Doc. no.: P2759R0
Date: 2023-01-15
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 2023 (Pre-Issaquah):
  - initial paper

Table of Content

1 Abstract **1**
2 Current state **2**
3 Basic Tenets **6**
4 The Process **6**
5 Towards a safer future **7**
6 Call to action **8**
7 Acknowledgements **8**
8 References **9**

1 Abstract

This paper describes the opinion of the DG on the matter of Safety of 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 solution. We do aim to define the structure needed to evolve such a solution that will work for C++ in all domains in a coherent manner. We aim to define the process, and offer opinion on the following

- basic tenets
- safety-by-default, or opt-in,
- safety as part of tooling, or built-in to the language, and by extension, the compiler
- backwards compatibility

- 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 provide 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++. But we must evolve. C++ will still be a General purpose programming language for close to all metal domains.

2 Current state

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

Today, many have noted that safety critical applications have proliferated even more than ever, putting more strain on programming languages to be safe. This trend seems certain to accelerate now and in the future. Previously, safety has always been paramount in some domains, such as Embedded , Medical, Aerospace, and avionics. 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 and recent increased concerns on the safety of power infrastructures, transportation networks, OS, chat messaging, there is 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 demand for their safety aspect to be more evident.

Prior Art

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

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

There have also been static checkers. Some of these in include:

- clang, gcc, MS tools, Coverity, cppcheck, lint, LDRA tools

In addition to static analysis tools, there are runtime, dynamic analysis tools which perform code coverage, memory error detection, Fault localization, invariant inference, security analysis, concurrency errors, 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 occur through 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 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 advising the Safety applications not to use C/C++ from the NIST [NIST] and NSA [NSA] seems to have ignited a widespread discussion of safety within C++. Both NIST and NSA seem to suggest using an alternate language.

But even before then, an increasing number of people within C++ have started working in the autonomous vehicle business and have been alerted to the need for safety in C++ since 2016. In response, C++ has started a UB 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 security perspective.

Within DG discussion, we have been looking at these safety issues throughout meetings in 2021- 2022, with RUST, and the google comparison paper on emulating Borrowed Checker 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 declaration could lead to several possible outcomes:

- non-government entities might ignore government directive AND/OR,
- government directive locks C++ out of certain market, and indirectly leads 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 champion that we stay calm and true to ourselves, and this is better than a reactive jump on the bandwagon.

SG23

Several particular C++ reflector threads, one by Gaby Dos Reis on the NIST declaration in early 2022 [2022/6/27][2022/11/10], a Future of C++ evening discussion at the Kona F2F, and another reflector one by Bjarne Stroustrup after the Kona F2F [2022/11/28] resulted in further actions within C++. This led to the creation of SG23 chaired by Roger Orr.

There were several notable points as a result of these discussion in 2022 reflector, among multiple threads from our point of view. As there were many related threads, we have tried our best to read as much we can, but can not claim we have not missed some posts.

- there was a large fragmentation of opinions, with no uniform approach/framework

- as a group, we are quick to call out immediate solutions to real and imaginary problems to address various UB, ambiguous behavior, overflow, GC, concurrency safety, race conditions, integer/ signed and unsigned issues

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. While we have co-opted 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 bottommost HW. 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 package we offer in the future. The result of which may not even be noticed by the public that we have done these changes. 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 safety critical industries. Further, we might build it. But if it's not what the safety community wants, or the safety community has evolved/changed to something different, then would they even use it?

C++ Image

C++ appears, at least in public image, less competitive than other languages in regards to safety. This seems true especially when compared to languages that advertise themselves more heavily/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++.

Yet what has been lost in the noise is that C++ has made great strides in recent years in matters of dangling, resource and memory safety [P2687]. C++ benefits from having a specification, active community of users and implementers. Other "safe" languages may not even have any specification, at least not yet. 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. The unarguable truth that we all know is that vulnerabilities are found in all languages, but it usually takes time. Newer languages have less vulnerabilities because they have not been through the test of time. Today, even RUST has had vulnerabilities discovered recently [Rust2][Rust3][Rust4] and time will expose more vulnerabilities and weaknesses for general use.

The important properties of stability, standard specification, applicability in many domains, and ISO process are ignored as the public seems to demand something that is visibly demonstrable in the language or the compiler, in the form of safe-by-defaults, or compile-time support for safety.

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

We could combat that public image, or is time better used to focus on what we do best and that is to develop a great language for all domains, ignoring what others say. Some would say we should copy RUST, or some other safe C++ variants? We are strongly against copying any "safe" language approach. They were designed for their domain, and work well there. We are a general purpose language with many domains, some of which, like High Performance computing, does not necessarily benefit from safety concerns, though even that could change as HPC moves to the cloud.

There are now also several attempts to address C++ issues, which some would call possible future contenders. The various C++ variants (e.g. circle, carbon, val, cpp2) all seem to serve specific domains, and focus on solving specific problems. They are not successors to C++ in all of its major domains. At best those languages are dialects of C++, or even simply different languages. WG21 is supposed to protect against dialects. If you want a dialect or a different language, that's not the job of WG21.

We see that when C++ has reached a certain size and complexity, people are tempted to create new language variants to satisfy their own special needs rather than deliver it inside C++. But new languages start small and simple, then (if they succeed) grow significantly such as 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++'s 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 greater language, but none of the current contenders are such. We should never abandon the millions of lines of existing code, some of which does not cry out for safety. 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, 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. So we should not make it the same for every industry.

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

- for programming language, in the middle of the stack,
- above are the higher 4g languages, modeling languages, tensor frameworks, template libraries
- below are the intermediate languages, then drivers, HW

3 Basic Tenets

It is best to establish some basic tenets. We are not precluding a lot of possible solutions, but it would be good to have a few basic rules of where we do not want to go.

- 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 inability to express the abstractions that are a pillar of C++ strength.
- do not leave us with a “safe” subset of C that eliminates C++’s productivity advantages.
- must not are purely run-time and thus impose overheads that eliminates C++’s strengths in the area of performance.
- should not imply that there is exactly one form of “safety” that must be adopted by all.
- do not hold the promise of delivering complete guaranteed type-and-resource safety.
- do not offer paths to gradual and partial adoption – opening paths to improving the billions of lines of existing C++ code.
- must not imply a freeze on further development of C++ in other directions.
- must not imply that code written in different environments with the same types have different semantics.

Our proposed solution is as follows, and will be expanded in the remainder of the paper:

- aim at a framework which is noticeable publicly
- place safety changes within that framework
- agree on where we make those changes (tools, language),
- agree on 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 complimentary. We would like to see them work together, but can not enforce it. Historically, we have been very good at creating an SG, but usually only after a few solutions have been generated and can benefit from the procedural iteration of an SG process. SG are good at iterating through a specific feature, but not great at brainstorming solutions. Recently SG23 was created but it could become bogged down in technical details, philosophical differences, or dialect creation.

SG23 needs a charter, and this document could be a good starting point for discussion.

Another approach aiming for a more coherent approach is to have small groups work together outside of an SG, periodically going 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

This 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), and can mix with normal code. Individual features may not be very visible, but will be more visible when packaged. It seems this will also address the image of C++ as having been augmented with Safety packages that can trigger compile time, or runtime analysis. 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 runtime
- 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 Tooling to enable more global analysis in identifying hard for humans to identify safety concerns.

<u>Backwards Compatibility</u>

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.

<u>Profiles</u>

To support more than one notion of “safety”, we need to be able to name them. We call a collection of restrictions and requirements that defines a property to be enforced, a “profile.” 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 CG [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 are OK with subsets and are not explicitly forbidding it. It is our opinion that it is just not a generally good 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, 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 see various profiles can appear in source code that potentially automatically trigger analysis. We do not restrict profiles just for safety, but could also support performance, embedded, or other aspects. These cross-cutting aspects can cover different domains: automotive, aerospace, avionics, nuclear, medicine.

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.

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. 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. 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 to use something like this as a starting point for a charter for SG23, then we need volunteers to write papers to propose a mechanism for following this framework. These papers would:

- resolve outstanding questions
- adjust our priorities
- maintain or adjust our directions
- update from industry experts
- detail exposition on the profile mechanics
- rules of composition, overriding 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