Doc. no. P2739R0
Date: 2022-12-6
Project: Programming Language C++
Audience: All
Reply to: Bjarne Stroustrup (Bjarne@stroustrup.com)

A call to action: Think seriously about "safety"; *then* do something sensible about it

Bjarne Stroustrup
Columbia University
www.stroustrup.com

Consider this from the NSA: https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF :

> the overarching software community across the private sector, academia, and the U.S. Government have begun initiatives to drive the culture of software development towards utilizing memory safe languages. [3] [4] [5]
>
> …
>
> NSA advises organizations to consider making a strategic shift from programming languages that provide little or no inherent memory protection, such as C/C++, to a memory safe language when possible. Some examples of memory safe languages are C#, Go, Java, Ruby™, and Swift®.

That specifically and explicitly excludes C and C++ as unsafe. As is far too common, it lumps C and C++ into the single category C/C++, ignoring 30+ years of progress. Unfortunately, much C++ use is also stuck in the distant past, ignoring improvements, including ways of dramatically improving safety.

Now, if I considered any of those "safe" languages superior to C++ for the range of uses I care about, I wouldn't consider the fading out of C/C++ as a bad thing, but that's not the case. Also, as described, "safe" is limited to memory safety, leaving out on the order of a dozen other ways that a language could (and will) be used to violate some form of safety and security.

Now, I can't say that I am surprised. After all, I have worked for decades to make it possible to write better, safer, and more efficient C++. In particular, the work on the C++ Core Guidelines specifically aims at delivering statically guaranteed type-safe and resource-safe C++ for people who need that without disrupting code bases that can manage without such strong guarantees or introducing additional tool chains. For example, the Microsoft Visual Studio analyzer and its memory-safety profile deliver much of the CG support today and any good static analyzer (e.g., Clang tidy, that has some CG support) could be made to completely deliver those guarantees at a fraction of the cost of a change to a variety of novel "safe" languages.

Please consider

- Bjarne Stroustrup and Gabriel Dos Reis: Design Alternatives for Type-and-Resource Safe C++. P2687R0. 2022-20-15.
- B. Stroustrup: Type-and-resource safety in modern C++. P2410r0. 2021-07-12.
- B. Stroustrup, H. Sutter, and G. Dos Reis: A brief introduction to C++'s model for type- and resource-safety. Isocpp.org. October 2015. Revised December 2015.
- The C++ Core Guidelines: https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md and especially it's safety profiles https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#pro-profiles

There is not just one definition of "safety", and we can achieve a variety of kinds of safety through a combination of programming styles, support libraries, and enforcement through static analysis. P2410r0 gives a brief summary of the approach. I envision compiler options and code annotations for requesting rules to be enforced. The most obvious would be to request guaranteed full type-and-resource safety. P2687R0 is a start on how the standard can support this, R1 will be more specific. Naturally, comments and suggestions are most welcome.

Not everyone prioritizes "safety" above all else. For example, in application domains where performance is the main concern, the P2687R0 approach lets you apply the safety guarantees only where required and use your favorite tuning techniques where needed. Partial adoption of some of the rules (e.g., rules for range checking and initialization) is likely to be important. Gradual adoption of safety rules and adoption of differing safety rules will be important. If for no other reason than the billions of lines of C++ code will not magically disappear, and even "safe" code (in any language) will have to call traditional C or C++ code or be called by traditional code that does not offer specific safety guarantees.

Ignoring the safety issues would hurt large sections of the C++ community and undermine much of the other work we are doing to improve C++. So would focusing exclusively on safety.

What might "something sensible to do" be? I suggest making a list of issues that could be considered safety issues (including UB) and finding ways of preventing them within the framework of P2687R0. That's what I plan to do.

And anyway, what is "the overarching software community"? To the best of my knowledge, no experts from the ISO C++ standards committee were consulted.