

Profiles: implementation status and experience

Document #:	P4415R0
Date:	2026-10-22
Project:	Programming Language C++
Audience:	SG23 (Safety and Security), EWG
Authors:	Michael Wong fraggamuffin@gmail.com ; Krystian Stasiowski ; Ville Voutilainen
Special note:	Information only. Not for schedule.

1. Abstract

Profiles now have two independent implementations. Krystian Stasiowski's Clang implements the P3589R2 framework, except `[[profiles::exempt]]`, and most of the P4222R2 initialization profile `std::init`. Ville Voutilainen's GCC implements the framework, `std::init`, and the first working invalidation profile, `std::invalidation`, on the design of P3446R0 with the default-deny discipline of P4296R0. Both run in a browser on Compiler Explorer today. This paper reports what each implementation does, where the two agree and where they diverge, what was measured and how, and what the implementation experience says the papers should change. It is a status report, not a design proposal. Every result is dated and most carry a link that reproduces it.

The initialization profile works on two compilers and they agree on its core rules. Invalidation works on one, and it rejects all nineteen of a set of adversarial dangling, use-after-free and escaping-address cases. Enforcing `std::init` leaves Clang's generated code byte-identical, which demonstrates the "same meaning" property on a real compiler. The divergences between the two are few, specific, and each one is a question the papers have not yet answered.

2. Revision history

R0: initial revision, for the 2026-10 pre-Búzios mailing.

3. Scope and method

This paper reports only behavior we have observed on the running implementations. Where a statement rests on an implementer's own report rather than on an independent run, we say so. Statements about committee papers name the paper and section.

Two kinds of evidence are used. The first is each implementer's own test suite and measurements: for Clang, the implementation status report of 2026-10-08 [Stasiowski-2026-10]; for GCC, the commit history and testsuite of the [p4324](#) branch through commit [fae893e](#) of 2026-10-08 [Voutilainen-p4324]. The second is an independent dual-compiler suite that the first author runs against both Compiler Explorer builds. It holds 78 cases, each compiled on both compilers, with a separate expected outcome for each compiler and a recorded cross-check of whether the two agree. The most recent full run, on 2026-10-06, compiled all 156 arms with no API errors.

Both Compiler Explorer builds trail their development branches. Where a feature is in a branch but not yet on Compiler Explorer, we say so. The design papers have moved since the implementations were checked against them: P4222R3 and the invalidation paper P4418R0 are in this mailing, with proposed wording in P4416R0 and P4417R0, and P3589R3 renames the framework subclauses. Where this paper names a revision or a section, it is the one the implementation was checked against.

	Clang	GCC
Implementer	Krystian Stasiowski	Ville Voutilainen
Source	github.com/cppalliance/clang, branches profiles , profiles-init	github.com/villevoutilainen/gcc, branch p4324
Compiler Explorer	"x86-64 clang (std::init profile - P4222)"	"x86-64 gcc (P4324 contracts, and profiles)"
Binaries	profiles-init-latest prerelease of cppalliance/clang	build from source
Profiles covered	framework, std::init	framework, std::init , std::invalidation
Flags used in this paper	-std=c++26 -fprofiles (Compiler Explorer default) or	-std=c++23 -pedantic-errors

	Clang	GCC
	<code>-std=c++23</code> <code>-pedantic-errors</code>	

4. The picture in one table

Feature	Clang	GCC
<code>[[profiles::enforce]]</code> , <code>[[profiles::suppress]]</code>	yes	yes
Suppression of a single rule (<code>rule:</code>)	yes, twelve named <code>std::init</code> rules	syntax accepted; no <code>std::init</code> rule name is recognized yet
<code>[[profiles::require]]</code> on import, module advertisement	yes, including header units	not measured
<code>[[profiles::exempt]]</code>	no; system headers exempt by default instead	yes, transitive across <code>#include</code> ; system headers also exempt by default
Command-line enforcement	<code>-fprofiles-enforce=</code>	<code>-fprofiles-enforce=</code> , and <code>-fprofiles-warning=</code> for a warning-only preview
Enforcing an unknown profile name	accepted, enforces nothing	rejected as an error
<code>[[uninit]]</code> , <code>[[ref_to_uninit]]</code>	yes	yes
<code>construct_at</code> / <code>destroy_at</code> lifecycle against an unmodified standard library	yes, annotated by the compiler	yes, annotated in the library
<code>uninitialized_*</code> , <code>destroy</code> , <code>ranges</code> forms	yes (development branch for some forms)	<code>uninitialized_fill_n</code> not annotated as of 2026-10-06

Feature	Clang	GCC
<code>[[must_init]]</code> on a parameter	no; <code>[[now_init]]</code> on the function instead	yes
Initialization bridge	<code>[[now_init]]</code> attribute	<code>std::now_init</code> function in <code><utility></code>
Class-type handles (<code>span</code> , <code>unique_ptr</code> , iterators) with <code>[[ref_to_uninit]]</code>	yes (development branch; work continuing)	not measured
<code>std::invalidation</code>	no	yes
<code>[[owner]]</code> , owning factories, <code>std::as_owner</code>	no	yes
<code>std::no_dangling</code> , <code>std::no_escape</code>	no	yes
<code>[[not_invalidating]]</code>	no	yes
Same meaning with the profile off	yes, LLVM IR byte-identical	yes, attributes inert with the profile off

The invalidation rows are the most consequential. A program that enforces `std::invalidation` on Clang today compiles clean because Clang does not know the profile. That is the framework's specified behavior for an unknown profile name, and it is why section 5.3 asks the framework paper to reconsider it.

5. The framework (P3589R2)

5.1 Clang

Clang implements every P3589R2 framework feature except `[[profiles::exempt]]` [Stasiowski-2026-10, section 2]. Enforcement applies from the attribute to the end of the translation unit and is positional everywhere, including in the runtime-check emitter. Suppression covers exactly the declaration or statement it is written on, which P3589R2 calls its dominion: on a function definition's declarator-id it covers the whole definition, and on one declarator of a group it covers only that declarator. Module interfaces advertise their enforcements, importers check them with `[[profiles::require]]` using exact designator matching, a non-partition implementation unit inherits the interface's enforcement, and importing

an enforcing module does not enforce anything in the importer. Redeclaration compatibility ([decl.attr.enforce]p5) is implemented.

Every profile rule is its own diagnostic group, `-Wprofile-<profile>-<rule>`, nested in `-Wprofile-<profile>` and `-Wprofiles`. Without enforcement, a `-W` option previews a profile's rules as warnings, which lets a team measure a codebase before adopting the profile. Under enforcement a violation is an error that `-w` and `-Wno-...` cannot turn off.

Two framework defects were found and fixed during the 2026-09 work and are worth recording because they would affect any implementation. A module compiled to object code from its built module interface, the normal two-phase modules build, emitted no runtime checks at all, because enforcements were deliberately not recorded in the module file to keep them from leaking into importers. And an implementation unit could never open with its own enforcement, because the implicit import of its own interface counted as a preceding declaration. Both are fixed.

5.2 GCC

GCC implements `[[profiles::enforce]]` and `[[profiles::suppress]]` on statements and functions, `[[profiles::exempt]]` transitively across `#include` chains, automatic exemption of system headers, `-fprofiles-enforce=` for enforcement without touching the source, and `-fprofiles-warning=` for a warning-only preview. Misattached `enforce` and `exempt` attributes are diagnosed. Rule-level suppression parses and no longer crashes the compiler (fixed 2026-09-11), but no `std::init` rule name is recognized yet: as of 2026-10-06, `rule: "ref_to_uninit"` is rejected with "profile 'std::init' has no sub-rule 'ref_to_uninit' to suppress". Module behavior on GCC was not measured for this paper.

5.3 What the framework paper should take from this

The two implementations converged independently on the same command-line spelling, `-fprofiles-enforce=`, and both exempt system headers by default. Those are good candidates for a recommended practice note.

They diverge on an unknown profile name. P3589R2 says enforcing an unknown profile is well formed and enforces nothing. Clang follows that; GCC rejects it. The cost of the paper's rule is now concrete: a user who enforces `std::invalidation` on a compiler that does not implement it gets no checking and no warning. We suggest the framework require at least a diagnostic for an enforced profile the implementation does not know, while keeping `[[profiles::require]]` on import as the mechanism that makes the absence a hard error.

Rule names are the second gap. Clang has twelve named `std::init` rules; GCC accepts the `rule:` syntax but has none yet. A suppression written for one compiler is meaningless on the

other. If rule-level suppression is to be portable, the rule names belong in the profile's specification, not in each implementation.

6. The initialization profile `std::init` (P4222R2)

6.1 Clang

Clang enforces P4222R2's guarantee, that no object is read or written before it is initialized, entirely at compile time using local definite-assignment analysis [Stasiowski-2026-10, section 3]. Every object is initialized or marked `[[uninit]]`; a marked object must be assigned on every path before it is read; a marked object cannot also have an initializer. Pointers and references to uninitialized memory carry `[[ref_to_uninit]]`, checked in both directions. Constructors must initialize every member and base unless the member is `[[uninit]]`. Non-local objects need constant initialization; function-local statics are exempt. `std::byte` is exempt. Templates are checked per instantiation.

The lifecycle rules work against unmodified libstdc++ and libc++. Clang recognizes `std::construct_at`, the `uninitialized_*` family with its `_n` and `ranges::` forms, and `allocator_traits::construct` as initializing; `destroy_at`, `destroy`, `destroy_n` and `allocator_traits::destroy` as destroying; `malloc`, `operator new`, `allocator::allocate` and the `*_for_overwrite` factories as returning uninitialized memory; and `memset`, `memcpy` and `memmove` as initializing their destination. A second `construct_at` on live storage, a `destroy_at` on destroyed storage, and a `destroy_at` on never-constructed storage are each rejected.

```
[[profiles::enforce(std::init)]];

#include <cstdlib>

#include <memory>

struct X { int v; X(int x) : v(x) {} };

void f() {

    X *p [[ref_to_uninit]] = static_cast<X *>(std::malloc(sizeof(X)));

    std::construct_at(p, 1);    // OK: initializes *p

    int a = p->v;                // OK

    std::construct_at(p, 2);    // error: *p is already initialized
```

```

    std::destroy_at(p);           // OK: *p is uninitialized again

    std::destroy_at(p);           // error: destroyed twice
(double_destroy)

    std::free(p);

}

```

Two marker attributes go beyond the paper. `[[now_init]]` declares that a function initializes the storage passed to its `[[ref_to_uninit]]` parameters, and is Clang's form of the paper's `[[must_init]]`. `[[now_uninit]]` declares that a function ends the lifetime of its argument's storage, which is the record P4222R2 section 4.4 says is missing for `destroy_at`. Neither name is settled: GCC spells the second `std::now_uninit`, and P4222R3 is to add a marker for the destroy side under a name, `now_uninit` or `must_uninit`, still to be agreed between the paper and the implementations.

In the development branch, any non-union class can carry `[[ref_to_uninit]]`: `std::span`, `std::unique_ptr`, `std::shared_ptr`, iterators, `std::ranges::subrange`, or a user class. A marked handle's element accesses behave like the same accesses through a marked pointer, and a marked `unique_ptr` owns its referent, so its destruction, `reset` and reassignment are destroys. This work is continuing and reaches Compiler Explorer with its next build.

```

void init(std::span<X> s [[ref_to_uninit]], int i) {

    std::construct_at(&s[0], 10); // OK

    X y = s[i];                  // error: copies from uninitialized
memory

    auto n = s.size();           // OK: uses only the handle

}

```

Not implemented in Clang: the `[[must_init]]` parameter spelling and verification of a callee's initializing obligation (P4222R2 section 6.2), the exposure analysis of section 5.3, and overloading on `[[ref_to_uninit]]` (section 4.8), which the framework's rule that profiles never change overload resolution excludes. Unions marked `[[uninit]]` are rejected.

6.2 GCC

GCC implements `[[uninit]]` and `[[ref_to_uninit]]` with definite-assignment analysis on GIMPLE, per-field analysis for `[[uninit]]` aggregates, constructor-body analysis for `[[uninit]]` members (P4222R2 sections 5.1 to 5.3), and the ban on `[[uninit]]` union members (section 5.6). It implements `[[must_init]]` on parameters and checks the obligation against the function's own body, including through a traced pointer variable. `[[ref_to_uninit]]` is supported on a function's return value and is preserved through pointer arithmetic. The library supplies `std::now_init`, `std::now_uninit`, `std::now_init_in_place`, and two address-taking escape hatches, `std::escape_uninit` and `std::escape_ref_to_uninit`. Since 2026-10-21 the branch rejects whole-object `[[uninit]]` on a class with any non-trivially initialized subobject, so a mixed class marks its members individually, and rejects non-local objects with non-constant initialization. It accepts member-wise definite assignment of an `[[uninit]]` aggregate, which the P4222R3 draft of 2026-10-22 rejects; that question is with the author (section 6.4).

```
[[profiles::enforce(std::init)]];

int f(int* arg [[must_init]]) { return 0; }

// error: function may return without the pointee of 'arg' being
// definitely

//          assigned, under the 'std::init' profile
```

6.3 Conformance against the paper's own examples

Krystian compiled every P4222R2 code example that states an outcome. Of 87, 59 match the paper and 9 have no stated verdict. The 19 that differ fall into three groups. Twelve are examples that conflict with a rule stated elsewhere in the paper; Clang follows the rule. Four are deliberate deviations: element-wise construction and destruction, and treating an escape as initialization. Three are unimplemented features: `[[must_init]]`, the callee obligation, and a marker on an array of pointers [Stasiowski-2026-10, section 4]. The twelve internal conflicts are the most useful output of this exercise for the paper's author, and the list is available for the next revision.

6.4 Where the two compilers agree and disagree

Of the 40 initialization and framework cases in the dual-compiler suite, as of the 2026-10-06 run, 22 agree in outcome for the same reason, 4 agree for different reasons, 9 diverge, and 5 exercise features only GCC has. No internal compiler error remains; the two that the suite carried were fixed in GCC on 2026-09-11. One divergence closed during September:

`construct_at` through a `[[ref_to_uninit]]` pointer now initializes the pointee on Clang, as it already did on GCC.

The core rules agree: a read of an `[[uninit]]` scalar or class subobject before initialization, an unmarked uninitialized object, `[[ref_to_uninit]]` bound to an object in the wrong state, and the address of an `[[uninit]]` object passed to a plain pointer are rejected on both, and the attributes are inert on both with the profile off.

The divergences are design questions, not bugs, and each needs an answer in P4222:

Question	Clang	GCC	P4222R2
Does a member write (<code>x.y = 5</code>) complete an <code>[[uninit]]</code> class object?	no; only whole-object initialization counts	yes	section 4.6 says a class object is initialized by <code>construct_at()</code> ; sections 5.4 and 5.5 are less clear; the R3 draft of 2026-10-22 makes member assignment an error
Does copy assignment complete an <code>[[uninit]]</code> class object?	no	yes	Same as above
<code>[[ref_to_uninit]]</code> on a reference	accepted	rejected	section 4.3
How a callee reports that it initialized its argument	<code>[[now_init]]</code> on the function	<code>[[must_init]]</code> on the parameter	section 6.2 uses <code>[[must_init]]</code>
The bridge from uninitialized to initialized storage	<code>[[now_init]]</code> attribute	<code>std::now_init</code> function	section 4.4 shows a function
Where the markers go	declaration positions only	declaration positions	some examples use type positions

There is today no spelling of the initialization bridge that compiles on both implementations. That is the single most practical item to settle in the next revision of P4222.

7. The invalidation profile `std::invalidation` (P3446R0, P4296R0, P4418R0)

7.1 What GCC implements

GCC implements `std::invalidation` on the P3446R0 design, with the zero-false-negatives, default-deny discipline of P4296R0 [Voutilainen-p4324]. The implementation has two layers. A negative baseline rejects every pattern it cannot prove safe. Two positive rules then accept what can be proven: Rule #0 is type-structural (unrelated types cannot alias, decided without data flow), and Rule #1 is local data flow (two locals, or a local and a parameter, proven not to invalidate each other). The analysis is intraprocedural and runs on GIMPLE after early inlining.

Member functions are classified by constness: a `const` member function does not invalidate, a non-`const` member function is assumed to invalidate, and `[[not_invalidating]]` on a non-`const` function or parameter opts out and is checked where the definition is visible. A virtual call is classified by the constness of the declared function.

Ownership is tracked with `[[owner]]`. The checker requires that an owner is consumed exactly once on every path: not leaked, not deleted twice, not read after it is consumed, and not passed to two owner parameters in one call. A fresh `new T` is an owner. An owning factory is written with `[[owner]]` on the function declaration; returning a fresh allocation from an unmarked function is rejected. `std::as_owner` adopts a pointer that the analysis cannot see is an owner, for example one obtained through an indirect call.

The escape hatches are library functions in `<utility>`: `std::no_dangling` for a pointer stored into a returned object, where the storing site is the only place the compiler can see the store; `std::now_valid` for manual revalidation of one object; and, since 2026-10-08, `std::no_escape` for reference-binding false positives.

7.2 Nineteen adversarial cases

To test the profile against the cases most often named as the ones a borrow checker solves and C++ cannot, we ran a set of nineteen: escaping stack addresses (directly, inside a returned `vector<int*>`, a returned struct with a reference member, a returned lambda capturing by reference, and through an unannotated local), use-after-free (straight-line, conditional, inside a callee, double delete, leak, the same code with no annotations, and `unique_ptr::get()` followed by `reset()`), container invalidation (iterator and raw pointer into a `vector` grown in a callee, `vector::data()` across `push_back`, `string::data()` and `string::c_str()` across `append`, and a `map` reference across `insert`), and a raw `new` returned from a function.

On 2026-09-24 GCC rejected seventeen of them. One apparent miss was a bug in our test, a `(void)*p` that is a discarded-value expression and never reads; with a real read it was always rejected. The other was real: there was no way to write an owning factory, so `int* f(){ return new int{9}; }` compiled clean. Ville added the rule the same day (commit [03f0b3c](#)), and on 2026-09-25 GCC rejected all nineteen. Clang rejects none of them, because it has no invalidation profile. Appendix A gives every case with a link.

The factory fix came with fifteen cases that pin down the rule set, also in Appendix A. `[[owner]] int* g(){ return new int{42}; }` is accepted; `int* [[owner]] f()` warns that the attribute does not apply to types and is then rejected as an unmarked factory. A marked function returning a non-owner is rejected. A caller receiving into an `[[owner]]` and deleting is accepted; receiving into a plain pointer is rejected, as a leak if never deleted and as a delete of a non-owner if deleted.

```
[[profiles::enforce(std::invalidation)]];

int* f() { return new int{9}; }           // error: unmarked
owning factory

[[owner]] int* g() { return new int{42}; } // OK

void use() {

    int* p [[owner]] = g();

    delete p;                             // OK

    int* q = g();                           // error: owner into a
plain pointer, leaked

}
```

7.3 What implementation taught the design

Three findings from the GCC work bear directly on the open questions between P3446R0 and P4296R0.

First, a sound owner discipline needs path-sensitive may-analysis across branch merges. On 2026-09-28 the branch fixed a soundness hole: `if (0 < x) delete p; *p = 10;` compiled clean, because the "consumed" diagnostics asked whether the owner was consumed on every path, and at a merge where one arm deletes and the other does not that test is silent. The fix (commit [accefed](#)) adds a forward "may already be consumed on some path" analysis. That fix then rejected the correct `if (c) delete p; if (!c) delete p;`, which commit

[b346dfa](#) accepts again by recognizing a condition that is the exact inverse of a dominating condition on the same operand, and deliberately stopping short of relational reasoning across variables. The same class of dominance-only unsoundness had already been fixed in the Rule #0 and Rule #1 tracking. P3446R0 avoids flow analysis; the implementation shows that zero false negatives requires some local flow analysis, and that the specification should state the rules as may-rules on the source program with a stated minimum for branch correlation.

Second, the analysis must ignore what the implementation generates. Commit [445a281](#) had to exclude compiler-synthesized scope-exit clobbers and the C++26 erroneous-behavior deferred-initialization placeholder from counting as owner reassignment. A specification written against the source program, not against an intermediate representation, keeps that a quality-of-implementation matter.

Third, the cost of the conservative default is measurable. Because a non-`const` member function is assumed to invalidate, `int* p = &v[0]; auto it = v.begin(); *p = 4;` is rejected, although nothing was invalidated. The remedy is `[[not_invalidating]]` on `begin()`, `end()`, `data()`, `operator[]`, `front()`, `back()` and similar. A count over the standard containers and pointer wrappers finds 286 non-const member function overloads that do not invalidate against 364 that do, counting each overload because each carries its own attribute (M. Bracco, personal communication, count of 2026-10-16, over 24 standard containers and wrappers from vector to variant). The split follows the data structure: vector, string, the unordered containers and the smart pointers are three quarters or more invalidating, while list, forward_list, set and map are mostly not, and array, span, string_view and weak_ptr never are. So the annotation is written once by library implementers, and flipping the default would raise the count from 286 to 364. Range-based `for` is the open case: by its specification, `for (auto& x : v)` calls `v.begin()` and `v.end()` before the body runs, so the analysis must know those calls do not invalidate for `x` to be usable.

The most recent work, on 2026-10-07 and 2026-10-08, closed two false negatives found by checking the implementation against a written set of lifetime rules: a chained call whose intermediate result is never named (`v.at(0).push_back(...)`), and a container constructed directly in the caller's return slot and then mutated through `push_back`, where an escaping local address could slip through.

7.4 Other invalidation work

The 6it team (Ignatchenko, Davidson, Ivanchykhin, Bracco) has a separate prototype on the P4296R0 design that was demonstrated on 2026-08-27. It is not yet publicly runnable, so its results are not reproduced here; 6it has confirmed that it may be cited, and a Compiler Explorer build is planned.

9. Same meaning, cost, and test coverage

The "same meaning" property is that a profile adds checking and changes nothing else. On Clang it is demonstrated directly: the LLVM IR is byte-identical whether `std::init` is enforced, previewed as warnings, absent, or disabled, and whether or not the markers are written, at `-O0` and at `-O2`. The profile emits no runtime code [Stasiowski-2026-10, section 4]. On both compilers the attributes are inert with the profile off, and a program using them compiles unchanged.

Compile-time cost on Clang, compared with a plain compile: including `<bits/stdc++.h>` costs 2 to 4 percent with system headers exempt, which is the default, and 11 percent with the standard library itself checked. A file of 300 functions using marked spans, `unique_ptrs` and iterators costs 5 percent. A synthetic file of 200 classes dense with `[[uninit]]` members costs 14 percent.

False positives on unmarked code: none, on Clang, across 45 common idioms over `span`, `unique_ptr`, `shared_ptr`, `weak_ptr`, iterators, `string_view`, ranges views, containers, `optional` and `function`, with both `libstdc++` and `libc++`.

Tests: Clang has 62 dedicated test files with about 1,800 checked diagnostics, and its full test suite passes. GCC's `g++.dg/profiles` testsuite recorded 1,512 passes as of 2026-09-28, alongside 4,306 passes in the contracts suites with 2 known expected failures. Both implementations document their limitations: Clang lists 21 deliberate strictnesses and 43 missed diagnostics for `std::init`, and 6 framework gaps [Stasiowski-2026-10, section 5].

10. What the papers should take from the implementations

For P4222 (initialization): settle whole-object versus member-wise completion of an `[[uninit]]` class object, on which the R3 draft and the GCC branch now differ; settle one spelling of the callee-initializes contract (`[[must_init]]` on the parameter or `[[now_init]]` on the function) and one spelling of the bridge (attribute or function); decide whether `[[ref_to_uninit]]` applies to references; record `[[now_uninit]]` or its equivalent for `destroy_at`; resolve the twelve examples that conflict with the paper's own rules; and state where the markers go, since both implementations accept only declaration positions.

For P3589 (framework): require a diagnostic for an enforced profile the implementation does not know; specify rule names as part of each profile so that rule-level suppression is portable; and consider recommending `-fprofiles-enforce=` and default system-header exemption, on which both implementations already agree.

For invalidation: specify the rules as may-rules over the source program, with a stated minimum for branch correlation; adopt the declaration-position `[[owner]]` as the owning-factory spelling, which is the one spelling that works today; record that `[[invalidating]]` was considered and rejected as restating the default; and specify range-based `for` and algorithm iteration under that default.

11. What is not implemented anywhere

Invalidation exists on one compiler. Clang has none of it. The exposure analysis of P4222R2 section 5.3 and overloading on `[[ref_to_uninit]]` (section 4.8) are implemented nowhere. Of the profiles on the list Bjarne Stroustrup presented at CppCon 2026, two have implementations: initialization and invalidation. Ranges and bounds, type, arithmetic, casting, concurrency, RAII, unions, pointers and algorithms exist as design, not as code. We say this plainly because the strongest version of this report is the accurate one.

12. How to reproduce

Every result in this paper can be reproduced on Compiler Explorer with the two compilers named in section 3. On Clang the Compiler Explorer default is `-std=c++26 -fprofiles`, so a snippet needs only its `[[profiles::enforce]]` line. On GCC, use `-std=c++23 -pedantic-errors`. On both, `[[profiles::enforce(...)]]`; is an empty-declaration attribute and must come before any other declaration, including those from `#include`.

Example	Compiler	Link
Enforce and rule-level suppress	Clang	https://godbolt.org/z/cos3Yh5jI
Definite assignment, static runtime initialization	Clang	https://godbolt.org/z/qz1s9ex7h
<code>[[ref_to_uninit]]</code> in both directions	Clang	https://godbolt.org/z/afGzx9njg
Lifecycle: double <code>construct_at</code> , double <code>destroy_at</code>	Clang	https://godbolt.org/z/WrZrEnjvY
Owning factory, after the fix	GCC	https://godbolt.org/z/dzaWPrheg

Example	Compiler	Link
The nineteen adversarial cases and the factory rule set	GCC	Appendix A

Compiler Explorer short links are snapshots of one editor state. If a compiler has advanced since a link was made, recompile it.

13. Disclosure of AI use

The first author prepared this paper with the help of an AI assistant. He wrote the outline and the measurement plan, designed, built and ran the dual-compiler and adversarial suites; the assistant drafted most of the prose from his implementation notes and from the implementers' reports and proposed edits that he accepted, rejected or rewrote. The second author's Clang progress and status reports, including [Stasiowski-2026-09] and [Stasiowski-2026-10] cited here, were written with an AI assistant, with his feedback and review. He is the author of the Clang implementation of profiles and used an AI assistant in developing it. The authors reviewed every sentence before submission. The conclusions, and the responsibility for any errors, are the authors' own.

14. Acknowledgements

Bjarne Stroustrup designed the initialization and invalidation profiles and Gabriel Dos Reis the framework. The C++ Alliance funds the Clang implementation. Sherry Ignatchenko, Guy Davidson, Dmytro Ivanchykhin and Marcos Bracco of 6it wrote P4296R0 and its prototype, and Marcos Bracco supplied the library annotation count in section 7.3. The participants in the Profiles implementation calls reviewed many of these results as they came in. Matt Godbolt and the Compiler Explorer team host both compilers. Bjarne Stroustrup, Krystian Stasiowski and Marcos Bracco commented on the draft of this revision, and Sherry Ignatchenko agreed to the mention of the 6it prototype.

15. References

[P3589R2] Gabriel Dos Reis, "C++ Profiles: The Framework".

[P4222R2] Bjarne Stroustrup, "An initialization profile", 2026-07-11.

[P3446R0] Bjarne Stroustrup, "Profile invalidation, eliminating dangling pointers", 2024-10-14.

[P4296R0] Sherry Ignatchenko, Guy Davidson, Dmytro Ivanchykhin, Marcos Bracco, "Default-Deny + Provable-Whitelist Invalidation (Safer Than Borrow-Checker in the Long Run)".

[Stasiowski-2026-10] Krystian Stasiowski, "C++ Profiles in Clang: Implementation Status", 2026-10-08. <https://gist.github.com/sdkrystian/7ac165c757325eb22c4d30a34f03e67f>

[Stasiowski-2026-09] Krystian Stasiowski, Clang profiles progress report, 2026-09-03. <https://gist.github.com/sdkrystian/864fdc886e41d4a6fe410645ed7b11d8>

[Voutilainen-p4324] Ville Voutilainen, GCC branch [p4324](#), tip [fae893e](#), 2026-10-08. <https://github.com/villevoutilainen/gcc/tree/p4324>

[cppalliance-clang] C++ Alliance Clang, [profiles-init-latest](#) prerelease. <https://github.com/cppalliance/clang/releases/tag/profiles-init-latest>

[P3589R3] Gabriel Dos Reis, "C++ Profiles: The Framework", P3589R3, 2025-12-01. The implementations were checked against R2; R3 renames the subclauses.

[P4222R3] Bjarne Stroustrup, "An initialization profile", P4222R3, 2026-10-22, in this mailing.

[P4418R0] Bjarne Stroustrup, "An invalidation profile", P4418R0, 2026-10-22, in this mailing.

[P4416R0] Michael Wong, Bjarne Stroustrup, "Proposed wording for the initialization profile", P4416R0, 2026-10-22, in this mailing.

[P4417R0] Michael Wong, Bjarne Stroustrup, "Proposed wording for the invalidation profile", P4417R0, 2026-10-22, in this mailing.

Appendix A. The adversarial and owning-factory cases

All results are GCC, `std::invalidation` enforced, compiled 2026-09-25 on build [abfe4d8](#) and re-run nightly since; still unchanged on 2026-10-06. Clang accepts every case, because it has no invalidation profile.

#	Case	GCC	Link
A1	<code>return &x</code> of a local (also caught by <code>-Wreturn-local-addr</code>)	rejected	https://godbolt.org/z/nPr7aW17
A2	<code>&x</code> escapes inside a returned <code>vector<int*></code>	rejected	https://godbolt.org/z/84PcsK1ov

#	Case	GCC	Link
A3	escapes inside a returned struct with a reference member	rejected	https://godbolt.org/z/hYvrhhrPr
A4	escapes inside a returned lambda capturing by reference	rejected	https://godbolt.org/z/hnjEEjoYs
A5	<code>int* p = &x;</code> <code>return p;</code> with no annotations	rejected	https://godbolt.org/z/ohMKbW9Go
A6	<code>delete p;</code> <code>*p = 9;</code>	rejected	https://godbolt.org/z/aTzjxdTjf
A7	conditional delete, then use	rejected	https://godbolt.org/z/4qP463zfT
A8	delete inside a callee, use after it returns	rejected	https://godbolt.org/z/GscKT9W3K
A9	double delete	rejected	https://godbolt.org/z/4aa3c43eh
A10	<code>[[owner]]</code> never released	rejected	https://godbolt.org/z/7js4cGcxE
A11	same code with no annotations	rejected	https://godbolt.org/z/1Mhc5drad
A12	<code>unique_ptr::get()</code> , then <code>reset()</code> , then use	rejected	https://godbolt.org/z/zbfM49Ec6
A13	iterator into a <code>vector</code> grown in a callee	rejected	https://godbolt.org/z/jqqrvrKnB
A14	raw pointer to <code>v[0]</code> , <code>vector</code> grown in a callee	rejected	https://godbolt.org/z/ah6raszEY

#	Case	GCC	Link
A15	<code>vector::data()</code> , then <code>push_back</code>	rejected	https://godbolt.org/z/9hnPhs3Wq
A16	<code>string::data()</code> , then <code>append</code>	rejected	https://godbolt.org/z/hYvdP9x8E
A17	<code>map</code> reference held across <code>insert</code>	rejected	https://godbolt.org/z/PPcacWd5s
A18	<code>string::c_str()</code> , then <code>append</code> , then read	rejected	https://godbolt.org/z/eYTfddsvf
F1	<code>int* f(){ return new int{9}; }</code>	rejected	https://godbolt.org/z/Wb1WM8Edq
F2	<code>[[owner]] int* g(){ return new int{42}; }</code>	accepted	https://godbolt.org/z/ore69r8az
F3	marked function returns an <code>[[owner]]</code> local	accepted	https://godbolt.org/z/Pen5ah8jh
F4	marked function returns a non-owner	rejected	https://godbolt.org/z/r54P1Kqo5
F5	<code>int* p [[owner]] = g(); delete p;</code>	accepted	https://godbolt.org/z/qeh689a7x
F6	<code>delete std::as_owner(p) ;</code>	accepted	https://godbolt.org/z/fsejfn5GG
F7	<code>int* p = g();</code> never deleted	rejected	https://godbolt.org/z/67j5WvfPo
F8	<code>int* p = g();</code> <code>delete p;</code>	rejected	https://godbolt.org/z/zG4jM88hr
F9	non-owner into <code>[[owner]]</code> without <code>as_owner</code>	rejected	https://godbolt.org/z/aEdv43qgb

#	Case	GCC	Link
F10	<code>int* [[owner]] f()</code> , type position	rejected	https://godbolt.org/z/bMKTTnqMj
F11	<code>[[owner]]</code> local returned from an unmarked function	rejected	https://godbolt.org/z/cMrssfdhy
F12	unmarked factory, caller receives into <code>[[owner]]</code>	rejected	https://godbolt.org/z/jfbqxr46T
F13	<code>int* p = new int{7};</code> never deleted	rejected	https://godbolt.org/z/sKPh4YPW7
F14	<code>int* p [[owner]] = new int{7}; delete p;</code>	accepted	https://godbolt.org/z/Pbzcasz93
D1	pointer stored into a returned object	rejected, by design	https://godbolt.org/z/Ta4WWrsha
D2	the same, through <code>std::no_dangling</code>	accepted	https://godbolt.org/z/sfecMTT9T

Two controls stay in the suite and must be accepted by both compilers, because a discarded-value expression is not a read: `(void)*p` after `c_str()` and `append`, and the same after `vector::data()` and `push_back`.