Safety Strategy Requirements for C++

- Document Number: P3874R0
- Date: 2024-10-06

- Reply-to: Jon Bauman <jonbauman@rustfoundation.org>

- Audience: SG23, Safety and Security

Abstract

C++ is the dominant systems[1] language and for much of its existence has had no serious competition in that niche. That has begun to change and though it seems inevitable that C++ will continue to be *used* for the foreseeable future, whether it continues to be a first-choice language for new code and greenfield projects depends on how it addresses the problem of memory safety which has been identified by users, industry, government and academia as the source of most serious defects and vulnerabilities[2]. In its four decade history, C++ has changed dramatically, and while the work to add memory safety to the language would be significant, there is demonstrable evidence it is technically feasible while maintaining compatibility with legacy code. There is no way to guarantee the safety of legacy C++ code, but developing a safe mode of operation could ensure C++ remains viable for *new* development if two key criteria are met:

1. Safety by construction: no undefined behavior without explicitly invoking unsafe code such as legacy C++
2. Backwards compatibility: direct access to existing C++

Research has shown that the vast majority of vulnerabilities reside in new or recently modified code. Since there are now viable alternative languages providing safety by construction, C++ must achieve parity. The benefit of retaining access to all unsafe legacy code is a compelling benefit that enables intentional, targeted migration to memory safety and ensures forward progress. Therefore, C++ should adopt these requirements for its language safety strategy. This committee clearly takes safety seriously and has already initiated major efforts to improve language safety, but the specific language goals are not yet sufficient. Making this clear and decisive commitment is in the best interests of C++, its users, and all people who are affected by the technologies it powers.

---

[1] Used here in the sense of resource-constrained programming. In addition to operating systems and embedded programming, this includes applications such as games, databases, and web browsers.

[2] See statements from Microsoft, Google, OpenSSF, NSA, the cybersecurity authorities of the United States, Australia, Canada, the United Kingdom, and New Zealand, and ACM

What Is Memory Safety?

Outside of systems languages, nearly all modern programming languages are considered memory safe. To a programmer and user, there is no inherent benefit to memory *unsafety*. Rather, it is a sometimes necessary sacrifice for performance or low-level access necessary in systems programming. The advent of systems languages which provide memory safety by default[3] has led to much concern about C++'s lack of such facilities. While it's possible to give a good definition of memory safety, the real problem is the consequence of unsafe memory usage: undefined behavior (UB). Most discussion tends to focus on memory safety because UB is an abstruse, technical concept and because ensuring memory safety (particularly lifetime and data race safety) is the hardest part of eliminating undefined behavior. Also, while the most common and severe defects often stem from memory unsafety in particular, the potential consequences are the same regardless of how UB is encountered. To be precise, this paper is advocating for a commitment to develop a mode of operation where C++ is free of UB. But since the conversation in industry tends to be in terms of memory safety, this paper uses the term "safe" to mean no undefined behavior. After all, memory safety doesn't provide any sort of meaningful guarantees in the face of UB which can be caused in many other ways. What's really important is the distinction between heuristically reducing UB, and UB freedom by construction.

Why not just *reduce* UB?

C++ has seen numerous improvements in its history, and many changes have been intended to make it easier for developers to avoid defects, including UB. Additionally, static and dynamic analysis tools have made UB reduction more efficient. Despite this, UB and the vulnerabilities it entails remain common and its consequences are devastating. Part of the problem is that while a reduction in code defects generally leads to a proportionate reduction in operational failures, UB is particularly dangerous because of the potential for exploitable vulnerabilities. And unlike failures which are stochastically encountered by regular users, adversaries seek out and exploit vulnerabilities, so even a very low rate of UB in the context of many billions of lines of C++ represents a high degree of risk.

It is still very valuable to reduce UB as much as possible, and that work, especially that which is applicable to legacy code should and will continue. However, there are two reasons why a mode of C++ which is free of UB by construction is important. First, research has shown that vulnerabilities decay exponentially with the age of code[4], meaning that very old code which hasn't changed is far less likely to contain latent UB and that avoiding UB in new code has an outsized effect on the rate of defects and vulnerabilities in a codebase. Second, in a systems context it is sometimes necessary to write unsafe code, but since the vast majority of code can

[3] Rust and Swift are the most popular examples. Go is memory safe in the absence of concurrency.
[4] See "How Long Do Vulnerabilities Live in the Code? A Large-Scale Empirical Measurement Study on FOSS Vulnerability Lifetimes", "Memory Safe Languages in Android 13", and "Eliminating Memory Safety Vulnerabilities at the Source"

be written safely, explicitly opting-in to the unsafe mode of execution drastically reduces the possibility of UB and facilitates auditability since the origin of any UB must be a bug in unsafe code[5].

Safety By Construction

The concept of programming languages which are free of undefined behavior by default but which enable explicit unsafe operations is not new. The value of this paradigm is both difficult to overstate and hard to appreciate until it's been experienced. Owing to its long history and close connections to C, unsafe operations have always been pervasive in C++, but over time, better facilities within libraries and toolchains have reduced the need to use unsafe mechanisms to great benefit. Additionally, improved analyzers have helped discover latent UB both statically and at runtime. Still, there are numerous ways to trigger UB in C++ and by its very nature, the results are unpredictable and nonlocal; a crash or vulnerability in one part of a program could be the result of UB anywhere in the executable. And though analysis tools have improved greatly, there is no way to guarantee that any particular subset of a C++ program is free of undefined behavior.

Encapsulation is a familiar concept in C++ and it's an incredibly powerful tool for enforcing abstraction and enabling the construction and maintenance of large, complex systems. Similarly, approaches like RAII improve safety and maintainability by reducing the need for programmers to hold the full complexity of the system in their heads: local analysis and interfaces with clear guarantees make scaling systems feasible. As such, it's very consistent with C++'s philosophy to encapsulate unsafe operations. Though there will be significant amounts of unsafe C++ code for many years, the best way to ensure progress towards greater overall safety is to provide facilities for writing new code whose safety is assured. Especially since new code is the most likely to contain defects, ensuring those defects cannot cause UB guarantees progress when applying techniques to discover legacy UB.

Since the existence of a safe mode may not be familiar to C++ programmers, it's worth explaining the remarkable power of encapsulating unsafe operations. Within code which is free of UB by construction, it is *impossible* to cause any behavior which violates the abstract machine model of the language. This makes dealing with many difficult tasks like handling untrusted input or concurrent data manipulation much less hazardous. And still it is possible to enjoy the level of performance which requires inherently unsafe operations because safe code can surround narrowly-scoped unsafe blocks. Importantly, accessing unsafe facilities is not viral: a safe function may execute unsafe operations within its implementation that is opaque to its interface[6]. What makes this possible is that the rules of the safe mode preclude any operations which can cause UB and unsafe code is explicit in its interface about the requirements for upholding safe usage. The need for unsafe mode arises from operations which cannot be automatically proven safe through local analysis or where runtime checking has unacceptable

[5] Or the compiler/toolchain/operating system
[6] "Creating a Safe Abstraction over Unsafe Code" example from *The Rust Programming Language*

costs. Within an unsafe block, it is the caller's responsibility to ensure safety conditions for unsafe interfaces are upheld. And it is the responsibility of implementations of unsafe operations to document the requirements ensuring safe usage. Of course, bugs are inevitable, and since unsafe code *can* cause UB, it is a serious implementation error if UB can be caused when an unsafe function is used in accordance with its documented safety requirements. The technical term for such an implementation is "unsound"[7], and since unsafety can be encapsulated, unsoundness can lead to UB downstream of purely safe code. There is nothing magical happening, but the amazing power of safety encapsulation means that *any* UB encountered must ultimately arise from a bug in *unsafe* code. Since unsafe blocks can be narrowly scoped, the vast majority of programming can occur in safe mode, making an otherwise global search very clearly targeted.

The subset of superset approach[8] has been applied to improving C++ and one way of looking at unsafe mode is as a superset of the safe default which enables a small set of superpowers[9]. These are sometimes necessary in systems programming, but since they are the exception, it's possible to apply the greater responsibility that comes with them. To operate in a mode where UB can occur anywhere accepts unnecessary levels of risk, but it's not because C++ doesn't care about safety; it merely predates the existence of this approach. Cars did not always have seat belts, but it's not too late to incorporate new approaches to safety that have been shown to be effective. The philosophy of C++ is pragmatic: useful concepts pioneered elsewhere are regularly incorporated and the default can trade efficiency to gain safety as long as the tools to open the hood and thoughtfully access maximum performance are provided.

There's Too Much Legacy C++ To Update

In languages that are free of UB by default, the vast majority of code is written in the safe mode of operation. Since all existing C++ is unsafe, it would be a monumental task to convert it all to a future safe mode of operation. Fortunately, that's neither required nor advisable. Research has shown that the longer code lives, the less likely it is to contain defects. Even written in an unsafe language, it makes more sense to leave code which is functioning acceptably alone. However, there is a perpetual need to write new code and providing a facility to do so free of UB can ensure that C++ continues to be a vital language. While converting existing code to a UB-free mode of operation may be trivial and therefore worthwhile in many cases, there is virtually no downside to writing new code in such a mode since access to all legacy code is merely an unsafe block away.

Like any major technical transition, incorporating safe code will cause inconvenience, but it will largely be syntactic. Since any access to legacy code will require an unsafe block, there will initially be many blocks. There is both a risk here of making code less readable and of normalization of deviance blunting the sharp sense of caution that ideally accompanies the use

[7] See "How Safe and Unsafe Interact"
[8] See "C++ safety, in context"
[9] In Rust, there are only five unsafe superpowers.

of unsafe code in default safe languages. However, clever use of syntactic sugar and organizing code into regions of local safe logic separated from unsafe blocks accessing legacy code can mitigate these issues. Furthermore, while it may be distressing to see a pervasive reminder that many lines of code are unsafe, the status quo is that 100% of C++ is unsafe; that the syntax of the language is silent on this fact is true of all defaults does not make their semantic consequences any less real. In many ways it is a boon to make less desirable code unattractive as a gentle incentive to improvement.

What about the standard library?

Another natural concern in a transition to a safe mode is that all existing C++ libraries are unsafe, and updating even the standard library would be a major undertaking. Again, calling from safe code into unsafe library code is an improvement over the status quo. And though it would require some additional syntax to enable the unsafe operations, the calling code can remain safe rather than virally propagating this syntax to its callers. There is no additional syntax required for calling safe code from unsafe. With regards to the standard library in particular, there are two reasons safety is less of a problem.

First, just because code is written in unsafe mode (as all existing C++ is) does not mean that it *does* cause UB; merely that it *could*. It is entirely possible to write C++ that is memory safe in practice, although it is difficult and depends on how much one is willing to trade off between performance, ergonomics and resilience. The challenge in calling and writing unsafe code safely is that the programmer is responsible for upholding the invariants required for safety. If existing code were written such that there were no such invariants because all pointers and bounds were checked and there were no dependencies on caller-controlled lifetimes, it could be declared safe to call and the underlying unsafe operations encapsulated in local unsafe blocks. It is already the case that programmers must abide by the standard library's documented requirements for safe usage. Introduction of a safe mode merely formalizes it.

Second, as some of the oldest and most thoroughly used and reviewed code, authored by some of the most experienced and knowledgeable C++ programmers, the standard library is among the least likely *sources* of unsoundness. It may be easy to cause UB through consumer misuse, but that is the status quo and highlights the incredible power of *safe* interfaces. Assuming sound implementations, the nature of a safe interface makes it *impossible* to cause undefined behavior. To be clear, a safe interface may have non-safety requirements[10], and failing to uphold them may result in *incorrect* behavior, but never *undefined* behavior; the language is not broken; nasal demons are safely restrained.

Finally, wrapping existing unsafe code in safe interfaces with additional checking facilitates the creation of a safe standard library with code reuse where possible. The emphasis on performance and correctness in these kinds of foundational facilities makes judicious use of unsafe code encapsulated in safe interfaces a ubiquitous paradigm in languages with default

[10] Such as that a comparison predicate implements a total order

safe modes. This additionally allows the exposure of both “safe but checked” and “unsafe but unchecked” versions of the same operations[11] to provide the appropriate tools to different consumers based on need.

Perhaps the biggest concern with safe standard library interfaces is that implementing a useful safe mode of C++ may depend on new safe versions of existing types. If unsafe code required modification to use safe interface types, that could potentially be a viral effect which inhibits adoption of safe code. While this is an important concern, it is out of the scope of this paper, because it does not advocate for a particular implementation of safety; merely that a mode free of undefined behavior is an essential goal for C++. We have many different approaches to achieving this kind of safety in different languages, and active research[12] may yield others. Furthermore, C++ has always been a multi-paradigm language, so it’s even possible to have different approaches to achieving the same safety requirements which allow users to select the best compromise for their use case[13]. C++ has a long history of sophisticated syntax and rich options for implicit type conversions, so challenges regarding new types for the sake of safety are almost certainly surmountable.

Is this even feasible?

The success of Rust and Swift have demonstrated that safety by construction is enormously practical and valuable for a systems language. And Sean Baxter’s Circle compiler[14] is a compelling proof of concept that a similar approach can be used to extend C++ while retaining backward compatibility. While the amount of new features and syntax for that approach is significant, it is no more of a departure than modern C++ is from older versions which are still commonly used today. Furthermore, this paper does not advocate for any specific implementation, merely that safety by construction should be an explicit requirement of the C++ safety strategy. Concerns that this would require a new standard library are understandable, and given the language upon which to build it, such libraries will surely come, both as *additions* to the standard itself, which has often added new functionality to improve safety and as 3rd party libraries, which often pioneer such concepts and provide valuable implementation experience before standardization. In either case, a standard library making full use of new language safety features is not a prerequisite for enjoying the benefits of safety by construction. As previously mentioned, the greatest benefits are to new code.

What about profiles and the UB white paper?

When “Safe C++” (P3390) was presented as a plan to add memory safety to C++, it was considered as an alternative to Safety Profiles. In the poll, Safety Profiles received more support. Subsequently, Safety Profiles did not achieve consensus for inclusion in C++26 and the

[11] Such as `get()` vs `get_unchecked()`
[12] See “ Balancing the Books: Access Right Tracking for C++ - Lisa Lippincott - C++Now 2025 ”
[13] Swift and Rust take different approaches to achieve similar memory safety goals.
[14] https://www.circle-lang.org

work to pursue language safety was delegated to a whitepaper. A proposal for a profiles framework achieved consensus, but will not by itself provide safety improvements. As originally envisioned, Safety Profiles aim to provide heuristic improvements to safety by highlighting unsafe constructs through static analysis. This is a potentially valuable approach for discovering latent UB in legacy code, but it does not appear feasible to achieve memory safety by construction using this method[15]. As such, future work on safety via the Profiles framework should be seen as a complement to a safety by construction strategy, but not an alternative. The former can help improve the massive amount of legacy C++, but the latter is required to bring the safety of writing new code up to the standard of alternative languages. The approach to UB is currently the responsibility of the whitepaper which has a goal of creating a comprehensive list. This is a clear and necessary first step, but currently there is no commitment to providing UB freedom. Related to this, “Implicit Contract Assertions” (P3100R2) aims to treat all UB through the contracts mechanism, which requires runtime enforcement support. If successful, using the *quick-enforce* semantic could deliver a C++ which is memory safe. What the tradeoffs are in terms of program termination on otherwise benign UB or reduced performance to provide checking which cannot be achieved statically is unknown, but it is a good first step. UB freedom by construction would be a huge boon for writing new code in C++. Especially if heuristic approaches such as Safety Profiles statically identify latent defects, writing new code to fix them would be far more risky if the changes transformed the UB into subtler forms which elude the heuristics rather than completely eliminating it. Since alternative languages have been designed to provide safety with minimal runtime cost, it will likely be necessary to pursue language changes to facilitate higher performance while maintaining UB freedom, but given the level of concern about safety itself, ensuring the future for C++ as a language for new development demands that the first priority be UB itself.

Why commit now without a full design?

WG21 is a careful and deliberative body befitting the enormous impact of a language as important and pervasive as C++. Though there have been changes which many would argue were regrettable in retrospect, the broad consensus model means the default choice is to leave things alone. As with old code that is more likely to be free of bugs, this is a wise default for a technology which is well established and has demonstrated enormous value.

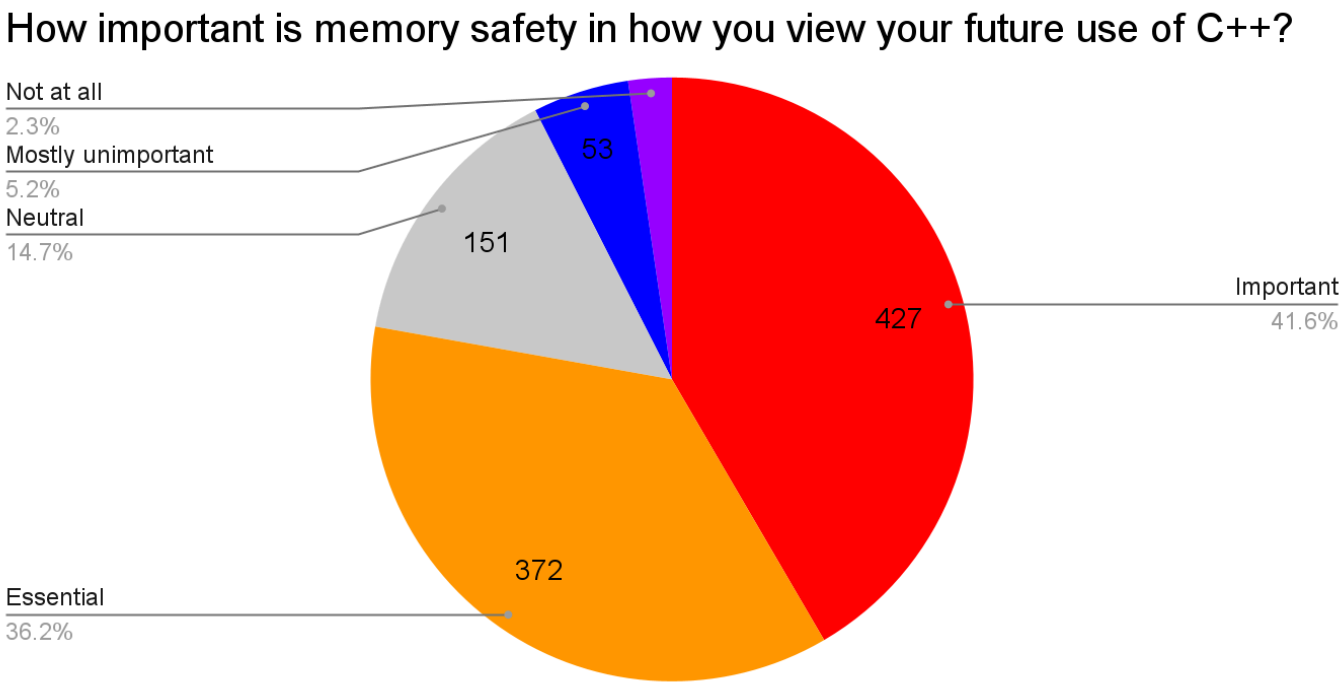
The cost of this judicious approach is that it can be difficult to take decisive action swiftly and in a body comprised of technologists, the tendency to debate the details of specific implementations can preclude committing to an important goal and communicating a clear direction to the huge community of C++ users. The current moment is a potential inflection point for C++. As safe-by-default systems languages are moving steadily through the technology adoption life cycle[16] from “early adopter” to “early majority” and significant calls are being made to move towards memory safety, even if that means moving away from C++. Staking out a clear

[15] “Why Safety Profiles Failed” provides an analysis of why the current language lacks the information necessary to provide such guarantees.
[16] See https://en.wikipedia.org/wiki/Technology_adoption_life_cycle

vision for the language has the potential to make the difference between a long future as a proven option among several capable systems languages or as a primarily legacy tool.

As things stand currently, the most public statement the committee and its leadership have made regarding memory safety describe it as “a very small part of security”[17] and though there is much valuable work underway to improve safety and reduce undefined behavior, there’s no reason to believe that memory safety by construction is likely to come to C++ in the near future. For users who find this kind of safety important, and there is clear evidence that many do, including some of the largest organizations in the world, they are currently obligated to make plans that do not include C++ to achieve it. Given the track record of C++ implementations delivering complex features in a timely and performant manner and the proofs of concept for achieving this kind of safety in a systems language, the fundamental obstacle seems to be the commitment to the vision.

Finally, this is what C++ developers clearly want. Based on the most recent survey, 77.8% described memory safety as “important” or “*essential*”. Only 7.5% said it was “mostly” or “not at all” important.



When asked “If you could wave a magic wand and change one thing about any part of C++” 2 of the 7 primary themes were “Memory Safety and Better Defaults” and “Remove Implicit Conversions and Undefined Behavior (UB)”. Additionally, a “Safe/Restrictive Subset or Mode” is one of the secondary themes. Per the summary:

[17] See “Request for Information: Open-Source Software Security: Areas of Long-Term Focus and Prioritization” and “DOE RFI Response”

> *There is strong consensus around modernizing the toolchain, improving safety and usability, and embracing breaking changes where necessary. Standardized package/build systems, memory safety mechanisms, and reflection are the clearest high-impact opportunities. Implementing these would directly address user pain points and significantly enhance daily productivity, maintainability, and adoption of C++.*

And in “Takeaways for Committee and Product Stakeholders”:

> *Top Priority: Deliver static reflection and safety features effectively; these are overwhelmingly the most anticipated changes.*

In many ways, the committee is already building towards these changes. It is in everyone’s interest to clearly communicate this to the wider world and commit to this ambitious and enormously valuable goal.

Conclusion

C++ stands at a crossroads. The committee has acknowledged the importance of memory safety and is working to identify and reduce UB, but hasn’t yet established whether C++ intends to provide memory safety. Meanwhile, memory-safe languages have demonstrated real-world benefits significant enough for organizations with some of the world’s largest investments in C++ to prefer them for new development. Public statements may have led observers to believe that the committee does not think memory safety is important or that it is too hard to add to the language. I do not believe either to be the case. Especially in light of the clear signal from users that such features are highly desired, plotting a clear course towards achieving memory safety will help ensure the best future for the language.

Acknowledgements

Thank you to all the members of WG21 who have been so welcoming to me as a new member and their curiosity about Rust and its approach to memory safety. In attendance at two meetings so far I have learned a great deal through discussion with too many individuals to name. Special thanks to David Sankel for reviewing this draft, and Jonathan Müller (“A principled approach to safety profiles”), Peter Bindels (“Making Safe C++ Happen”, “Subsetting”), Joshua Berne and Timur Doumler (“Implicit contract assertions”) for feedback and the initial inspiration.