Doc. No. P3984R0
Date: 2026-02-22
Audience: EWG, SG12, SG20, SG23
Author: Bjarne Stroustrup
Reply to: bjarne@stroustrup.com

A type-safety profile

Bjarne Stroustrup (bjarne@stroustrup.com)
Columbia University

Abstract

First the concept of and rationale for Profiles [BS25b, BS22b, BS23] are summarized and some suggested alternatives are mentioned. Next the type-safety and resource-safety profiles are presented in some detail as an example to illustrate the specification and implementation requirements of two of the most essential and difficult profiles.

This is a description of Profiles in general terms aimed at understanding, not a standards text aimed at standards experts only. However, it is sufficiently detailed to be translated in "standardese" with relatively few technical changes. What is described has been prototyped [HS21, CG, GDR25] and some parts used in the form of guidelines.

Finally, a plan for progress is outlined (§5).

1. Introduction

My initial motivation to develop C++ was to combine the best of C and Simula:

- C's ability to access hardware and manipulate data without overhead.
- Simula's ability to organize programs and guarantee type safety.

To this, I immediately added

- constructors and destructors to provide the ability to eliminate resource leaks and uninitialized objects.
- uniform treatment of built-in types and user-defined types to lay the foundation of generic programming.

C compatibility concerns and the zero overhead principle prevented me (and later the C++ standards committee) from enforcing complete type and resource safety. I (still) consider compatibility/stability and the zero-overhead principle essential for the evolution of C++ and its foundational libraries. Where this is not adhered to, problems emerge.

However, not everybody needs every old feature and not everybody needs ultimate performance. In fact, many (probably most) C++ users don't need all of the constraints and facilities needed to conform to either or both compatibility and performance ideals. Also, they often have other concerns, as do I for various projects. This is the motivation for all coding guidelines and for the (still evolving) Profiles design [BS23, BS24b, GDR25]. In particular, we can design a Profile to guarantee type safety and resource safety (the absence of resource leaks) without imposing inessential overheads.

These guarantees are the ones I want for most application code:

- ***Type safety***: Every object is used exclusively according to its definition.
- ***Resource safety***: No resource is leaked.

We can achieve that by careful use of contemporary C++, but I want a Profile to guarantee it, as do important C++ users. Otherwise, the approach won't scale. This implies that code that needs to violate that guarantee or can't be proven not to must be isolated. Examples of such necessary code include the implementation of key abstractions (e.g., **vector**). It also implies that the guarantee must be opt-in because essentially no existing code consistently meets the requirements needed to provide this guarantee (even correct code because "correct" doesn't mean "provably correct").

The fundamental assumptions behind the Profiles design are

- Some people need guaranteed properties of their code.
- Not everybody has the same needs.
- It is impossible to prove an arbitrary program safe for any reasonably definition of "safe" in a language able to manipulate hardware and implement fundamental abstractions.
- A guarantee must be validated by software, not just by a review by developers.
- A desired guarantee must be explicitly requested in code.

This implies that we need a framework that allows programmers to request a variety of guarantees and prevent the use of features deemed unsafe for the requested guarantees.

The fundamental technique for implementing a profile is to subset a superset of ISO standard C++ [BS05, BS05a]:

- ***Superset***: Extend the C++ to be used with the abstractions needed to conveniently express what needs to be expressed for a given application area.
- ***Subset***: Restrict the set of facilities to be used in application code to what can be used while maintaining the desired guarantee (leaving "unsafe" uses to the implementation of the abstractions).

Supersetting is done by adding libraries and eliminating undefined behavior (supported by run-time checks where necessary). Subsetting is done through static analysis (ideally by the compiler)

Subsetting alone cannot deliver many of the most desired guarantees because the implementation of the abstractions requires use of techniques that cannot be verified (e.g., pointer arithmetic and linked structures).

Supersetting alone can't deliver many of the most desired guarantees because the low-level features would still be available to break them.

By supersetting, we create a language that's easier for analyzers (e.g., compilers) to understand. Use of banned features outside explicitly trusted sections of code is simply considered errors. This solves the problem that large chunks of C++ (especially old-style code) are too complex to analyze efficiently relative to a requested guarantee.

By specifying requirements as enforceable profiles, we avoid having to rely on developers following guidelines and sprinkling annotations all over the code. That approach is labor intensive, error prone, and basically not feasible at scale. It doesn't provide the guarantees required by many user organizations.

Profiles do not interfere with traditional C++ design and implementation techniques (e.g., OO, GP, and functional styles) except when we define a profile to do so and explicitly require that profile. Rather, a good profile eliminates misuses of those techniques.

The result of compiling a TU using profiles is ISO standard C++ that can be linked with other TUs [GDR25].

A profile cannot change the semantics of a program beyond defining the meaning of some forms of undefined behavior. For example, signed arithmetic overflow is UB so a profile can define it to be wraparound like unsigned arithmetic (though I wouldn't do that), to be saturated arithmetic, or to throw an exception. However, wraparound of unsigned arithmetic is well-defined so a profile cannot define to be anything else. This implies that a program without UB will behave identically when compiled with or without a profile enforced. The aim of key profiles is to insert run-time checks to ensure that an error action is triggered rather than reaching the point of UB.

If a profile needs a language change (e.g., the addition of a new standard attribute), then that change must be done to the language rather than just to the profile.

Profiles are not just about "safety" (for a variety of kinds of safety). They address the widespread need for support of a better style of code (for a variety of definitions of "better") from both developers and educators.

Some, but not all, profiles will be part of the standard. Profiles is an open framework so that profiles serving specific needs can be added "locally." One limiting factor here will be the lack of a standard internal representation of C++ (e.g., IPR [GDR11, CDR21], Clang AST [MK13]).

1.1. Alternatives to profiles

Why do we need a framework? After all, people can and do address problems with libraries, static analysis, compiler options, pragmas, etc. That variety is the problem. These features tend to address limited problems, be platform specific, be incompatible, require detailed understanding of compilation and build systems, be poorly specified, be non-portable, require consistent use of coding guidelines, be non-standard, and the like. The variety of genuinely helpful facilities that differ in their style of invocation is a recipe for chaos. For the key facilities that we want to be part of the standard, we need a standard way to specify them and document their use in the code. For all facilities that are not simply sets of libraries, we need a standard way of requesting their use.

Offering long lists of specific tests and restrictions for developers to select from are ever-popular among people who desire maximal freedom of choice. However, having hundreds of thousands of developers choosing from a set of (say) 70 options leads to unreadable code, incompatible code, and sets of options that do not add up to a guarantee that can be understood, named, and remembered. Languages, libraries, and systems have failed, being rejected in favor of simpler alternatives. Part of the aim of profiles is to reduce the set of alternatives to something manageable; to move management of complexity from implementation details to language and library design.

By defining sets of optional libraries, tests, and implementation alternative, rather than defining guarantees as language features, we could not just avoid making potentially erroneous design decisions, we could also avoid problems of evolving guarantees. If we in the future find a profile needs changing (as we surely will), we have exactly the same options as when we find that a language feature or a standard library needs a change:

- Vote in a change, trying to minimize impact on existing code (e.g., making it a pure extension).
- Making a new profile with a new name and potentially deprecating the old one (as we did when replacing **auto_ptr** with **unique_ptr**).

Evolution is difficult, but none of the problems are unique to profiles. Making design decisions is a key part of language design.

2. Specification

As an example, consider

- ***Type safety***: Every object is used exclusively according to its definition.
- ***Resource safety***: No resource is leaked.

That's the definition of two profiles. In principle, from these simple definitions, we could implement Profiles to enforce these requirements just as we write compilers and libraries to implement specifications. That is, the detailed requirements can in most cases be deduced from the general requirements. However, it is not fair, safe, or manageable to leave the interpretation of these requirements to implementers. In particular, we need a clear definition of "object" and "resource." Beyond that, we need detailed implementation guidance to avoid implementers (possibly ourselves) having to repeatedly reinvent techniques and rediscover problem areas.

The border between what should be specified in the standards document and what should be in an implementation guide is not trivial to define. Different profiles will require different degrees of detail.

That's not a new problem. However, not everybody – not even in WG21 – is used to think about this in the context of defining a language feature or a library component.

Code examples can be found in the references.

2.1. Object

An object is a region of memory that has been given a type and been initialized according to the definition of that type.

Accessing an object exclusively according to its definition implies that we must not access

- an area of memory as an object before its initialization/construction has completed (except from within its constructor). That is, before it becomes an object.
- an object after its destruction has started (except from within its destructor). That is, after it ceases to be an object.
- an area of memory that hasn't been initialized/constructed according to the type we use to access it. That is, where it might not be an object, or it might be an object of a different type.

That last point implies that we must eliminate range errors and access through dangling pointers.

2.2. Resource

A resource is an entity that must be acquired before use and released after its last use.

That's a very broad definition that cannot be directly translated into verifiable rule. For example, a FILE (a C stream) is a resource because it needs to be opened before use and closed (possibly implicitly at the end of a program). Even an **int** can be a resource. For example, it could be required that it be increased when a scope is entered and decreased as that scope is exited.

To achieve this in a reasonably simple way, we must require that to be considered a resource an entity must have a constructor and a destructor that manages it. That is, an entity that requires explicit acquisition and/or explicit release is simply not considered a resource until it has been encapsulated in a type with a constructor/destructor pair.

2.3. Constructors/destructors

A constructor is supposed to correctly initialize an object, and a destructor is supposed to release every resource. To what extent are we willing to trust that?

We can ensure that every member is initialized in a constructor (except members explicitly marked to remain uninitialized). There should be a separate profile to guarantee initialization for people who cannot adopt the complete type- and-resource profile. That initialization profile should then be used as part of the type- and-resource profile.

The easiest way of ensuring that a destructor matches a constructor is to use only resources managed by constructor/destructor pairs. That is, have only members of types that do not require explicit destruction. For example, using a **vector** or a **unique_ptr** rather than a (pointer,size) pair to represent owned memory. However, that solution doesn't cover every case, e.g., **unique_ptr**'s constructors and destructors.

We can

- Trust the definition of constructors and destructors to be correct (e.g., not leak).
- Require non-trivial constructor/destructor pairs to use explicitly unverified code.

The latter is the most theoretically correct solution. The former would be manageable.

How do we handle errors in destructors caused by an inability to release a resource? For example, a socket that can't be closed because the other end of a communication channel won't cooperate. We can handle that with the techniques we do today, such as transferring the obligation to close an uncooperative socket to some other code, e.g., by putting it on a to-be-handled-later error list.

2.4. Copies and moves

To be handled similar to constructors and destructors.

2.5. Ranges and nullptr

The two obvious ways of avoiding accessing an object that is not of the right type (or no type at all) is to prevent

- Range errors
- Access through nullptr

This is (relatively) easily done by using hardened libraries doing range checking for types for which the number of elements is known (e.g., **vector** and **span** [KV25]) and a pointer type that's checked for nullptr (e.g., **gsl::not_null** [CG]).

That solves half of the problem. The other half is to ensure that unsafe operations are not used (e.g., subscripting a pointer for which the number of elements pointed to is unknown). For that, we need simple static analysis (e.g., the compiler) to ban or restrict

- Pointer arithmetic
- Subscripting raw pointers

Together, these two approaches are a classical example of the subset-of-superset technique. Without the hardened library, we couldn't ban those basic uses of raw pointers and without the ban of those basic uses of raw pointers we couldn't guarantee the absence of range errors and nullptr violations.

2.6. Invalidation

There is a nasty third way that we might access an object of the wrong type (or no type at all):

- access through a invalidated pointer; that is, a pointer to an object that had been deleted or gone out of scope

That's also known as the dangling pointer problem. Preventing invalidation by run-time measures implies complications and overheads (e.g., garbage collection). However, it is possible to eliminate access through invalid pointers through static analysis [HS19, BS21, CG]. That technique has been prototyped and is currently used as a guideline (at best partly enforced).

Here is an example that must be rejected because it invalidates:

```
void f(vector<int>& v)
{
        v.push_back(7);
}

void g()
{
        vector v = {1,2};
        auto p = &v[1];
        f(v);
        *p = 3;                 // must be prevented
}
```

The problem/error occurs because an alias (**p** for **&v[1]**) is used after the object it referred to has been moved. The basic solution is relatively simple and completely compile time.

When analyzing a function:

- Don't access through an invalid pointer.
- Don't let a pointer to a deleted object or to an object that is about to be destroyed out of the function.
- If an alias to an object has been taken, don't call a function that can invalidate it use the object.
- Return only pointers that are known to
  - Have been received as arguments
  - Point to something static
  - Point to something on the free store

By "pointer", we include anything the refers to an object (e.g., references and lambda captures).

By "return", we include any way of passing a pointer out of a scope (e.g., assigning to a static object).

When analyzing a set of functions:

- Assume that a non-**const** function and a function accessing an object through a non-**const** pointer argument invalidates.
- Assume that a **const** function and a function accessing an object through a **const** pointer argument doesn't invalidate.
- Add a **[[not_invalidating]]** attribute to be used to speed up analysis by marking non-**const** functions and functions that take non-**const** pointer arguments that don't invalidate (e.g., **vector<T>::size()**) [BS24c].

  Note that a **[[not_invalidating]]** attribute can be verified when the function definition is compiled. It is not an annotation that relies on programmers using it correctly, just an optimization to save work for the compiler.

The implementation problem here is that what is returned from a function and what has been deleted can be flow-dependent. So, we must accept only code that's not too complex to analyze.

- Some examples are fundamentally impossible to statically analyze.
- Some examples are so complex that a static analyzer would slow down compilation to an unacceptable degree.

An example of code that is fundamentally too complex to accept:

```
int glob = 7;

int* f(int i)
{
        int x = 5;
```

```
        return (0<i) ? &glob : &x;   // might return a pointer to a destroyed object
}
```

We need implementer input to choose what limitation is practical. In particular, to preserve portability, we need a definition of "too complex" so that all compilers agree on what to reject. I suggest:

- If a condition cannot be evaluated at compile time, whatever is guarded by that condition is assumed to be executed.

That is, conditional use of some constructs is treated as unconditional:

- creation of an alias,
- the return of an invalid pointer,
- the invalidation of a pointer,
- the call of an invalidating function, or
- the use of an invalid pointer.

In older code and code written in older styles, this limitation will be a burden, but encouraging the use of simpler code is beneficial in its own right.

3. The Profiles framework

The Profiles framework (based on the subset-of-superset strategy (§1) [BS05, BS05a]) has been developed over several years to cover a wide range of requests involving a variety of notions of safety and style involving combinations of compile-time and run-time techniques [BS25a]. The Framework, complete with syntax and WP wording is described in detail [GDR25]. In particular, it introduces notation for enforcing, requiring, and suppressing profiles.

The framework is to be common to all profiles and independent of any particular profile.

4. Other profiles

To be complete the type-and-resource profile will depend on other profiles. To allow gradual adoption of profiles, such profiles should be independently defined so that they can be used on their own [BS25c]. Examples of possible standard profiles are:

- ***initialization***: every object is properly constructed/initialized (or explicitly declared not to be).
- ***casting***: no old-style non-verifiable casts.
- ***union***: no union use.
- ***ranges***: all pointer arithmetic and all subscripting are checked for range errors.

Obviously, when some construct is banned (outside unverified/trusted code), alternatives must be provided, e.g., **variant**s instead of **union**s). The profiles listed above are relatively easy to specify and thus good candidates for early implementation.

Most likely separate from the type-and-resource profile, but related, we need:

- ***arithmetic***: eliminate implicit narrowing conversions, overflow, and underflow.
- ***algorithms***: use only ranges (rather than iterator pairs) and prevent dereferencing of **end()** iterators.
- ***concurrency***: The type-and-resource profile assumes that there are no data races. To ensure that, a separate concurrency profile is needed. Specifying that is non-trivial and beyond the scope of this paper. There must eventually be a concurrency profile and part of that will undoubtedly be ways of minimizing sharing [CG].

These profiles are important, but harder to specify. Note that libraries for checking arithmetic exists (e.g., [BS25d, HH25]).

5. What needs to be done soon

To make progress, we need to specify, implement, and try out a few simple profiles to gain experience. To avoid chaos, we urgently need the framework to be approved [DV26]. Many ideas that are part of profiles have been prototyped and/or used, but to make progress we must build on the proposed Profiles framework [GDR25] and start experimenting with specific profiles within that framework.

For that to happen, we need the framework available and accessible on one or more C++ implementations, ideally on all major implementations. Otherwise, using different initial/experimental profiles will require too much boilerplate code and different interfaces to different tool chains (e.g., in-code annotations, compiler options, and build-system settings). That will impede portability and possibly introduce incompatibilities that will be hard to root out in the future. Currently, there is – to the best of my knowledge – a specification [GDR25] and an experimental implementation is being conducted in a major C++ compiler. Also, an implementation guide is being written. We encourage implementers to coordinate and if possible, collaborate to avoid inessential incompatibilities [DV26].

Beyond the framework implementation(s), we need to design and precisely specify a few initial profiles. We need a reasonable common style of Profile descriptions. That – and bringing the older documents in line with the C++26 specification – is a necessary first step on the way to standardization. A start is made in [BS24].

6. References

- [BS05] B. Stroustrup A rationale for semantically enhanced library languages
- [BS05a] B. Stroustrup and G. Dos Reis: Supporting SELL for High-Performance Computing. LCPC05. October 2005.
- [BS21] B. Stroustrup: Type-and-resource safety in modern C++
- [BS22b] B. Stroustrup and G. Dos Reis: Design Alternatives for Type-and-Resource Safe C++. WG21 P2687R0. 2022.
- [BS22c] B. Stroustrup: A Tour of C++ (3rd Edition). Addison-Wesley 2022. ISBN 978-0136816485.
- [BS23] B. Stroustrup and G. Dos Reis: Safety Profiles: Type-and-resource Safe programming in ISO Standard C++. P2816R0. 2023-02-16
- [BS23a] B. Stroustrup and G. Dos Reis: Safety Profiles: Type-and-resource Safe programming in ISO Standard C++. P2816R0. 2023-02-16.
- [BS23b] B. Stroustrup: Concrete suggestions for initial Profiles. P3038R0. 2023-12-16.
- [BS24] B. Stroustrup: A framework for Profiles development. P3274R0. 2024-05-5.
- [BS24] B. Stroustrup: Programming: Principle and Practice using C++. Addison-Wesley. 2024. ISBN 978-0-13-830868-1.
- [BS24b] B. Stroustrup: A framework for Profiles development. WG21 P3274R0. 2024.
- [BS24c] B. Stroustrup: Profile invalidation - eliminating dangling pointers . WG21 P3346R0.
- [BS25] B. Stroustrup: 21st Century C++. P3650R0. 2025-3-6.
- [BS25a] B. Stroustrup: Dealing with pointer errors: Separating static and dynamic checking). P3651R0. 2025-3-6.
- [BS25b] B. Stroustrup: What are profiles? P3704R0. 2025-05-19.
- [BS25c] B. Stroustrup: Profiles syntax. P3447R0. 2024-10-14.
- [BS25d] B. Stroustrup: Concept-Based Generic Programming in C++. 2025-10.
- [CG] The C++ Core Guidelines.
- [DV26] D. Vandevoorde, J. Garland, P. E. McKenney, R. Orr, B. Stroustrup, and M. Wong: Profiles and Safety: a call to action. D3970R0. 2026-01-16.
- [GDR11] G. Dos Reis and B. Stroustrup: A Principled, Complete, and Efficient Representation of C++. Journal of Mathematics in Computer Science Volume 5, Issue 3 (2011).
- [GDR21] G. Dos Reis: In-memory and Persistent Representations of C++ . CppCon 2021.
- [GDR25] G. Dos Reis: C++ Profiles: The Framework. P3589R2. 2025-05-19.
- [HH25] H. Hinnant: Better Behaved Integers (bbi). 2025.
- [HS19] H. Sutter: Lifetime safety: Preventing common dangling. P1179 R1. 2019-11-22
- [KV25] K. Varlamov and L. Dionne: Standard library hardening. P3471R4. 2025-02-14.
- [MK13] M. Klimek: The Clang AST - a Tutorial . LLVM Euro 2013.
- [ML25] M. Laverdière, C. Lapkowski, and C. Gros,: A Safety Profile Verifying Initialization. P3402R3. 2025-05-16.

Acknowledgements

Thanks to Howard Hinnant and Joshua Berne for constructive comments on drafts of this paper.