Doc. No. D3447R0
Date: 2024-10-14
Audience: SG23
Reply to: Bjarne Stroustrup (bjarne@stroustrup.com)

Profiles syntax

Abstract

This note defines the syntax for Profiles and names a few profiles that should become standard. It also summarizes a few rationales for using that syntax .

This note outlines a design that still needs to be implemented and tested. This is a note for comments.

1. Introduction

Profiles is a system for explicitly requiring certain guarantees (such as type safety, resource safety, and arithmetic safety). Such guarantees are provided by a combination of restrictions (eliminating “unsafe code”) and run-time checks (enforcing proper usage).

The aims of profiles are:

- To offer a small set of coherent guarantees, rather than a large “random” set of tests and modifiers.
- To localize requests for guarantees in a few well-specified places, rather than spread throughout the code.
- To ensure that the resulting program is ISO standard C++, rather than having the meaning of constructs incompatibly changed by annotations.

The last aim ensures that the meaning of a program is exactly the same as if the profile hadn’t been specified with the caveat that unspecified behavior might be handled differently (e.g., range-checking might have been applied as required by a profile). This again implies that validated code can be combined with unvalidated code, as is necessary for experimentation, gradual introduction, partial application, and validating on one system and re-compiling on another.

It is impossible to offer meaningful guarantees without restrictions. This approach ensures that most contemporary C++ facilities are usable with most profiles. Most restrictions fall

on antiquated programming styles for which there are already better alternatives in ISO Standard C++.

2. Previous work

This paper is based on previous work:

- H. Hinnant, R. Orr, B. Stroustrup, D. Vandevoorde, M. Wong: DG opinion on safety for ISO C++. P2759R1. 2023-01-23.
- B. Stroustrup and G. Dos Reis: Design Alternatives for Type-and-Resource Safe C++. P2687R0. 2022-01-15.
- B. Stroustrup, G. Dos Reis: Safety Profiles: Type-and-resource Safe programming in ISO Standard C++. P2816R0. 2023-02-16.
- B. Stroustrup: Concrete suggestions for initial Profiles. P3038R0. 2023-12-16.
- B. Stroustrup: A framework for Profiles development. P3274R0. 2024-5-5.

Further references can be found in those papers. Naturally, those papers reflect progress of the ideas over time.

3. Syntax

Profiles use the attribute syntax. The attribute syntax is defined in §9.12.1 “Attribute syntax and semantics” [dcl.attr.grammar].

Profiles uses three attribute specifiers, each prefixed by **profiles::**

- **profiles::enable**
- **profiles::enforce**
- **profiles::suppress**
- **profiles::require**

These specifiers are followed by an *attribute-argument-clause*. For example:

[[profiles::enable(ranges)]]

[[profiles::enable(ranges, experimental)]]

The experimental versions are there to ease experimentation, to allow partial implementation of a profile, and gradual introduction of profiles. An experimental version should be a subset of the full profile, but its detailed meaning should not be standardized.

An experimental version is not interchangeable with a “final” version. They are considered as having different names.

Questions:

- Is it best to use the plural **profiles?** (yes)
- Is it best to capitalize Profile names? (no, to keep in the ISO standard style)
- Is it best to require the prefix **profiles::** for profile directives? (yes)
- Is it best to require the prefix **profiles::** for profile names? (no)

The general idea is to make Profile directives to be highly visible in code. They will be few and highly significant.

In principle, a Profile attribute could be applied to any scope (e.g., a function definition) but because of header files it will probably be best – at least initially – not to allow them in the global scope.

A profile name is an *attribute-token*.

3.1. Profiles::enforce

[[profiles::enforce(p)]] must be applied to a module or a scope to ensure that **p** is enforced. For example:

- **namespace [[profiles::enforce(union)]] N { ... }**
- **export module DataType.Array [[profiles::enforce(memory)]];**

The position of the attribute for **namespace** and **import** is odd. Am I using it wrong here?

3.2. Profiles::enable

[[profiles::enable(p)]] must be applied to an import directive or a scope and enforces **p** for uses of that imported module. For example:

- **import M [[profiles::enable(arithmetic)]];**
- **namespace [[profiles::enable(ranges)]] M { /* ... */ }**

3.3. Profiles::suppress

[[profiles::suppress(p)]] must be applied to an import directive or a scope for a module or a scope. For example:

- **import M [[profiles::suppress(type)]];**
- **namespace [[profiles::suppress(type)]] N { /* ... */ }**

A typical use would be to use unverified/trusted code to implement abstractions needed to support a profile.

3.4. Profiles::require

[[profiles::require(p)]] must be applied to an import directive for a module or a scope. For example:

- **import M [[profiles::require(type)]];**

The **import** fails unless the imported module was compiled with that profile.

4. Initial profiles

The initial set of profiles to be standard is:

- **algorithms**
- **arithmetic**
- **casting**
- **concurrency**
- **initialization**
- **invalidation**
- **pointers**
- **ranges**
- **RAII**
- **type**
- **union**

Draft specifications and rationales for these profiles can be found in P3274R0.

Many of these profiles are “profile fragments” meant to be used in combinations.

We could add _**safety** to every profile name because all of the profiles (so far) are concerned with safety. I thought that redundant and verbose. Also, these profiles can be seen a more directly concerned with correctness than mere safety. It is of course easy to be safe according to some criteria yet yielding wrong results.

5. Why the [[…]] syntax

The profiles use the attribute syntax, e.g., **[[profiles::enable(ranges)]]**, rather than a “proper language syntax using keywords” (possibly context dependent keywords), e.g., **profiles::enable(ranges);** to ease experimentation and gradual introduction. This choice of

syntax does not imply that enforcement is optional in implementations that support profiles. However, using the attribute syntax allows us to have a single code base for code that must be compiled with a variety of compilers, some pre-profiles. In particular, it allows us to support a likely important use case: First check your program with the best compiler for Profiles support, then port the code and compile with an older compiler. Without **#ifndef** hackery.

This use of the **[[...]]** syntax was approved by a SG23 vote at the Saint Luis meeting:

> ***POLL: We are in favour of the [[Profiles::enable(...)]] syntax.***
>
> **SF WF N WA SA**
>
> 12 6 1 3 0

6. Supporting attributes

To support some profiles some attributes can be necessary or merely helpful. Many avoid false positives by simplifying analysis.

Such annotations can take the form of attributes (e.g., **[[not_invalidating]]**) or type aliases (e.g., **owner**) and tend to apply to just one profile.

6.1. Profiles: initialized

profiles::initialized requires that objects are initialized, but there are situations where uninitialized memory is essential. To avoid verbose and relatively frequent use of suppression an **uninitialized** attribute is helpful:

- **[[uninitialized]]**

This can be used can be used to mark define objects that needs to be uninitialized (e.g., input buffers). For example:

- **int buf[10'000] [[uninitialized]];**

6.2. Profiles: invalidate

invalidate prevents invalidating operations in non-**const** member functions and on non-const arguments. That can be overly strict. In that case, we can use:

- **[[not_invalidating]]**

[[not_invalidating]] can be statically validated, so it is not a potential source of error.

- **owner**

Signifies that a pointer must be **deleted**. Type aliases are semantically equivalent to attributes when used for Profiles, but less syntactically "noicy."

- **[[not_local]]**

Indicates that a pointer returned from a function does not depend on an argument. **[[not_local]]** can be statically validated, so it is not a potential source of error.

- **[[not_returned]]**

Indicates that parts of an object isn't used in the return value of a function. **[[not_returned]]** can be statically validated, so it is not a potential source of error.

- **[[invalidating]]**

Suppresses invalidation protection for operations of a function argument.

7. Potential restrictions and extensions

What is described above is a minimal scheme. Naturally, extensions must be considered.

7.1. Combination of profiles

Some profiles are combinations of others. For example, **type** requires **initialization**, **ranges**, **invalidation**, and more. We could add a directive for that. For example:

- [[**profiles::combine(type**, i**nitialization**, **ranges**, **invalidation)**]]

However, we shouldn't support that idea until we learn whether this really belongs in the language, rather than in some external system for implementing profiles.

7.2. Control of run-time violation handling

The selection of violation response must be a global property and will have to be known to the implementors of standard library components, such as vector and span. I suggest we have the option to set that in code. For example

- **[[profiles::enforce(range, runtime-violation-response, throw)]]**

The build system must ensure that all requests for such a violation response are identical.

See also : A framework for Profiles development.

8. Further work

This paper is focused on syntax. Pinning down the semantics of the various constructs and profiles is more important and interesting, but we need a shared syntax to effectively do that. I plan to write a note on each of the mentioned profiles with sufficient detail for a good developer to implement a first version and supply feedback to improve the definition. With the framework (P3274R0) in place, others have many places to contribute (text or code).

Please help. It would be unfortunate if we had to wait for specification and implementation until I could draft all these documents.

9. Acknowledgements

Many thanks to Xavier Bonaventura, Ilya Burylov, Christoff Meerwald, Gabriel Dos Reis, Andreas Weiss, Michael for comments and especially tricky examples.