Hello. This is Paul Krill of InfoWorld. I am aware of the budding controversy in which the US National Security Agency advises against C and C++ usage, prompting your short paper to refute this. I was hoping you could answer some questions as a followup:

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2739r0.pdf

https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF

- How is C++ safer than C? As you note, the two often get lumped together.
    - Yes, far too many people talk about the mythical C/C++ language and then often proceed to focus on the weaknesses of the C part. Many of those weaknesses can be avoided in C++; typically, by writing more efficient code that more directly expresses the intent of the programmer.
    - Before going further, let's try to pin down what "safety" might mean. For a start, I aim for type-and-resource safety. That is, every object is used according to its type and no resource is leaked. For C++, that implies some run-time range-checking, eliminating access through dangling pointers, and avoidance of misuses of casts and unions. That's essentially impossible to statically (compile-time) guarantee in C without dropping to a very unproductive low level of code whereas C++ offers high-level, facilities, such as containers, span, range-for loops, and variants that with some additional effort can offer guarantees without damaging efficiency or productivity.
    - Now, there are other safety concerns that can be addressed, such as integer arithmetic overflow and excessive response time, but let's consider security, which is the ultimate aim of many of the safety efforts. Security is a systems property. "Bad guys" will attach/exploit the weakest link they can find. If they can get physical access to your computer, they probably won't have to bother with the type systems of the languages you use. Similarly, if they can get through poor input validation, e.g., use SQL injection, then again, they don't have to bother about programming languages. It is important to remember that a programming language and the styles of code used in it is only a small part of a system. We need a comprehensive and balanced approach to development of quality systems.
    -
- What makes C++ as safe as some of the languages cited, including C#, Go, Java, Ruby™, and Swift?
    - First note that sooner or later every system must use hardware and that effective hardware access is rarely safe. All the languages mentioned, and we can add Rust, either support unsafe code or rely on calls to languages supporting unsafe code. Thus, all

languages are vulnerable through code that is not statically verified. Often the "unsafe code" user by "safe" languages is C or C++ so to do its job and support those safe languages C++ must be able to accept some unsafe/unverified code.

- My strategy for safer C++, potentially delivering completely verified code for some code fragments, rests on three pillars:
  - Static analysis to verify that no unsafe code is executed.
  - Coding rules to simplify the code to make industrial-scale static analysis feasible.
  - Libraries to make such simplified code reasonably easy to write and ensure run-time checks where needed.
- Leave out any of those three and you have a near impossible problem on your hand. People have repeatedly tried that, and failed. Some of the criticisms I have seen on the Web simply fail to mention that we are not trying to do the impossible, say to guarantee type-and-resource safety exclusively through static analysis or by simply subsetting C++.
- This approach has been used with some success in the C++ Core Guidelines for years, should be strengthened and receive some support in the standard. The paper by Gabriel Dos Reis and me from last year give some details (https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2687r0.pdf ). It might worth noting that neither the problem nor my approach to its solution is new. I worked on industrial coding rules, wrote academic papers about this, and worked on static analysis software in the early 2000s. My view is that the software industry in general has been slow to appreciate the magnitude of the problem and the need to address it.
-

- You mentioned C++ use being stuck in the distant past. So developers are not taking advantage of the safety improvements made?
  - Well, many C++ developers take good advantage of newer – C++11, C++17, etc. – features and reap significant benefits, but sadly, many are still stuck in an old-C mindset with undisciplined messes of pointers and macros, or they are stuck in environments that enforces pre-C++11 styles. It is those programmers I worry about, and please remember that I never said *all* C++ programmers.
  -
- What do you see as the key safety improvements made to C++ over the years?
  - The STL containers and algorithms, range-for and span, the resource management pointers (shared_ptr and unique_ptr), RAII, and exceptions. Fundamentally, the classes with constructors and destructors underly every improvement.
  -
- Re: "Now, if I considered any of those "safe" languages superior to C++ for the *range of uses I care about*." What would those uses be?
  - Primarily current uses of C++: aerospace, medical instruments, AI/ML, graphics, audio, animation, communications, finance, games, automotive, bio-medicine, scientific computation, high-energy physics, simulation. The list is endless. There are millions of C++ programmers with billions of lines of code "out there." I – and many others – feel an obligation to help them maintain their flexibility and performance, as well as improving their safety – for appropriate definitions of safety for their domains – and productivity.

- Any other safety improvements planned for C++ in C++ 23 that should especially be noted?
  - Nothing much specifically for C++23. C++23 was always meant to be a relatively small increment on C++20 and Covid got in the way of some of the features that we had hoped for. On the other hand, I hope that this uproar about safety will encourage the

committee and the industry to back some of the safety ideas that I have worked on for years.
    - You can significantly benefit from the C++ Core Guidelines today, but to get guarantees for industrial-scale software, you need support from static analysis. The analysis offered by the Visual Studio analyzer (especially the type, bounds, and lifetime profiles) and Clang tidy are not yet perfect, but they are steadily improving.
- Other comments?
    - Enabling and encouraging a gradual adoption of new features supporting expressiveness, performance, and safety is essential. A “clean break,” e.g., to a new language would imply that billions of lines of C++ would remain essentially unchanged for years or decades. We can do better than that.

Thanks!