Doc. No. P3704R0
Date: 2025-05-19
Audience: EWG, SG21, SG23
Author: Bjarne Stroustrup
Reply to: bjarne@stroustrup.com

What are profiles?

Bjarne Stroustrup (bjarne@stroustrup.com)
Columbia University

1. Introduction

A profile is a set of guarantees, such as type safety, absence of resource leaks, and range errors (§3). Typically, a profile is implemented by banning language features and libraries that could compromise its guarantees plus a set of libraries to ease the writing of code that conforms to the profile.

"Profiles" is a framework for requesting the enforcement of C++ profiles and for suppressing a profile where necessary [GDR'25].

A profile is opt-in. The result of using a profile is ISO standard C++ with no change of meaning of code without undefined behavior compared with the same code used without enforcement.

This paper explains general ideas, rather than being a fully-fleshed-out standards proposal. There are many more details in papers in the reference list.

2. Profiles and guidelines

A profile is an enforced set of guidelines (domain-specific rules). A good guideline is a coherent set of rules aimed at helping to develop good code, where "good" usually involves maintainability, comprehensibility, and performance. However, it is not possible for a group of developers to follow guidelines in a large codebase without help. Consequently, enforcement is needed, preferably in a compiler. That is, a profile.

There can be no profile or set of profiles acceptable to all C++ users. The range of application areas is too large for that. Consequently, the dream of a single profile for all – that is, a single redefined language – is just a dream. There can be no ideal language for everything and everybody.

The Profiles framework is open. That is, some profiles will be standard but users can define and use their own to serve their needs without affecting other users or waiting for the standards committee to decode and vote.

3. Who needs profiles?

In an ideal world, nobody needs profiles because they simply enforce what can be done without them. That is similar to saying that nobody needs high-level languages because everything can be done in assembler. In reality, just about everybody needs – or at least would be helped by – profiles to cope with complexity. One way of looking at a profile is as a domain-specific language embedded in a general-purpose language.

- One of the simplest and potentially most effectful uses of a profile would be to support teaching of programming. Essentially every language has traps and pitfalls that are best avoided by novices and features that are best not introduced early. Most languages in industrial use has more features and libraries than can be learned initially. C++ is arguably a prime example of this, but even early C had "traps and pitfalls" papers and books written about it. My recent books [BS'22, BS'23] follow the C++ Core guidelines except when warning against bad habits. Those guidelines prevent most dangers that people warn against, but guidelines are not consistently enforced. A teaching profile would greatly simplify teaching, thereby resolving the dilemma of whether to you teach programming using a "toy" teaching language or a large language in real-world use.
- These days, there is much demand for guaranteed memory safety. That's reasonable and achievable but shouldn't be confused with absolute safety or security guarantees. As usual with a popular topic, there isn't a single universally definition of "memory safety" but its essence includes two aspects:
  - Run-time prevention of range errors and nullptr dereferences. This is most easily achieved by using checking libraries (hardened libraries [KV'25, CC'24]) and prevention of the use of uncheckable subscripting.
  - Static (compile time) elimination of dereferencing of dangling pointers [BS'1515b, HS'19, BS'24b].
- In many contexts, resource safety; that is, the elimination of leaks is essential to ensure that a program can run for a long time. For most embedded systems, that's essential. Resource safety is not just prevention of memory leaks. Leaking file handles and forgetting to release locks can leave a system frozen or crashed. In addition to preventing leaks, we usually want to minimize reduce resource contention. Holding all resources for twice as long as is necessary (as is not uncommon in many languages and programming styles) implies the need for twice the compute power and twice the energy

consumption. RAII is key to resource safety, but its consistent would benefit from being enforced by a profile.
- From the earliest days, my ultimate aim for C++ has been complete type safety. That is every object is accessed only in according to its definition. To ensure that, we need guaranteed initialization, absence of narrowing conversions, memory safety, resource safety, and more [BS'94].
- Real-time guarantees are essential for many systems. For example, the general free store (new/delete) is often banned in favor of static memory and specialized memory pools. Similarly, there are often limits on stack usage. I have seen exceptions banned where they would have been ideal except for the lack of a way of reliably estimating the time needed to get from a throw to its catch. Profiles can address much of this.

For more potential profiles and more details about them see [BS'94]. The key point here is that we need a variety of profiles to address the variety of needs. I aim for complete type safety, but that is hard to achieve in a language that is widely used for machine-near code, with a long history, and billions of lines of existing code.

In particular, we need a common framework to ensure that the many significant efforts to improve the use of C++ don't degenerate to a mess of incompatible tools [GDR'25].

4. Perspective, experiments, and experience

This paper explains general ideas, rather than being a fully fleshed standards proposal. On the other hand, "Profiles" as presented here and in previous papers [GDR'25, BS'22, BS'24, BS'24b, BS'25] is the result of decades of work and is thus based on significant practical experience.

"Profiles" is the concrete manifestation of decades-old ideas. It is a general notion, rather than just the C++ manifestation proposed for ISO standard C++.

No programming language is perfect for everything and everybody. However,

- A general-purpose language is under constant pressure to expand its area of use in various directions, to be extended.
- To serve collaborating groups with diverse needs, these extensions need a common substrate (shared language base).
- A general-purpose language is under constant pressure to be simplified to approximate the ease of use of special-purpose languages.

Implementers respond to the demands of more features with extensions of the language, compiler, platform specific language extensions, libraries, and tools.

We cannot change the language incompatibly without causing serious trouble for many users. However, we can change the use of the language.

"Profiles" is a conceptual framework for managing the complexity of a programming language with diverse uses. Many people seem to consider the proposals for profiles for ISO standard C++ as nothing but a belated response to Rust's borrow checker. That is flat wrong. Profiles are not just for "safety" but (among other things) also for managing technical debt and educationalists have asked for support to keep students focused on modern styles. The first profiles for the C++ Core Guidelines were deployed in 2014 and my first academic paper outlining some of the fundamental ideas were in 2005 [BS'05, BS'05b]. I gave three CppCon keynotes related to guidelines [BS'15, BS'17, BS'23]. The general model for "safety" in C++ based on static analysis and a few essential run-time checks was outlined in 2015 [BS'15b].

Ada has a profiles framework, ("The Ravenscar Profile") for safety critical applications [Ada'12]. Ideas of a common framework for varying languages go back to at least Landin [PJL'66] and Backus [JB'78] who looked for a common framework for diverse uses. Domain Specific Languages are (rightfully) popular and typically needs to be embedded in a language substrate that provides access to generale-purpose facilities and other special purpose languages. C and the JVM are examples of such substrates. They have been spectacularly successful but lower the level of interoperability far below modern programming languages. The don't offer facilities for enforcing high-level programming styles.

Almost from the start of C, Lint [SCJ'78] was developed to perform checks that the compiler couldn't do because of they were too complex for the compiler, related to separate compilation, or not everybody wanted them. Much was sucked into the C++ type system, but especially the "not everybody wanted them" (or could afford them) has spawned a multitude of tools. Lint separated what made sense from what was legal but unwise. It started the use of static analyzers in the C world.

In the form of coding guidelines support, precursors to profiles have been used for decades. The C++ Core Guidelines has significant support in the Visual Studio analyzer, Clang Tidy, CLion, and probably in more tools. The VS support for the C++ Core Guidelines has for years supported 4 profiles. Gabriel Dos Reis has an experimental version of the Profiles framework itself [GDR'25].

Hardened libraries, that is versions of the C++ standard library with run-time range checking, is widely deployed [KV'25, CC'24].

The problems with the multitude of tool support are

- It is hard to keep track of what is available, and where.
- No significant tool related to guidelines enforcement is universally available.

- Many tools are focused on finding flaw in old-style code, rather than aiming for modernization and serious reduction of technical debt.
- Most tools are incompatible with each others.
- The many overlapping tools leave holes in the checking and fail to offer comprehensive guarantees.
- There is no standard way to request the installation and use of a checker.
- People first trying out C++ don't know of these tools, don't know how to get them installed and invoked, don't use them, and suffer from the lack of their support.

This is a clear case for standardization of at least the Profiles framework and a few key Profiles.

References

- [Ada'12] The Ada Ravenscar Profile .
- [CC'24] C. Caruth: Story-time: C++, bounds checking, performance, and compilers. November 2024.
- [GC] The C++ Core Guidelines .
- [GDR'25] C++ Profiles: The Framework (Revision 1) February 2025.
- [BS'05] B. Stroustrup and Gabriel Dos Reis: Supporting SELL for High-Performance Computing. LCPC05. October 2005.
- [BS'05b] B. Stroustrup: A rationale for semantically enhanced library languages. LCSD05. October 2005.
- [BS'94] B. Stroustrup: The Design and Evolution of C++. Addison-Wesley. 1994.
- [BS'15] B. Stroustrup: Writing Good C++14 CppCon 2015.
- [BS'15b] 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.
- [BS'17] B. Stroustrup: Learning and Teaching Modern C++. CppCon2017.
- [BS'22] B. Stroustrup and G. Dos Reis: Design Alternatives for Type-and-Resource Safe C++. October 2022.
- [BS'23] B. Stroustrup: Delivering Safe C++. CppCon 2023.
- [BS'24] B. Stroustrup: A framework for Profiles development. May 2024.
- [BS'24b] B. Stroustrup: Profile invalidation – eliminating dangling pointers. October 2024.
- [BS'25] B. Stroustrup: Note to the C++ standards committee members. February 2025.
- [HS'19] H. Sutter: Lifetime safety: Preventing common dangling. November 2019.
- [HS'25] H. Sutter: Core safety profiles for C++26. February 2025.
- [JB'78] J. Backus: Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programming. CACM August 1978.

- [KV’25] K. Varlamov and L. Dionne: Standard library hardening. February 2025.
- [PJL’66] P. Landin: The next 700 programming languages March 1966.
- [SCJ’78] Wikipedia: Lint (software)