P3589R1 2025-02-02 Reply-To: gdr@microsoft.com
Audience: EWG

C++ Profiles: The Framework

Gabriel Dos Reis
Microsoft

This document offers an overview of the design and rationale for the "C++ Profiles" framework (Stroustrup, 2024) as discussed in previous papers and presentations (Stroustrup, 2024) (Stroustrup & Dos Reis, 2023) (Stroustrup, 2023) (Stroustrup & Dos Reis, 2022) (Stroustrup, 2024) and approved by SG23. The framework is independent of any specific "profile" such as memory safety profile, type safety profile, style profiles, arithmetic safety profile, hardened standard library profile, etc. This is *not* a competing proposal with respect to Herb Sutter's proposal (Sutter, 2025). Rather, the aim here is to "factor out" the mechanisms of the profile framework protocol from considerations and discussions of any specific profile in order to help the community focus attention on the right sets of concerns in the appropriate contexts. Questions regarding the *Profiles framework* should, in general, be considered independently of a given specific profile. Conversely, questions regarding a given profile should be considered independently of the framework. Of course, there will arise questions regarding interactions between profiles or how a given profile should work given the general framework. The aim here is to help structure conversations around profiles in order to make forward progress.

1 DESIGN

At its core, the idea of "profiles" is to allow a programmer to state that they desire their program to abide by the conventional ISO C++ rules and *additional rules*. A set of such additional rules, called *profile*, **are not to change the meaning (i.e. set of permitted behaviors) of a well-formed program with no undefined behavior**. To address the contemporary challenge of memory safety concerns, we need some standard profiles related to type and memory safety, guaranteed to be available in all C++ implementations. Herb Sutter's proposal (Sutter, 2025) is focused on such profiles, to plug into the general C++ profiles framework. The Profiles framework is independent of any specific profile.

As explained in (Stroustrup, 2024), a framework is essential to enable gradual adoption of safety measures, to support incremental improvements to guarantees and support tools, to become a tool for fighting the problems of technical debts affecting safety, and to ensure portability of guarantees across toolchains. This document is not to rehash past conversations; it is not a new proposal; rather, it is to provide a crisp overview, rationale, and specification for the Profiles framework, unencumbered by considerations of any specific profile details. Without a common framework, the options from different suppliers addressing common problems (e.g., range errors) will be significantly different, and code relying on one cannot be relied on to work on another or to offer the same guarantees. Without a framework, we will get an incompatible mess of checks and improvements that will be impossible to unify in a future standard. This serious problem can be addressed by the options implementing specific Profiles, just as if they had been specified in-code.

A companion paper investigates the practical matters of implementation strategies.

1.1 AIMS

The Profiles framework aims to

1. Provide syntactic support for a program (fragment) to express the set of guarantees it enjoys, i.e. set of profiles enforced by the C++ translator. Each profile is formulated to ensure the program is free of a certain class of problems (e.g. memory safety, resource-leak free, etc.) (Stroustrup & Dos Reis, 2022)
2. Provie ability for a program to selectively and locally suppress the application of a profile (or a rule thereof) to a construct, possibly with a justification message string. The granularity of such suppression is limited to a declaration or a statement
3. Support an open-ended set of profiles, some standard, some implementation-defined, some provided by third parties, etc.
4. Provide a mechanism for a library to express an interface along with guarantees that it offers such as through a module interface, etc.
5. Provide a mechanism for a program to take dependencies on other components (modules, libraries, etc.) and to request specific guarantees from those components

A profile may have an effect on the runtime behavior of a program such as enabling runtime instrumentation (e.g. bound checking in array indexing). However, its static semantic effects are as-if applied only after translation phase 7. It is not possible for a profile to change the outcome of overload resolution or template instantiation, nor is it possible to "SFINAE out" failure of a program to satisfy a profile requirement.

The description in the following sections is based on the paper (Stroustrup, 2024) which was approved by SG23 at the Wrocław meeting. Of the four proposed syntactic constructs in section 3 of that paper, only ***three*** are retained, namely:

- `[[profiles::enforce]]` (see sections 1.1.1 and 2.2)
- `[[profiles::require]]` (see sections 1.1.5 and 2.3)
- `[[profiles::suppress]` (see sections 1.1.2 and 2.4)

The fourth, `[[profiles::enable]]`, is deemed redundant with `[[profiles::require]]` in its semantic effects at this point.

1.1.1 Request for profile enforcement

The simplest way for a program to request a profile enforcement is to state something like

`[[profiles::enforce(`*profile-designator-list*`)]];`

as the first thing in a source file (translation unit) before any declaration is seen. We call that a profile-enforcement attribute. Here, *profile-designator-list* is comma-separated list of *profile-designator*s A *profile-designator* is something as simple as a profile name (e.g. `std::type`, `std::lib::hardened`, etc.) or a profile name followed by a parenthesized comma-separated *profile-argument* list such as `acme::hardened(fortify: 3, sanitize: thread)`. Any identifier used in a *profile-argument* is not subject to name lookup rule. The *profile-argument* list is interpreted by the implementation in an implementation-defined way.

The rules in a profile designated in a profile-enforcement attribute are enforced in the dominion of that attribute, which is defined as the set of tokens starting from the end of that attribute till the end of the translation unit containing the attribute.

A profile-enforcement attribute is not to appear anywhere other than at global scope before any declaration. For example, the following program fragment is accepted

```
module;
[[profiles::enforce(std::lib::hardened)]];
export module My.Module;
import Kai.Utils [[profile::require(std::ranges)]];
export enum class ID : int { };
```

but the program fragment

```
int my_abs(int);
[[profiles::enforce(std::lib::hardened)]]; // error: profile enforcement after my_abs decl
import Kai.Utils [[profiles::require(std::type)]]; // OK.
```

is rejected because the profile-enforcement attribute appears after the declaration of the function `my_abs`.

Normally, a source file will state the set of profiles it desires to see enforced as a comma-separated *profile-designator*s as a request at one place. The constraint that a profile-enforcement attribute is to appear before any declaration allows for a row of profile-enforcement attributes before any declarations, as in:

```
[[profiles::enforce(std::lib::hardened)]];
[[profiles::enforce(acme::hardened(fortify: 3, sanitize: thread))]];
#include <my/lib.h>
String greet(const Neighbor& n) { /* … */ }
```

It is allowed to repeat a profile enforcement attribute as long as the repetitions are exactly the same token sequences.

```
[[profiles::enforce(std::lib::hardened)]];
[[profiles::enforce(std::type)]];
[[profiles::enforce(std::lib::hardened)]];  // OK.
[[profiles::enforce(acme::hardened(fortify: 3, sanitize: thread))]];
[[profiles::enforce(acme::hardened(fortify: 3))]];   // error.
[[profiles::enforce(acme::hardened(sanitize: thread))]]; // error.
```

When valid, the subsequent repetitions have no effect.

1.1.2 Local suppression of enforcement

Occasionally, there is a need to locally suppress the enforcement of a profile or a rule thereof. The simplest way to achieve that is to use a profile-suppression attribute on a statement or on a declaration; such a profile-suppression attribute has the form

`[[profiles::suppress(`*profile-name*`)]]`

The dominion of a profile-suppression attribute is the sequence of tokens making up the declaration or the statement to which the attribute appertains. A profile-suppression attribute exempts the dominion of the nominated profiles from enforcement of that profile. For example, in

```
[[profiles::enforce(std::type)]];
extern int read(char* buf, int n);
int main()
{
   [[profiles::suppress(std::type)]] char buffer[1024];  // OK.
   int len = read(buffer, 1014);
}
```

The declaration of the local variable `buffer` suppresses the `std::type` profile requirement that all variables must be initialized at definition point.

Often, from software engineering perspective (code evolution, maintenance, validation processes, etc.) it is necessary to state a reason for why a suppression is in effect. The profile-suppression attribute supports that with a "j`ustification:`" argument as in

```
[[profiles::suppress(std::type,
    justification: “buffer is filled in by read()”)]]
char buff[1024];  // OK.
```

Although this form of profile-suppression was not explicitly shown in the examples from profiles papers published earlier, it wasn't the intent to exclude it. This form is based on experience with `[[gsl::suppress]]` which itself evolved based on years of in-field deployments and feedback from users. The "`justification:`" argument is required to be a string.

Finally, frequently, when locally suppressing a profile, it is not the totality of all rules in the profile that are desired to be non-enforced. Rather, it is often just one specific rule from that profile that one wishes to suppress. This is especially the case when one wants to avoid unintended consequences with relatively large suppression dominion. The profile-suppression attribute supports that scenario with a "`rule:`" argument when a profile has named rules. Here is an example

```
[[profiles::suppress(acme::type,
     rule: “acme.cast.reinterpret”)]]
int* ptr = reinterpret_cast<int*>(0xdeadbeef);
```

for suppressing a ban of `reinterpret_cast` use as dictated by the `acme::type` profile. Although this form of profile-suppression was not explicitly shown in profile papers published earlier, it is based on a decade old experience with `[[gsl::suppress]]` (Editors, 2025). The argument for the "`rule:`" can be any non-comma *balanced-token* – a string literal is typical.

Of course, it is possible to suppress a particular rule from a profile while also supplying a justification for that suppression:

```
[[profiles::suppress(acme::type,
     rule: “acme.cast.reinterpret”,
```

```
        justification: “thumbstone pointer value”)]]
int* ptr = reinterpret_cast<int*>(0xdeadbeef);
```

In that example, the code manufactures a pointer value to be used as a singularity, presumably to catch uses of an invalid pointer.
The relative order of the “`justification:`” argument and the “`rule:`” argument is not important; but the profile name must appear first. While those are the additional arguments to `profiles::suppress` that are explicitly needed in this proposal, we suggest to allow a slightly more generalized form of profile-suppression attribute syntax to accommodate for implementation-defined arguments. In summary, a profile suppression can also be expressed in the general form

`[[profiles::suppress(`*profile-name* `,` *profile-argument-list*`)]]`

1.1.3 Open-ended set of profiles

The profiles framework is designed to accommodate not just standard profiles (e.g. `std::type`, `std::lib::hardened`, etc.) but also implementation defined profiles as well as third-party profiles. For instance, an implementation may “package” existing compiler options into a single coherent profile designed for a specific purpose for programs to use in their source files:

```
[[profiles::enforce(Vendor::Ruleset(“CoreCheck”, fortify: 3))]];
```

More generally, implementations may use this notation to provide “code analysis plugin” hooks for third-party tools providers, and to allow for more experimentations in the design of specific profiles. For instance, since profiles effects notionally take effects after translation phase 7, an implementation can present a profile plugin with the complete abstract semantic graph of a translation unit for profile enforcement before proceeding to “machine code generation”. The profiles framework presents a tooling opportunity with a minimal common notation for programmers to express desire of guarantees.

1.1.4 Interface guarantees

A program usually has dependencies on libraries or other components the interfaces of which can be expressed as module interfaces. A library may express guarantees offered by its primary module interfaces or module partition interfaces through a profile-enforcement as an attribute in its *module-declaration*. For example:

```
module;
#include <unistd.h>
export module Kai.Utils [[profiles::enforce(std::type)]];   // OK.
export using ::getpid;
// …
```

The meaning of a profile-enforcement attribute in a *module-declaration* is that the nominated profile is enforced in its dominion in that module unit, and if the *module-declaration* is that of a primary module interface unit then the dominion extends to all module implementation units of that module, and only to those. Furthermore, the profile names (or rather the *profile-designator*s) are checked in *module-import-declaration*s that request a particular profile in a translation units that import the

module. A profile-enforcement attribute that appears before a module-declaration only affects that particular module unit.

1.1.5 Request of guarantees provided by an interface

When a program takes dependencies on another component (expressed as a module), it may request that the component's interface is indeed advertised as obeying the rules of specific profiles. For instance,

```
import Kai.Utils[[profiles::require(std::type)]];
```

is OK if the module `Kai.Utils` was declared with the *profile-designator* `std::type` in a profile-enforcement attribute in the *attribute-specific-seq* of its *module-declaration*. Otherwise, that *module-import-declaration* is an error.

The same behavior is expected if, instead of a named module, a header unit was imported. In that case, the expectation is that the requested *profile-designator* was used in the profile-enforcement in the header (file) corresponding to the header unit.

2 PROPOSED WORDING

2.1 DIAGNOSABLE RULE

Failure to satisfy a constraint dictated by a profile is a diagnosable rule. Accordingly, modify paragraph [intro.compliance.general]/1 as follows:

> The set of *diagnosable rules* consists of all syntactic, ~~and~~ semantic, and profile rules in this document except for those rules containing an explicit notation that "no diagnostic is required" or which are described as resulting in "undefined behavior". An implementation is permitted to provide additional profile rules if they are active only under the appropriate implementation-defined profile.

2.2 PROFILE ENFORCEMENT

The enforcement of a profile is requested via a *profile-designator* defined as follows. Add these new productions to the list or productions in paragraph [decl.attr.grammar]/1

profile-designator:
    profile-name
    profile-name (*profile-argument-list*)
profile-name:
    identifier
    profile-name : : *identifier*
profile-argument-list:
    profile-argument
    profile-argument-list , *profile-argument*
profile-argument:
    non-operator-non-punctuator-token

identifier : *non-comma-balanced-token*

non-operator-non-punctuator-token:
Any *token* other than an *operator-or-punctuator*

non-comma-balanced-token:
Any *balanced-token* other than comma

2.2.1 Syntax

Modify paragraph [decl.pre]/1 as follows:

empty-declaration:
attribute-specifier-seq$_{opt}$;

Modify paragraph [decl.pre]/15 as follows

An *empty-declaration* has no effect. The optional *attribute-specifier-seq* appertains to the *empty-declaration*.

2.2.2 Static Semantics

A profile may have a dynamic semantics (e.g. requesting array bound checking) in addition to static semantics. The static semantics is conceptually applied after translation phase 7.

Add a new subsection titled "Profile enforcement [decl.attr.enforce]"

1. A *profile-enforcement attribute* is an *attribute* where the *attribute-token* is `profiles::enforce`. It shall appear only in the *attribute-specifier-seq* (if any) of a *module-declaration*, or in an *empty-declaration* and that *empty-declaration* shall precede any non-*empty-declaration*, if any, in the *translation-unit*.

2. A profile-enforcement attribute shall have an *attribute-argument-clause*, and that *attribute-argument-clause* shall have the syntactic structure `(` *profile-designator* `)`. The *profile-name* in a *profile-designator* specifies a *profile*, which is a set of additional language restrictions applied after translation phase 7. [*Example*:

```
module;
[[profiles::enforce(lib::hardened)]];
export module My.Module;
import Kai.Utils [[profile::require(acme::ranges)]];
export enum class ID : int { };
```

--end example]

[*Example*:

```
int my_abs(int);
[[profiles::enforce(acme)]]; // error: profile enforcement after my_abs decl
import Kai.Utils [[profiles::require(acme::type)]]; // OK.
```

--end example]

[*Example*:

```
[[profiles::enforce(lib::hardened)]];
[[profiles::enforce(acme(fortify: 3, sanitize: thread))]];
```

```
[[profiles::enforce(acme::ruleset("CoreCheck"))]];
#include <my/lib.h>
String greet(const Neighbor& n) { /* … */ }
```

--end example]

3. It is permitted to repeat a profile-enforcement attribute naming the same profile as long as the repetitions are exactly the same token sequences. When valid, the subsequent repetitions have no effect. [*Example*:

```
[[profiles::enforce(lib::hardened)]];
[[profiles::enforce(acme::type)]];
[[profiles::enforce(lib::hardened)]];  // OK.
[[profiles::enforce(acme(fortify: 3, sanitize: thread))]];
[[profiles::enforce(acme(fortify: 3))]];   // error.
[[profiles::enforce(acme(sanitize: thread))]]; // error.
```

--end example]

4. In a given translation unit, the *dominion* of a profile *P* is the sequence of tokens starting after a profile-enforcement attribute whose *profile-designator* nominates P, and extending to the end of that translation unit. The additional language restrictions enabled by the profile *P* apply only to its dominion, possibly except the dominion of a profile-suppression ([decl.attr.suppress]) that specifies *P*. If the profile-enforcement attribute appears in the *export-declaration* of the primary interface unit of a module *M*, then the dominion of *P* includes all module implementation units of *M*.

5. If a *declaration D* appears in the dominion of a profile P_1, all other redeclarations of *D*, if any, shall appear in the dominion a profile P_2 and any such P_2 shall be compatible with P_1. Two profiles are *compatible* if they are the same or proclaimed as such by the implementation. All standard profiles are compatible with each other.

2.3 PROFILE REQUIREMENT

Add a new section titled "Profile requirement [decl.attr.require]":

1. A *profile-requirement attribute* is an *attribute* where *attribute-token* is `profiles::require`. That *attribute* shall have an *attribute-argument-clause* the syntactic structure of which shall be `(` *profile-designator* `)`.
2. A profile-requirement attribute shall appear only in a *module-import-declaration*. If that *module-import-declaration* specifies a named module or a module partition *M*, then the *profile-designator* shall appear in a profile-enforcement attribute contained in the *export-declaration* of the module interface unit of *M*. If the *module-import-declaration* specifies a *header-name H*, then the *profile-designator* shall appear in a profile-enforcement attribute ([decl.attr.enforce]) of an *empty-declaration* in the header unit corresponding to H. [Example:

Translation unit #1:

```
export module Kai.Utils [[profiles::enforce(lib::hardened, acme::type)]];
// …
```

Translation unit #2:

```
import Kai.Utils [[profiles::require(acme::type)]];       // OK.
// …
```
Translation unit #3:
```
import Kai.Utils [[profiles::require(acme::sanitize)]];   //error.
// …
```

--end example]

[Example:

Translation unit #1:
```
export module Kai.Utils [[profiles::enforce(lib::hardened, acme:type)]];
// …
```
Translation unit #2:
```
import Kai.Utils [[profiles::require(lib::hardened)]];
import Kai.Utils [[profiles::require(lib::hardened)]];    // OK.
import Kai.Utils [[profiles::require(acme::type)]];       // OK.
```

--end example]

2.4 Local Profile Suppression

Add a new section titled "Local profile suppression [decl.attr.suppress]":

1. A *profile-suppression attribute* is an *attribute* where the *attribute-token* is `profiles::suppress`. That *attribute* shall have an *attribute-argument-clause* of the form `(` *profile-name* `)` or `(` *profile-name* `,` *profile-argument-list* `)`.
2. If a *profile-argument* in a profile-suppression attribute is of the form
   `justification` `:` *non-comma-balanced-token*

   then that *non-comma-balanced* shall be a *string-literal*.
3. The *dominion of a profile-suppression attribute* is the sequence of tokens starting after the attribute extending till the last token of the declaration or statement to which the attribute appertains. If a profile-suppression attribute designates a profile *P*, none of the rules of *P* is enforced in the dominion of that attribute.
4. A profile *P* may support an implementation-defined way to specify, via *non-comma-balanced-token*, a specific rule in *P*. If a *profile-argument* in a profile-suppression attribute is of the form
   `rule` `:` *non-comma-balanced-token*

   then only the rule of P designated by that *non-comma-balanced-token* is not enforced in the dominion of that attribute; all other rules of *P* are in effect. [Example
   ```
   [[profiles::enforce(acme::type)]];
   extern int read(char* buf, int n);
   int main()
   {
      [[profiles::suppress(acme::type)]] char buffer[1024];  // OK.
      int len = read(buffer, 1014);
   }
   ```
   --end example]
5. [Example:

```
    [[profiles::suppress(acme::type,
         rule: “acme.cast.reinterpret”,
         justification: “thumbstone pointer value”)]]
    int* ptr = reinterpret_cast<int*>(0xdeadbeef);
--end example]
```

3 CHANGELOG

3.1 REVISION R0 -> R1

- Add examples of usage of profiles.
- Add more contexts and references to previous papers presented to SG23 and EWG.
- Clarify that a request to a profile can have arguments; add examples.
- Add examples of suppressions with justification message and named rule.
- Add examples of guaranteed profiles and requested profiles
- Completely rework the proposed formal wording section.

4 ACKNOWLEDGMENT

Thanks to all of you who provided feedback on earlier feedback drafts of this paper, in particular Bjarne Stroustrup, and Ville Voutilainen.

5 REFERENCES

Editors, C. C. G., 2025. *C++ Core Guidelines*. [Online]
Available at: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

Stroustrup, B., 2023. *Concrete suggestions for initial Profiles*. [Online]
Available at: https://open-std.org/JTC1/SC22/WG21/docs/papers/2023/p3038r0.pdf

Stroustrup, B., 2024. *A framework for Profiles development*. [Online]
Available at: https://open-std.org/JTC1/SC22/WG21/docs/papers/2024/p3274r0.pdf

Stroustrup, B., 2024. *Profile invalidation - eliminating dangling pointers*. [Online]
Available at: https://open-std.org/JTC1/SC22/WG21/docs/papers/2024/p3446r0.pdf

Stroustrup, B., 2024. *Profiles syntax*. [Online]
Available at: https://open-std.org/JTC1/SC22/WG21/docs/papers/2024/p3447r0.pdf

Stroustrup, B., 2024. *Profiles syntax*. [Online]
Available at: https://open-std.org/JTC1/SC22/WG21/docs/papers/2024/p3447r0.pdf

Stroustrup, B. & Dos Reis, G., 2022. *Design Alternatives for Type-and-Resource Safe C++*. [Online]
Available at: https://open-std.org/JTC1/SC22/WG21/docs/papers/2022/p2687r0.pdf

Stroustrup, B. & Dos Reis, G., 2023. *Safety Profiles: Type-and-resource Safe Programming in ISO Standard C++*. [Online]
Available at: https://open-std.org/JTC1/SC22/WG21/docs/papers/2023/p2816r0.pdf

Sutter, H., 2025. *Core safety Profiles: Specification, adoptability, and impact*. [Online]
Available at: https://open-std.org/JTC1/SC22/WG21/docs/papers/2025/p3081r1.pdf