Doc. no.: P2759R1
Date: 2023-01-22
Programming Language C++
Audience: All WG21
Reply to: Bjarne Stroustrup (bjarne@stroustrup.com)

DG Opinion on Safety for ISO C++

H. Hinnant, R. Orr, B. Stroustrup, D. Vandevoorde, M. Wong

Revision History

- R0: Jan 15 2023 (Pre-Issaquah):
  - initial paper
- R1: Jan 22, 2023
  - modifications to initial paper based on early feedback

Table of Content

1 Abstract **2**

2 Current State **2**

Prior Art 2

Why is Safety on the map now? 3

SG23 4

C++ Image 5

Safety 6

3 Basic Tenets **6**

4 The Process **7**

5 Towards A Safer Future **7**

Backwards Compatibility 8

Profiles 8

6 Call To Action **9**

7 Acknowledgements **9**

8 References **9**

1 Abstract

This paper describes the opinion of the DG on the matter of Safety for C++, starting post-Kona 2022. As this is an evolving area, we anticipate there will be continued refinement of this opinion and, as such, DG has unanimously agreed to work on a paper to guide that opinion. We do not aim to define a specific solution. We do aim to outline the structure needed to evolve such a solution, one that will work for C++ in all domains in a coherent manner. We aim to define a process and offer opinion on the following

- basic tenets
- by-default vs. opt-in safety
- safety as part of tooling vs. safety built-into the language (and thus the compiler)
- backwards compatibility
- a notion of “profiles” for safety, but also for other domains, such as performance

The goal of this paper is to motivate, guide, provide perspective and streamline discussion on this very important matter to C++. Above all, we aim to offer a balanced perspective that has served C++ well. This discussion of safety is important and necessary, but we must not lose sight of other domains that have been and remain key components of C++. On the flip side, C++ must continue to evolve to keep its relevance. C++ will remain a *general-purpose* programming language for all close-to-the-metal domains.

2 Current State

We summarize the current state of C++ with regards to C++ safety as of the end of 2022 with the goal of setting a direction.

Many have noted that safety-critical applications are proliferating, increasing more than ever that programming languages “be safe”. This trend seems certain to accelerate now and in the future. Safety has always been paramount in some domains, such as medical, aerospace, avionics, and various other embedded applications. In these areas, C++ has always made great strides because of its flexibility and safety potential. With the proliferation of browsers, edge devices, autonomous vehicles, etc. and recent increased concerns regarding the safety of power infrastructures, transportation networks, OSs, chat/messaging systems, and the like, there are definitely increased demands for more formal constraints with regards to safety.

In reality, critical infrastructures have always required safety but there are now more of them and more demands for their safety properties to be more evident.

Prior Art

There has been much previous notable work in support of safety in C++, mostly outside of the committee, but in recent years also from within WG21. This is evident in the large number of safety guidelines and coding rules developed specifically for C++. While not exhaustive, some of these include:

- MISRA [MISRA], JSF++ [JSF], CERT [CERT], HICPP [HIC], C++CG [C++CG], and AUTOSAR [AUTOSAR]

The broader C++ ecosystem includes a plethora of static analysis tools and new ones — both open-source and commercial — come on the market every year. The following is but a small sample of such tools:

- Clang, GCC, MS tools, Coverity, cppcheck, lint, LDRA tools

In addition to static analysis tools, there are also run-time/dynamic analysis tools — like valgrind — which perform code coverage, memory error detection, fault localization, invariant inference, security analysis, concurrency error detection, program slicing and performance analysis. Most use a form of instrumentation or transformation.

Why is Safety on the map now?

It has always been on the map to some degree, but it has clearly picked up steam in recent years. In early years, the US military supported ADA as the language for military software. Ada didn't take over the world, but is still widely used and is a very nice language for its domains. Applications such as embedded, automotive, avionics, medical, and nuclear were obvious applications that require safety if programmed in C++. So along the way, there were safety guidelines developed for most of these. The Internet explosion brought in browsers which were increasingly targets of hacking as more commercial transactions occurred through these browsers. Rust, originally from Mozilla, built on top of C++ became the poster child of a safe browser language. Increasingly we have seen Rust's safety claims tested in more applications beyond browsers, e.g. drivers and Linux kernel [rust1]. As cars become autonomous vehicles (essentially software systems on wheels) they drive the various increasingly complex applications of Advanced Driving-Assistance Systems (ADAS), pedestrian prediction, route planning, and machine learning. This has motivated the United Nations to put forward the United Nations Economic Commission for Europe (UNECE) WP.29 directive on Automotive Safety [UNECE] which in turn has started a number of ISO committees on safety of autonomous driving, such as TC22/SC32 [SC32], and UL4600 [UL]. The use of C++ in these and other applications have really put safety on the map, even though it was already there in some form.

More recently, two developments involving US government publications — from the NIST [NIST] and NSA [NSA] — advising safety-sensitive applications not to use C/C++ appear to have ignited a widespread discussion of safety within C++. Both NIST and NSA suggest using alternative languages.

But even before these events, an increasing number of people within the C++ community started working in the autonomous vehicle business and alerted us about the need for more safety in C++ (at least since 2016). In response, WG21 started a UB (undefined behavior) study group in SG12, increased collaboration with safety groups (such as WG23, MISRA, and AUTOSAR), and added a passive study group in SSSG with the explicit direction to comment on any proposal from a safety and security perspective.

Within DG discussions, we have been looking at these safety issues throughout meetings in 2021–2022, with Rust, and the Google comparison paper on emulating borrow checking in C++ [Borrowed]. We urged the creation of an SSRG in our P2000R3 paper of late 2021 in section 7.2.3 [P2000r3]. Some of our members have been involved with safety for over a decade (e.g. JSF++, C++CG, MISRA, AUTOSAR, ISO safety groups TC22/SC32, SC42, UL4600, SAE).

Government endorsement or declaration does not guarantee success or failure. ADA is a good example of that. The NIST and NSA declarations could lead to several possible outcomes:

- non-government entities might ignore government directive and/or,
- government directives might lock C++ out of certain markets, and indirectly lead to a push away from C++.

Nobody knows which way this will go, and there could be other outcomes we have not anticipated. The rest of this paper will argue to stay calm and true to ourselves, and against a reactive jump to one of various poorly-understood bandwagons.

SG23

In 2022 the committee saw:

- notable C++ reflector threads started by Gabriel Dos Reis regarding the perception of C++ safety (see [2022/6/27] and [2022/11/10]),
- a “Future of C++” evening discussion at the Kona meeting in November, and
- another reflector thread started by Bjarne Stroustrup after the Kona meeting (see [2022/11/28]).

This led to the creation of a new study group on the topic — SG23 chaired by Roger Orr.

There were several significant points made during these discussions. We (DG) have tried our best to read as much as we can, but can not claim we have not missed some posts. We note that

- There were a large variety of opinions, with no suggestion for a uniform approach/framework to tackle the underlying issues, and
- As a group, we are quick to call out immediate solutions to real and imaginary problems to address various issues such as UB, ambiguous behavior, overflow, GC, concurrency safety, race conditions, signed and unsigned integer issues, etc.

Many of us are engineers, and only a few of us are also safety experts. Even within the safety industry, the needs are constantly changing as we watch how ADAS has changed in recent years, as we understand more about how difficult it is to make machine learning algorithms safe, etc. While we have obtained some cooperation from the safety industry, we need to be cognizant of our limitations.

Safety is an overall system property involving multiple layers in the stack, from the top-most application to the bottom-most hardware. Each layer must have safety built-in and overlapping with the other layers. As such, any one particular safety concept, e.g. type safety is not enough to make the overall system safe.

It is our opinion that addressing these as separate features will lead to incoherence in any safety measures we offer in the future. Such incoherence could jeopardize the chances of success of the measures in the marketplace. Indeed it may not be what the safety community really wants as we could be over/under-engineering the solution. Any safety package also needs to be flexible and adaptable to the various safety critical industries. We might build things, but if it’s not what the safety community wants, or the safety community has evolved/changed to something different, then the work would be for nought.

C++ Image

C++ appears, at least in the public eye, less competitive than other languages in regards to safety. This seems true especially when compared to languages that advertise themselves more aggressively/actively/brazenly/competently than C++. In some ways, they appear especially to satisfy an executive-suite definition of safety, which makes it attractive for executives to ask for a switch from C++.

Further, the US agencies lumping together C and C++ is likely conflating two languages' properties.

Yet what has been lost in the noise is that C++ has made great strides in recent years in matters of resource and memory safety [P2687]. C++ benefits from having a formal specification and an active community of users and implementers. In contrast, some languages regarded as safe may lack a formal specification, which introduces its own safety concerns (e.g., how to ensure a consistent semantic view of code). These important properties for safety are ignored because we are less about advertising. C++ is also time-tested and battle tested in millions of lines of code, over nearly half a century. Other languages are not. Vulnerabilities are found with any programming language, but it takes time to discover them. One reason new languages and their implementations have fewer vulnerabilities is that they have not been through the test of time in as many application areas. Even Rust, despite its memory and concurrency safety, has experienced vulnerabilities (see, e.g., [Rust2], [Rust3], and [Rust4]) and no doubt more will be exposed in general use over time.

The important properties of stability, formal specification, applicability in many domains, and ISO process are often ignored because they are not as visibly demonstrable in the language or the compiler (e.g., in the form of safe-by-default semantics or compile-time support for safety).

So what do we do about the image of C++?

Should we combat that public image or is time better used to focus on what we do best, to develop a great language for all domains. Some would say we should copy Rust, or some other memory-safe C++ variant. We are strongly against narrowly copying any "safe" language approach. They were designed for their domain and work well there. We are a general purpose language with many application domains, some of which, like "high performance computing" do not necessarily benefit from safety measures in the same way (though admittedly even that could change as HPC moves to the cloud).

Recently, several projects have been started within the C++ community to attempt to solve some C++ issues, in particular Carbon, Cpp2, and Val. All of these are experiments in *very* early development stages whose viability in commercial/industrial settings is an open question. That doesn't mean that C++ cannot borrow ideas from these projects, but it seems inadvisable to bet the farm on any of them at this time.

C++ has reached a certain size and complexity. People are naturally tempted to create new language variants to satisfy their own special needs rather than deal with that complexity to deliver the desired functionality within C++. All new languages start small and simple, but if they succeed they inevitably grow significantly (see, e.g., Java, Python, and C++).

So how do we succeed? We strongly believe that we succeed by doing what we think is best for the language, not simply by copying some other languages. We succeed by learning from others and drive a process that facilitates this evolution of C++ towards better safety support. We succeed by not ignoring other domains. C++ as a high-performance general-purpose language is what made it successful. There

might come a time when C++ will pass on its torch to another great language, but none of the current contenders are such. We should never abandon the millions of lines of existing code, some of which do not cry out for particular safety measures. We should recognize the urgency to support safety in C++ is one of the issues of our time. The question is how to do that, and how much safety to support?

Safety

We are not strangers to safety, with some DG members having written JSF++[JSF], and participate in some of the key safety ISO and UL groups regarding automotive safety (TC22/SC32 ISO 26262, 21448, PAS8800, SAE, UL4600 and AUTOSAR), machine learning safety (SC42 TR5469), MISRA, and WG23, as well as C++ Core Guidelines for over a decade. What we have learned is that safety changes and evolves over time as we learn more and as the industry changes. We believe we should not force safety on everyone, especially those who don't need or want it. Safety should not be static, but evolving, as we learn more, and are informed more by outside safety experts as to what they really need. It is different for different domains which also evolve and change at different rates. For example, aerospace safety is different from medical safety.

Safety in language is also only one component but is one part of the entire programming stack with each part of the stack doing its part

- in the middle of the stack, the programming language itself
- in the upper layers of the stack are modeling languages, frameworks, template libraries, etc.
- in the lower layers are the intermediate languages, drivers, and, ultimately, hardware

3 Basic Tenets

It is best to establish some basic tenets. We are not precluding a lot of possible solutions, but it is useful to have a few basic rules that delimit areas we do not want to end up in.

- Do not radically break backwards compatibility – compatibility is a key feature and strength of C++ compared to more modern and more fashionable languages.
- Do not deliver safety at the cost of an inability to express the abstractions that are currently at the core of C++ strengths.
- Do not leave us with a "safe" subset of C that eliminates C++'s productivity advantages.
- Do not deliver a purely run-time model that imposes overheads that eliminate C++'s strengths in the area of performance.
- Do not imply that there is exactly one form of "safety" that must be adopted by all.
- Do not promise to deliver complete guaranteed type-and-resource safety for all uses.
- Do offer paths to gradual and partial adoption – opening paths to improving the billions of lines of existing C++ code.
- Do not imply a freeze on further development of C++ in other directions.
- Do not imply that equivalent-looking code written in different environments will have different semantics (exception: some environments may give some code "undefined behavior" while others give it (a single!) defined behavior).

Our proposed process — which will be expanded in the remainder of this paper — centers on the following:

- Aim at a framework which is noticeable publicly.
- Place safety changes within that framework.
- Agree on where we make those changes (tools, language, library).
- Agree on a backward compatibility direction.
- Prioritize the most important items to focus our attention on.

4 The Process

We see multiple approaches to this process which may or may not be complementary. We would like to see them work together, but cannot enforce that. Historically, we have been very good at creating an SG, but usually only after a few solutions have been independently generated and can benefit from the procedural iteration of an SG process. SGs are good at iterating through a specific proposal, but not great at brainstorming different potential solutions. Recently SG23 was created but it could become bogged down in technical details, philosophical differences, or dialect creation.

SG23 needs a charter; we are hoping this document might be a good starting point for the discussion thereof.

Another approach is to have small groups work outside of an SG and periodically present work to the SG. This can work faster than an SG, but will need someone's will and leadership.

Ultimately whatever the direction, there needs to be a coherent approach, but we will likely have to unify it for the language, library, and tools aspects. It still remains a question as to whether we need to support every nagging concern regarding safety composition. Next we will try to tackle this topic of safety composition.

5 Towards A Safer Future

The process of whole-group discussion on the reflector and at Kona has informed the solution space based on member opinions. After reviewing almost all the discussion, we will make a recommendation as to how to move forward with safety in C++ under the process we suggested above.

We now support the idea that the changes for safety need to be not just in tooling, but visible in the language/compiler and library. We believe it should be visible such that the "safe code" section can be named (possibly using *profiles* — see below), and can mix with traditional code. Individual features may not be very visible, but will be more visible when packaged. This is important for the following reasons

- Intention: This will make it possible for developers and tools to determine intent.
- Visibility: This makes it detectable, most importantly at compile time, but also at run time.
- Composition: This will create composition of profiles, through imports, includes, library calls, and binary inclusions.

While we continue to favor doing more of the early experience with safety concepts in tooling as we have done for decades, we now clearly support safety features in language and library, but packaging

several features into *profiles*. We also support pushing/encouraging tools that enable more global analysis to pinpoint safety concerns that are hard for humans to identify.

Backwards Compatibility

We continue to feel strongly in favor of backwards compatibility of safe code with conventional code. If some change requires breaking this backwards compatibility, then that needs to be discussed within the whole WG21 community with the input of safety experts.

Profiles

To support more than one notion of "safety", we need to be able to name those notions. We call "profile" a collection of restrictions, requirements, and related semantics that define a safety property to be enforced. A typical profile will not be a simple subset of C++ language features. For example, a range-safety profile cannot simply ban the current unchecked subscripting, but needs to provide a run-time checked alternative for many cases.

Initial work on the idea of profiles can be found in the Core Guidelines (CG, see [C++CG]) and in Section 7 in a recent paper by Stroustrup and Dos Reis [P2687]. Profiles package up several features to make it visible for a code region. Profiles do not limit code in such a way that it reduces the language expressivity like subsets do. We do recognize some domains can deal with subsets and are thus not opposed to a profile-specific subset. However, it is our opinion that subsetting is not a suitable solution for a general purpose language.

Profiles are needed to enforce semantically coherent sets of rules, rather than having individual developers select from a large set of restrictions on individual language features, library facilities, and coding rules.

We like to think profiles do not fragment the ecosystem but increase diversity. Fragmentation occurs when we solve the same problem in different ways. But diversity provides different ways to solve different problems.

We envision that various profiles can appear in source code and automatically trigger analysis. We do not restrict profiles to just address safety concerns, but that they could also support performance concerns, embedded constraints, etc. These cross-cutting aspects can cover different domains: automotive, aerospace, avionics, nuclear, medicine, and so forth. For example we might even have safety profiles for safe-embedded, safe-automotive, safe-medical, performance-games, performance-HPC, and EU-government-regulation.

The set of profiles is open, and only a few would be standardized by WG21 with the rest contributed by industry.

Once we have profiles, we will need to define rules for composition or overriding of different profiles. This is a known difficult problem, resembling the problems involved in passing information between different programming languages. We invite proposals to cover these. It is important to define

- How do different profiles within the same code, or across TU work together?
- How do called libraries of different profiles work together with different profiles?
- How do imported modules, binaries with different profiles work together?

Profiles impose restrictions on use where they are activated. They do not change the semantics of a valid program (except to turn UB into a specific well-defined behavior or vice versa). In particular, a piece of code means the same in every profile (or no profiles). This property is the crucial difference between dialects and our approach.

6 Call To Action

Let's assume WG21 agrees on this as a framework for moving forward or uses something like this as a starting point for a charter for SG23. We'd then need volunteers to write papers to propose a mechanism for following this framework. These papers would (among others):

- resolve outstanding questions,
- adjust our priorities,
- maintain or adjust our directions,
- provide updates from industry experts,
- detail exposition on the profile mechanics, and
- determine rules of composition of profiles.

7 Acknowledgements

Many of the arguments and points of view have a long history in the committee. Thanks to all who contributed. In particular, thanks to the authors of the documents we reference here.

8 References

- [C++CG] https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#pro-profiles
- [MISRA]https://www.misra.org.uk/misra-c-plus-plus/
- [HIC] https://www.perforce.com/resources/qac/high-integrity-c-coding-standard/concurrency
- [CERT] https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046460
- [JSF] https://www.stroustrup.com/JSF-AV-rules.pdf
- [AUTOSAR] https://www.autosar.org/fileadmin/standards/adaptive/18-03/AUTOSAR_RS_CPP14Guidelines.pdf
- [rust1] https://www.linuxfoundation.org/webinars/rust-for-linux-writing-abstractions-and-drivers
- [unece] https://unece.org/transport/vehicle-regulations/world-forum-harmonization-vehicle-regulations-wp29

- [SC32] https://www.iso.org/committee/5383636.html
- [UL] https://users.ece.cmu.edu/~koopman/ul4600/index.html
- [NIST] https://www.nist.gov/itl/executive-order-14028-improving-nations-cybersecurity/recommended-minimum-standards-vendor-or
- [NSA] https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF
- [Borrowed] https://docs.google.com/document/u/1/d/e/2PACX-1vSt2VB1zQAJ6JDMaIA9PlmEgBxz2K5Tx6w2JqJNeYCy0gU4aoubdTxlENSKNSrQ2TXqPWcuwtXe6PlO/pub?urp=gmail_link
- [P200r3] https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2000r3.pdf
- [2022/06/27] [isocpp-news] The pressure/political campaign targeting C++ is upping
- [2022/11/10] [isocpp-news] More News from the Safety Front
- [2022/11/28] [isocpp-lib-ext] Safety concerns
- [P2687] https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2687r0.pdf
- [Rust2]https://blog.g5cybersecurity.com/cve-2020-25792-an-issue-was-discovered-in-the-sized-chunks-crate-through-0-6-2-for-rust/
- [Rust3] https://blog.sonatype.com/this-week-in-malware-may-13th-edition
- [Rust4] https://portswigger.net/daily-swig/rust-patches-sneaky-redos-bug