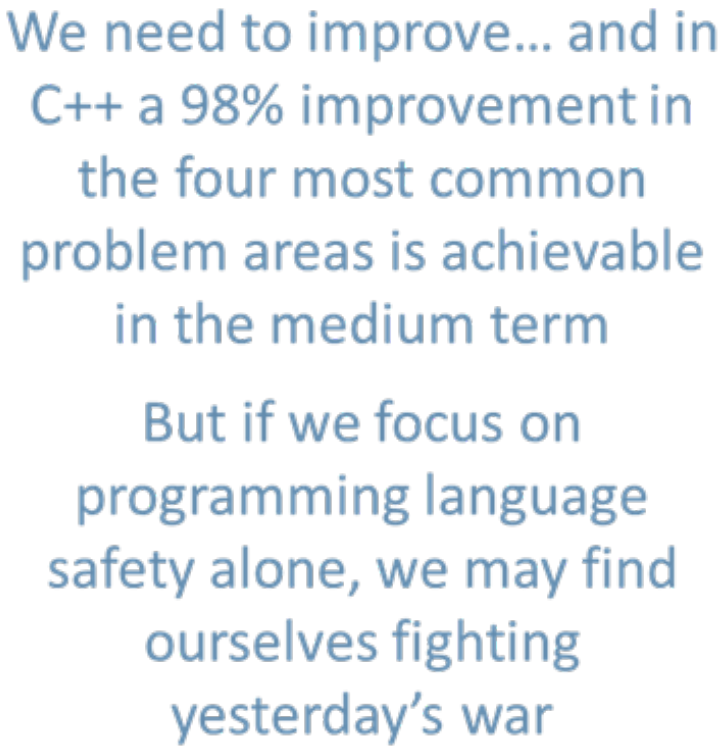
But there have been misconceptions, too, including focusing too narrowly on programming language safety as our industry's primary security and safety problem — it isn't. Many of the most damaging recent security breaches happened to code written in MSLs (e.g., Log4j (https://www.cisa.gov/news-events/news/apache-log4j-vulnerability-guidance)) or had nothing to do with programming languages (e.g., Kubernetes Secrets stored on public GitHub repos (https://blog.aquasec.com/the-ticking-supply-chain-attack-bomb-of-exposed-kubernetes-secrets)).

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.



(https://herbsutter.com/wp-content/uploads/2024/03/image.png)

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

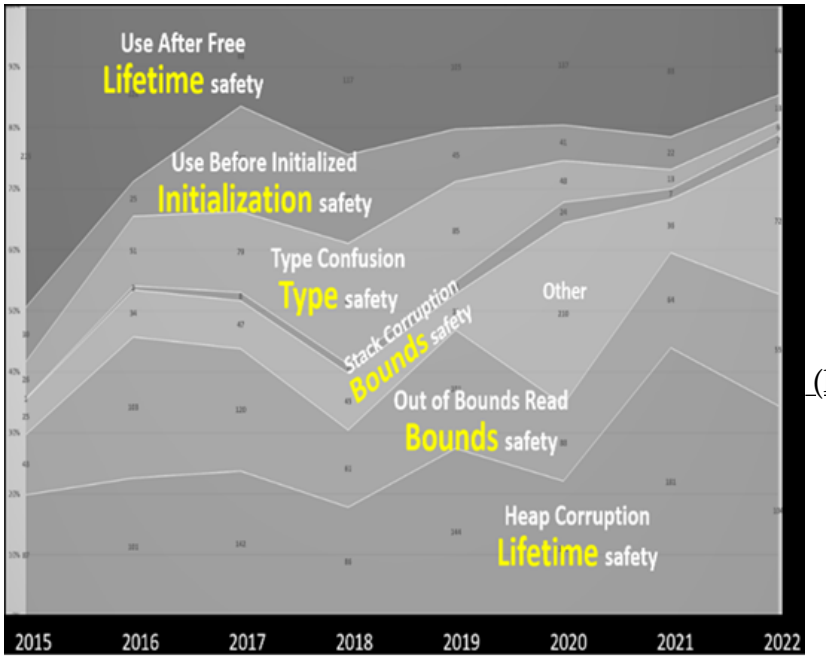
1	CWE-787	Out-of-bounds Write	63.72
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.54
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	34.27
4	CWE-416	Use After Free	16.71
5	CWE-78	Improper Neutralization of 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

(https://herbsutter.com/wp-content/uploads/2024/03/image-1.png)

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++ (example (https://media.defense.gov/2023/Dec/06/2003352724/-1/-1/0/THE-CASE-FOR-MEMORY-SAFE-ROADMAPS-TLP-CLEAR.PDF)), so by definition addressing these four would address the immediate NIST/NSA/CISA/etc. issues with C++. (More on this under "The problem 'isn't'… (1)" below.)

And in all recent years including 2023 (see figures 1's four highlighted rows, and figure 2), these four constitute the bulk of those oft-quoted 70% of CVEs (https://en.wikipedia.org/wiki/Common_Vulnerabilities_and_Exposures) (Common [Security] Vulnerabilities and Exposures) 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" (https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html#tableView) did not involve language safety and so are outside that denominator. More on this under "The problem 'isn't'… (3)" below.)



(2) Specifically memory safety CVEs:
Root cause proportions by patch year

(https://herbsutter.com/wp-content/uploads/2024/03/image-2.png)

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).

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).

> ***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 10-50x 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). *[Updated: Removed count of 2024 Rust vs C/C++ CVEs because MITRE.org search doesn't have a great way of accurately counting the latter.]* 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 in "The problem 'isn't'… (2)" below.
- 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 pointers); it's not just about linked lists (https://rust-unofficial.github.io/too-many-lists/) 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 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 conform 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++ (https://open-std.org/JTC1/SC22/WG21/docs/papers/2023/p2816r0.pdf). 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 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 fixits 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 problem 'is'…" above). 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 (https://learn.microsoft.com/en-us/cpp/safeint/safeint-library?view=msvc-170) for C++ to handle integer overflows, which is opt-in. C# has a checked arithmetic language feature (https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/checked-and-unchecked) 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 *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 problem 'is'…" above).

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 (https://thenewstack.io/unsafe-rust-in-the-wild/), compared for example to 25% of Java libraries (https://dl.acm.org/doi/abs/10.1145/2814270.2814313).
- 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 (https://rustc-dev-guide.rust-lang.org/sanitizers.html) too, including ThreadSanitizer and undefined behavior sanitizers (https://github.com/rust-lang/miri), 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 (https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-

security/) 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 (https://www.horizon3.ai/analysis-of-2023s-known-exploited-vulnerabilities/).)

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) (https://www.crn.com/news/security/10-major-cyberattacks-and-data-breaches-in-2023), and attackers go after the slowest animal in the herd. Most of the issues listed in NISTIR-8397 (https://nvlpubs.nist.gov/nistpubs/ir/2021/NIST.IR.8397.pdf) affect all languages equally, as they go beyond memory safety (e.g., Log4j (https://www.cisa.gov/news-events/news/apache-log4j-vulnerability-guidance)) 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 (https://learn.microsoft.com/en-us/cpp/code-quality/build-reliable-secure-programs?view=msvc-170), for which I was the editor. (More on this in the Call to Action.)
- MSLs get CVEs too, though definitely fewer (again, e.g., Log4j (https://www.cisa.gov/news-events/news/apache-log4j-vulnerability-guidance)). For example, see MITRE list of Rust CVEs (https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=rust), including six so far in 2024. And all programs use unsafe code; for example, see the *Conclusions* section of Firouzi et al. (https://www.researchgate.net/publication/344892072_On_the_use_of_C_Unsafe_Code_Context_An_Empirical_Study_of_Stack_Overflow)'s study of uses of C#'s unsafe on StackOverflow and prevalence of vulnerabilities, and that all programs eventually call trusted native libraries or operating system code.
- 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) (https://www.cve.org/Media/News/item/news/2023/08/29/Python-Software-Foundation-Added-as-CNA) for Python and pip distributions, 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.

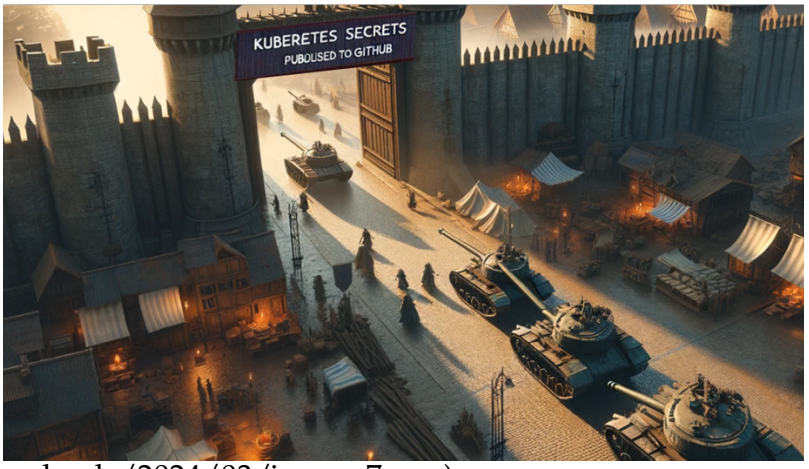
Call(s) to action

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

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 (https://blog.aquasec.com/the-ticking-supply-chain-attack-bomb-of-exposed-kubernetes-secrets) on public GitHub repos), misconfigured servers (e.g., DarkBeam (https://cybernews.com/security/darkbeam-data-leak/#google_vignette), Kid Security (https://cybernews.com/security/kidsecurity-parental-control-data-leak/)), lack of testing, supply chain vulnerabilities, social engineering, and other problems that are independent of programming languages. Apple's white paper (https://www.apple.com/newsroom/pdfs/The-Continued-Threat-to-Personal-Data-Key-Factors-Behind-the-2023-Increase.pdf) about 2023's

(https://herbsutter.com/wp-content/uploads/2024/03/image-7.png)



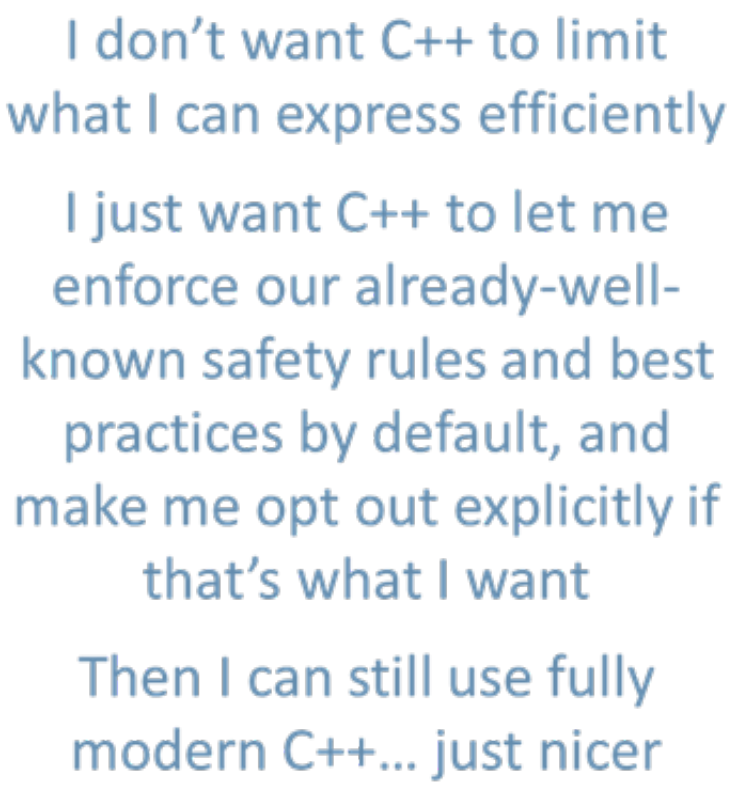
rise in cybercrime emphasizes improving the handling, not of program code, but of the data: "it's imperative that organizations consider limiting the amount of personal data they store in readable format while making a greater effort to protect the sensitive consumer data that they do store [including 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.



(https://herbsutter.com/wp-content/uploads/2024/03/image-3.png)

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.

(https://herbsutter.com/wp-content/uploads/2024/03/image-5.png)

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 it 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 (https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#pro-profiles), a set of about two dozen enforceable rules for type and memory safety of which I am a coauthor. 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* (https://www.amazon.com/C-Programming-Language-4th/dp/0321563840)'s advice on type safety; *C++ Coding Standards* (https://www.amazon.com/Coding-Standards-Rules-Guidelines-Practices/dp/0321113586)' section on type safety; the *Joint Strike Fighter Coding Standards* (https://www.stroustrup.com/JSF-AV-rules.pdf); High Integrity C++ (https://www.perforce.com/resources/qac/high-integrity-cpp-coding-standard); the C++ Core Guidelines section on safety profiles (https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#pro-profiles) (a small enforceable set of safety rules); and the recently-released MISRA C++:2023 (https://misra.org.uk/product/misra-cpp2023/).

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 (https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#SS-type) **by default.** 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++ run-time 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 (https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#probounds-bounds-safety-profile) **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 (https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#SS-type) and ES.20 (https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-always); and possibly zero-initialize data by default as currently proposed in P2723 (https://wg21.link/p2723). 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 (https://youtu.be/ELeZAKCN4tY?si=HKzgS8CUBdGREDAN&t=4305); definite-first-use rules (https://youtu.be/ELeZAKCN4tY?si=MnhZGU5xoRhTGF_V&t=4556).

Lifetime safety

Enforce the Pro.Lifetime safety profile (https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#SS-type) **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 RAII 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, is to default to an implicit return *this; (https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2973r0.html) 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 (https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0893r1.html), 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 (see: [a] (https://stackoverflow.com/q/8889522/2069064) [b] (https://stackoverflow.com/q/5939077/2069064) [c] (https://stackoverflow.com/q/14433884/2069064) [d] (https://stackoverflow.com/q/46806239/2069064) [e] (https://stackoverflow.com/q/25965157/2069064) [f] (https://stackoverflow.com/q/38643022/2069064) [g] (https://stackoverflow.com/q/45385837/2069064) [h] (https://stackoverflow.com/q/20989496/2069064) [i] (https://stackoverflow.com/q/35564553/2069064) [j] (https://stackoverflow.com/q/42335710/2069064) [k] (https://stackoverflow.com/q/37470518/2069064)).

- 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 (https://github.com/hsutter/cppfront/), 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 (https://learn.microsoft.com/en-us/cpp/code-quality/understanding-sal?view=msvc-170)" 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: 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: Define mathematically valid comparison chains Default return *this; for C assignment operators that return C& (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 post's conclusion: **Call(s) to action**.

Tagged:
coding,
cybersecurity,
programming,

security,
technology

Published by Herb Sutter

Herb Sutter is an author and speaker, a technical fellow at Citadel Securities, and serves as chair of the ISO C++ standards committee and chair of the Standard C++ Foundation. View all posts by Herb Sutter