Should C++ be a memory-safe language?

- Document Number: P3874R1
- Date: Feb 23, 2026
- Reply-to:
    - Jon Bauman <jonwg21@shumi.org>
    - Timur Doumler <papers@timur.audio>
    - Nevin Liber <nevin@cplusplusguy.com>
    - Ryan McDougall <mcdougall.ryan@gmail.com>
    - Pablo Halpern <phalpern@halpernwightsoftware.com>
    - Jeff Garland <jeff@crystalclearsoftware.com>
    - Jonathan Müller <jonathanmueller.dev@gmail.com>
- Audience: Evolution Working Group (/ug/)

Abstract

In recent years there have been numerous recommendations from industry, academia and government to prefer "memory-safe languages"[1]. Rust is consistently described as a memory-safe language while C++ is not — despite the fact that both can exhibit undefined behavior violating many of the commonly understood memory safety guarantees such as protection against buffer overflows and dangling pointers. This raises two important questions for which WG21 currently lacks consensus. Achieving it is essential to effectively direct the evolution of C++:

1. What is a memory-safe language?
2. Should C++ become one?

We argue that the term "memory-safe language" indicates systematic construction to guarantee the memory safety property, which implies freedom from any undefined behavior, and that the heuristic *reduction* of memory safety bugs and other forms of undefined behavior is insufficient to achieve the memory safety property. The purpose of this paper is to answer the two questions above and propose a clear goal for C++ to become a memory-safe language.

[1] CISA: "The Urgent Need for Memory Safety in Software Products", "Secure by Design: Google's Perspective on Memory Safety", Microsoft: "We need a safer systems programming language", and ACM: "It Is Time to Standardize Principles and Practices for Software Memory Safety" to name a few

Contents

[Motivation]
[What is a memory-safe language?]
 [Why is Rust considered a memory-safe language?]
[Should C++ become a memory-safe language?]
 [Defining C++ as a memory-safe language]
 [A memory-safe C++ is about new code]
 [Lifetime safety is the primary challenge]
 [Rust and Circle as a theoretical design]
 [What about the standard library?]
[Conclusion]
[Appendix: Encapsulating Unsafety]
[Acknowledgements]
[Revision History]

Motivation

The amount of C++ in existence today is difficult to measure, but clearly enormous. This includes much of the world's critical infrastructure, and many organizations must both maintain large existing C++ codebases and integrate new components. It is therefore to the benefit of all people that C++ evolves such that it can be the best choice for both existing and new code, providing the potential to build and expand islands of memory safety without having to switch languages entirely, which is often impractical or prohibitively expensive.

To address the interest in "memory-safe languages", it is essential to have a clear definition, but even the collection of twenty-one co-authors, spanning academia and industry, with expertise in memory-safety research, deployment, and policy who published ACM's, "It Is Time to Standardize Principles and Practices for Software Memory Safety" acknowledge that there isn't one:

> "During the last two decades, a set of research technologies for strong memory safety—memory-safe languages, hardware and software protection, formal approaches, and software compartmentalization—have reached sufficient maturity to see early deployment in security-critical use cases. However, there remains no shared, technology-neutral terminology or framework with which to specify memory-safety requirements"

In order to build consensus around a strategy, we will propose a definition for WG21 which will satisfy the requirements of a "memory-safe language" as the term is used in such recommendations. With a consensus definition, we can then decide whether or not WG21 will pursue such a goal.

Regardless of whether WG21 pursues the goal of C++ as a memory-safe language, existing work to *improve* language- and functional-safety[2] should continue. The goal proposed in this paper is complementary to those efforts, including safety profiles[3], the profiles framework[4], contracts, implicit contract assertions[5] and the core language UB white paper[6]. This proposal does not seek to make the perfect the enemy of the good, but if there is consensus that WG21 should pursue C++ becoming a memory-safe language, it is essential that consensus is achieved in full knowledge of the requirements of achieving such a goal. There is no point in going down a road that does not reach the desired destination. No amount of precondition assertions, safety profiles, or standard library hardening will suffice to make C++ a memory-safe language unless paired with superset features that provide a subset that is both useful and systematically free of undefined behavior.

What is a memory-safe language?

To define "memory-safe language", we examine its use. Assuming defects such as buffer overflows and use-after-free bugs represent memory-safety violations, and that all memory-safety violations entail undefined behavior[7], a language that is free of undefined behavior must be a memory-safe language. However, this definition is too narrow to capture the common use of the term since both Rust and C++ can exhibit undefined behavior arising from memory-safety violations. UB is a necessary feature of systems languages, which demand maximum performance and low-level hardware access, so strictly speaking only the default "safe" subset of Rust is a memory-safe language, but these recommendations are speaking at a high level and categorizing based on observed results.

To understand the different classifications of Rust and C++, we must consider *why* the recommendations to prefer memory-safe languages are being made. Consistently, the observation is made that memory-safety bugs lead to critical security vulnerabilities and that the use of memory-safe programming languages can eliminate them. These recommendations generally do not elaborate on *why* memory-safety violations lead to critical security vulnerabilities. Is there something special about memory-safety violations, or is this a property of undefined behavior more broadly?

The relationship between vulnerabilities and UB can be understood through the concept of a "weird machine"[8]. In contrast to the abstract machine which describes the semantics of a

---

[2] As defined by P3578 R1 'What is "Safety"?'
[3] P3081 R2 "Core safety profiles for C++26", P3446 R0 "Profile invalidation - eliminating dangling pointers"
[4] P3589 R2 "C++ Profiles: The Framework"
[5] P3100 R5 "A framework for systematically addressing undefined behaviour in the C++ Standard"
[6] P3656 R1 "Initial draft proposal for core language UB white paper: Process and major work items"
[7] P3100 R5 § 2.2.1: "The [undefined behavior] categories of Initialisation, Bounds, and Type and Lifetime correspond to the common terms *initialisation safety*, *bounds safety*, *type safety*, and *lifetime safety*, respectively, and collectively represent UB that is commonly referred to with the umbrella term *memory safety*"
[8] "Exploit Programming: From Buffer Overflows to "Weird Machines" and Theory of Computation"

programming language, a weird machine represents a computational environment which is unconstrained by language semantics and therefore vulnerable to exploitation. In a theoretical sense, UB means anything *can* happen, so no safety properties can hold. More practically, UB leads to an environment where the security[9] of a system may be compromised.

According to Ralf Jung[10], "it doesn't matter *why* your program has UB, what matters is that a program with UB defies the basic abstractions of the language itself, and this is a perfect breeding ground for vulnerabilities. Therefore, we shouldn't call a language 'memory safe' if the language does not systematically prevent Undefined Behavior." This refers to *any* kind of UB, including all categories described in P3100 R5, "A framework for systematically addressing undefined behaviour in the C++ Standard".

WebKit Architect Geoff Garen defines a memory-safety bug as "a bug that can create a weird machine", and that "weird machine bugs are Turing-complete, so even one weird machine bug is a sufficient gadget to reprogram an entire process", that they are "the most powerful primitives available to an attacker today", and that "any bug that can create a weird machine eventually will be used to."[11]

If the real concern is security vulnerabilities and those arise from undefined behavior and the weird machines they create, why is the discourse in terms of "memory safety" rather than "undefined behavior"? As a social construct, it is difficult to know why particular terminology becomes prevalent. Perhaps it is because memory-safety bugs are among the most common forms of UB and because lifetime- and concurrency-related bugs are among the hardest to avoid. Also, in systems languages, memory safety in the sense of the absence of any undefined behavior has only recently become a viable option. Finally, it seems to be the case that some forms of UB may be more exploitable either because they are easier to discover or because the exploit itself is simpler or more reliable[12]. Regardless, since the absence of UB makes software easier to reason about, it is a highly desirable guarantee for both security and maintainability.

Why is Rust considered a memory-safe language?

If one were to consider only the default mode of Rust which has no undefined behavior by definition, it would be easy to classify it as a memory-safe language. However, the use of "unsafe Rust" enables the 8 operations[13] which can lead to many forms of UB[14]. The vast majority of code is written in the default, UB-free subset of the language, but nearly all programs depend on both unsafe Rust code in the standard library as well as system libraries written in C or C++.

---

[9] That is, the ability to defend against threats that can lead to unauthorized information disclosure, theft or damage to hardware, software, or data
[10] Formal verification researcher, Author of Miri, member of many Rust Project teams, including lead for the Rust operational semantics team and member of the Rust compiler team. From "There is no memory safety without thread safety"
[11] "C++ Memory Safety in WebKit" Geoffrey Garen - C++Now 2025 (Slides)
[12] See CWE "Top 25 Most Dangerous Software Weaknesses" and "Top 10 KEV Weaknesses"
[13] The Rust Reference: Unsafety
[14] The Rust Reference: Behavior Considered Undefined

The authors of these recommendations are aware of these facts, and the dependencies on system libraries apply to essentially all languages considered memory safe. The reason Rust is considered a memory-safe language is that its guarantees against UB are systematic and syntactically explicit. This results in a much lower surface area for potential UB, making it more amenable to auditing and detection with automated tooling[15]. The consequence is that in practice, Rust achieves reliability and resistance to vulnerabilities similar to other memory-safe languages[16]. This reliability depends on the mechanism by which Rust encapsulates unsafe code. For more details, see the appendix on encapsulating unsafety.

Should C++ become a memory-safe language?

Given the recommendations to prefer memory-safe languages, how should WG21 respond?

1. **C++ is already safe enough**: Promote the message that modern C++ achieves acceptable memory safety when used correctly and combined with good engineering practices such as using containers and smart pointers, testing, code reviews, coding guidelines, and consistent use of tooling such as sanitizers and static analyzers. Any remaining memory-safety bugs are the cost for getting the highest performance in a high-level language.
2. **C++ will incrementally become safe enough**: Improvements to memory safety are already underway through mechanisms such as standard library hardening, contracts and safety profiles. New language features to increase memory-safety may be added, but nothing that makes the language “not C++”. The goal is not to eliminate all UB, even from a restricted subset of the language, merely to incrementally reduce it.
3. **C++ will become a memory-safe language**: The subset-of-superset approach is pursued to the same end as Rust and Swift, resulting in a useful subset with no UB. When using this subset similarly, C++ meets the definition of a memory-safe language.

While some members may support (1), the current path of the committee seems to be (2). Proposals to additionally pursue (3) have thus far failed to achieve consensus to pursue from SG23[17].

The previous revision of this proposal aimed to separate design considerations from the goal of making C++ a memory-safe language and achieved consensus in SG23[18]. In revising the proposal for consideration in EWG, it has become clear that WG21 lacks a consensus definition of “memory-safe language”. This revision aims to provide that definition and settle the question of whether C++ should evolve to become one. Though this proposal does not commit to a

[15] Miri is an Undefined Behavior detection tool for Rust that can detect unsafe code that fails to uphold its safety requirements
[16] See “Secure by Design: Google’s Perspective on Memory Safety” § “Preventing Memory Safety Bugs through Safe Coding”
[17] P2771 R1 “Towards memory safety in C++” (June 2023), P3390 R0 “Safe C++” (November 2024)
[18] P3874 R0 “Safety Strategy Requirements for C++” (November 2025)

particular design to achieve this goal, it will consider what sort of approach would achieve the goal and the costs associated with implementation and standardization.

Defining C++ as a memory-safe language

This paper argues that the classification of "memory safe language" depends on automatically-enforced guarantees against **all** uses of memory which violate the semantics of the abstract machine. Since undefined behavior places no requirements on program behavior, its presence is incompatible with any such guarantees.

However, these guarantees are not absolute. Even using a fully UB-free language doesn't guarantee *global* UB-freedom in practice because the computation environment depends on more than just your program. The presence of logic errors in the toolchain or underlying system can lead to undefined behavior on program execution. The guarantees against UB which make a "memory-safe language" are locally enforced both during compilation and at runtime and mean that the UB didn't *originate* in the UB-free subset.

The differentiating factor between Rust and C++ is that Rust contains a syntactically-explicit, compiler-enforced language subset which is free of UB. If C++ can define a language subset which is similarly free of UB *and* useful enough for the vast majority of programming tasks, it too can be considered a memory-safe language. In fact, C++ already *has* a subset which is free of UB in the form of *constexpr*. This is of great benefit, but not sufficient for general programming. And while it would be technically possible to define a subset excluding all runtime-UB-triggering operations, a subset that does not allow pointer dereference or arithmetic would not be useful. This is why it is necessary to add new features that *replace* those which must be restricted for the sake of avoiding UB.

This paper proposes a subset-of-superset strategy[19] for C++ to achieve memory safety such that:

1. The subset is syntactically explicit and well-defined; no undefined behavior can be exhibited within it and only code within the subset may be accessed.
2. Code outside the subset may exhibit undefined behavior, but can still directly access all code within the well-defined subset.

Fundamentally, the subset is restrictive for the purpose of providing a guarantee of the memory safety property. This is not to be confused with the absence of memory safety bugs. Diligent use of many programming techniques can effectively reduce the rate of bugs, but guarantees depend upon provable properties and automatic enforcement. The reason the memory safety property is important is because it implies a program is well-formed and its behavior can be described by the abstract machine. Any program which can exhibit UB under some inputs provides no guarantees whatsoever. Calls for the use of memory-safe languages are based on

[19] "A rationale for semantically enhanced library languages", Bjarne Stroustrup

the observation that UB leads to exploitable vulnerabilities and is especially challenging to debug.

A memory-safe C++ is about *new* code

Given the amount of C++ already written, it will remain a relevant language for the foreseeable future. Whether C++ maintains its popularity as a choice for new code in the long-term is less certain. The value of a memory-safe subset is focused on *new* code. As previously mentioned, existing efforts to reduce memory-safety defects which apply to existing code such as safety profiles, standard library hardening and implicit contract assertions should continue. Developing a memory-safe subset for C++ with guarantees comparable to other systems languages[20] will ensure the greatest freedom to choose the best tool for the job.

Lifetime safety is the primary challenge

Both C++ and Rust have relatively comparable amounts of explicit undefined behavior[21], and in the case of both languages, some kinds are easier to avoid than others. Bounds safety, for instance, entails a performance cost, but is straightforward to enforce. The most challenging form of undefined behavior to address appears to be lifetime safety. Unsurprisingly, this is the reason for Rust's most notable feature: references whose lifetimes are enforced by the borrow checker. This is purely a compile-time phenomenon and based on local analysis that considers only the signature of called functions rather than whole-program analysis. However, the borrow checking implementation itself is fairly complex and depends upon a novel mid-level intermediate representation (MIR) which preserves the necessary control flow[22]. Swift takes a different approach based on automatic reference counting, but this entails additional runtime overhead. There are other approaches which could be successful, but it seems clear that if C++ is to achieve memory-safety, some significant new feature will be required.

The other kind of safety often considered to be especially challenging is data race safety. Somewhat surprisingly, the approach Rust takes to solving this problem is not even a language-level construct. Rather, it is based on unsafe traits[23] that are responsible for enforcing the necessary synchronization to allow data to be referenced across or moved between threads. Paired with standard library primitives like `Arc` and `Mutex` which syntactically prevent unsynchronized access, it is a comparatively simple matter to achieve data race safety.

[20] Notably Rust and Swift, but Carbon, Mojo and other active research aims to provide memory-safety and systems-level performance
[21] See The Rust Reference: "Behavior considered undefined" and P3100 R5: "A framework for systematically addressing undefined behaviour in the C++ Standard"
[22] The Circle Language also uses the same borrow-checking approach. See P3390 R0 "Safe C++" § 2.2: Borrow checking for a thorough discussion.
[23] Circle also uses this approach, described in § 2.7: `send` and `sync`

Rust and Circle as a theoretical design

This proposal is explicitly about the definition of a memory-safe language and the decision to pursue this goal for C++. It does not *commit* to a particular design, but for the sake of considering the commitment involved, it is worth considering both Rust and Circle as points of comparison. Rust is a memory-safe language designed in response to C++ and targeting the same domain with many of the same principles. Circle is an extension of C++ based on Rust's design for memory safety which implements the subset-of-superset strategy and satisfies the "no undefined behavior" definition of memory safety in this proposal[24]. Rust has 10 years of experience in deployment at scale and significant adoption across nearly all the core industries as C++[25]. Circle has a functional implementation available in binary form and on Compiler Explorer which is demonstrative of the fundamental effort to apply this design to C++[26].

What about the standard library?

Perhaps the greatest concern with regards to a memory-safe C++ is that it will require an entire new standard library before it is useful. Since code which can cause UB is disallowed in the memory-safe subset, using the existing standard library would require frequent escapes. This represents no loss of functionality, but it would be considerably unergonomic. However, any standard library interfaces which are free of UB currently[27] could be simply declared as part of the memory-safe subset. For the rest, addressing this requires new *interfaces* in the memory-safe subset, but in many cases it should be possible for these interfaces to soundly encapsulate existing implementations. This technique is used even in the Rust standard library where "unsafe Rust" is used internally to achieve maximum performance. See the appendix on encapsulating unsafety for details. In at least some cases, wholly different APIs will be required. For example, the requirements that `begin` and `end` iterators reference the same container cannot be easily enforced for syntactically separate entities, so the amount of work may be significant, but the memory-safety benefits of the subset are fundamentally *accessible* once the language features are implemented. The reality of decades of legacy C++ existing is unavoidable, but it does not preclude the value of adding memory-safety for new code.

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 become a memory-safe language. Meanwhile, memory-safe languages have demonstrated real-world benefits significant enough for organizations with some of the world's largest

[24] See § 1.2: Extend C++ for safety
[25] Safety-critical is perhaps the most notable exception, but functional-safety should not be confused with language-safety. See P3578 R1 "What is "Safety"?", and the Safety-Critical Rust Consortium for more.
[26] See the minutes of the presentation of P3390 to SG23 for a discussion of the effort involved.
[27] For example, likely all of the `std::variant` API except for `get_if` could be declared memory-safe

investments in C++ to prefer them for new development[28]. 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[29]. Neither is the case, but leading the evolution of C++ demands clarity and consensus. Especially in light of the clear signal from users that such features are highly desired, plotting a clear course towards making C++ a memory-safe language will help ensure the best future for all.

Appendix: Encapsulating Unsafety

Understanding the relationship between the default, memory-safe subset of languages like Rust and Swift and the nature of the guarantees they provide is both complex and subtle. Fundamentally, UB is a property of an execution which depends both on the binary produced by the toolchain and the execution environment. To say that using memory-safe languages prevents the class of defects arising from UB is both true and incomplete. The discourse around memory safety ranges from high-level articles targeting an executive audience[30] to academic papers written for programming language theorists engaged in formal verification proofs[31]. The needs of C++ evolution lie somewhere between and require abstractions and guarantees which are useful for specification and implementation.

In practice, a memory-safe language, whether a subset of a systems language or a complete non-systems language is free of UB because it restricts operations which necessarily risk UB for the sake of performance, such as unchecked array access or mutable aliasing. In non-systems languages, performance is compromised to achieve memory safety and in the memory-safe subset of systems languages, expressiveness is compromised. By using the “escape hatch” of explicit memory-unsafe operations, both full performance and expressiveness is available, but the cost is that avoiding UB relies on the programmer upholding guarantees that cannot be checked by the compiler[32]. The advances in theory, developed in response to C++’s three decades of global success and the persistent challenges posed by memory-safety bugs have given rise to systems languages with memory-safe subsets with sufficient expressiveness and performance for the vast majority of uses. Additionally, their memory-unsafe code can be effectively encapsulated and used from the memory-safe subset without risk of triggering UB. To explain how that is possible requires a few additional definitions.

Memory-Unsafe API: an interface that might cause UB if safety conditions[33] are not upheld.

[28] “Rust in Android: move fast and fix things”, “Microsoft is Getting Rusty: A Review of Successes and Challenges”, “Adobe’s memory safety roadmap: Securing creativity by design”
[29] See “Request for Information: Open-Source Software Security: Areas of Long-Term Focus and Prioritization” and The Direction Group’s “DOE RFI Response”
[30] “Improving Europe's cybersecurity posture through memory safety”
[31] “RustBelt: Securing the Foundations of the Rust Programming Language”
[32] Compilers have bugs too, of course. The Rust compiler maintains a label for bugs which represent soundness issues.
[33] See The Rust Reference. Safety conditions are similar to the C++ definition of preconditions.

Memory-Safe API: an interface that can be used from the memory-safe language subset without restrictions. The implementation may contain potentially memory-unsafe operations encapsulated within an abstraction boundary.

Soundness: a property of an abstraction boundary that prevents misuse of the memory-safe language subset from exhibiting undefined behavior.

For a simple example of a memory-unsafe API, consider the `get_unchecked()` method on the Rust vector type. This method returns a reference without doing bounds checking, which runs the risk of UB. It is trivial to implement the memory-*safe* API `get()` in terms of the memory-unsafe one, and as of writing, this is indeed how the Rust implementation works:

```
fn get(self[34], slice: &[T]) -> Option<&T> {
  if self < slice.len() {
    // SAFETY: `self` is checked to be in bounds.
    unsafe { Some(slice_get_unchecked(slice, self)) }
  } else {
    None
  }
}
```

This is roughly equivalent to the following C++:

```
template <class T>
auto get(size_t idx, span<T const> slice) -> optional<T const&> {
  if (idx < slice.size ()) {
    return slice[idx];
  } else {
    return nullopt;
  }
}
```

Though the implementation of `get()` contains memory-unsafe operations that require the use of the `unsafe` keyword, the `get()` API is memory safe and properly encapsulated: it is impossible to misuse in a way that exhibits undefined behavior. We say that such an API is *sound*. Clearly, in order to deliver memory-safety guarantees at the program level, memory-safe APIs must be sound. This is analogous to how type-safe APIs can be implemented in terms of properly encapsulated type-unsafe constructs in C++, such as the example of `std::variant` being implemented in terms of a union.

[34] Note that `self` in this case is a `usize` since the implementation is in terms of the helper trait `SliceIndex`. This allows various linearly-indexed containers to share the implementation.

In other cases, a memory-safe API relies on termination to avoid UB. Consider this example of vector insertion from The Rustonomicon:

```
pub fn insert(&mut self, index: usize, elem: T) {
    // Note: `<=` because it's valid to insert after everything
    // which would be equivalent to push.
    assert!(index <= self.len, "index out of bounds");
    if self.len == self.cap { self.grow(); }

    unsafe { /* insertion logic omitted for brevity */ }

    self.len += 1;
}
```

In practice, the approach to providing sound APIs is to limit the use of memory-unsafe constructs as much as possible. Since most commonly-used data structures and algorithms that cannot be expressed through Rust's memory-safe subset are provided in the standard library, there is a high degree of confidence in the soundness of the memory-safe APIs, which is reflected in the extremely low incidence of memory-safety defects in practice.

One common point of confusion is whether a defect in the memory-safe subset can trigger UB through a chain of causality. Consider this example which exhibits UB:

```
let v = vec![1, 2, 3];
let i = v.len();
unsafe { v.get_unchecked(i); }
```

One could say that setting `i` to an invalid index is the *cause*, but the use of the `unsafe` keyword enables operations that the compiler cannot check and puts the responsibility for avoiding UB on the programmer to uphold the documented safety conditions. In this case:

> Calling this method with an out-of-bounds index is *undefined behavior* even if the resulting reference is not used.

One of the major benefits of explicit `unsafe` block syntax is auditability. There is a clear indication of all code regions where upholding safety conditions are the responsibility of the programmer. An audit of `unsafe` code blocks would lead to the point where the `get_unchecked` method was called out of contract, even if the eventual fix occurred outside the `unsafe` block. This is a very simple example, but since all operations that ultimately trigger UB are disallowed in the memory-safe context, all UB is the result of a similar pattern: failure to uphold the safety conditions on which code in a syntactic `unsafe` context depends for well-formedness.

In general, safety conditions can depend on the maintenance of safety invariants in program state[35] that can be invalidated by code in the memory-safe subset[36], so encapsulation depends on thoughtful design and abstraction boundaries. Again, this is very similar to the way that C++ and other strongly-typed system languages achieve type safety. Ultimately, there is no free lunch and achieving both memory-safety and maximum performance depends on human diligence. But through more powerful language constructs, many of them pioneered or popularized by C++, we can make dramatic improvements in the scope of guarantees that can be delivered with the assistance of our mechanical companions.

For a full treatment of the challenges associated with writing correct memory-unsafe code in Rust, see *The Rustonomicon* and *Rust's Unsafe Code Guidelines Reference*. For an in-depth, practical example of how these techniques are brought to bear to maximize performance *and* memory-safety, see “Implementing Vec”.

Acknowledgements

Special thanks to Barry Revzin, Oliver Hunt, Ian Petersen, and David Sankel for reviewing this draft, and Jonathan Müller (“A principled approach to safety profiles”), Peter Bindels (“Making Safe C++ Happen”, “Subsetting”), Joshua Berne (“Implicit contract assertions”) for feedback on the previous revision and the initial inspiration. Finally, this work would not have been possible without the support of the Rust community, the Rust Project and especially Mara Bos and Ralf Jung. The value of expertise increases exponentially when paired with a desire to share knowledge. Their high-quality writing about the subtle and complex topics of memory safety and patiently answering questions in an accessible manner have been the foundation of this work.

[35] One effort underway to improve the ergonomics of implementing sound interfaces is unsafe fields
[36] See "Working with Unsafe" from the *The Rustonomicon*

Revision History

P3874 R0 was titled “Safety Strategy Requirements for C++”. It was longer and more focused on motivations for this direction. That revision was presented to SG23 during the November, 2025 meeting in Kona and resulted in the following polls:

SG23 encourages the creation of a design in line with P3874R1:

SF	***F***	***N***	***A***	***SA***
13	5	5	2	0

Result: consensus for

SG23 agrees that P3874R1 should be forwarded to EWG:

SF	***F***	***N***	***A***	***SA***
10	4	8	2	1

Result: consensus for

SG23 encourages further work on P3874 with the goal of forwarding the paper to EWG to be incorporated as part of a single Standing Document:

SF	***F***	***N***	***A***	***SA***
18	4	2	0	1

Result: consensus for