

Document #:	P4417R0
Date:	2026-10-22
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 invalidation profile

Note: This paper is proposed working-draft wording only. The design, its rationale and its examples are in "An invalidation profile", P4418R0 [BS'26c]; read that first. The wording tracks the P4418R0 draft of 2026-10-22. 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 invalidation profile described in the design paper "An invalidation profile", P4418R0 [BS'26c]. 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. A companion paper, "Proposed wording for the initialization profile" [MW'26a], does the same for the initialization 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 invalidation profile prevents access through a pointer, reference, iterator, or pointer-holding object to an object whose lifetime has ended or whose storage has been relocated. The design paper [BS'26c] gives three guarantees: a pointer can be dereferenced only if it points to an existing object; no pointer can

escape beyond the lifetime of what it points to; and a non-static `[[owner]]` will be deleted exactly once. It delivers them with a small set of rules enforced by local static analysis of each function body, two attributes, `[[not_invalidating]]` and `[[owner]]`, and one library function, `no_dangling()`.

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.

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: `profiles::enforce`, `profiles::require`, `profiles::suppress` and `profiles::exempt` in `[decl.profile.enforce]`, `[decl.profile.require]`, `[decl.profile.suppress]` and `[decl.profile.exempt]`; the grammar *profile-designator*, *profile-name* and *profile-argument*; the term *dominion*; and the framework's change to `[intro.compliance.general]` by which failure to satisfy a constraint dictated by a profile is a diagnosable rule, so every "shall" below is a profile rule whose violation requires a diagnostic. The *profile-name* used is `std::invalidation`, following the `std::` prefix of the examples in P3589R3; the design paper does not name its profile and the spelling is a placeholder for the committee to settle. Subclause numbers follow N5054 and are for orientation only; 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 library element labels in italics; notes and examples with italic labels in roman brackets. 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 three conditions of "Local Invalidation" and the every-path and loop sentences there; the three assumptions of "Invalidation through relocation" and the verification sentence for `[[not_invalidating]]`; the two lists under "Pointers" and "Containers"; the escape cases of the Introduction and of "When is a pointer valid?"; the five ownership rules under "Attribute `[[owner]]`"; the `no_dangling()` paragraphs under "Profile suppression"; the Summary; and the three headings of "Draft WP text", which this paper expands. 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. The run-time checks for dereference of a null pointer, pointer arithmetic, out-of-range access and subscripting are the subject of a separate profile and are not covered here.

4. Guarantees: subclause Invalidation profile `[profiles.inval]` † of clause Profiles `[profiles]` †

Add to the clause `[profiles]` (see `[MW'26a]` for the clause and for the subclause `[profiles.general]` shared by the standard profiles) the following subclause.

34.y Invalidation profile `[profiles.inval]` †

34.y.1 General [profiles.inval.general]

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

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

(2.1) — [G1] every indirection through a pointer designates an object within its lifetime ([basic.life]), and every use of a reference or of an iterator designates an object within its lifetime;

(2.2) — [G2] no pointer to an object leaves the scope in which that object is known to exist ([profiles.inval.escape]);

(2.3) — [G3] every object created by a *new-expression* ([expr.new]) is deleted at most once, and every object owned ([profiles.inval.owner]) by a pointer that does not have static or thread storage duration is deleted exactly once, by a *delete-expression* ([expr.delete]) applied to a pointer that owns it.

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.inval.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 invalidation profile has the same meaning when the profile is not enforced, and the attributes of [dcl.attr.notinvalidating] and [dcl.attr.owner] 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 invalidation profile, except within the dominion of a suppression ([decl.profile.suppress]) or an exemption ([decl.profile.exempt]) naming it. The attributes `not_invalidating` and `owner` on declarations outside the dominion are taken into account; a function declared outside the dominion without these attributes is treated as specified in [profiles.inval.terms] paragraph 5. [Note 3: A non-const member function, or a function taking an object by non-const pointer or reference, is assumed to invalidate unless it says otherwise, whatever its definition does. — end note]

6 [Note 4: The three guarantees of the design paper are (2.1), (2.2) and (2.3). The rules of [profiles.inval.terms] through [profiles.inval.owner] are stronger than what (2.1) through (2.3) alone require, so that they can be checked under [G4]. Dereferencing a null pointer value, pointer arithmetic that leaves an array, and subscripting outside the bounds of an array are not covered by this profile ([Note 5: They are the subject of a separate profile. — end note]). Each rule is annotated with the guarantee it delivers. — end note]

[Drafting note: the design paper states the third guarantee for a "non-static [[owner]]". (2.3) therefore requires the delete only of objects owned by a pointer with automatic storage duration or by a non-static data member; an owner with static or thread storage duration may hold its object for the whole program. The "at most once" half of (2.3) applies to every object created by a *new-expression* regardless of who owns it.]

5. Terms and analysis model

34.y.2 Terms [profiles.inval.terms]

1 For the purposes of this subclause, an expression or object *refers to* an object *o* if it is

(1.1) — a pointer whose value is the address of *o* or of a subobject of *o* ([basic.compound]);

(1.2) — a reference bound to *o* or to a subobject of *o* ([dcl.init.ref]), or a name that denotes *o* ([basic.lookup]);

(1.3) — an iterator obtained from a container object *c* ([container.reqmts]) where *o* is an element of *c*, or an iterator obtained from such an iterator by an iterator operation;

(1.4) — an object of class type having a subobject that refers to *o*, including a closure object with a capture by reference ([expr.prim.lambda.capture]);

(1.5) — a pointer to, a reference to, or an array of objects that refer to *o*; or

(1.6) — 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 an expression that refers to *o*.

[*Note 1*: This is the design paper's broad sense of "pointer": pointers, references including the names of objects, classes with pointer members, lambdas with capture by reference, `jthread` objects holding pointers, `unique_ptr` and `shared_ptr`, iterators, pointers to pointers, references to pointers, and arrays of pointers. — *end note*]

2 An object *c* is a *container of o* if *c* refers to *o* through a subobject of non-const pointer type, a non-const reference, or an iterator, or if *c* is an object of a container type ([container.reqmts]) of which *o* is an element. [*Note 2*: A container is anything that could relocate or destroy what it refers to. A container may be a pointer in the sense of paragraph 1 and a pointer may be a container. An object referred to only through pointers and references to const cannot be invalidated through them; casting away constness is restricted by a separate profile. — *end note*]

3 [G1] An *invalidating operation* on an object *o* is any of the following:

(3.1) — the end of the lifetime of *o* ([basic.life]), including by a *delete-expression* whose operand refers to *o*, by a call to `std::destroy_at`, `std::destroy`, `std::destroy_n` or their `std::ranges` counterparts whose argument refers to *o*, by an explicit destructor call, or by exit from the block in which *o* was declared;

(3.2) — a call to a non-static member function of a container *c* of *o* on *c*, unless the member function is declared const or the attribute `not_invalidating` ([dcl.attr.notinvalidating]) appertains to it;

(3.3) — a call to a function in which a container *c* of *o*, or a pointer or reference to *c*, is the argument for a parameter of non-const pointer or reference type, unless the attribute `not_invalidating` appertains to the parameter or to the function;

(3.4) — a function call in which a pointer that owns *o* ([profiles.inval.owner]) is the argument for a parameter to which the attribute `owner` ([dcl.attr.owner]) appertains, or a transfer of ownership of *o* to an owner that is not a variable of the function body being analyzed ([profiles.inval.owner] paragraph 3); or

(3.5) — an assignment to, or a destruction of, a subobject through which *c* refers to *o*, where *c* is a container of *o*.

[Note 3: (3.2) and (3.3) are the design paper's defaults: a non-const member function and a function accessing an object through a pointer or reference to non-const are assumed to invalidate; a const member function and a function accessing an object only through a pointer or reference to const are assumed not to. The attribute `not_invalidating` marks the exceptions and is verified ([`dcl.attr.notinvalidating`]). Reallocation by `vector::push_back` is the motivating case of (3.2). — *end note*]

4 [G1] A pointer, reference, iterator, or class object *p* that refers to an object *o* is *valid* at a point *P* if no invalidating operation on *o* occurs on any path ([`profiles.inval.analysis`]) from the point at which *p* came to refer to *o* to *P*; otherwise *p* is *invalid* at *P*. [Note 4: Whether *p* is valid is a property of the program text determined by this subclause, not a property of *p* at run time. — *end note*]

5 [G1] A function is *invalidating with respect to* a parameter *q* of pointer or reference type if *q* is not a pointer or reference to const and the attribute `not_invalidating` appertains neither to *q* nor to the function; a non-static member function is *invalidating with respect to* its object parameter if it is not declared const and the attribute `not_invalidating` does not appertain to it.

6 [G2] An object *o* is *known to outlive* a block *B* of a function *f* if

(6.1) — *o* has static or thread storage duration ([`basic.stc.static`], [`basic.stc.thread`]);

(6.2) — *o* is referred to through a parameter of *f* of pointer type or of non-const lvalue reference type, or is an element of a container so referred to; or

(6.3) — *o* was created by a *new-expression* and a pointer that owns *o* ([`profiles.inval.owner`]) is itself known to outlive *B*.

[Note 5: An object referred to through a parameter of const lvalue reference type or of rvalue reference type is not known to outlive the function, because the reference may be bound to a temporary object ([`class.temporary`]). A variable with automatic storage duration declared in *B* or in a block nested in *B* is not known to outlive *B*. — *end note*]

7 [G2] A pointer, reference, iterator, or class object *p* that refers to *o* *escapes* a block *B* if *p* is

(7.1) — the operand of a `return` statement of the function whose body is or contains *B* ([`stmt.return`]);

(7.2) — the operand of a `throw` expression ([`expr.throw`]);

(7.3) — used to initialize or assign to an object that is known to outlive *B*, or a subobject of such an object, or an object referred to through an expression that itself escapes *B*;

(7.4) — captured by a closure object that escapes *B*; or

(7.5) — used to initialize or assign to a subobject of an object of class type that escapes *B*, or passed as an argument to a constructor or member function of such an object other than one to which `not_invalidating` appertains.

[Drafting note: paragraphs 6 and 7 are the escape rules of the design paper's Introduction and "When is a pointer valid?" written as one relative-lifetime rule: a pointer may leave a scope if what it refers to is known to outlive that scope. The design paper's cases, received as an argument, points to something static,

points to something on the free store and is responsible for it, are (6.2), (6.1) and (6.3). Note 5 states the consequence for parameters of const lvalue reference and rvalue reference type, which the design paper leaves implicit. (6.3) is what makes the design paper's `glob_ptr = new int{9}` an error: the result of the *new-expression* is the owner, and it outlives nothing, so the only escapes for a freshly created object are into an owner that outlives the block or through an `owner`-attributed return ([profiles.inval.owner] paragraph 3).]

34.y.3 Analysis [profiles.inval.analysis]

1 [G4] Whether a pointer, reference, iterator, or class object is valid at a point *P* in a function body ([profiles.inval.terms] paragraph 4), whether a pointer owns an object at *P* ([profiles.inval.owner]), and whether an expression escapes a block, 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. An operation that occurs on any path to *P* is treated as having occurred at *P*; an operation that is required at *P* shall occur on every path to *P*. [Note 1: This is the design paper's formulation: every deletion and every access through a pointer is considered unconditional; every action the profile considers, a deletion, a required deletion, an invalidation, or a transfer of ownership, must appear on every path through the code; and loops are considered to be executed zero or more times. 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 analysis only through the declaration of the called function. — *end note*]

2 [G1, G4] A use of a pointer, reference, iterator, or class object *p* that is invalid at *P* is ill-formed ([profiles.inval.use]). [Note 2: Code whose validity depends on a condition the analysis does not evaluate can be rewritten, can use `std::no_dangling` ([mem.nodangling]), or can suppress the profile ([decl.profile.suppress]). — *end note*]

3 [Example 1:

```
void f(int x, int y) {
    int* q [[owner]] = new int{8};
    delete q;
    if (0 < x) *q = 11;           // error: q is invalid
    *q = 9;                      // error: q is invalid
    int* p [[owner]] = new int{7};
    if (0 < x) delete p;
    *p = 10;                     // error: p is invalid on one path
    if (0 < y) *p = 1;           // error: p is invalid on one path
    // error: p owns an object on one path at the end of the block
}
```

— *end example*]

[Drafting note: the design paper's version of this example declares *q* and *p* without `[[owner]]` and counts five errors. Under the ownership rules of the design paper and of [profiles.inval.owner], initializing a non-owner from a *new-expression* and deleting a non-owner are two further errors on each of *q* and *p*. The example above adds `[[owner]]` so that it shows exactly the five invalidation errors; the design paper's Local Invalidation section may wish to do the same, or to say that `[[owner]]` catches the

unowned news.]

6. Rules

34.y.4 Uses of pointers [profiles.inval.use]

1 An expression *uses* a pointer, reference, iterator, or class object *p* that refers to an object *o* if it

(1.1) — applies indirection ([*expr.unary.op*]), subscripting ([*expr.sub*]), or class member access through a pointer ([*expr.ref*]) to *p*;

(1.2) — applies the lvalue-to-rvalue conversion ([*conv.lval*]) to a glvalue designating *o* obtained through *p*, or assigns to such a glvalue;

(1.3) — invokes a member function on *o* through *p*, or calls a function with *p* as an argument for a parameter of pointer or reference type;

(1.4) — applies an iterator operation ([*iterator.requirements*]) to *p* other than assignment to *p* or destruction of *p*; or

(1.5) — initializes, assigns, or returns a pointer, reference, iterator, or class object from *p* ([*profiles.inval.terms*] paragraph 1).

2 [G1] An expression that uses *p* at a point at which *p* is invalid is ill-formed.

3 [G1] Let *p* be a pointer, reference, iterator, or class object that refers to an element *o* of a container *c*. Between the point at which *p* came to refer to *o* and a use of *p*, there shall be no invalidating operation on *o* ([*profiles.inval.terms*] paragraph 3). [*Note 1*: This follows from paragraph 2 and the definition of validity and is stated because it is the rule the design paper gives for invalidation through relocation. — *end note*]

4 [*Example 1*:

```
void f(vector<int>& vi) {  
    vi.push_back(9);           // may relocate the elements of vi  
}  
  
void g() {  
    vector<int> vi1 { 1, 2 };  
    f(vi1);                   // OK: no pointer to an element exists  
    vector<int> vi2 { 1, 2 };  
    auto p = vi2.begin();     // p refers to the first element  
    f(vi2);                   // invalidating operation on *p  
    *p = 7;                   // error: p is invalid  
}
```

— *end example*]

34.y.5 Escape [profiles.inval.escape]

1 [G2] A pointer, reference, iterator, or class object *p* that refers to an object *o* shall not escape a block *B* ([*profiles.inval.terms*] paragraph 7) unless *o* is known to outlive *B* ([*profiles.inval.terms*] paragraph 6).

[*Note 1*: In particular, a pointer to a variable with automatic storage duration declared in a function shall not be returned from that function, assigned to a variable with static storage duration, stored in an object

passed in by pointer or reference, or stored in a container that is returned. A pointer received as an argument, or a pointer to an object with static storage duration, may. — *end note*

2 [G2] A class object that refers to an object *o* through a subobject escapes a block only if every object it refers to is known to outlive that block. [Note 2: The result of a call to a member function of such an object is treated as referring to every object the class object refers to ([profiles.inval.terms] (1.4) and (1.6)); a program that knows better uses `std::no_dangling` ([mem.nodangling]). — *end note*]

3 [Example 1:

```
int glob = 5;
int* glob_ptr = nullptr;

void f(int* p1) {
    int x = 7;
    glob_ptr = &x;           // error: x does not outlive f
    glob_ptr = p1;          // OK: *p1 was passed in
    // error: the new object is owned by no one that outlives f
    glob_ptr = new int{9};
    glob_ptr = &glob;        // OK: static storage duration
}

vector<int*> h(int* p) {
    vector v1 { p };
    return v1;               // OK: *p outlives h
    int x = 7;
    vector v2 { p, &x };     // v2 is usable here
    return v2;               // error: x does not outlive h
}
```

— *end example*]

4 [Example 2:

```
class WidgetFactory {
    Window* window_;
    Logger* logger_ = nullptr;
public:
    WidgetFactory(Window& w, Logger& l) : window_(&w), logger_(&l) {}
    WidgetFactory(Window& w) : window_(&w) {}
    Widget create_widget();           // returns an object holding a pointer
};

Widget inner(Window& window) {
    Logger log;
    WidgetFactory factory{window, log};
    // error: the result may refer to log, which does not outlive inner
    return factory.create_widget();
}

Widget inner2(Window& window, Logger& log) {
    WidgetFactory factory{window, log};
    return factory.create_widget(); // OK: window and log were passed in
}
```

— *end example*]

34.y.6 Ownership [profiles.inval.owner]

1 [G3] A pointer *owns* an object *o* at a point *P* if it is the holder of the obligation to delete *o* at *P*, as

determined by this subclause. At each point, at most one pointer owns *o*. [*Note 1: Ownership, like validity, is a property of the program text. — end note*]

2 [G3] The result of a *new-expression* owns the object it creates. A variable, parameter, non-static data member, or function result to which the attribute `owner` ([`dcl.attr.owner`]) appertains is an *owner*; any other pointer is a *non-owner*.

3 [G3] Initializing or assigning an owner from a pointer that owns *o* transfers ownership of *o* to the owner; the source, if it is an object, becomes a non-owner that refers to *o*. Initializing or assigning a non-owner from a pointer that owns *o* makes the non-owner refer to *o* and leaves the ownership unchanged. Passing a pointer that owns *o* as the argument for a parameter that is an owner transfers ownership of *o* to the callee ([`profiles.inval.terms`] (3.4)); passing it as the argument for any other parameter does not. Returning a pointer that owns *o* from a function to whose result the attribute `owner` appertains transfers ownership of *o* to the caller, where the call expression owns *o*.

4 [G3] A pointer that owns *o* and is the result of a *new-expression* or of a function call shall be used to initialize or assign an owner, or be the operand of a *delete-expression*, in the full-expression in which it is produced. [*Note 2: `int* p = new int{7};` with `p` a non-owner throws away the ownership and is ill-formed; `use(q, new int{9})` with the second parameter an owner is well-formed. — end note*]

5 [G3] A pointer that owns *o* and is a variable with automatic storage duration shall, on every path from the point at which it came to own *o* to the end of its block and to every `return` statement in that block, either be the operand of a *delete-expression* or transfer ownership of *o* under paragraph 3. [*Note 3: An owner that is a non-static data member is deleted or transferred by the destructor of the enclosing class, which is analyzed as a function body in its own right. An owner with static or thread storage duration is not required to delete the object it owns. — end note*]

6 [G3] The operand of a *delete-expression* shall be a pointer that owns the object it refers to at that point. After a *delete-expression*, every pointer that refers to the deleted object is invalid ([`profiles.inval.terms`] (3.1)), and no pointer owns it.

7 [G1, G3] After a transfer of ownership of *o* to a callee or to an owner that is not a variable of the function body being analyzed, every pointer in the function body that refers to *o* is invalid ([`profiles.inval.terms`] (3.4)). [*Note 4: The callee may have deleted the object. A pointer passed to a non-owner parameter is not affected by the call unless the function is invalidating with respect to that parameter ([`profiles.inval.terms`] paragraph 5). — end note*]

8 [*Example 1:*

```
void use(int* p, int* q [[owner]]) {
    delete p;                // error: p is not an owner
    delete q;                // required on every path
}

void f() {
    int* p = new int{7};      // error: ownership thrown away
    int* q [[owner]] = new int{7};
    use(p, q);               // ownership of *q passes to use
    *p = 7;                  // OK: p is not invalidated by the call
    delete q;               // error: q no longer owns the object
}
```

```

void g() {
    int* q [[owner]] = new int{7};
    use(q, new int{9});           // q is passed as a non-owner
    *q = 11;                      // OK
    delete q;                    // OK: q still owns the object
}

```

— end example]

[Drafting note: paragraphs 2 and 3 are the five ownership rules of the design paper. The sentence on passing an owner to a non-owner parameter, which the design paper shows by its second `use` example, is stated as a rule: the owner is not consumed and no annotation is needed. The `owner`-attributed return in paragraphs 2 and 3 is the form the experimental GCC implementation [Impl1] provides; without it, the design paper's rule that a pointer cannot be returned unless it was passed in, points to a static, or points to an owned free-store object has no legal way to return a freshly created object. Paragraph 7 states what the design paper leaves implicit after `use(p, q)`: every alias of the transferred object is invalid, not only the former owner. The additional annotations of the experimental implementation for consuming an owner and for treating a non-owner as an owner are not specified here.]

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

Add the following two subclauses after `[dcl.attr.indet]` (9.13.6), or after the subclauses added by [MW'26a] if that paper is adopted first. The structure follows `[dcl.attr.indet]` and `[dcl.attr.nodiscard]`. No change to the grammar is proposed: an attribute-token is an identifier (`[dcl.attr.grammar]`), so a new standard attribute adds a subclause and not a production, and every position used below is one where an attribute-specifier-seq may already appear.

9.13.u Not-invalidating attribute `[dcl.attr.notinvalidating]` †

1 The attribute-token `not_invalidating` specifies that a function does not invalidate the objects referred to by one of its parameters or by its object parameter. No *attribute-argument-clause* shall be present. The attribute may be applied to the declaration of a non-static member function, or 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. [Note 1: Applied to a member function, it refers to the object parameter; applied to a parameter, it refers to that parameter. — end note]

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 2: The attribute is information for the invalidation profile (`[profiles.inval]`). Within the dominion of that profile a function to which it appertains, or whose parameter it appertains to, is not invalidating with respect to that parameter (`[profiles.inval.terms]` paragraph 5), and the definition of the function is checked (`[profiles.inval.verify]`). A const member function and a parameter of pointer or reference to const need no attribute. — end note]

4 [Example 1:

```

template<class T> class vector {
public:
    size_t size [[not_invalidating]] () const;      // redundant: const
    T* data [[not_invalidating]] ();                // non-const, verified
    void push_back(const T& x);                     // invalidating
};
void fill(vector<int>& v [[not_invalidating]], int x);

```

— end example]

9.13.v Owner attribute [dcl.attr.owner] †

1 The *attribute-token* `owner` specifies that a pointer holds the obligation to delete the object it refers to. No *attribute-argument-clause* shall be present. The attribute may be applied to the declaration of a variable, a non-static data member, or a parameter of pointer type, to such a *parameter-declaration* by an *attribute-specifier-seq* at its start, or to the declaration of a function whose return type is a pointer 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 invalidation profile ([profiles.inval]). Within the dominion of that profile an entity to which it appertains is an owner ([profiles.inval.owner]) and the rules of that subclause apply to it. It does not affect the type of the entity. — end note]

4 [Example 1:

```

int* q [[owner]] = new int{7};          // q owns the new object
void use(int* p, int* q [[owner]]);      // use takes ownership via q
int* make [[owner]] (int v);             // the result is owned
struct Node { Node* next [[owner]]; };   // deleted by ~Node

```

— end example]

34.y.7 Verification of `not_invalidating` [profiles.inval.verify]

1 [G1] Let f be a function and q a parameter of f , or the object parameter of f , to which the attribute `not_invalidating` appertains or with respect to which f is declared not invalidating. In the definition of f there shall be no invalidating operation ([profiles.inval.terms] paragraph 3) on any object referred to through q , and q shall not be passed as the argument for a parameter with respect to which the called function is invalidating. [Note 1: The attribute makes the programmer's intent explicit and is verified when the definition is compiled; it is not trusted. — end note]

[Drafting note: this is the design paper's sentence "No invalidating operation is allowed in a function specified [[not_invalidating]]". It is placed in [profiles.inval] rather than in the attribute subclause because, by the pattern of [dcl.attr.indet], an attribute subclause carries no profile rules.]

8. Library wording

8.1 Container requirements [container.reqmts]

Add a paragraph to [container.reqmts] after the paragraphs on invalidation of references, pointers, and iterators:

For the purposes of the invalidation profile ([profiles.inval]), the attribute `not_invalidating` ([dcl.attr.notinvalidating]) is treated as appertaining to every non-const member function of a container, and to every function in this Clause taking a container by non-const pointer or reference, whose specification does not state that it may invalidate references, pointers, or iterators to elements of the container. [Note: `vector::push_back` may invalidate ([vector.modifiers]); `vector::data`, `vector::begin`, `vector::operator[]`, `vector::front` and `vector::back` do not. — end note]

[Drafting note: this paragraph replaces the per-function annotation of the standard library that the design paper's "Tool support" section discusses. The working draft already states, function by function, which container operations may invalidate references, pointers and iterators, so the profile can key off those statements rather than off attributes. A count over 24 standard containers and pointer wrappers, taken per overload because each overload would carry its own attribute, finds 286 non-const member function overloads that do not invalidate against 364 that do; the paragraph covers the former without touching their declarations, and keying off the existing invalidation statements rather than off attributes is also what keeps the default from having to flip. It also settles the status of non-const `begin()`, `end()`, `data()`, `operator[]`, `front()` and `back()`, which would otherwise count as invalidating under the default of [profiles.inval.terms] (3.2).]

8.2 Smart pointers [unique.ptr], [util.smartptr.shared]

Change the following declarations in [unique.ptr.single.general] and [unique.ptr.single.modifiers]:

```
constexpr explicit
    unique_ptr(type_identity_t<pointer> p [[owner]]) noexcept;
constexpr unique_ptr(type_identity_t<pointer> p [[owner]],
    const D& d) noexcept;
constexpr unique_ptr(type_identity_t<pointer> p [[owner]],
    remove_reference_t<D>&& d) noexcept;
constexpr pointer release [[owner]] () noexcept;
constexpr void reset(pointer p [[owner]] = pointer()) noexcept;
```

Make the corresponding changes for the array specialization in [unique.ptr.runtime], and in [util.smartptr.shared.const] and [util.smartptr.shared.mod] for the constructors of `shared_ptr` taking a raw pointer and for `shared_ptr::reset` taking a raw pointer.

Add to [unique.ptr.single.general] after the paragraph describing the stored pointer:

For the purposes of the invalidation profile ([profiles.inval]), a `unique_ptr` object owns the object its stored pointer refers to, `get()`, `operator*` and `operator->` return non-owners that refer to it, and the destructor and `reset` are destructions of it.

Add the corresponding paragraph to [util.smartptr.shared.general], with "the last `shared_ptr` object sharing ownership" in place of "a `unique_ptr` object".

[Drafting note: this is the list of cases that the design paper's "Draft WP text" asks for under Library

functions: the places where a standard ownership abstraction takes over or gives up the responsibility to delete. `make_unique` and `make_shared` need no annotation, since they never expose the owned pointer. The declarations above omit the template-heads that precede them in the synopsis.

`allocator::deallocate` and `allocator_traits::deallocate` are not annotated: the invalidation profile tracks objects created by *new-expressions*; raw storage is the concern of the initialization profile [MW'26a].]

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

Add:

```
// [mem.nodangling], dangling assertion
template<class T>
constexpr T&& no_dangling(T&& x) noexcept;
```

8.4 Dangling assertion [mem.nodangling] †

Add a new subclause at the end of [memory]:

```
template<class T>
constexpr T&& no_dangling(T&& x) noexcept;
```

1 *Returns:* `std::forward<T>(x)`.

2 *Remarks:* For the purposes of the invalidation profile ([profiles.inval]), the result refers to no object that is not known to outlive the innermost enclosing block of the call ([profiles.inval.terms] paragraph 6), and the result is valid ([profiles.inval.terms] paragraph 4). The rules of [profiles.inval] do not apply to the definition of this function.

3 *Recommended practice:* An implementation should issue a diagnostic for a call to `no_dangling` if it can determine from the function body containing the call that the argument refers to an object that is not known to outlive that block, or that the argument is invalid at the point of the call.

4 [*Note 1:* `no_dangling` asserts that an expression the profile would reject as escaping or as invalid does not in fact refer to an object that will have been destroyed. It is a localized form of suppression of the profile, in the way that `std::now_init` is for the initialization profile [MW'26a]; an implementation that can see that the assertion is wrong may say so, but the profile does not require it to try. — *end note*]

5 [*Example 1:*

```
Widget inner(Window& window) {
    Logger log;
    WidgetFactory factory{window, log};
    // OK: the program asserts that the result does not refer to log
    return std::no_dangling(factory.create_widget());
}
```

— *end example*]

[Drafting note: the design paper describes `no_dangling()` as an intrinsic that asserts that a function that would otherwise be considered as causing dangling does not, shows it applied to the returned `Widget`, and says that an analyzer that finds a `no_dangling()` to be wrong is allowed to issue an error. The function is specified as a template on the asserted expression, which is the form the design paper's

WidgetFactory example needs and the form the experimental GCC implementation [Impl1] provides; a pointer argument is covered by the same template. Paragraph 3 gives the permission to diagnose as recommended practice, so that two implementations may differ in whether they catch a wrong assertion without differing in which programs they accept. The design paper's open question whether the function is needed at all is left to the committee; the wording is arranged so that removing 8.3 and 8.4 and the two notes that cite them leaves the rest intact.]

8.5 Feature-test macros [version.syn]

Add:

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

9. Change to [basic.life]

Add a note at the end of [basic.life] paragraph 8 (the paragraph on using a pointer after the lifetime of the object it points to has ended):

[*Note:* Within the dominion of the invalidation profile ([profiles.inval]), a program that would perform such a use is ill-formed. — *end note*]

[Drafting note: this note is the only change to [basic.life]. The profile does not change when lifetimes begin or end.]

10. Consistency check: guarantees against rules

Reading down a column answers "which rules deliver this guarantee?"; reading across a row answers "which guarantee justifies this rule?". The right-hand column lists the places in the working draft that each rule touches or relies on.

Rule	G1 deref	G2 escape	G3 owner	G4 local	G5 meaning	Places in N5054 touched or relied on
[profiles.inval. terms] 1 (refers to)	x	x				[basic.compound], [dcl.init.ref], [basic.lookup], [container.reqmts], [expr.prim.lambda.capture], [conv.ptr], [expr.add]
[profiles.inval. terms] 2 (container of)	x					[container.reqmts]
[profiles.inval. terms] 3 (invalidating operation)	x		x			[basic.life], [expr.delete], [specialized.destroy], [class.dtor], [stmt.dcl], [class.mfct.non.static], [dcl.attr.notinvalidating]
[profiles.inval.	x			x		none new

Rule	G1 deref	G2 escape	G3 owner	G4 local	G5 meaning	Places in N5054 touched or relied on
terms] 4 (valid)						
[profiles.inval. terms] 5 (invalidating with respect to)	x					[dcl.fct], [class.mfct.non.static], [dcl.attr.notinvalidating]
[profiles.inval. terms] 6 (known to outlive)		x	x			[basic.stc.static], [basic.stc.thread], [dcl.fct], [class.temporary], [expr.new]
[profiles.inval. terms] 7 (escapes)		x				[stmt.return], [expr.throw], [expr.prim.lambda.capture], [class.base.init]
[profiles.inval. analysis] 1, 2 (every path)	x	x	x	x		[stmt.select], [stmt.iter], [expr.cond], [expr.log.and], [expr.log.or]
[profiles.inval. use] 1, 2, 3 (uses)	x					[expr.unary.op], [expr.sub], [expr.ref], [conv.lval], [expr.call], [iterator.requirements]
[profiles.inval. escape] 1, 2 (escape)		x				[stmt.return], [basic.stc], [class.mfct]
[profiles.inval. owner] 1 to 3 (ownership and transfer)			x			[expr.new], [dcl.init], [expr.assign], [expr.call], [stmt.return], [dcl.attr.owner]
[profiles.inval. owner] 4, 5 (obligation discharged)			x	x		[expr.new], [expr.delete], [stmt.jump.general], [class.dtor], [basic.stc.static]
[profiles.inval. owner] 6, 7	x		x			[expr.delete]

Rule	G1 deref	G2 escape	G3 owner	G4 local	G5 meaning	Places in N5054 touched or relied on
(delete; after transfer)						
[profiles.inval.verify] 1 (not_invalidating verified)	x			x		[dcl.attr.notinvalidating], [dcl.fct.def]
[dcl.attr.notinvalidating], [dcl.attr.owner]	x	x	x		x	[dcl.attr.grammar] 9.13.1, [dcl.meaning.general], [dcl.attr.indet] 9.13.6 (pattern)
Library 8.1 (container requirements)	x					[container.reqmts], [vector.modifiers], [sequence.reqmts]
Library 8.2 (smart pointers)	x		x			[unique.ptr.single.general], [unique.ptr.single.modifiers], [unique.ptr.runtime], [util.smartptr.shared.const], [util.smartptr.shared.mod]
Library 8.3, 8.4 (no_dangling)	x	x				[memory.syn], [mem.nodangling]
Library 8.5 (feature-test macro)						[version.syn]
[basic.life] note	x				x	[basic.life] 6.8.4

Three observations follow. G2 and G3 are delivered by [profiles.inval.terms] 6 and 7 with [profiles.inval.escape], and by [profiles.inval.owner], respectively; the design paper's sections "When is a pointer valid?" and "Attribute [[owner]]" exist to deliver exactly these two guarantees, which is why the design paper now states three guarantees rather than one. The rows serving G4 alone ([profiles.inval.terms] 4 and the every-path paragraph) are where implementers will test the wording against the two experimental implementations. And the library row for 8.1 is the one place where a single paragraph removes over a hundred annotations; if the committee prefers explicit attributes, every non-invalidating non-const container member needs the attribute individually, and the design paper's

"Tool support" section becomes a wording task rather than a tooling one.

11. Relationship to the initialization profile wording

This paper and [MW'26a] 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 dominion: for the initialization profile an unannotated declaration means initialized; for the invalidation profile an unannotated non-const function means invalidating. Both are intended to live under one clause [profiles] with one subclause [profiles.general], proposed in [MW'26a], so that the framework text of P3589R3 is stated once.

12. 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.

13. 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 invalidating defaults, and to the members of the profiles group for their review of the design paper and of earlier drafts of this wording.

14. References

- [BS'26c] B. Stroustrup: An invalidation profile. P4418R0. 2026-10-22.
- [BS'26b] B. Stroustrup: An initialization profile. P4222R3. 2026-10-22.
- [BS'26] B. Stroustrup: What are Profiles? P3704R1. 2026-10-22.
- [BS'24] B. Stroustrup: Profile invalidation, eliminating dangling pointers. P3446R0. 2024-10-14.
- [GDR'25] G. Dos Reis: C++ Profiles: The Framework. P3589R3. 2025-12-01.
- [MW'26a] M. Wong, B. Stroustrup: Proposed wording for the initialization profile. P4416R0. 2026-10-22.
- [MW'26b] M. Wong: Profiles: implementation status and experience. P4415R0. 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>