C++26 Needs Contract Checking

{Safety, Security, Performance} – *All Three, Required Now*

Christian Eltzschig <christian@ekxide.io> ekxide IO GmbH
Mathias Kraus <mathias@ekxide.io> ekxide IO GmbH
Ryan McDougall <ryanm@applied.co> Applied Intuition
Pez Zarifian <pez@applied.co> Applied Intuition

Abstract

Contract Checking is the single most important way to address C++ memory safety from C++ in a simple, easy, and backward compatible way. The opportunity cost of not having Contract Checking in C++26 is too high.

Background

C++ is the domain language for a large array of important industries. Some safety critical industries are represented in more detail in P2026. Historically C++ has been known to be fast and efficient, but like C before it – not necessarily “computationally safe”. By “computationally safe” we mean the union of all features that make it hard to make incorrect programs, such as type-safety, memory-safety, thread-safety, lifetime-safety, bounds-safety, etc. This is in contrast to “physical safety”, by which we mean all the things that keep Life and Property from Harm in the physical world.

Some of the harms that can happen in the physical world come from incorrect software, and some of those harms are serious: loss of billions, death of individuals, or – increasingly – vulnerability of state infrastructure to hostile (even state) actors. This is why the Office of the President of the United States of America has urged projects to avoid C++. While it is fun to speculate about the C++ prowess of the President – and take note that this advice is in no way binding – it is also clear that this unprecedented intervention is driven by real world safety concerns, and many industry partners are starting to ask about the safety and security of the products the C++ community are offering.

While Rust is a fine language, many of such companies have large investments in existing C++ code bases, and moving to Rust in order to gain better memory and thread-safety is not feasible. It involves significant effort and will necessarily result in transition phases which will inject their fair share of risks as well. Safety is better served with micro-evolution from a well understood and proven field. Moving to Rust is a significant change in terms of technology management and therefore, represents a risk.

While the end result might be more memory safe by being in Rust, there are significant financial costs – including developer training, hiring, tooling, security training etc. More importantly however, it does *not* account for the **opportunity-cost** of taking the same amount of time to make the C++ computationally safer.

The C++ community wants the latter – *more tools to make C++ computationally more safe* – ***now***. We want to be investing this time in making C++ better, not migrating it to Rust.

What makes software “Safe”?

Safety – like performance, security, or usability – is a **system property**, i.e., it cannot be measured as merely the sum of its parts. A defect in a single link in a chain of responsibility has the potential to ruin the entire system. For example, a single defect in a Rust `unsafe` block could cause a memory safe code to fail in a cascade. That is to say – *computational safety* is ***necessary but not sufficient*** for assuring *physical safety*.

Physically safety-critical software is made by adding layer after layer of redundant checks and fallbacks, then testing the system in trial after trial in a representative environment – until we’re able to make a statistical argument that harm is unlikely to happen. This is discussed further in P1517 – but simply put we:

1. Think really hard about everything that could cause harm, write it down in a big table, and make sure that by the end we’ve got some kind of answer for each one of those items.
2. Think really hard about every best practice we know of, and make sure it’s in force – even if it sometimes feels a bit annoying.
3. Write down – in detail – every responsible party for every single component. This is accountability.
4. Write up the set of assumptions that each component will have – that’s its “contract”.
5. Write up all the ways we can message the software that something has gone wrong – those are “errors”.
6. Write up all the ways the software can message us that something’s gone wrong – that is “health”.
7. Have the code fix, report, fail over, and fail safe for each of those things.
8. Have and regularly run tests at *every* layer of integration:
    a. Unit
    b. Component
    c. System
    d. Deployment
    e. Acceptance
    f. etc…
9. Try to run tests that cover even highly unlikely scenarios – stuff that’s hard to dream up. This is where fuzz testing, simulation, and real-world proving are used.

10. Think really hard about how much of the end result is covered by one of these layers, and then make a statistical argument that harm is therefore unlikely.
11. Get a third party to agree with your reasoning – often a state based regulator.

Notice that the majority of software engineering with Rust or C++ only really happens in step 7 – so even if we had the same memory-safety tools as Rust, and thus eliminated all memory safety problems – we would still need to do most of **the same** redundant checking and testing.

Redundancy is important because *mistakes happen by accident* – we need to reduce the odds that any one mistake can slip past all possible checks. We can tell the installers of airplane panels to be careful about fastening all the bolts, but if someone isn't double (or more) checking that all bolts are fastened, then one day an unlikely event will happen, and a Boeing 737 MAX 9 panel will fall off mid flight.

Why do we need Contract Checking?

Run-time contract checking is redundant checking that ensures that design time contracts are in force at run time. Good software engineers are already encoding compile-time contracts in the type system – which the compiler has already evaluated and "proven" – so the potential holes in our contract enforcement exist at run-time.

Contract Checking in C++26 is the single most impactful way a C++ programmer can use the language to contribute to program correctness. Regular unit tests can test that an *expected* thing happens, but they have a harder time with **unexpected** things. Contract checking is a natural complement to unit testing, because the conditions for incorrectness are explicitly built into the code.

On the surface contract checking does not address the main source of bugs in C++ – (lack of) memory safety. However, with a bit of effort, contract checking is huge improvement in memory safety too, because you can now declare bounds-safety in your contract, e.g.

```
void f(const auto& container, size_t index)
     pre(index < container.size());
```

Type-safety could also be used here, but that greatly complicates the type system through combinatorial explosion of type interactions, which may make code harder to reason about. For example, we can wrap the bounds-unsafe code in a bounds-safe container:

```
void f(const CheckedIndexContainer<T>& container);
f(CheckedIndexContainer{container, index});
```

The index still must be checked at run-time, so it's just moving the contract requirements from the function interface to a per-contract type in another file. When passing data between different

functions, the programmer will have to request an explicit type conversion, which would incur both syntax and runtime overhead. C++26 Contract Checking syntax is a simple and expressive way to quickly and easily add checking to C++. Ease and flexibility matter at scale.

Many of the systems we use today are physically safe because they are effectively minor variations on systems that have already been mass deployed before. For a 2024 model automobile, the crash test results for the physical chassis are rarely divergent from the 2023 model's. The same conservatism applies to software – re-using proven code is preferred to major rewrites. As consumers demand faster innovation, that conservatism will no longer be protective – it must be the software that works to ensure software is safe. While macro, assert, or exception based contract checking is in wide use by many firms, to keep pace with expectations, our tools need to be *better*.

Based on a simple analysis of several safety-critical code bases on the order of ~10MLoC, there are >50KLoC of macro-based contract checks. That's over 50,000 points where a flow of control – even an extremely unlikely one – could otherwise lead to serious consequences. If this results in a security breach on a power plant for example, that one "hole" is all that's needed to affect a city.

Why do we need Contract Checking Now?

Because the opportunity cost of waiting any longer – *for basic functionality* – is too high.

In order to both address the industry concerns about the viability of C++, and to measurably improve the safety and security of the C++ programs today, programmers should be improving their checking and testing *without delay*. If we cannot offer even the most basic functionality until 2029, many rational firms will decide to use that time to **migrate off of C++**.

What's more, there's **no excuse** for not shipping Contract Checking in C++26. While the proposal in P2900 could always use improvement and refinement – the smallest feature subset that will both displace previous contract checking mechanisms, and make it easier to quickly add more checking is:

1. Function-appertaining Syntax that
    a. Declares its Preconditions.
    b. Can be Parsed by Static Analysis.
2. Injects run-time Checks that
    a. Run regular C++ code.
    b. Inhibits "Time-Travel" Optimization within checks.

This is based on over a decade of experience working directly on safety critical software – writing tens of thousands of individual contract checks – and plotting internal defect metric dashboards. The majority of contract checks, by a large margin, are of the following kinds:

1. That a resource (like a pointer) is not invalid (eg. null).
2. That a value (such as an index) is within expected bounds.
3. That an “unreachable” state has not been entered.

These kinds of checks are *universal* to all languages at all times – and even with good error handling and type-safe best practice – there will be **thousands** of them. If every postcondition is the precondition for another function, then all invalid flows can be captured by preconditions.

The code that is required to meet these 3 cases are simple, but does call into arbitrary `size`, `begin`, or `end` (and equivalent) of methods of container-like objects. The safety and correctness of such checks is not a significant source of defects – as long as the checks are not optimized away, then checks can simply and reliably consist of “regular C++ code”.

There have often been comments to the effect of “if Contract Checking doesn’t have `<feature>` then it’s pointless/useless/not viable”. While the statements may be well intended, they are objectively wrong according to established industry practice. While its usefulness might be greatly diminished for many users, it is still **broadly useful** to most programmers, who just want to check that their most basic contract assumptions hold. Extensions to this basic functionality can be added over time.

There has also been some concern specifically with P2900 in specific that it defers too much to implementation defined behavior. This is by design, and is not a concern for safety critical users. Such users are relying *explicitly* and directly on specific compiler versions during all their testing, and deploying with a different compiler would represent deploying an untested product. Having more standardized behavior between implementations is certainly an improvement when porting to a new platform, but it is assumed this is something that can be done incrementally.

Why do we not have Contract Checking yet?

A big reason is the industry and culture of writing high quality software has not fully matured into a shared perspective to the point where all the design decisions are “obvious”. There are a few firms that are at the leading edge of safety and security – but the reality is most present day software doesn’t pose much of a threat of harm if it fails. Websites can be reloaded. Mobile apps can be replaced. Both run in a “sandbox”. Most programmers have come to understand “software quality” from their own perspective, and the imperatives for improving quality aren’t universal.

But a less flattering reason is that ISO, WG21, and SG21 (to their own degrees) are not fully representative of the **actual end users** of C++ – the ones writing code today. These groups are volunteer based – and often the people busiest writing the code that puts rubber to the road are the ones *least available to attend committee meetings*. The end result is that conversations in meetings tend to be dominated by a minority of voices, sometimes those who have learned to raise the temperature in order to discourage participation. Some of the loudest voices come

from finance or “big tech”, and – while those are important industries – they **do not** represent the best interests of automotive, aerospace, or robotics companies (to name a few). Disproportionate representation of actual C++ users is a miscarriage of our professional duty to do what’s right by the community and industry at large.

Who Watches the Watcher?

What happens if contract checking logic itself contains errors or undefined behavior? This is the current state of the industry, which all use macro-based checks:

```
void f(const auto& container, size_t index) {
     CHECK_PRECONDITION(index < container.size());
```

Which might be fairly uniformly implemented as equivalent to:

```
#define CHECK_PRECONDITION(expr) \
     if (!expr) std::abort();
```

Let’s consider the insightful example given in Mr. Dos Reis’s excellent P3285:

```
int f(int a) { return a + 100; }
int g(int a) pre(f(a) > a) {
     int r = a - f(a);
     return 2 * r;
}
```

Because signed integer overflow is Undefined Behavior, GCC (with a trial implementation of contract checking and optimization on O3) optimizes that code into:

```
int g(int a) {
     return 200;
}
```

Which defeats the precondition check entirely – but is exactly what happens to our macro version as well. With a primitive implementation of contract checking we are *no worse off* than we are today.

In practice this is **not a fundamental problem** for safety critical software – because “safety” (as discussed in the previous section) is a *system property*. A system is only “safe” when it is tested as an integrated whole – layer after layer. If the system corresponding `g` failed at the *unit* level for the input `INT_MAX - 90`, succeeded at all other layers of system validation, but then failed

under use – then the blame rests more concretely on system validation – than failure of WG21 make contract checking harder to get wrong.

It would be *better* if there was a practical way to remove UB from all contract checking logic – but the authors of this paper believe that the **opportunity cost** of waiting for such a specification far outweighs the real world benefits.

What would a truly minimal Contract Checking facility look like?

P2900 represents the current best consensus of SG21 – and along with P1494 – represents a truly acceptable and useful Minimum Viable Product based on years of industry experience. It is this paper's position that both P2900 and P1494 should be accepted by EWG.

However, if that should somehow become impossible, it would be better for C++ as a whole to have a “more minimal” solution in C++26, rather than a “better” solution in C++29 or later. The opportunity cost is too high.

For argument sake, a sketch of such a “more minimal” solution might look like:

- P1494 or wording to the same effect that would create a partial optimization barrier so that contract checks are not elided.
- The following sections of P2900
    - Syntax: 3.2.
    - Restrictions: 3.3.
    - Name Lookup and Access Control: 3.4.1
    - Postconditions: Referring to the Result Objet: 3.4.3
    - Point of Evaluation: 3.5.4
- Other considerations being considered Implementation Defined.

While leaving so much to the implementation would compromise portability of many useful corner cases – most notably lacking an official set of semantics and an official violation handler – it does handle the majority use case for safety critical software: runtime checking for the purpose of verifying memory safety.

Also note that this section is a brief sketch for illustrative purposes. If EWG is unable to pass P2900 and P1494, then a full paper detailing this sketch would be required.

Calls to Action

To the Committee At Large

1. Take an interest in Safety and Security. Show up to SG21 and SG23. Ask questions.
2. Think about users who may not be the ones you're most familiar with.
3. Think about the quality of conversations you're having. Don't let loudest voices dominate.

To SG21

1. Too many of our users are currently underserved. Let's deliver practical results for users. Any single company – for example a "financial" company, or a "big tech" company – do not wholly represent our users.

To SG23

1. Take a strong stance on what safety means for C++.
2. Help define for EWG what it can do to support safety in C++. Set the goalposts.

To EWG

1. Consider unmet safety and security needs to be an **existential** problem for C++.
2. Prioritize delivering a **minimum viable feature** that *at least*
    a. Function-appertaining Syntax that
        i. Declares its Preconditions.
        ii. Can be Parsed by Static Analysis.
    b. Injects run-time Checks that
        i. Run regular C++ code.
        ii. Inhibits "Time-Travel" Optimization within checks.
3. Understand the opportunity cost of pursuing the ideal feature. Code is being written, run, and scrutinized in the years without a simple contract checking facility.