

Document #:	P4416R0
Date:	2026-10-23
Project:	Programming Language C++
Audience:	SG23, CWG, LWG
Authors:	Michael Wong (fraggamuffin@gmail.com); Bjarne Stroustrup (bjarne@stroustrup.com)
Special Note:	Information only; not yet ready for review

Proposed wording for the initialization profile

Note: This paper is proposed working-draft wording only. The design, its rationale and its examples are in "An initialization profile", P4222R3 [BS'26b]; read that first. The wording tracks the P4222R3 draft of 2026-10-13. Where the two differ, the design paper governs and a drafting note here says where. The wording will be revised after the design is approved and before it is handed to CWG and LWG.
--

Abstract

This paper supplies proposed working-draft wording for the initialization profile described in the design paper "An initialization profile", P4222R3 [BS'26b]. The design paper states what the profile guarantees and why; this paper states the rules in the form the working draft needs, so that the two can be reviewed together and the wording can be handed to CWG and LWG once the design is approved. The wording is written against the current C++29 working draft N5054 and the profile framework of P3589R3. It also proposes the clause [profiles] and its general subclause under which all standard profiles would be specified, and offers the layout used here as a common style for specifying them. A companion paper, "Proposed wording for the invalidation profile" [MW'26c], does the same for the invalidation profile and shares its clause structure with this one.

1. Revision history

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

2. Introduction

The initialization profile prevents the use of an object before it has been initialized. The design paper [BS'26b] gives its guarantees as: every object is initialized before it is used; an object is constructed once only and destroyed once only; every static object is initialized before use; and the checking is local to each function body, so that no complex flow analysis is required and all conforming implementations accept and reject the same programs. It delivers them with a small set of rules, four attributes, `[[uninit]]`, `[[ref_to_uninit]]`, `[[must_init]]` and `[[now_init]]`, and one library function, `now_init()`.

The wording below follows the method of the design papers: the guarantees are specified first, as properties of a well-formed program within the dominion of the profile; every rule is tagged with the guarantee it delivers; and a table in section 10 checks the rules against the guarantees in both directions. The intent is that a reader can see, for every "shall", which guarantee justifies it, and for every guarantee, which rules deliver it. The rules were checked against the two experimental implementations [Impl1], [Impl2], whose status and the experience with them are reported in [MW'26b]; the place of this profile among the profiles is described in [BS'26]. This answers directly the three requests of the design paper's section "How to specify this in the standard?": that the guarantee be specified and not only the places affected; that the list of affected places be given in addition; and that there be a common style for specifying standard profiles, which section 11 extracts from the layout used here.

3. Scope and conventions

The wording is drafted against N5054, the C++29 working draft of 2026-07-16 (editors' report N5055), and assumes the profile framework of P3589R3, section 2: the attributes `profiles::enforce`, `profiles::require`, `profiles::suppress` and `profiles::exempt` in the subclauses `[decl.profile.enforce]`, `[decl.profile.require]`, `[decl.profile.suppress]` and `[decl.profile.exempt]`; the grammar *profile-designator*, *profile-name* and *profile-argument* added to `[decl.attr.grammar]`; the term *dominion* for the region of program text in which a profile is enforced; the statement that a profile is a set of additional language restrictions applied after translation phase 7; and the framework's change to `[intro.compliance.general]` by which failure to satisfy a constraint dictated by a profile is a diagnosable rule. Consequently every "shall" below is a profile rule whose violation is a violation of a diagnosable rule within the dominion of the initialization profile, and the implementation issues a diagnostic; nothing below is "no diagnostic required" except where `[dcl.attr.indet]` already uses that form for cross-translation-unit attribute mismatches, which is copied. The *profile-name* used is `std::init`, following the `std::` prefix that P3589R3's examples use for standard profiles (`std::type`, `std::lib::hardened`); P3589R3 does not list the standard profiles and P3402R3 [LLG'25] used `std::initialization`, so the final spelling is for the framework author and the committee. Subclause numbers follow N5054 and are given only for orientation; stable names are authoritative. Labels marked † are new.

Typography follows the working draft: code and attribute tokens in monospace; comments in code in *italic*; grammar terms, defined terms, placeholders and the labels of library elements (*Constraints*, *Effects*, *Returns*, *Remarks*) in *italics*; notes and examples with italic labels in roman brackets; no bold in body text. Every code block is held to 72 columns, with a comment placed on its own line above the statement it describes whenever the pair would not fit. Editorial instructions are in *italics*. Drafting notes in square brackets are addressed to reviewers and are not proposed text.

The source for every rule is the design paper: the sections "What is initialization?", "Definite assignment", "Static analysis: No complex flow analysis", "Static objects", "[[uninit]] rules", "Passing uninitialized memory around", "void*", "Library support", "Templates and [[uninit]]", "Lifetimes", "Existing support for uninitialized", "Class objects" and its subsections, "Summary", "How to specify this in the standard?", "Early draft standard text" and the appendix on attribute placement. Where this paper states a rule more precisely than the design paper, or supplies a rule the design paper only shows by example, a drafting note says so. Nothing in this wording specifies diagnostics or their text, in keeping with P3589R3, which makes profile violations diagnosable rules under [intro.compliance.general] and leaves message quality to implementations; P3589R3's named-rule suppression (the `rule:` argument of `profiles::suppress`) is implementation-defined, so no rule names are proposed here either. If named rules are wanted for `std::init`, they belong in the framework paper first.

4. Guarantees: new clause Profiles [profiles] †, subclauses General [profiles.general] † and Initialization profile [profiles.init] †

Add a new clause holding one general subclause and one subclause per standard profile. P3589R3 adds [decl.profile.enforce], [decl.profile.require], [decl.profile.suppress] and [decl.profile.exempt] but no home for the profiles themselves; a clause after Clause 33 [thread], or an annex, are both workable and the text below moves unchanged.

34 Profiles [profiles] †

34.1 General [profiles.general] †

1 This Clause specifies the standard profiles ([decl.profile.enforce]). Each standard profile is specified in one subclause of this Clause, which names the profile by its *profile-name*, states the properties that a well-formed program has within the dominion of the profile, and states the rules by which those properties are delivered.

2 The rules of a standard profile are constraints dictated by that profile ([intro.compliance.general]). They apply to the declarations and to the potentially-evaluated expressions and statements within the dominion of the profile ([decl.profile.enforce], [decl.profile.require]), except within the dominion of a suppression ([decl.profile.suppress]) or an

exemption ([decl.profile.exempt]) naming the profile. [*Note 1*: A violation of a rule of a standard profile within its dominion is a violation of a diagnosable rule, and an implementation issues a diagnostic. — *end note*]

3 The rules of a standard profile do not change the meaning of a program. [*Note 2*: A program that is well-formed within the dominion of a standard profile has the same meaning when the profile is not enforced. The attributes a profile introduces are information for the profile and have no effect on the type of any entity, on overload resolution, or on template argument deduction. — *end note*]

[Drafting note: this subclause states once what every standard profile would otherwise restate: where the rules apply, that they are diagnosable, and that they are semantics-preserving. Its content is taken from P3589R3 section 2 and from the shared General paragraphs of this paper and of [MW'26c]; if the framework paper supplies an equivalent subclause, this one is withdrawn and the profile subclauses cite the framework's.]

34.x Initialization profile [profiles.init] †

34.x.1 General [profiles.init.general]

1 The initialization profile is the standard profile with *profile-name* `std::init` ([decl.profile.enforce]).

2 Within the dominion of the initialization profile, a well-formed program has the following properties:

(2.1) — [G1] every object is initialized before it is used ([profiles.init.use]);

(2.2) — [G2] no object is initialized while it is initialized, and no object is destroyed while it is not initialized ([profiles.init.life]);

(2.3) — [G3] every non-local variable with static or thread storage duration has static initialization ([basic.start.static], [profiles.init.decl]).

3 [G4] Whether a program satisfies the rules of this subclause is determined for each function body separately, from the program text of that body and from the declarations it refers to, without regard to the values of expressions, the number of times a statement is executed, or the definitions of called functions ([profiles.init.analysis]). [*Note 1*: Two conforming implementations therefore accept and reject the same programs under this profile. — *end note*]

4 [G5] The rules of this subclause do not change the meaning of a program. [*Note 2*: A program that is well-formed within the dominion of the initialization profile has the same meaning when the profile is not enforced, and the attributes of [dcl.attr.uninit], [dcl.attr.reftouninit],

[dcl.attr.mustinit] and [dcl.attr.nowinit] have no effect on the type of any entity, on overload resolution, or on template argument deduction. — *end note*

5 The rules of this subclause apply to declarations and to potentially-evaluated expressions and statements within the dominion of the initialization profile, except within the dominion of a suppression ([dcl.profile.suppress]) or an exemption ([dcl.profile.exempt]) naming it. The attributes `uninit`, `ref_to_uninit`, `must_init` and `now_init` on declarations outside the dominion are taken into account; a declaration outside the dominion without these attributes is treated as declaring initialized objects and as referring to initialized objects. [Note 3: A function declared without these attributes is assumed to require initialized arguments and to return initialized objects, whatever its definition does. — *end note*]

6 [Note 4: The rules of [profiles.init.terms] through [profiles.init.class] are stronger than what [G1] through [G3] alone require, so that they can be checked under [G4]. Each rule is annotated with the guarantee it delivers. — *end note*]

[Drafting note: paragraph 2 is the design paper's "Guarantees" list made normative. "Constructed once only" and "destroyed once only" are stated as two prohibitions in (2.2) because an object that is destroyed and whose storage is reused is constructed again legitimately. The design paper's guarantee that only local, every-path analysis is used is a property of the specification rather than of programs, so it is paragraph 3 rather than a (2.x) item. Paragraph 5 is the migration rule: an unannotated declaration outside the dominion is initialized. The C allocation functions, `operator new` and `allocator::allocate` are annotated in section 8 so that `malloc` and allocator-based containers work without a separate `uninit_malloc`; if those library annotations are not accepted, this paragraph needs the `uninit_malloc` fallback from the design paper's section on `void*`. P3589R3's `profiles::exempt` gives a second route for C headers: a translation unit can exempt `<stdlib.h>` from `std::init` and keep the annotations out of the header entirely; the wording accommodates both.]

5. Terms and analysis model

34.x.2 Terms [profiles.init.terms]

1 For the purposes of this subclause, each object has at each point in a function body an *initialization state*, which is one of *initialized*, *uninitialized*, or *indeterminate*, and each pointer, reference, or object of class type at each such point either *refers to uninitialized objects* or *refers to initialized objects*. Both are properties of the program text determined by this subclause; neither is a property of an object at run time. [Note 1: The initialization state conservatively tracks whether the lifetime of an object has begun ([basic.life]). It does not alter the value an object holds; an object whose storage holds an erroneous value ([basic.indet]) is uninitialized here only if the rules say so. — *end note*]

2 The initialization of an object *o* leaves *part of o uninitialized* if *o*, or a subobject of *o*, is default-initialized and no initialization is performed for it ([dcl.init.general]). [Note 2: This is the case for an object of scalar type without an *initializer*, for an array of such objects, and for a class object whose default constructor is not user-provided and which has such a member without a default member initializer. It is not the case for a class object with a user-provided default constructor, even one with an empty body; such a constructor is constrained by [profiles.init.class]. — *end note*]

3 [G1] An object is *uninitialized at its definition* if

(3.1) — the attribute `uninit` ([dcl.attr.uninit]) appertains to it;

(3.2) — it is designated by indirection, subscript, or class member access ([expr.unary.op], [expr.sub], [expr.ref]) through a pointer, reference, or object of class type that refers to uninitialized objects;

(3.3) — it is created by a *new-expression* ([expr.new]) whose initialization leaves part of it uninitialized; or

(3.4) — it is a subobject of an uninitialized object.

Otherwise it is *initialized at its definition*. [Note 3: An object with an *initializer*, an object default-initialized by a constructor, a zero-initialized object with static or thread storage duration ([basic.start.static]), a parameter, and a temporary object are initialized at their definition. — *end note*]

[Drafting note: the design paper says that a dynamically created object "with a default initializer or a default constructor is considered initialized" but does not say what `new int` yields. (3.3) and (6.3) below make the result of a *new-expression* whose initialization leaves part of the object uninitialized refer to uninitialized objects, so `int* p = new int;` is ill-formed and `int* p [[ref_to_uninit]] = new int;` is well-formed. This is the only reading consistent with the "neither or both" rule for pointers in [profiles.init.ref].]

4 [G1, G2] An *initialization* of an object *o* is any of the following:

(4.1) — a simple assignment ([expr.assign]) whose left operand designates *o* as a whole, where *o* is of scalar type;

(4.2) — a call to `std::construct_at` or `std::ranges::construct_at` ([specialized.construct]) whose first argument points to *o*;

(4.3) — a *new-expression* with a *new-placement* that designates the storage of *o* and whose initialization does not leave part of *o* uninitialized;

(4.4) — a function call in which *o*, a pointer to *o*, or a reference to *o* is the argument for a parameter to which the attribute `must_init` ([`dcl.attr.mustinit`]) appertains;

(4.5) — a call to `std::now_init` ([`specialized.now`]) whose argument points to *o*; or

(4.6) — for an array *o*, an initialization of every element of *o*.

After an initialization of *o*, *o* and its subobjects are initialized.

[Drafting note: this list does not treat `memcpy`, `memmove` or `start_lifetime_as` as initializations. For trivially copyable types [`basic.types.general`] makes a byte copy value-preserving, so an argument can be made that `memcpy` into uninitialized storage should count as initializing it. Nothing in the design paper asks for it, so it is left for a later revision; it is noted here so that the omission is a decision rather than an oversight.]

5 [G2] A *destruction* of an object *o* is any of the following:

(5.1) — a call to `std::destroy_at` or `std::ranges::destroy_at` whose argument points to *o*, or a call to `std::destroy`, `std::destroy_n`, or their `std::ranges` counterparts ([`specialized.destroy`]) whose range includes *o*;

(5.2) — an explicit destructor call or pseudo-destructor call ([`expr.prim.id.dtor`], [`class.dtor`]) whose object expression designates *o*; or

(5.3) — for an array *o*, a destruction of every element of *o*.

After a destruction of *o*, *o* and its subobjects are uninitialized. [*Note 4*: The implicit destruction of an object at the end of its lifetime ([`basic.life`], [`stmt.jump`], [`class.dtor`]) is not a destruction in this sense; it is constrained by [`profiles.init.life`]. — *end note*]

6 [G1] A pointer, reference, or object of class type *refers to uninitialized objects* if

(6.1) — the attribute `ref_to_uninit` or `must_init` appertains to it;

(6.2) — it is the result of a call to a function to which the attribute `ref_to_uninit` appertains;

(6.3) — it is the result of a *new-expression* whose initialization leaves part of the created object uninitialized;

(6.4) — it is the result of a conversion ([`conv.ptr`], [`conv.qual`], [`expr.static.cast`], [`expr.reinterpret.cast`], [`expr.cast`]) or of pointer arithmetic ([`expr.add`]) applied to a pointer that refers to uninitialized objects;

(6.5) — it is the result of indirection, subscript, class member access, a member function call, or a conversion function applied to an object of class type that refers to uninitialized objects, where the result is of pointer, reference, or class type; or

(6.6) — it is initialized or assigned from a pointer, reference, or object that refers to uninitialized objects.

Otherwise it *refers to initialized objects*. [Note 5: A null pointer value refers to initialized objects. A pointer of type `void*` is governed by the same items; it refers to uninitialized objects if and only if the pointer it was converted from did, and the result of converting it to another pointer type refers to uninitialized objects if and only if it did. — *end note*]

[Drafting note: (6.5) is what makes `unique_ptr<T> p [[ref_to_uninit]]`, `span<T> s [[ref_to_uninit]]`, and an iterator parameter of `uninitialized_fill` work without the profile seeing inside the class: `*p`, `s[i]`, `*first` designate uninitialized objects. The design paper lists the class types it has in mind under "Passing uninitialized memory around". Whether a class may have a `ref_to_uninit` member and also be used unannotated is a design question not settled here.]

34.x.3 Analysis [profiles.init.analysis]

1 [G4] The initialization state of an object *o* at a point *P* in a function body, and whether a pointer, reference, or class object refers to uninitialized objects at *P*, are determined from that function body alone, as follows. Let a *path* to *P* be a sequence of the statements and full-expressions of the function body that could be executed, in order, from the start of the body to *P*, where every condition is treated as able to have either value, the substatement of every iteration statement is treated as able to be executed any number of times including zero, and every function call is treated as returning normally. Then

(1.1) — *o* is initialized at *P* if on every path to *P* the last initialization or destruction of *o* before *P* is an initialization, or there is none and *o* is initialized at its definition;

(1.2) — *o* is uninitialized at *P* if on every path to *P* the last initialization or destruction of *o* before *P* is a destruction, or there is none and *o* is uninitialized at its definition;

(1.3) — otherwise, the initialization state of *o* at *P* is indeterminate.

[Note 1: No knowledge of the values of conditions, of the number of iterations of a loop, or of the body of a called function is used. A call affects the state of an argument only through the attributes on the called function's declaration ([profiles.init.ref]). — *end note*]

2 [G4] Whether a pointer, reference, or class object refers to uninitialized objects at P is determined in the same manner from [profiles.init.terms] paragraph 6, with (6.6) applied along every path; if the paths disagree, every use of it at P other than as an operand of `std::now_init` is ill-formed.

3 [G1, G4] An element of an array designated by a subscript or by indirection through a pointer whose offset from the array's first element is not a constant expression ([`expr.const`]) has the initialization state of the array as a whole if the array is initialized or uninitialized at that point, and indeterminate initialization state otherwise. [Example 1:

```
void f(int i) {
    X a [[uninit]] [3];
    // a[0] initialized; a[1], a[2] uninitialized; a indeterminate
    std::construct_at(&a[0]);
    // error: a[i] has indeterminate state
    std::construct_at(&a[i]);
    X x = a[0];           // OK
    X y = a[1];           // error: a[1] is uninitialized
}
```

— end example]

4 [G1, G2] An initialization, destruction, or use ([`profiles.init.use`]) of an object whose initialization state is indeterminate at that point is ill-formed, except as the operand of `std::now_init`. [Note 2: Code whose initialization structure cannot be expressed under these rules can be rewritten, can use `std::now_init` ([`specialized.now`]) or a function to which `now_init` appertains ([`dcl.attr.nowinit`]), or can suppress the profile ([`decl.profile.suppress`]). — end note]

[Drafting note: paragraph 1 is the design paper's single analysis rule, "a desired action must be present on every path from definition to use and an undesirable action absent on every path", with loops treated as zero-or-more. This deliberately rejects the range-for initialization example under "Definite assignment", which the design paper itself says would require the analyzer to recognize the loop; the `uninitialized_*` algorithms and `now_init` are the sanctioned routes. Paragraph 3 gives "random access to an uninitialized array must be banned" a precise form: constant offsets are tracked per element, anything else inherits the whole array's state or is indeterminate. The experimental implementations [Impl1], [Impl2] should confirm that they track per element. If they do not, paragraph 3 is replaced by "an element of an uninitialized array has indeterminate state unless the array as a whole is initialized or uninitialized", which routes all array initialization through the `uninitialized_*` algorithms and `now_init`.]

6. Rules

34.x.4 Declarations [profiles.init.decl]

1 [G1] If the initialization of a variable, or the initialization of a non-static data member for which neither a *mem-initializer* nor a default member initializer is provided, would leave part of the object uninitialized ([profiles.init.terms]), the attribute `uninit` shall appertain to the variable or member.

2 [G1, G2] If the attribute `uninit` appertains to a variable or non-static data member, the variable shall have no *initializer*, the member shall have no default member initializer and shall not be named in a *mem-initializer*, and the initialization of the object shall perform no initialization for the object and for every subobject of it. [Note 1: An object of a class type with a non-trivial default constructor, or with a member of such a type, cannot have the attribute `uninit`: part of it would be initialized. Such an object is initialized at its definition or is created in storage that refers to uninitialized objects. — *end note*]

3 [G2] A non-static data member to which the attribute `uninit` appertains shall be of a trivially destructible type.

4 [G1] The attribute `uninit` shall not appertain to a parameter or to a reference. A variable of pointer type shall have an *initializer* or the attribute `uninit`. [Note 2: A pointer may be initialized with a null pointer value. — *end note*]

5 [G3] Every non-local variable with static or thread storage duration shall have static initialization ([basic.start.static]). [Note 3: The variable is constant-initialized or zero-initialized; a `constexpr` specifier ([dcl.constinit]) makes a violation a diagnostic even outside the dominion. The order of dynamic initialization across translation units ([basic.start.dynamic]) is not expressible by local analysis. A block variable with static storage duration may have dynamic initialization, since its initialization is sequenced before its first use ([stmt.dcl]). — *end note*]

6 [Example 1:

```
int glob;                                // OK: zero-initialized
// error: zero-initialized and uninit (paragraph 2)
int glob2 [[uninit]];
// error: dynamic initialization (paragraph 5)
int glob3 = f();
struct S { int x; std::string s; };
void g() {
    // error: leaves loc uninitialized without uninit (paragraph 1)
    int loc;
    int loc2 [[uninit]];                // OK
```

```

// error: initializer together with uninit (paragraph 2)
int loc4 [[uninit]] = 4;
// error: s2 is initialized (paragraph 2)
std::string s2 [[uninit]];
// error: s3.x would be uninitialized (paragraph 1)
S s3;
// error: s4.s would be initialized (paragraph 2)
S s4 [[uninit]];
int* p1; // error: no initializer (paragraph 4)
int* p2 = nullptr; // OK
}

```

— *end example*]

[Drafting note: paragraph 3 is what makes the implicit destruction of the enclosing object checkable without non-local analysis (see [profiles.init.life] paragraph 3): under [profiles.init.class] paragraph 2 a `uninit` member stays uninitialized outside the constructors, so a member that needed destroying could not be shown to be initialized where the destructor runs. The `Vector_memory` member in the design paper's "Vector example", whose type has a user-provided destructor, does not satisfy paragraph 3 as written; and the `Label lab1 [[uninit]]` example under "Classes exposing uninitialized memory to users" violates paragraph 2 because `Label` has `std::string` members. Either the examples change or paragraph 3 is relaxed to "every constructor initializes it", which is non-local for out-of-line constructors. These are design points, not wording points; the wording follows the explicit rule statements under "[[uninit]] rules" and "Classes without constructors".]

34.x.5 Uses of objects [profiles.init.use]

1 An expression *uses* an object *o* if it

(1.1) — applies the lvalue-to-rvalue conversion ([conv.lval]) to a glvalue designating *o* or a subobject of *o*;

(1.2) — is a class member access ([expr.ref]) or subscript ([expr.sub]) whose object expression designates *o*, other than one whose result is the operand of unary `&` or the initializer of a reference that refers to uninitialized objects;

(1.3) — is an assignment, compound assignment, increment, or decrement ([expr.assign], [expr.pre.incr], [expr.post.incr]) whose left operand designates a subobject of *o*, or designates *o* itself where *o* is of class type;

(1.4) — binds a reference that refers to initialized objects to *o* or to a subobject of *o*, or initializes or assigns a pointer that refers to initialized objects from the address of *o* or of a subobject of *o*;

(1.5) — copy-initializes a parameter, a return value, or a temporary from *o*;

(1.6) — invokes a member function, including an implicitly defined copy or move operation, on *o*; or

(1.7) — applies `typeid` to a polymorphic glvalue designating *o*, or `dynamic_cast` to a glvalue designating *o*.

2 [G1] An expression that uses an object that is uninitialized at that point is ill-formed.

3 [Note 1: The following are not uses: taking the address of *o*, or binding a reference that refers to uninitialized objects to *o* ([profiles.init.ref]); an initialization of *o* ([profiles.init.terms]); and the unevaluated operand of `sizeof`, `alignof`, `decltype`, `noexcept`, or `typeid` applied to a non-polymorphic glvalue ([expr.context]). An assignment to *o* as a whole where *o* is of scalar type is an initialization, not a use. — end note]

4 [Note 2: Writing to an uninitialized object of class type is ill-formed; such an object is initialized with `std::construct_at` or a placement *new-expression*. Writing to a subobject of an uninitialized object, including a member of an uninitialized union or aggregate, is ill-formed by (1.3); an object is initialized as a whole. — end note]

5 [Example 1:

```
struct P { int x, y; };
union U { int x; std::string s; };
void f() {
    int i [[uninit]];
    int j = i;           // error: i is uninitialized
    i = 7;              // OK: initialization of i
    int k = i;          // OK
    P p [[uninit]];
    // error: member assignment to an uninitialized object (1.3)
    p.x = 1;
    std::construct_at(&p, 1, 2); // OK: initialization of p
    U u [[uninit]];
    u.x = 9;             // error: (1.3)
    std::byte buf [[uninit]] [64];
    // error: subobject of an uninitialized array (1.3)
    buf[0] = std::byte{0};
}
```

— end example]

[Drafting note: (1.3) and Note 2 state the design paper's rule that an uninitialized object is initialized as a whole, so that an assignment to one member of a `uninit` aggregate is an error rather than a partial initialization. Keeping the state of an object binary (initialized or not) is what lets paragraph 1 of [profiles.init.analysis] track objects rather than members. A relaxation under which an aggregate becomes initialized once every member has been assigned and before any part of it is odr-used would keep the state binary and remain local, at the cost of tracking per member between the assignments; it is not proposed here. The experimental GCC implementation [Impl1] accepts the relaxed form; the choice is the design paper's.]

34.x.6 Pointers, references, and calls [profiles.init.ref]

1 [G1] An object designated by indirection, subscript, or class member access through a pointer, reference, or object of class type that refers to uninitialized objects is uninitialized at that point ([profiles.init.terms]); an object so designated through one that refers to initialized objects is initialized at that point.

2 [G1] A pointer, reference, or object of class type that refers to uninitialized objects shall be initialized or assigned only from

(2.1) — the address of, or a reference to, an object that is uninitialized at that point, or a subobject of such an object;

(2.2) — a pointer, reference, or object of class type that refers to uninitialized objects; or

(2.3) — a null pointer value.

3 [G1] A pointer, reference, or object of class type that refers to initialized objects shall not be initialized or assigned from the address of, or a reference to, an object that is uninitialized at that point, nor from a pointer, reference, or object of class type that refers to uninitialized objects.

[Note 1: A pointer to uninitialized objects is converted into a pointer to initialized objects only by `std::now_init` ([specialized.now]) after the objects have been initialized, and a pointer to initialized objects is converted into one to uninitialized objects only by a destruction ([profiles.init.terms]). — end note]

4 [G1] In a function call, including the implicit object argument, the argument for a parameter that refers to uninitialized objects shall satisfy paragraph 2, and the argument for any other parameter of pointer, reference, or class type shall satisfy paragraph 3.

5 [G1] Let f be a function and p a parameter of f to which the attribute `must_init` appertains. Unless the attribute `now_init` appertains to f ([dcl.attr.nowinit]), in the definition of f every object designated through p shall be initialized at every `return` statement of f and at the end of its body. A call to f is an initialization ([profiles.init.terms]) of the objects designated through the

argument for *p*. [Note 2: For a parameter of pointer or reference to array type, or of a class type that designates a range, this is the whole array or range. — *end note*]

6 [G1] In a function to which the attribute `ref_to_uninit` appertains, the operand of every `return` statement shall refer to uninitialized objects. In any other function returning a pointer, reference, or object of class type, the operand of every `return` statement shall refer to initialized objects.

7 [G5] An object of a type that is a template parameter, or of a type that depends on a template parameter, is treated like any other object; its initialization state follows from the declaration that introduces it. [Note 3: In a template definition, a parameter of type `T*` refers to initialized objects unless `ref_to_uninit` appertains to it. A constraint ([temp.constr]) cannot observe the attributes of an argument expression, so a function that accepts uninitialized objects is a distinct function with its own declaration. — *end note*]

8 [Example 1:

```
void fill(int* p [[must_init]], int n);           // initializes p[0..n)
int* buffer [[ref_to_uninit]] (int n);           // returns uninitialized ints

void g() {
    int a [[uninit]] [10];
    // error: p refers to initialized objects (paragraph 3)
    int* p = a;
    int* q [[ref_to_uninit]] = a;                 // OK (paragraph 2)
    fill(a, 10);                                  // OK: initialization of a
    int x = a[3];                                  // OK
    int* r = buffer(4);                            // error (paragraph 3)
    int* s [[ref_to_uninit]] = buffer(4);          // OK
    int y = s[0];                                  // error: *s is uninitialized
    std::construct_at(s, 1);                       // OK: initialization of s[0]
    // OK: t refers to initialized objects
    int* t = std::now_init(s);
    void* v = s;                                    // error (paragraph 3)
    void* w [[ref_to_uninit]] = s;                 // OK
}
```

— *end example*]

34.x.7 Construction and destruction [profiles.init.life]

1 [G2] An initialization ([profiles.init.terms]) shall not be applied to an object that is initialized at that point. A destruction shall not be applied to an object that is uninitialized at that point.

2 [G2] An object with automatic storage duration whose type is not trivially destructible shall be initialized at every point at which its lifetime would end by exiting its scope ([stmt.jump.general], [stmt.dcl]), including by an exception. [Note 1: The implicit destructor call at the end of the scope is then applied to an initialized object. An object of trivially destructible type may be left uninitialized. — *end note*]

3 [G2] At the end of the body of a constructor, and at every `return` statement in it, every non-static data member that is uninitialized shall be of a trivially destructible type. [Note 2: This follows from [profiles.init.decl] paragraph 3 and is stated for completeness. — *end note*]

4 [Example 1:

```
void h(bool b) {
    X x [[uninit]];           // X has a non-trivial destructor
    if (b)
        std::construct_at(&x);
    // error: x has indeterminate state at scope exit (paragraph 2)
}

void k(X* p [[ref_to_uninit]]) {
    std::construct_at(p);     // OK: *p was uninitialized
    std::construct_at(p);     // error: *p is initialized
    std::destroy_at(p);       // OK
    std::destroy_at(p);       // error: *p is uninitialized
}
```

— *end example*]

34.x.8 Classes [profiles.init.class]

1 [G1] In the definition of a constructor, each non-static data member is initialized at the start of the constructor's *compound-statement* if it is initialized by a *mem-initializer*, by a default member initializer, or by default-initialization that does not leave part of it uninitialized; otherwise it is uninitialized there. [Note 1: By [profiles.init.decl] paragraph 1, a member that is uninitialized there has the attribute `uninit`. The *compound-statement* is then analyzed under [profiles.init.analysis]; an assignment to a `uninit` member of scalar type in the body is an initialization. — *end note*]

2 [G1, G4] In a member function other than a constructor, and in any other function, a non-static data member of an initialized object is initialized if the attribute `uninit` does not appertain to it, and uninitialized if it does; a member to which the attribute `ref_to_uninit` appertains refers to uninitialized objects. [Note 2: Whether some constructor initialized a `uninit` member is not determined by local analysis. A class that initializes such a member in its constructors and uses

it elsewhere converts it with `std::now_init`, initializes it with a placeholder value instead of marking it `uninit`, or uses a member of a type that manages its own initialization. — *end note*]

3 [G1] A member function to which the attribute `ref_to_uninit` appertains returns a pointer, reference, or object of class type that refers to uninitialized objects ([profiles.init.ref] paragraph 6); any other member function returning a pointer or reference to a member shall return one to an initialized member.

4 [Example 1:

```
class Slot {
    int x = 0;
    int y [[uninit]];
public:
    int& get_init() { return x; } // OK
    int& get_uninit [[ref_to_uninit]] () { return y; } // OK
    // error: y is uninitialized (paragraph 3)
    int& bad() { return y; }
};

struct X {
    int* p [[uninit]];
    int x;
    // OK: p is initialized in the body before its use
    X(int v) : x{v} { p = new int(v); use(p); }
    // error: p is uninitialized here (paragraph 2)
    int* get() { return p; }
};
```

— *end example*]

[Drafting note: paragraph 2 follows the design paper's "Member initializers": a `uninit` member stays uninitialized outside the constructors, which is what keeps the analysis of every member function local. The alternative, that a member is initialized everywhere once every constructor initializes it, requires looking at every constructor definition when analyzing any member function and is not proposed.]

7. Attributes: new subclauses in [dcl.attr]

Add the following four subclauses after [dcl.attr.indet] (9.13.6). The structure follows [dcl.attr.indet] and [dcl.attr.nodiscard]: attribute-token and what it may appertain to in paragraph 1, the argument-clause rule as its own sentence, the first-declaration and cross-translation-unit rule in paragraph 2, the semantic statement in paragraph 3, an example last.

No change to the grammar is proposed. An *attribute-token* is an *identifier* or an *attribute-scoped-token* ([dcl.attr.grammar]), so a new standard attribute adds a subclause, not a production; *nodiscard*, *indeterminate* and the others were added the same way. Every position the design paper uses for the four attributes is already a position where an *attribute-specifier-seq* may appear: after a *declarator-id* in a *noPtr-declarator*, which covers `int x [[uninit]]`; `int arr [[uninit]] [7]`; and `int* f [[ref_to_uninit]] (int)`; and makes the attribute appertain to the entity declared ([dcl.meaning.general]); at the start of a *parameter-declaration*, which covers an unnamed parameter; and at the start of a *simple-declaration* or *member-declaration*. The grammar for *profile-designator*, *profile-name* and *profile-argument* is supplied by P3589R3 and is not duplicated here. Two spellings in the design paper's appendix on attribute placement would need text, but not grammar: `void f2(int* [[ref_to_uninit]])` already parses, with the *attribute-specifier-seq* belonging to the *ptr-operator*, so only a sentence in [dcl.ptr] redirecting what it appertains to would be needed (see the drafting note under [dcl.attr.ref_touninit]); and `unique_ptr<int [[uninit]]>` already parses as an attribute on the *type-specifier-seq*, where it appertains to the type, which the design paper itself rules out. The one notation that would need a new production, `[[uninit]]` as an element of a braced *initializer-list*, is the one the design paper's appendix declines to propose.

9.13.x Uninitialized object attribute [dcl.attr.uninit] †

1 The *attribute-token* `uninit` specifies that an object is intentionally left without initialization. No *attribute-argument-clause* shall be present. The attribute may be applied to the definition of a variable of object type with automatic, static, or thread storage duration, or to the declaration of a non-static data member that is not a bit-field. [Note 1: An *attribute-specifier-seq* following a *declarator-id* appertains to the entity declared ([dcl.meaning.general]); the attribute is therefore written after the name, where an *initializer* would otherwise appear, or at the start of the declaration. — end note]

2 The attribute has no effect on the semantics of a program. [Note 2: The attribute is information for the initialization profile ([profiles.init]); within the dominion of that profile it marks the object as uninitialized at its definition ([profiles.init.terms]) and constrains the declaration ([profiles.init.decl]). Outside the dominion the attribute is accepted and has no effect. It does not affect what value the storage of the object initially holds ([basic.indet]), and it may be used together with the attribute *indeterminate* ([dcl.attr.indet]). — end note]

3 [Example 1:

```
int a [[uninit]];           // a is uninitialized
int arr [[uninit]] [7];    // arr is an uninitialized array
// error within the dominion of std::init ([profiles.init.decl])
int b [[uninit]] = 1;
// error within the dominion of std::init: s is initialized
std::string s [[uninit]];
```

```
struct X { int* p [[uninit]]; };
```

— *end example*]

9.13.y Reference-to-uninitialized attribute [dcl.attr.ref_touninit] †

1 The *attribute-token* `ref_to_uninit` specifies that a pointer, reference, or object of class type refers to zero or more uninitialized objects. No *attribute-argument-clause* shall be present. The attribute may be applied to

(1.1) — the declaration of a variable, a non-static data member, or a parameter of pointer type, reference type, or class type;

(1.2) — a *parameter-declaration* ([dcl.fct]) of such a type, by an *attribute-specifier-seq* at its start; or

(1.3) — the declaration of a function whose return type is a pointer type, reference type, or class type, in which case the attribute specifies the property for the result of a call to the function.

2 If the attribute is applied to a function or to a parameter of a function, it shall appear on the first declaration of the function in a translation unit. If a function or a parameter of a function is declared with the attribute in one translation unit and the same function or parameter is declared without it in another translation unit, the program is ill-formed, no diagnostic required.

3 The attribute has no effect on the semantics of a program. [*Note 1: The attribute is information for the initialization profile ([profiles.init]). It does not affect the type of the entity, overload resolution, or template argument deduction. Within the dominion of the profile the entity refers to uninitialized objects ([profiles.init.terms]) and the rules of [profiles.init.ref] apply. — end note*]

4 [Example 1:

```
void f(X* p [[ref_to_uninit]]);           // *p is uninitialized on entry
void g([[ref_to_uninit]] X*);             // same, unnamed parameter
// the result refers to uninitialized ints
int* alloc [[ref_to_uninit]] (std::size_t n);
auto alloc2 [[ref_to_uninit]] (std::size_t n) -> int*;
struct Pool { std::byte* buf [[ref_to_uninit]]; };
std::span<int> sp [[ref_to_uninit]] = make_slots(n);
```

— *end example*]

[Drafting note: the wording uses the positions the current attribute grammar supports: after the *declarator-id* for named entities and functions, and at the start of a *parameter-declaration* for

unnamed parameters. The design paper's appendix on attribute placement prefers `void f2(int* [[ref_to_uninit]])`, with the attribute after the `*`. Under `[dcl.ptr]` an *attribute-specifier-seq* in that position appertains to the pointer type, which is exactly what the design paper says it must not do; adopting that spelling needs one sentence in `[dcl.ptr]` redirecting what it appertains to, not a grammar change. The appendix's `unique_ptr<int [[uninit]]> up;` appertains to the type and is not supported by any text here. Implementer input, which the appendix itself asks for, decides this.]

9.13.z Must-initialize attribute `[dcl.attr.mustinit]` †

1 The *attribute-token* `must_init` specifies that a function initializes the objects referred to by one of its parameters. No *attribute-argument-clause* shall be present. The attribute may be applied to the declaration of a parameter of pointer type, reference type, or class type, or to such a *parameter-declaration* by an *attribute-specifier-seq* at its start. The attributes `must_init` and `ref_to_uninit` shall not both appertain to the same parameter.

2 A parameter to which the attribute `must_init` appertains is treated as if the attribute `ref_to_uninit` also appertained to it (`[dcl.attr.ref_touninit]`); paragraph 2 of that subclause applies.

3 The attribute has no effect on the semantics of a program. [*Note 1*: Within the dominion of the initialization profile, the definition of the function is required to initialize what the parameter refers to, and a call to the function is an initialization of the argument (`[profiles.init.ref]`). — *end note*]

4 [*Example 1*:

```
// after a call, *p is initialized
void init(X* p [[must_init]], int v);
```

— *end example*]

9.13.w Initializing function attribute `[dcl.attr.nowinit]` †

1 The *attribute-token* `now_init` specifies that a function initializes the objects referred to by its parameters by means not expressed in its definition. No *attribute-argument-clause* shall be present. The attribute may be applied to the declaration of a function at least one of whose parameters has the attribute `ref_to_uninit` or `must_init`.

2 A function to which the attribute `now_init` appertains is treated as if the attribute `must_init` appertained to each of its parameters to which the attribute `ref_to_uninit` appertains, except that the definition of the function is not required to initialize the objects they refer to (`[profiles.init.ref]` paragraph 5). Paragraph 2 of `[dcl.attr.ref_touninit]` applies to the attribute.

3 The attribute has no effect on the semantics of a program. [*Note 1: The attribute asserts, without verification by the initialization profile, that the function initializes its uninitialized arguments. It is the function-level counterpart of `std::now_init` ([specialized.now]) and is intended for existing initialization functions whose bodies cannot be checked under [profiles.init.analysis]. — end note]*

4 [Example 1:

```
// a call initializes *p
int read [[now_init]] (std::byte* p [[ref_to_uninit]], int max);
```

— end example]

[Drafting note: the design paper names a function `now_init()` under "Existing support for uninitialized" and a function attribute `[[now_init]]` in its appendix on attribute placement ("void fct1 [[now_init]] (int* p [[ref_to_uninit]]) is what is needed to fit existing initialization functions into the validation framework"). Both are specified here. The attribute is defined by canonical equivalence to `must_init` with the body check waived, so there is exactly one rule for "a call initializes the argument" ([profiles.init.ref] paragraph 5) and the only difference is whether the body is checked. The design paper's statement that the library support consists of a single function is honored: there is no `now_uninit`, and destruction is expressed only through `destroy_at` and its relatives and through explicit destructor calls. The design paper does not say what `now_init` means for a function with several `ref_to_uninit` parameters; "all of them" is the only reading that needs no argument clause and is the one taken here. If selectivity is wanted, `[[must_init]]` on the chosen parameters plus `[[profiles::suppress(std::init)]]` on the definition expresses it with existing machinery and the attribute can be dropped.]

8. Library wording

8.1 Header `<cstdlib>` synopsis [cstdlib.syn] and C library memory allocation [c.malloc]

In the synopsis, change the declarations of the allocation functions as follows:

```
void* aligned_alloc [[ref_to_uninit]] (size_t alignment, size_t size);
void* calloc(size_t nmemb, size_t size);
void* malloc [[ref_to_uninit]] (size_t size);
void* realloc [[ref_to_uninit]] (void* ptr, size_t size);
```

Add a paragraph to [c.malloc]:

Remarks: For the purposes of the initialization profile ([profiles.init]), the storage returned by `aligned_alloc`, `malloc`, and `realloc` is uninitialized and the storage returned by `calloc` is initialized. [Note: The contents preserved by `realloc` are treated as uninitialized because the profile cannot determine how much of the new storage they occupy. — end note]

[Drafting note: the design paper's section on `void*` prefers annotating `malloc` in place and names `uninit_malloc` only as the fallback if the C headers cannot be changed. Since [cstdlib.syn] is the C++ synopsis, the attribute can be added there regardless of `<stdlib.h>`; the implementation then has to make `<cstdlib>` agree. If that is judged infeasible, replace this edit by `void* uninit_malloc [[ref_to_uninit]] (size_t size);` with *Effects*: Equivalent to `return malloc(size);`. `realloc` is annotated `ref_to_uninit` conservatively, so a program that relies on the preserved prefix has to pass it through `now_init`; the alternative is to leave `realloc` unannotated and document it as a hole in the profile. The rest of this section is unaffected either way.]

8.2 Allocators [allocator.members], [allocator.traits.members], [allocator.requirements.general]

In [allocator.members], change the declaration of `allocate`:

```
constexpr T* allocate [[ref to uninit]] (size_t n);
```

In [allocator.traits.members]:

```
[nodiscard]] static constexpr pointer
allocate [[ref_to_uninit]] (Alloc& a, size_type n);
[nodiscard]] static constexpr pointer
allocate [[ref_to_uninit]] (Alloc& a, size_type n,
                           const void pointer hint);
```

In [allocator.requirements.general], add to the row for `a.allocate(n)`: "For the purposes of the initialization profile ([profiles.init]), the storage is uninitialized."

8.3 Storage allocation [new.syn], [new.delete]

Add the attribute `ref_to_uninit` to each non-deleting allocation function declared in `[new.syn]`, for example:

[illegible]

[*Note*: This affects only direct calls to an allocation function. The result of a *new-expression* is governed by [profiles.init.terms] paragraph 6. — *end note*]

8.4 Header `<memory>` synopsis [memory.syn]

Add after the declarations of `construct_at` and `destroy_at`:

```
// [specialized.now], initialization assertion
template<class T>
    constexpr T* now_init(T* p [[ref_to_uninit]]) noexcept;
```

8.5 Specialized memory algorithms [specialized.algorithms]

Add a paragraph to [specialized.algorithms.general]:

For the purposes of the initialization profile ([profiles.init]), an algorithm in this subclause whose declaration carries the attribute `must_init` on a parameter initializes every object in the range it designates, and an algorithm whose declaration carries the attribute `ref_to_uninit` on a parameter accepts a range of uninitialized objects through it.

Change the declarations in [uninitialized.construct.default], [uninitialized.construct.value], [uninitialized.copy], [uninitialized.move], and [uninitialized.fill], and their `ranges::` counterparts, so that the parameter designating the output range carries the attribute `must_init`, and in the `_n` forms carries `ref_to_uninit`. For example:

```
template<class NoThrowForwardIterator>
    constexpr void uninitialized_default_construct(
        [[must_init]] NoThrowForwardIterator first,
        NoThrowForwardIterator last);
template<class NoThrowForwardIterator, class Size>
    constexpr NoThrowForwardIterator uninitialized_default_construct_n(
        [[ref_to_uninit]] NoThrowForwardIterator first, Size n);
template<class InputIterator, class NoThrowForwardIterator>
    constexpr NoThrowForwardIterator uninitialized_copy(
        InputIterator first, InputIterator last,
        [[must_init]] NoThrowForwardIterator result);
template<class NoThrowForwardIterator, class T>
    constexpr void uninitialized_fill(
        [[must_init]] NoThrowForwardIterator first,
        NoThrowForwardIterator last, const T& x);
```

For each `_n` form, and for `uninitialized_copy` and `uninitialized_move`, whose output range is not delimited by the attributed parameter, add to the Effects paragraph: "For the purposes of the initialization profile ([profiles.init]), the objects in the range [`result`, `result + n`] (respectively [`result`, `result + (last - first)`]) are initialized after the call."

[Drafting note: for the two-iterator forms the output range is delimited by `first`, `last` of the same pair, so `must_init` on `first` suffices under [profiles.init.ref] paragraph 5. For the single-iterator forms the length comes from another argument, which is why the sentence is needed. A cleaner alternative is an argument clause naming the length, `[[must_init(n)]]`; that is a design change and not proposed. The implementations of these algorithms are not checkable under [profiles.init.analysis] (they loop), so the library's own definitions would carry `now_init` or suppress the profile; this is invisible to users.]

8.6 `construct_at` [specialized.construct]

Change the declarations:

```
template<class T, class... Args>
    constexpr T* construct_at(T* location [[must_init]],
                              Args&&... args);

namespace ranges {
    template<class T, class... Args>
        constexpr T* construct_at(T* location [[must_init]],
                                  Args&&... args);
}
```

Add after the Effects paragraph:

Remarks: For the purposes of the initialization profile ([profiles.init]), `*location` is uninitialized before the call and initialized after it, and the returned pointer refers to initialized objects.

8.7 `destroy` [specialized.destroy]

Leave the declarations of `destroy_at`, `destroy`, `destroy_n` and their `ranges::` counterparts unchanged, and add after the Effects paragraph of each:

Remarks: For the purposes of the initialization profile ([profiles.init]), the object or objects destroyed are initialized before the call and uninitialized after it.

8.8 Initialization assertion [specialized.now] †

Add a new subclause after [specialized.destroy]:

```
template<class T>
    constexpr T* now_init(T* p [[ref_to_uninit]]) noexcept;
```

1 *Constraints:* `is_object_v<T>` is `true`.

2 *Returns:* `p`.

3 *Remarks:* For the purposes of the initialization profile ([profiles.init]), a call to `now_init` is an initialization of the object designated by `p` ([profiles.init.terms]) and the returned pointer refers to initialized objects. The rules of [profiles.init] do not apply to the definition of this function.

4 [Note 1: `now_init` asserts that the program has initialized the designated objects by means the profile cannot verify. Together with a function to which the attribute `now_init` appertains ([dcl.attr.nowinit]), it is the only way to obtain a pointer to initialized objects from a pointer to uninitialized objects without suppressing the profile. — *end note*]

8.9 Feature-test macros [version.syn]

Add:

```
#define __cpp_lib_now_init 2026XXL // also in <memory>
```

[Drafting note: a language-level feature-test macro for the profile itself, if any, belongs to P3589R3.]

9. Change to [basic.indet]

Add a note at the end of [basic.indet] paragraph 1:

[Note: Within the dominion of the initialization profile ([profiles.init]), a program that would apply the lvalue-to-rvalue conversion to an object holding an indeterminate or erroneous value is ill-formed. — *end note*]

[Drafting note: this note is the only change to [basic.indet]. The profile does not change what value an object holds; `uninit` is not `indeterminate` and the two compose.]

10. Consistency check: guarantees against rules

This table is the check the design paper asks for under "How to specify this in the standard?". Reading down a column answers "which rules deliver this guarantee?"; reading across a row answers "which guarantee justifies this rule?". A rule with no guarantee would be a candidate for deletion; a guarantee with no rule, or with a rule that depends on information [G4] forbids, would be a hole. Neither occurs. The right-hand column lists the places in the working draft that each rule touches or relies on, which is the design paper's list of affected places kept honest by the left-hand side.

Rule	G1 use	G2 once	G3 statics	G4 local	G5 meaning	Places in N5054 touched or relied on
[profiles.ini t.terms] 2 (leaves part uninitialize d)	x					[dcl.init.general] 9.5.1 ("no initialization is performed")
[profiles.ini t.terms] 3 (uninitialize d at definition)	x					[dcl.attr.uninit], [expr.new] 7.6.2.8, [expr.unary.op], [expr.sub], [expr.ref]
[profiles.ini t.terms] 4 (initializatio n)	x	x				[expr.assign], [specialized.construct] 26.11.8, [expr.new] placement, [dcl.attr.mustinit], [specialized.now]
[profiles.ini t.terms] 5 (destructio n)		x				[specialized.destroy] 26.11.9, [expr.prim.id.dtor], [class.dtor]
[profiles.ini t.terms] 6 (refers to uninitialize d)	x					[dcl.attr.reftouninit], [conv.ptr], [conv.qual], [expr.static.cast], [expr.reinterpret.cast], [expr.cast], [expr.add], [expr.new]
[profiles.ini t.analysis] 1, 2 (every path)	x	x		x		[stmt.select], [stmt.iter], [expr.cond], [expr.log.and], [expr.log.or] (treated as branches)
[profiles.ini t.analysis] 3 (subscripts)	x			x		[expr.sub], [expr.const] 7.7

Rule	G1 use	G2 once	G3 statics	G4 local	G5 meaning	Places in N5054 touched or relied on
[profiles.init.analysis] 4 (indeterminate state)	x	x		x		none new
[profiles.init.decl] 1, 2 (uninit required / forbidden)	x	x				[dcl.init.general], [class.base.init] 11.9.3, [class.default.ctor]
[profiles.init.decl] 3 (trivially destructible member)		x		x		[class.prop], [class.dtor]
[profiles.init.decl] 4 (parameters, references, pointers)	x					[dcl.fct], [dcl.ref], [dcl.ptr]
[profiles.init.decl] 5 (static initialization)			x			[basic.start.static] 6.10.3.2, [expr.const.init] 7.7.4, [dcl.constinit], [stmt.dcl]
[profiles.init.use] 1, 2 (uses)	x					[conv.lval], [expr.ref], [expr.sub], [expr.assign], [expr.pre.incr], [expr.post.incr], [dcl.init.ref], [expr.call], [expr.typeid], [expr.dynamic.cast], [expr.context]
[profiles.init.ref] 1	x					[expr.unary.op], [expr.sub], [expr.ref]

Rule	G1 use	G2 once	G3 statics	G4 local	G5 meaning	Places in N5054 touched or relied on
(indirection)						
[profiles.ini t.ref] 2, 3, 4 (neither or both; calls)	x					[dcl.init], [expr.assign], [expr.call], [over.match.funcs] (implicit object argument)
[profiles.ini t.ref] 5 (must_init obligation and effect)	x			x		[stmt.return], [dcl.attr.mustinit], [dcl.attr.nowinit]
[profiles.ini t.ref] 6 (returns)	x					[stmt.return]
[profiles.ini t.ref] 7 (templates)					x	[temp.dep], [temp.constr]
[profiles.ini t.life] 1 (construct once, destroy once)		x				none new
[profiles.ini t.life] 2, 3 (implicit destruction)		x		x		[stmt.jump.general], [stmt.dcl], [except.ctor], [class.dtor]
[profiles.ini t.class] 1, 2, 3 (members)	x			x		[class.base.init], [class.mfct], [dcl.attr.reftouninit]

Rule	G1 use	G2 once	G3 statics	G4 local	G5 meaning	Places in N5054 touched or relied on
[dcl.attr.uni nit], [dcl.attr.reft ouninit], [dcl.attr.mu stinit], [dcl.attr.no winit]	x	x			x	[dcl.attr.grammar] 9.13.1, [dcl.meaning.general], [dcl.attr.indet] 9.13.6 (pattern)
Library 8.1 to 8.8	x	x				[cstdlib.syn], [c.malloc], [allocator.members], [allocator.traits.members], [allocator.requirements.general], [new.syn], [memory.syn], [specialized.algorithms] 26.11, [specialized.construct], [specialized.destroy]
Library 8.9 (feature-te st macro)						[version.syn]
[basic.inde t] note	x				x	[basic.indet] 6.8.5

Two observations follow from the table. Every row serving only G4 ([profiles.init.decl] 3, [profiles.init.life] 2 and 3, the trivially-destructible rules) is a rule that exists purely to keep the analysis local; those are the rules to expect pushback on, and to defend as the price of implementation-independent acceptance or to trade for `now_init`. And G3 is delivered by exactly one rule, [profiles.init.decl] 5, which is the kind of one-to-one correspondence the method is meant to expose: if that rule is ever weakened, the guarantee is gone, not merely narrowed.

11. A common style for standard profiles

The design paper asks for "a common style for specifying standard profiles". The layout used above is offered as that style; it is independent of the initialization profile and is applied to the invalidation profile in [MW'26c].

A standard profile is specified in one subclause [profiles.name] of the clause [profiles], in six parts in a fixed order. The first part, *General*, names the profile, states its guarantees as normative properties of a well-formed program within the profile's domain, each with a tag [Gn], and states whether its rules are decidable per function body (the locality property) and that they do not change the meaning of a program. The second part, *Terms*, defines the profile's abstract state (here: initialization state and "refers to uninitialized objects") using existing terms of art with cross-references, and defines the operations that change that state. The third part, *Analysis*, defines how the state is propagated through a function body; for every profile that is to be implementation-independent this will be some form of the every-path rule, and writing it once per profile rather than once in the framework keeps profiles free to differ. The fourth part, *Rules*, holds the "shall" statements, grouped by the clause of the working draft they most relate to (declarations, expressions, functions, classes), each tagged with the guarantee it delivers. The fifth part, *Attributes*, adds the profile's attributes as ordinary subclauses of [dcl.attr] in the pattern of [dcl.attr.indet], always with the sentence that the attribute has no effect on the semantics of a program. The sixth part, *Library*, annotates existing declarations and adds any escape hatches, using the elements of [structure.specifications] and never prose where a *Remarks* element will do. A consistency table of guarantees against rules closes the specification; it is not normative text but is what reviewers use to check that every rule has a guarantee and every guarantee a rule.

Two style decisions are worth settling at the framework level rather than per profile. The first is the violation vocabulary. P3589R3 makes failure to satisfy a constraint dictated by a profile a diagnosable rule, so a profile's rules can be plain "shall" statements and [intro.compliance.general] supplies the consequence; P3402R3 [LLG'25], written against P3081R2 [HS'25], instead ends each rule with "profile-rejected by `std::initialization`". The former is used here because it reuses [intro.compliance.general] and needs no new term; the committee should pick one and all standard profiles should use it. The second is the home of the profile subclauses: the clause [profiles] proposed in section 4, or an annex. The text above is indifferent, but cross-references from [dcl.attr] and the library into it are not, so the decision should precede any profile's CWG review. A third, smaller point for the framework author: P3589R3 uses "profile rule" and "enforcement" without defining them; a one-sentence definition of each in [decl.profile.enforce] would let every profile use them without restating, and [profiles.general] above could then be shortened accordingly.

12. Relationship to the invalidation profile wording

This paper and [MW'26c] share a structure: a General subclause stating the guarantees and the properties of locality and of unchanged meaning; a Terms subclause defining the abstract state the profile tracks; an Analysis subclause stating the every-path model once; the rules, each tagged with its guarantee; one attribute subclause per attribute, following [dcl.attr.indet]; and library wording with an escape hatch of the form "the rules of [profiles.x] do not apply to the definition of this function". The two profiles differ in the state they track, initialization state against validity, ownership and escape, and in their defaults at the boundary of the domain: for

the initialization profile an unannotated declaration means initialized; for the invalidation profile [BS'26c] an unannotated non-const function means invalidating. Both live under the one clause [profiles] with the one subclause [profiles.general] proposed in section 4 of this paper, so that the framework text of P3589R3 is stated once; [MW'26c] adds its own subclause to that clause and does not restate the general one.

13. Disclosure of AI use

I prepared this paper with the help of an AI assistant. I wrote the outline of the clause structure, the guarantees and the mapping from the design paper's sections to subclauses; the assistant drafted most of the paragraphs from outlines I approved and proposed edits that I accepted, rejected or rewrote. I checked the examples against the design paper and the experimental implementations. I reviewed every change before submitting this paper. The design decisions are mine, and so is the responsibility for any errors. A CWG and LWG expert assistance in the future would help verify the actual wording.

14. Acknowledgements

Thanks to Ville Voutilainen and Krystian Stasiowski for the experimental implementations against which the rules were checked, to Gabriel Dos Reis for the profile framework and for the discussion of member-wise assignment, and to the members of the profiles group for their review of the design paper and of earlier drafts of this wording.

15. References

[BS'26b] B. Stroustrup: An initialization profile. P4222R3. 2026-10-22.

[BS'26c] B. Stroustrup: An invalidation profile. P4418R0. 2026-10-22.

[BS'26] B. Stroustrup: What are Profiles? P3704R1. 2026-10-22.

[GDR'25] G. Dos Reis: C++ Profiles: The Framework. P3589R3. 2025-12-01.

[HS'25] H. Sutter: Core safety profiles for C++26. P3081R2. 2025-02-04.

[LLG'25] M.-A. Laverdière, C. Lapkowski, C.-H. Gros: A Safety Profile Verifying Initialization. P3402R3. 2025-05-16.

[MW'26b] M. Wong: Profiles: implementation status and experience. P4415R0. 2026-10-22.

[MW'26c] M. Wong, B. Stroustrup: Proposed wording for the invalidation profile. P4417R0. 2026-10-22.

[N5054] Working Draft, Programming Languages, C++. N5054. 2026-07-16.

[N5055] Editors' Report, Programming Languages, C++. N5055. 2026-07-16.

[Impl1] Experimental GCC branch. <https://github.com/villevoutilainen/gcc/tree/p4324>

[Impl2] Experimental Clang branch. <https://github.com/cppalliance/clang/tree/profiles-init>