

Doc. No. P4418R0

Date: 2026-10-22

Audience: EWG, SG12, SG20, SG23

Author: Bjarne Stroustrup

Reply to: bjarne@stroustrup.com

An invalidation profile

Bjarne Stroustrup (bjarne@stroustrup.com)

Columbia University

Abstract

Memory safety has been a popular topic over the last few years. It is a subset of type-and-resource safety which has been my ideal for C++ since its inception and for which the general solution for C++ was outlined in [BS'15, HS'15, BS'21]. This paper describes the enforcement part of the solution that relates to prevention of dangling pointers and its rationale.

This is a design document that in many ways it is a follow-up to [BS'24]. An experimental implementation exists as a GCC branch [Impl1] and a Clang branch [Impl2].

The key guarantees proposed are:

- 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.
- A non-static **[[owner]]** will be deleted exactly once.

Here, a “pointer” is anything that refers directly to an object, such as pointers, references, iterators, pointers to pointers (§4), and classes containing pointers (§5).

The guarantee is implemented based on just a handful of general rules (§10).

The run-time checks required to prevent violations through dereference of **nullptr**, pointer arithmetic, out-of-range access, and subscripting are handled by a separate profile.

Changes since [BS'24]

This paper isn't a revision of that paper, but it is a restatement of the design there.

- **owner<T>** has been renamed **[[owner]]** for uniformity with other profile annotations.
- Now that experimental implementations exist screenshots of profile checking are added.
- Many definitions have been made more precise.

1. Introduction

One of the nastier problems with C++ code that doesn't manage resources well is access through a pointer that doesn't point to an object, usually because the object that the pointer used to refer to has been destroyed or moved. The invalidation profile prevents access to invalidated objects through simple local static analysis (§2).

Consider a trivial example:

```
int* f()           // bad code
{
    int x = 5;
    int* p = new int {7};
    delete p;
    *p = 9;         // error, caught by profile
    return &x;      // error: pointer to soon to be destroyed object returned
}
```

This is of course bad code and using ownership abstractions such as `unique_ptr` and `vector` typically leads to simpler and as performant code. However drastic code reorganization is often not feasible and however we want to address problems we first have to detect their presence.

The invalidation profile relies on restrictions of language use that make static analysis feasible and two attributes to make local static analysis affordable. Importantly, those restrictions are not particularly onerous and do not force the abandonment of popular C++ programming techniques.

The use of the profile is optional – must be requested in the source code – and can be suppressed locally if that's needed [GDR'25, BS'26].

The basic approach to the solution is never to let a pointer escape from the scope of the object it points to. In particular, when analyzing a function:

- Don't access through an invalid pointer.
- If an alias to an object has been taken, don't call a function that can destroy or move it before a use of the object through the alias.
- Don't let a pointer out of its scope unless it is known to
 - Have been received as an argument
 - Points to something static
 - Points to something on the free store (and is responsible for what it point to; see §8)
 - Is not about to be destroyed upon exit from the scope.

By “pointer”, we include anything that refers to an object (e.g., references and lambda captures; see §4).

By “let out of scope”, we include any way of passing a pointer out of a scope (e.g., by putting it into a container that is about to be returned from the function; see §6).

2. Local Invalidation

Consider another simple example of bad code:

```
void f(int x, int y)      // very bad code
{
    int* q = new int{8};
    delete q;
    if (0<x) *q = 11;      // possible disaster
    *q = 9;                // likely disaster

    int* p = new int{7};
    if (0<x) delete p;
    *p = 10;               // possible disaster
    if (0<y) *p = 1;        // possible disaster

    // possible leak of *p
}
```

In large code bases, equivalent problematic code can be found, especially in code containing overly large complex functions. Such errors can and must be caught.

In general, perfect static analysis is impossible. Consider separate compilation, dynamic linking, and functions with loops depending on conditionals. For example, in the example above **x** and/or **y** might depend on input. Even when theoretically feasible, static analysis can be very expensive.

As explained in [BS’26b], to keep compile-time analysis feasible, cheap, comprehensible, and portable, only a simple linear flow analysis is used. To make static analysis conservative and cheap, we consider every deletion and every access through a pointer unconditional.

Every invalidation profile action considered (deletion, required, invalidation, or **[[owner]]** related (§8)) must appear on every path through the code. Loops are considered to be executed zero or more times.

Thus, we get:

```
void f(int x, int y)
```

```

{
    int* q = new int{8};
    delete q;
    if (0<x) *q = 11;    // error: possible disaster
    *q = 9;              // error: likely disaster

    int* p = new int{7};
    if (0<x) delete p;
    *p = 10;             // error: possible disaster
    if (0<y) *p = 1;     // error: possible disaster

    // error: possible leak of *p
}

```

Now, the five possible errors are caught by the profile. If this is too conservative, the programmer can suppress the invalidation profile locally.

There is a simple solution using the `[[owner]]` attribute (§8).

3. Invalidation through relocation

Invalidation can – and often does – involve more than one function. Still the invalidation profile must catch such errors without imposing complex, onerous, or expensive rules. Consider:

```

void f(vector<int>& vi)
{
    vi.push_back(9);    // may relocate vi's elements
}

void g()
{
    vector<int> vi1 { 1,2 };
    f(vi1);             // OK, no pointer to a vi1 element; reallocation is OK

    vector<int> vi2 { 1,2 };
    auto p = vi2.begin(); // point to first element of vi
    f(vi2);
    *p = 7;             // likely disaster, thus disallowed by invalidation profile
}

```

There are of course other ways to access memory that doesn't hold an object of the expected type (e.g., misuse of casts or access through an uninitialized pointer), but those are addressed separately (e.g., by the initialization profile [BS'26b]).

Experimental implementations exist (GCC and Clang):

The top screenshot shows the Compiler Explorer interface with the GCC compiler (x86-64 gcc (P4324 contracts) (Compiler #1)). The source code is a C++ program that uses the `std::invalidation` profile. The code defines a function `f` that takes a `vector<int>` and a `vi` object, and a function `g` that takes a `vi` object. The code uses `vi.push_back(9)` and `vi.begin()` to access the `vi` object. The output window shows an error message: "In function 'void g()': <source>:19:10: error: use of a value bound to 'vi2', potentially invalidated by an earlier mutation of 'vi2', not permitted under the 'std::invalidation' profile". The error points to the line `*p = 7;` in the `g` function.

The bottom screenshot shows the Compiler Explorer interface with the Clang compiler (clang 21.1.8 (invalidation profile) (Compiler #1)). The source code is the same C++ program. The output window shows an error message: "reference validation failed". The error points to the line `*p = 7;` in the `g` function. A note indicates that the value was invalidated here at line 16: `f(vi1);`.

When analyzing a set of functions:

- Assume that a non-**const** member function and a function accessing an object through a pointer to a non-**const** argument invalidates.
- Assume that a **const** function and a function accessing an object through a pointer to a **const** argument doesn't invalidate.

- Use a **[[not_invalidating]]** attribute to mark non-**const** functions and functions that take a pointer to a non-**const** arguments but don't invalidate (e.g., **vector<T>::size()**) [BS'24].

The first two rules prevent invalidation by default and can be validated when the function definitions are seen.

The **[[not_invalidating]]** attribute is needed to avoid a massive number of false positives while also avoiding unaffordable global static analysis. Note that a **[[not_invalidating]]** attribute is verified when the function definition is compiled: No invalidating operation is allowed in a function specified **[[not_invalidating]]**. It is not an annotation that relies on programmers using it correctly and consistently. It just makes the programmer's intent explicit and thereby saves work for the compiler. Thus, the safety guarantee is maintained and made affordable.

The aim is for static analysis to be cheap enough to be implemented in compilers and specified so that all compilers give identical results. Having different static analyzers apply different degrees of sophistication would seriously damage portability of code.

4. Pointers

In the context of invalidation, anything that refers to an object is considered a pointer:

- Pointers
- References (incl. name of objects)
- Classes with pointer members
- Lambdas with capture-by-reference or pointer members
- **jthreads** holding pointers (they are classes in a defined scope)
- **unique_ptrs** and **shared_ptrs** (they are classes with a single logical pointer member)
- Iterators
- Pointers to pointers
- References to pointers
- Arrays of pointers

5. Containers

In the context of invalidation, anything that could relocate is called a container (anything that contains a pointer to something; §4):

- Classes with non-**const** pointer members
- Lambdas with capture-by-reference or non-**const** pointer members
- **jthreads** holding non-**const** pointers (they are classes in a defined scope)

- Non-**const** **unique_ptr**s and non-**const** **shared_ptr**s (they are classes with a single logical pointer member)
- Pointers to non-**const** pointers
- References to non-**const** pointers
- Arrays of non-**const** pointers

To get invalidation, we need a pointer (§4) to an element of a container; that is an object pointed to by such a pointer.

Note that a “container” can be a “pointer” and vice versa.

A **const** pointer cannot be invalidated (except through a non-**const** pointer alias, and thus caught, or through casting) and casting must be restricted for any profile guarantee to hold. Thus, to cast, we need to suppress whichever profile controls it.

6. When is a pointer valid?

In the context of the invalidation profile, a pointer (for the general definition of pointer from §4) is valid as long as the object it points to hasn’t been destroyed. Consider:

```
int glob = 5;

void f(int* p1)           // pointer argument
{
    int x = 7;
    int* p2 = &x;         // pointer to local
    int* p3 = new int{9}; // pointer to free store object (but potential leak; see §8)
    int* p4 = &glob;       // pointer to static
    // ... p1, p2, p3, and p4 are usable here ...
}
```

However, not every valid pointer may escape from the scope in which it’s valid:

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

void f(int* p1)           // pointer argument
{
    int x = 7;
    glob_ptr = &x;         // error: pointer to local; will dangle
    // ...
    glob_ptr = p1;         // OK: pointer passed in
    // ...
}
```

```

    glob_ptr = new int{9};           // error: potential leak caught
    // ...
    glob_ptr = &glob;               // OK: pointer to static
    // ...
}

```

The return of a pointer to the object created by **new** is considered an error because it is not known whether the calling function will **delete** it (§8).

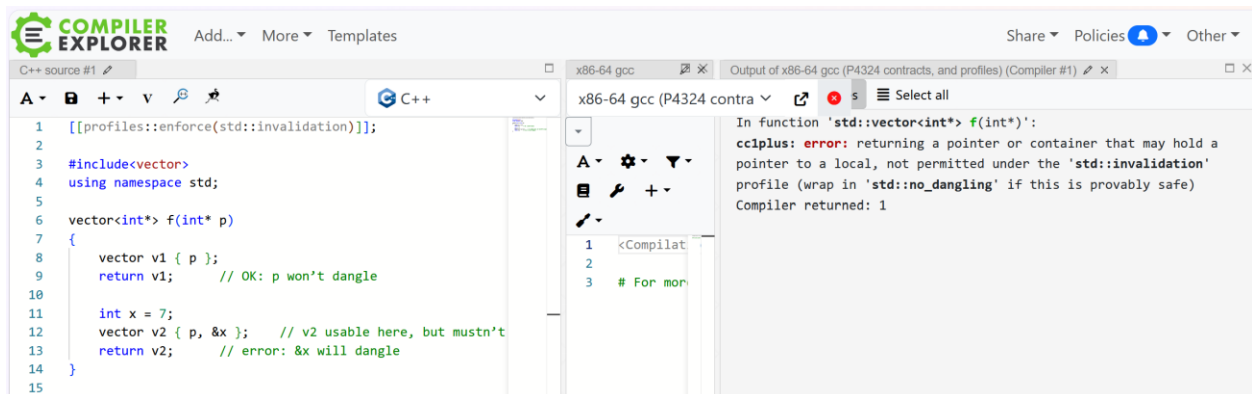
These rules, of course, also apply to containers of pointers:

```

vector<int*> f(int* p)
{
    vector v1 { p };
    return v1;           // OK: p won't dangle

    int x = 7;
    vector v2 { p, &x }; // v2 usable here, but mustn't escape
    return v2;          // error: &x will dangle
}

```



7. Profile suppression `no_dangling()`

Consider a tricky real-world example (slightly condensed):

```

class WidgetFactory {           // optionally log each creation
    Window* window_;
    Logger* logger_ = nullptr;
public:
    WidgetFactory(Window& w, Logger& l) :window_(&w), logger_(&l) {}
    WidgetFactory(Window& w) :window_(&w){}
    Widget create_widget()
    {

```

```

        if (logger_) logger_>log("Creating new Widget");
        return Widget{*window_}; // returns object containing a pointer
    }
};

```

How do we know that the **Widget** returned by **create_widget()** doesn't contain a pointer that will dangle?

```

Widget inner(Window& window)
{
    Logger log;
    WidgetFactory factory{window, log}; // factory holds pointers to window and log
    return factory.create_widget(); // error: the profile doesn't know that
                                     // the widget created doesn't depend on &log
}

```

A local static analyzer can't know that the **Widget** returned doesn't contain a pointer to **log**.

- A **WidgetFactory** is a kind of container of pointers
- It is OK to construct a container with pointers into different scopes
- It is not OK to return a container with a pointer to a local

So, by default, that example is an error!

In general, for static analysis to prove that this example didn't result in a dangling pointer would be complicated, non-local, and potentially very expensive. Relying on exclusively static analysis would end up slowing down compilation. Instead, we must suppress the profile.

Naturally, someone would try to bypass the protections. For example:

```

Widget inner(Window& window)
{
    auto log = make_unique<Logger>();
    WidgetFactory factory{window, *log}; // factory holds pointers to window and log
    return factory.create_widget(); // error: the profile doesn't know that
                                     // the widget created doesn't depend on &log
}

```

Fortunately, that is also caught by the profile.

The obvious solution is to construct the **Logger** outside the inner function:

```

Widget inner(Window& window, Logger& log)
{
    WidgetFactory factory{window, log}; // factory holds pointers to window and log
}

```

```

    return factory.create_widget();           // OK
}

```

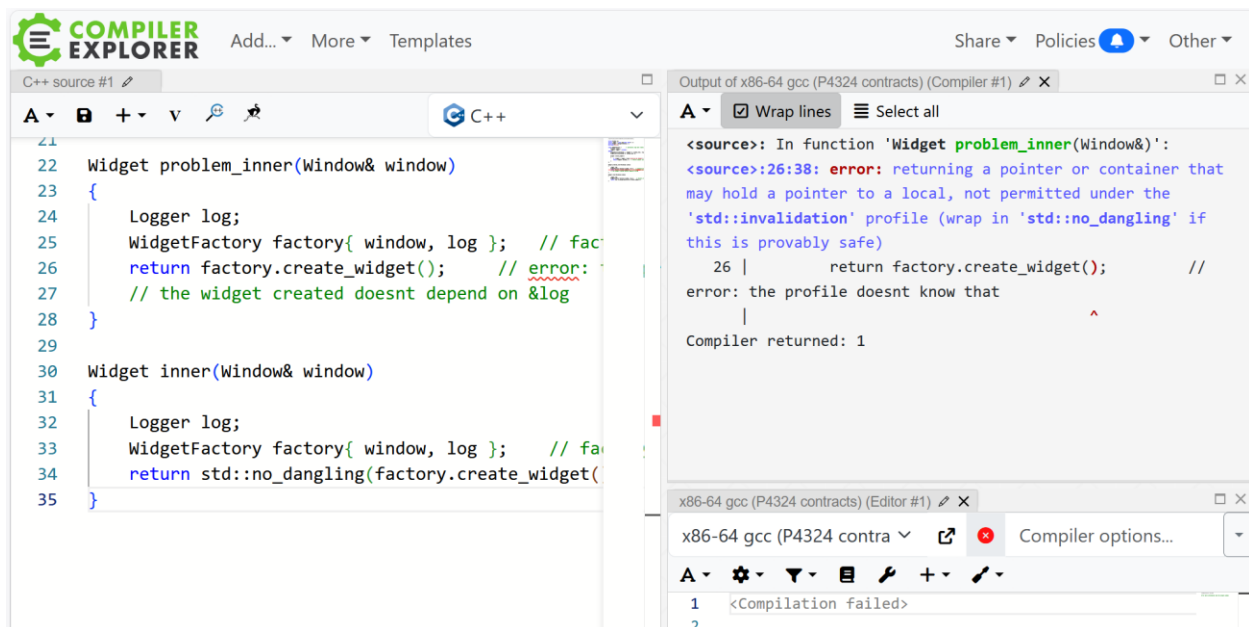
If that's considered too drastic a change, say to an existing program in production use, the invalidation profile can be suppressed. Suppressing the whole profile seems lax, verbose, and non-specific, so we provide an intrinsic function **no_dangling()** that asserts that a function that would otherwise be considered causing dangling does not:

```

Widget inner(Window& window)
{
    Logger log;
    WidgetFactory factory{window, *log};      // factory holds pointers to window and log
    return no_dangling(factory.create_widget()); // OK: profile suppressed
}

```

By default, **no_dangling()** is simply a form of profile suppression like **now_init()** in the initialization profile [BS'26b]. However, in many cases, **no_dangling()** can be validated at compile time. If, on the other hand, the analyzer finds a **no_dangling()** to be wrong, it is allowed to issue an error.



8. Attribute **[[owner]]**

A not uncommon source of dangling pointers is premature **deletes**. Consequently, we need to track and validate ownership of resources created using **new**. Similarly, a not uncommon source of resource leaks is missing or double **deletes**.

How do we represent “ownership”? That is, the responsibility to **delete** an object on the free store (on the heap, in dynamic memory)?

Ideally and most frequently, we use ownership abstractions, such as **vector** and **unique_ptr** to take care of that implicitly. That is, we use constructors and destructors in scoped objects (aka RAI). However, to implement such abstractions, we need a lower-level mechanism to avoid errors and/or having to suppress the invalidation profile. For that, we provide the attribute **[[owner]]** to indicate that a pointer is responsible for deleting what it points to. Most users need not explicitly use **[[owner]]** because its use will be localized in the interface to or implementation of the ownership abstractions, such as **vector** and **unique_ptr**.

[[owner]] can be applied to any pointer with the effect of the invalidation profile enforcing the ownership rules:

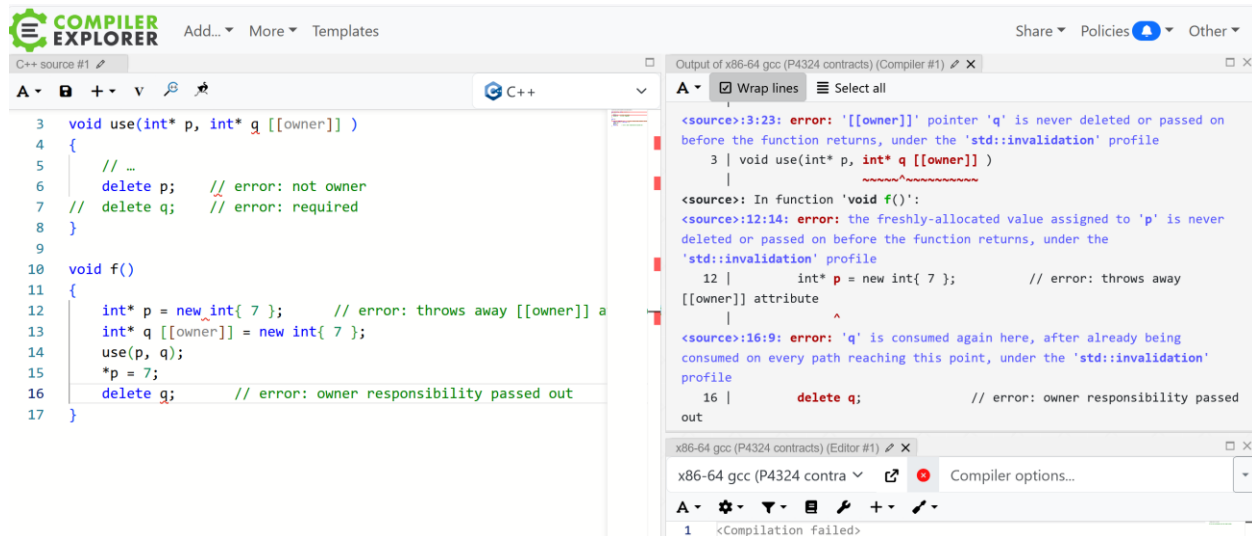
- An **[[owner]]** must be deleted or passed on to another **[[owner]]**.
- A non-**[[owner]]** mustn't be deleted.
- Assigning an **[[owner]]** to another **[[owner]]** transfers the obligation to delete and leaves behind a non-**[[owner]]** (an alias).
- An **[[owner]]** assigned to a non-**[[owner]]** creates an alias and leaves the **[[owner]]** unchanged.
- The result of a **new** is considered an **[[owner]]**.

For example:

```
void use(int* p, int* q [[owner]])
{
    // ...
    delete p;      // error: not owner
    // delete q;    // required
}

void f()
{
    int* p = new int{7}; // error: throws away [[owner]] attribute
    int* q [[owner]] = new int{7};
    use(p,q);
    *p = 7;
    delete q;         // error: [[owner]] responsibility was passed to use()
}
```

Note that like other profile attributes, **[[owner]]** is validated and does not change the semantics of a program using it.

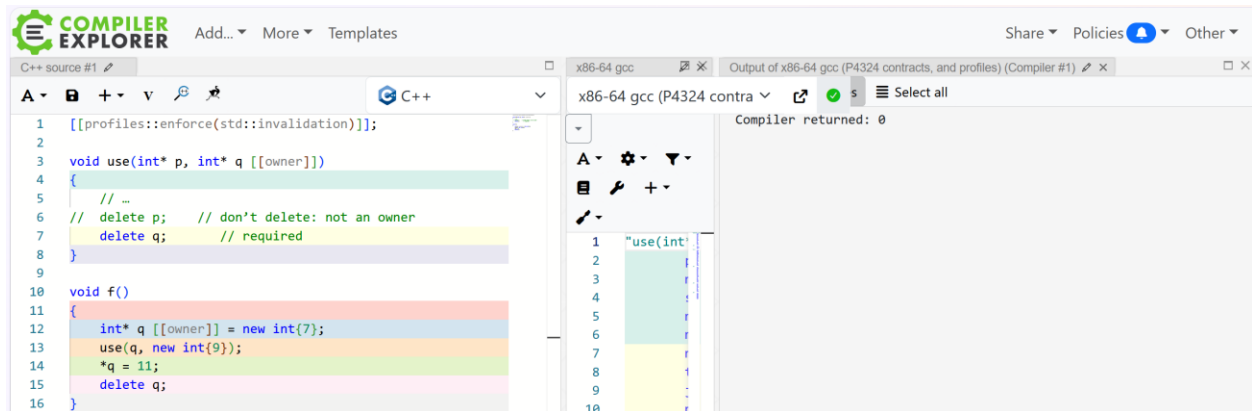


Consider a slightly different example:

```
void use(int* p, int* q [[owner]])
{
    // ...
    // delete p;      // don't delete: not an owner
    delete q;        // required
}

void f()
{
    int* q [[owner]] = new int{7};
    use(q, new int{9});
    *q = 11;
    delete q;
}
```

Here, a non-owner to `*q` is passed to `use()` (as required) so `*q` cannot be deleted and `*q=11` is OK. The ownership of the `int{9}` created by `new` is passed to `use()` (as required).



9. Tool support

Unfortunately, as stated, these rules require many `[[not_invalidating]]` annotations of library functions. The general solution to that is a tool that informs the invalidation profile checker about the semantics of library functions. For example, a tool that reads the effects sections of the ISO standard [GDR'26]. For the standard library, adding `[[not_invalidating]]` annotations manually where needed would take only a few hours so initially a tool for language-based solution may not be needed. An intermediate solution would be to allow the invalidation profile checker to read a list of functions deemed non-invalidating.

An alternative approach would be to build knowledge of common use patterns into the profile checker. For example, it could know that the implementation of ranges doesn't involve invalidation.

```
for(int x : v) fct(x);
for_each(v.begin(),v.end(), fct);
ranges::for_each(v,fct);
```

However, that approach is not general.

10. Summary

Brief summary:

- 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.
 - A non-static `[[owner]]` will be deleted exactly once.
- A pointer cannot be returned from a function unless
 - It was passed in as an argument

- Points to a static
- Points to an object on the free store

This generalizes to pointers passed out of a scope.

- An invalidating function for a container can't be called if pointer to an element has been taken
 - A non-**const** function is assumed to invalidate
 - A **const** function doesn't invalidate
 - A function is assumed to invalidate a non-**const** argument
 - A function doesn't invalidate a **const** argument
- Invalidation profile attributes
 - **[[not_invalidating]]** this argument will not be invalidated by this function
 - **[[owner]]** this pointer must be deleted or passed on
- A function to assert that its argument doesn't contain a dangling pointer
 - **no_dangling()** *an intrinsic function*
- The result of **new** is assumed to be an **[[owner]]**
- Zero run-time overhead for code obeying the profile
- No change of semantics

11. Questions

- Is the **no_dangling()** function to suppress the invalidation profile needed?
 - I bit more experience is needed to determine if it is worthwhile or general suppression of the invalidation profile is sufficient.

12. Draft WP text

This is an overview. It needs and CWG expert review. See the separate “Proposed Wording” paper [MW'26]. Note that the run-time checks required to prevent violations through dereference of **nullptr**, pointer arithmetic, out-of-range, and subscripting are handled by a separate profile.

12.1. 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.
- A non-static **[[owner]]** will be deleted exactly once.

12.2. Attributes

The profile defines two attributes:

- **[[not_invalidating]]** this argument will not be invalidated by this function.
- **[[owner]]** this pointer must be deleted or passed on.

12.3. Library functions

The standard library provides a single function:

- **no_dangling()** an intrinsic function that asserts that its argument doesn't contain a dangling pointer.
- Standard library ownership abstractions are decorated with **[[owner]]** attributes where they are given the responsibility to **delete**. This needs to be turned into a list of cases.

References

- [HS'15] H. Sutter and N. MacIntosh: *Lifetime Safety: Preventing Leaks and Dangling*. <https://github.com/isocpp/CppCoreGuidelines/blob/master/docs/Lifetimes%20I%20and%20II%20-%20v0.9.1.pdf>.
- [Impl1] Experimental GCC branch <https://github.com/villevoutilainen/gcc/tree/p4324>
- [Impl2] Experimental Clang branch <https://github.com/cppalliance/clang/tree/profiles-init>
- [Impl3] Experimental compiler <https://6it.dev/>
- [GDR'25] Gabriel Dos Reis: *C++ Profiles: The Framework*. P3589R3. 2025-12-01.
- [GDR'26] Gabriel Dos Reis: Function Summaries for Scaling Static Code Analysis of C++. D4422R0. 2026-10-22.
- [BS'15] B. Stroustrup, H. Sutter, and G. Dos Reis: [A brief introduction to C++'s model for type- and resource-safety](#). Isocpp.org. October 2015. Revised December 2015.
- [BS'21] B. Stroustrup: [Type-and-resource safety in modern C++](#). SG22/WG21 document. P2410r0. 2021-07-12.
- [BS'24] B. Stroustrup: [Profile invalidation – eliminating dangling pointers](#). P3446R0. 2024-10-14.
- [BS'26] B. Stroustrup: *What are Profiles?* P3704R1. 2026-10-22.
- [BS'26b] B. Stroustrup: *Initialization Profile*. P4222R3. 2026-10-22.
- [MV'26] Michael Wong: Proposed Wording for the invalidation Profile. P4417R0. 2026-10-22.

Acknowledgements

Thanks to the many comments and suggestions from Ville Voutilainen, Mike Wong, and several others. Especial thanks to the profile implementers.