Proposal for C2y

WG14 N3952

Title: Response to design concerns with _Optional

Authors, affiliation:

Mr. Christopher Bazley, Arm. (WG14 member in individual capacity.)
Professor Dr. Martin Uecker, Graz University of Technology.

Date: 2026-08-16

Proposal category: Informative

Target audience: Committee

Abstract: N3876 claims that it "does not insist that any particular existing prior art be standardized exactly as-is." That disclaimer is misleading: by insisting that the `_Optional` qualifier occupy the same declarator position as Clang's nullability annotations, N3876 necessarily imports the consequences of placing the qualifier on the pointer itself, or else requires new exceptions to C's established rules for qualification conversion, lvalue conversion and type compatibility. Yet N3876 provides no coherent alternative semantics for nullability analysis, nor for the closely related problem of type variance. This paper explains the differences between the two approaches and examines safety and type-system consequences that N3876 does not adequately address. Finally, it reviews the standardization and implementation history leading to the proposed Technical Specification. It recommends that WG14 continue developing the Technical Specification using the syntax and semantics already selected.

Prior art: N3089, N3422, N3597, N3876, N3951.

Response to design concerns with _Optional

Reply-to: Christopher Bazley (chris.bazley.wg14@gmail.com),
Martin Uecker (ma.uecker@gmail.com)
Document No: N3952
Date: 2026-08-16

Table of Contents

How the standardization process works—or rather, does not

When the original RFC was discussed on the LLVM forums, Aaron Ballman wrote:

> *The proposal is largely experimental in terms of design and needs a stronger indication that a standards body really believes in this design (in terms of syntactic choices, at the very least) and data demonstrating how this feature catches bugs that cannot be caught by other features in the same space (not just due to missing diagnostics that are possible to implement).*

Later in the same thread, Aaron Ballman also wrote:

> *This exact proposal is not accepted, but if WG14 came back showing strong support for it, that would be new information for our community and would certainly be worth revisiting the discussion over.*

That standards-body support was subsequently obtained twice.

The 69th meeting of WG14 (Strasbourg, 22-25 January 2024), voted 15/0/6 in favour of adding something along the lines of N3089, and later, in Graz (24–28 February 2025), voted 12/6/7 in favour of presenting N3422's proposed syntax and semantics in a Technical Specification. In fact, this second vote understates support for the design, because

> *At least one No vote was to indicate support for direct adoption into the IS.*

The minutes of the Strasbourg meeting also record:

> *Several expressions of delight at the proposal within the room. Explicit thanks and encouragement to the author for bringing this and for the diligent preparation.*

Acting on this committee direction entailed considerable additional work to resolve outstanding design questions, to develop an accurate and complete specification of the new qualifier, and to harden the two implementations of that specification that currently exist (SDCC and the original fork of Clang).

N3876 does not treat those outcomes as sufficient evidence that a standards body believes in the design. Instead, it presents the continuing opposition of its authors as a reason for WG14 to reconsider its own direction. WG14's judgment was invoked when the absence of standards-body support furnished a reason to reject the RFC, but WG14's judgment is discounted now that the requested support has been obtained.

Aaron Ballman's request for "data demonstrating how this feature catches bugs that cannot be caught by other features in the same space" is impossible to satisfy conclusively. Almost any example can be dismissed by positing additional diagnostics or a sufficiently powerful analyser. More importantly, diagnostic power is not the only metric by which such a feature should be assessed. Nevertheless, N3597 provided a detailed and frank analysis of use of the `_Optional` qualifier in real programs and libraries, including an example of a bug found through its use without flow-sensitive analysis. Many more such bugs have been found since.

The authors of N3876 claim that

> *The rationale for why only model (2) is supported… is not justified by existing implementation practice or a thorough investigation of how code is written in practice.*

but they neither cite nor substantively engage with N3597, which was written with exactly that purpose in mind. Have the authors of N3876 undertaken a thorough investigation of why neither model 1 nor model 3 has become a de facto standard for C programmers, and how practical (or not) the application of those models might be to large real-world C language projects?

Dismissing experience of using the qualifier in real-world projects as

> *appeals to authority like K&R C about "economy of expression"*

is patently absurd. Every reputable authority recognises the importance of economy of expression, including the creator of C++:

> *"I want concise expression of ideas. I actually want to be able to read your code and vice-versa. If it's a mess of bits and bytes, often I can't."*

(CppCon 2016: Bjarne Stroustrup "The Evolution of C++ Past, Present and Future")

Despite N3089 documenting both a working modification of Clang and its successful use on parts of Arm's user-space Mali driver, N3876 asserts that the proposed model "is only supported by SDCC." Yet there is no technical reason why it should not be supported by Clang too: the fork passes the Clang test suite, it is currently used to perform CI testing on a number of projects hosted on GitHub, and has found real bugs as a result of diagnostics that it produces. The differences between this fork (810f32a0) and mainline (4066590d) are a relatively modest 2064 insertions and 156 deletions (+1908 lines), including tests. The authors of N3876 may reasonably characterize this experience as limited; they cannot reasonably treat it as non-existent.

Their position creates a catch-22: a Technical Specification provides a stable, public specification against which implementations can be developed and deployment experience gathered. N3876 invokes the design's limited deployment against publishing the very instrument intended to broaden that deployment.

The solution that N3876 describes as model 2 ("mark only pointers which can be null") is a mischaracterisation of the facility afforded by the `_Optional` qualifier, since existence of qualified types does not preclude interactions with any of the other models.

Even if the authors of N3876 are correct that a type-based solution is impractical, despite growing evidence to the contrary, it is not their responsibility to prevent implementers and users from discovering that for themselves. Opposition from one implementation community should be weighed as evidence; it should not preclude other implementations and users from gathering contrary evidence.

What rational incentive is there for volunteers to contribute their time to improving the C programming language if neither implementation experience nor committee consensus is ever permitted to settle important design questions?

Reopening of design decisions

The design of the `_Optional` qualifier was discussed internally at Arm with experienced C and C++ programmers at every stage of its development before it was presented to WG14. Every potential drawback and counterproposal that has since been made was seriously considered at that time. The design has also been discussed extensively within WG14, and several apparent drawbacks have thereby been mitigated.

The authors of N3876 ask the committee to

> *leave aside the academic discussion about what a qualifier is or isn't*

But this is by no means an academic point. The ISO C standard specifies language syntax and semantics in terms of established terminology. It may be that a qualifier is not ultimately the appropriate mechanism for tracking pointer nullability, although in our view the evidence does not support that. Regardless, a qualifier-based mechanism deserves the chance to be properly evaluated. Prejudice against the mechanism does not constitute evaluation.

N3876 states:

> *However, the committee has since strengthened their consensus requirements and there is a risk that the planned TS will fail to gain consensus.*

The detached phrasing omits relevant context: two of N3876's authors recently participated in another effort to reopen a syntactic decision after WG14 had adopted it. It is hard to avoid the conclusion that this controversy contributed to a perceived need for stronger consensus requirements.

At its January 2024 meeting, WG14 voted 12/6/3 in favour of adding something along the lines of named loops to C2Y (N3195). That proposal was refined as N3355, which was adopted into the C2Y working draft following the September 2024 meeting. Its syntax uses an ordinary label to name a loop or `switch` statement:

```
outer:
for (int i = 0; i < n; ++i) {
    /* ... */
    break outer;
}
```

Shortly after that adoption, Ballman and Keane submitted N3377, which sought to replace the syntax already selected by WG14 with a new position for the loop name:

```
for outer (int i = 0; i < n; ++i) {
    /* ... */
    break outer;
}
```

N3377 raised valid questions about label scope and macro expansion. The relevant procedural point is narrower: adoption by WG14 was not treated as settling the syntactic design. Instead, members of the Clang community continued seeking to replace the committee's chosen syntax after it had entered the working draft. At Graz, however, the committee voted 6/11/9 on whether it wished to see a future paper changing the syntax and recorded "no direction."

Meanwhile, the adopted feature received positive reactions from programmers outside the committee. Public discussion welcomed the ability to identify an enclosing loop directly and expressed interest in bringing the same facility to C++.

Jan Schultke, author of the WG21 paper, defended N3355's reuse of familiar syntax over N3377's invention of new syntax for the name:

> *Both syntaxes (N3355, N3377) work just fine from a technical viewpoint, but N3355 is arguably much better because it's using familiar syntax instead of inventing something novel.*

Other commenters agreed, for example:

> *It reuses syntax that is already familiar, rather than inventing a new and confusing way of doing the same. The syntax `for outer(...)` looks like weird function call.*

These comments suggest that the syntax rejected by members of the Clang community was neither generally incomprehensible nor self-evidently unsuitable for C.

The scoping objection raised by N3377 is being addressed by N3658, for which WG14 subsequently expressed direction by 18/7/1. This is relevant to the design of `_Optional` because it shows that design issues can be addressed, if the committee provides clear direction *and its decisions are not reversed*.

A similar pattern now emerges with `_Optional`: WG14 first expressed strong directional support for N3089 and subsequently supported presenting N3422's specific syntax and semantics in a Technical Specification. N3876 nevertheless asks the committee to reopen the qualifier placement because the Clang community continues to oppose it.

Technical decisions must remain open to correction when new evidence reveals a real defect. But repeated reconsideration imposes real costs, particularly when the objections were already known before the original decision. N3876 itself states:

> *None of the information presented in this paper should be new to WG14*

This is not a point in its favour. A standardization process cannot function effectively if committee decisions are treated as provisional until they agree with the preferences of a particular implementation community.

Overstatement of opposition to the design

N3876 gives a one-sided account of the reaction to the design of `_Optional`, based on

> *informal polls and communications with users which are not publicly available to cite*

Meanwhile, it omits expert opinion and on-the-record support for the design of `_Optional`.

The design for `_Optional` was first presented to Arm's C++ guild and at another internal development tools conference. Had it received a negative reaction in those contexts, it never would have reached the attention of WG14.

The convener of the C++ guild later offered his enthusiastic support in the discussion on the LLVM forums:

> *"First of all, I think this is a great proposal. I think it's bold, and I think it's the kind of bold that C (and C++) would need to step up the memory safety game."*

(Emil Ohlsson, Feb 2023)

N3876 invokes repeated expert reaction that the qualifier is "in the wrong position" as evidence that the design is inappropriate in practice. The public reflector record is not so one-sided. The reflector also records experienced WG14 members arguing that pointer-side placement would repeat an existing design mistake:

> *Now, Chris fights a similar fight. People come up with arguments why `_Optional` is wrong "it must be on the pointer, because..."*
>
> *But similar to my fix, if you really understand the language both from practical experience **and** their abstract rules, then at some point you will hopefully realize that `_Optional` has to work exactly as Chris specified it.*
>
> *Ultimately, the argument "it says something about the pointer" is incorrect, because `_Optional` does not say anything meaningful about the pointer at all (as you have pointed out, it would then be meaningless as regular pointers can already be null).*
>
> *So what does `_Optional` say?*
>
> *In general, a qualifier in C says something about **accesses**. Once an object is accessed, the qualifier becomes irrelevant and the qualifier is removed during lvalue conversion.*
>
> *But null pointers are just valid values of a pointer type that can be loaded and stored without any restriction. Whether a pointer is null or not does not add any restriction on accesses to the pointer object itself! So why should then the qualifier be on the pointer? It should not!*

Uecker, SC22WG14.26923

Jens Gustedt endorsed this analysis:

> *Martin, you are exactly right.*
>
> *As another data point: we already have a qualifier that only works with pointers.*
>
> *And we got it all wrong, one of the reason why even the specification of restrict is so messy, is because it tries to map a feature on a pointer that basically is a property of the to-be-accessed object, not of the pointer itself.*
>
> *Restrict would be so much simpler if it would be qualifying the pointee and not the pointer. So we could perhaps think of a qualifier for objects (or maybe just an attribute) that replaces restrict.*

Gustedt, SC22WG14.26935

The design of `_Optional` is also properly understood by Alex Celeste:

> *`_Optional` in this model is propagating like `const`. It is not supposed to be removed on assignment and therefore assigning to non-`_Optional` not only shouldn't work, it's what it is guarding **against**.*

Celeste, SC22WG14.27052

These are not expressions of aesthetic preference: they correctly identify the qualifier's position as the mechanism by which constraints are preserved across assignments and calls.

Support for the `_Optional` qualifier's entirely conventional position has nothing to do with style wars about the best placement of the asterisk in declarations, and cannot be readily dismissed as

> *the author's opinions of how C code should be written*

as N3876 attempts to do.

The idea that a qualifier at the topmost level of type derivation is an appropriate mechanism for constraining values rather than accesses is simply fallacious. Even the `restrict` qualifier does not fall into that category because it is not a constraint: it is a permission (for optimisers, not programs).

The hypothetical alternative language design argued for by N3876 is unproven and incompletely specified; nor is it supported by

> *40+ years of prior art the committee could be drawing on*

The "prior art" cited by N3876 **is not prior art for a type-based solution**.

Further, the statement by the authors of N3876 that a design that adheres faithfully to C's established type qualifier rules is

> *not ready to be inflicted on users*

is unnecessarily hyperbolic. It is also contradicted by implementation and deployment experience.

The `_Optional` qualifier is simply a tool, like `const`. Its existence implies no compulsion, nor any single source of truth.

Source-code annotation language

N3876's citation of Microsoft source-code annotation language (SAL) does not establish prior art for its proposed placement `int * _Optional`. On the contrary, SAL annotations conventionally precede an entire parameter declaration, instead of qualifying a particular `*` in a declarator.

Microsoft's documentation says:

> *Optional pointer parameters*
>
> *When a pointer parameter annotation includes `_opt_`, it indicates that the parameter may be null.*

For example, the standard function `perror` can be annotated using SAL as follows:

```
void perror(_In_opt_z_ const char *msg);
```

This more closely resembles usage of the `_Optional` qualifier than it does usage of Clang's nullability attributes. Here is the same declaration with the `_Optional` qualifier:

```
void perror(_Optional const char *msg);
```

In other words, the `char` passed by reference is optional. Here is the same declaration with Clang's nullability attribute:

```
void perror(const char *_Nullable msg);
```

The above example also serves to refute a common complaint that the name of the new qualifier has no precedent.

Another example is the Annex K function, `fopen_s`, which can be annotated using SAL as follows:

```
errno_t fopen_s(_Outptr_result_maybenull_ FILE **stream,
                _In_z_ const char *filename,
                _In_z_ const char *mode);
```

`_Outptr_result_maybenull_` means that `stream` itself must not be null, but the pointer written to `*stream` may be null. This would be expressed with `_Optional` as:

```
errno_t fopen_s(_Optional FILE **stream,
                const char *filename,
                const char *mode);
```

In other words, the `FILE` returned by reference is optional. The corresponding declaration with Clang's nullability annotations would be:

```
errno_t fopen_s(FILE *_Nullable *_Nonnull stream,
                const char *_Nonnull filename,
                const char *_Nonnull mode);
```

Pedantic insistence on the position of `_Nullable` in the centre of such declarations seems like the kind of "academic" overthinking that the authors of N3876 accuse WG14's experts of indulging in.

Of course, it is possible to point to more complex parameter declarations that *do* require a qualifier to be part of a declarator, but there is a well-established legal maxim that "hard cases make bad law", which we believe applies in this case.

A false equivalence

The authors of N3876 assert that

> *the Clang community requires the pointer to be annotated rather than the pointee, as in* `int * _Optional`

Leaving aside the question of whether a value for which storage is allocated, and that can be copied, compared, converted or deallocated, can accurately be described as “optional”, this demand betrays a fundamental misunderstanding of the differences between the two mechanisms and brings into question the premise of their paper.

The `_Optional` qualifier is not a SAL mechanism for annotating pointers as maybe null; it is a first-class language feature that changes the type of lvalues. Nor does the presence or absence of `_Optional` indicate whether a pointer can be null.

In contrast, Clang’s annotations imply certainty that its analyser cannot guarantee. Path-sensitive analysis augmented by source code annotations produces very different results from a type-based approach that can usefully be enhanced by path-sensitive or flow-sensitive analysis.

Notably, the fact that Clang produces no diagnostic for the following program is dangerous:

```
static int bar(int *_Nullable y)
{
    return *y;
}

int foo(int *x)
{
    return bar(x);
}
```

The programmer who wrote the `bar` function might reasonably expect pointer dereferences therein to be checked. They are not. Surprisingly, it makes no difference whether bar has internal linkage or not.

This reply on the LLVM forums in 2023 is particularly illuminating:

> *So basically back in 2015 the static analyzer people realized that _Nullable and path-sensitive analysis aren’t meant for each other. This is why only _Nonnull-related checks are enabled by default in the static analyzer but _Nullable-related checks are in permanent alpha and will probably never be productized.*

In contrast, the version using the `_Optional` qualifier does produce a diagnostic, because the presence or absence of an optional qualifier does not signify that a value can or cannot be null. Instead, it provides concrete and actionable type information:

```
static int bar(_Optional int *y)
{
    return *y; // diagnostic message
}

int foo(int *x)
{
    return bar(x);
}
```

We believe that the semantics of Clang's nullability annotations are surprising, unstable under ordinary maintenance, computationally expensive to convert into diagnostics, dangerous and unteachable, but that is not our principal point.

It is misleading to suggest that the difference between an actual qualifier that pertains to lvalues and an annotation expressed using qualifier syntax comes down to "academic debates" or a question of "style". The differences between a type-based and an SAL-based approach cannot be reconciled simply by playing around with the syntax of the pointer declaration. For that reason, we consider it **preferable** that the two approaches can readily be distinguished syntactically.

The maintainers of Clang have had more than a decade to fix null pointer analysis. Why should any aspect of an alternative fully specified and proven type-based approach be revised based on a false equivalence between the two?

It would be irresponsible for WG14 to decide the fate of the `_Optional` Technical Specification based on a misunderstanding of the language extension described therein.

Type deduction in C++

N3876 gives the following example of type deduction:

```
template <typename Ty>
void func(Ty *ptr, Ty val);
_Optional int *ptr = foo();
func(ptr, 12);
```

and explains

> *In C++, this causes `Ty` to deduce to both `_Optional int` and `int` because the qualifier is not ignored for type deduction, so the deduction is ambiguous.*

First let's expand this into a complete example:

```
_Optional int *foo(void);
template <typename Ty> void func(Ty *ptr, Ty val);
int main(void)
{
    _Optional int *ptr = foo();
    func(ptr, 12); // error: no matching function for call to 'func'
    return 0;
}
```

As expected, the fork of Clang that supports the `_Optional` qualifier reports:

```
<source>:8:5: error: no matching function for call to 'func'
    8 |     func(ptr, 12);
      |     ^~~~
<source>:3:29: note: candidate template ignored: deduced conflicting types for
parameter 'Ty' ('_Optional int' vs. 'int')
    3 | template <typename Ty> void func(Ty *ptr, Ty val);
      |                             ^
```

The C programming language does not provide C++-style templates, so we must assume that the above example is part of a C++ program.

The whole point of the `_Optional` qualifier is to ensure that a pointer to an optional-qualified value is not treated like a pointer to an unqualified value (as in the example above). Whilst a C++ implementation could special-case template argument deduction so that `_Optional` is ignored when deducing `Ty` from `ptr`, doing so would be unsafe: it would allow `func` to dereference the resulting unqualified lvalue without the check that the qualifier was introduced to require.

The real solution is simple, and it is the same as that used for any call to a function that does not accept a pointer to an optional-qualified type: explicitly remove the qualifier in such a way that flow-sensitive analysis can check that the call is safe:

```
_Optional int *foo(void);
template <typename Ty> void func(Ty *ptr, Ty val);
int main(void)
{
    _Optional int *ptr = foo();
    if (ptr)
        func(&*ptr, 12);
    return 0;
}
```

With this modification, type deduction works perfectly well.

Some programmers may resent the need to explicitly remove the qualifier. We then need to ask who wrote the declaration of `foo` and why they decided it should return a pointer to an optional-qualified type if they did not intend its return value to be checked for null before being dereferenced.

Let's assume that the problematic declaration belongs to a C language project maintained by a third party who does not care about C++. That already means that, without compiler extensions:

- Compound literals in the same header are not valid C++:
  `#define NULL_HANDLE ((handle_t){0})`
- Designated initialisers in the same header are not valid C++17 (and may not be valid C++20 either):
  `handle_t new_data(int v) {handle_t h = {.second = v, .first = 0}; return h;}`
- Static minimum array bounds in the same header are not valid C++:
  `void send_data(handle_t h[static 1]);`
- Uses of `_Generic` in the same header are not valid C++:
  `#define get_str(h) _Generic(h, handle_t *: get_str, const handle_t *: get_cstr)(h)`
- Parameters of variably modified type in the same header are not valid C++:
  `void set_values(int n, handle_t (*h)[n]);`

The `_Optional` qualifier is not the biggest problem with the header file in C++; it is the smallest, because it is the most trivial to solve:

```
#define _Optional
#include <foolib.h>
#undef _Optional
```

Even if the solution were not so simple, it would be invidious to argue that C should not have first-class language features just because they might be used in header files. Compatibility is a legitimate consideration, but it cannot reasonably require every new C feature to preserve C++ template deduction or make C a syntactic subset of C++.

If the design of `_Optional` is indeed "hostile" to type deduction in C++, then so too are `const` and `volatile`:

```
int *foo(void);
template <typename Ty> void func(Ty *ptr, Ty val);
int main(void)
{
    const int *ptr = foo();
    func(ptr, 12); // error: no matching function for call to 'func'
    return 0;
}
```

We believe that no apology is necessary for the commonplace semantics of type qualifiers.

Trait interfaces in C++

N3876 also says:

> *Similar concerns surround existing C++ trait interfaces like `std::add_pointer`, `std::remove_pointer`, etc. as well as the new reflection facilities coming in C++26.*

Under the ordinary behaviour of C++ type traits, `std::remove_pointer_t<_Optional int *>` would be `_Optional int`, just as `std::remove_pointer_t<const int *>` is `const int`; applying `std::add_pointer_t` would reconstruct the original pointer type.

A C++ extension to support the `_Optional` qualifier would need to specify whether an optional-qualified type may be named outside the contexts permitted by the `_Optional` Technical Specification, or whether it should be removed by `std::remove_pointer_t`.

N3876 neither identifies a concrete failure nor specifies the behaviour it considers necessary. A new qualifier would require some integration with C++ traits wherever it is placed. Even if the syntax demanded by N3876 were used instead of the syntax implemented by SDCC, it is not clear that the natural behaviour of discarding such a top-level qualifier would be correct:

```
using P = int *_Optional;
using T = std::remove_pointer_t<P>; // int
using Q = std::add_pointer_t<T>;    // int *, not int *_Optional
```

This is an integration question for a hypothetical C++ extension, not evidence that the C design is defective.

Macro madness

N3876 gives the following example macro definition:

```
#if NEW_ENOUGH
#define OPTIONAL_PTR(type) _Optional type *
#else
#define OPTIONAL_PTR(type) type * _Nullable
#endif
```

It is reasonable to conjecture how authors of headers might choose to combine Clang's nullability annotations with use of the `_Optional` qualifier in a single declaration but deliberately providing broken code does not make for a compelling example. We see no reason why anyone sufficiently aware of the `_Optional` qualifier to write such a macro should not also understand its correct placement.

The following complaint shows that the authors of N3876 are already aware of the de facto standard solution to reliable type name composition using macros:

> *Further, the syntactic choice is irregular in that it cannot be applied to all type derivations without resorting to `typedef`/`typeof` or allowing for qualified function types.*

That de facto standard solution would be:

```
#define OPTIONAL_PTR(type) _Optional typeof(type) *
```

The definition of `OPTIONAL_PTR` that expands to use `_Nullable` instead of `_Optional` does not work without `typeof` in the general case either, which has **nothing to do with placement of the qualifier**:

```
#define OPTIONAL_PTR(type) type * _Nullable

// error: brackets are not allowed here; to declare an array, place the brackets
after the name
OPTIONAL_PTR(OPTIONAL_PTR(int [3])) a;

// error: expected unqualified-id
OPTIONAL_PTR(OPTIONAL_PTR(int (float))) b;
```

Ultimately, the question of what might be "the correct macro" seems unnecessary because declarations of pointers to qualified types can also incorporate annotations, if so desired:

```
_Optional int (*_Nullable _Optional *_Nullable a)[3];
```

This is undeniably ugly, but it is ugly in exactly the same way as a declaration of a "nullable" pointer to a `const`-qualified "nullable" pointer to a `const`-qualified array.

Whereas the `_Optional` qualifier permits assignment to a pointer to a so-qualified type, `_Nullable` is not a permission: a value that Clang's analyser believes could be null can also be assigned to an unannotated pointer. Clang's analyser only diagnoses assignments of values it believes could be null to objects that have been annotated with `_Nonnull`. Together with the fact that Clang's analyser does not treat external declarations annotated with `_Nullable` as a reliable source of information, such a declaration in a header has no real power.

For example, no diagnostic message is generated for the following program:

```
int (*_Nullable *_Nullable a)[3];
int main()
{
    int (*_Nonnull *_Nonnull b)[3];
    b = a; // no diagnostic
    (void)b;
    return 0;
}
```

A meaningful example would therefore need to be a function definition, which is less likely to appear in a shared header file. For example, Clang's analyser diagnoses assignment of this return value:

```
int *_Nonnull foo(int *_Nullable a)
{
    return a; // diagnostic
}
```

A compiler that supports both `_Optional` and `_Nullable` does not require both to be specified:

```
int *_Nonnull foo(_Optional int *_Nullable a)
{
    return a;
}
```

The fact that a constraint violation is diagnosed when type `_Optional int *` is assigned to type `int *` makes the additional warning from Clang's analyser redundant. In general, the semantics of Clang's annotations and C's type system are so different that we do not believe such mixed usage makes sense. They are not simply different spellings of the same thing.

In summary, use of the `OPTIONAL_PTR` macro proposed by the authors of N3876 to declare function parameters using `_Nullable` or `_Optional` is flawed:

- In function declarations, any such existing use of `_Nullable` is principally documentation.
- In function definitions, any such existing use of `_Nullable` has semantics that are incompatible with those of the `_Optional` qualifier.

Type variance

The question of whether Clang's nullability annotations are part of the type of a so-declared object or parameter is not merely of academic interest. The fact that they lie outside the type system makes them dangerous in other unexpected ways:

```
static void accept_nullable(int *_Nullable y)
{
    if (y) *y = 1;
}

static void require_nonnull(int *_Nonnull y)
{
    *y = 1;
}

void pass_nullable(void (*)(int *_Nullable));

void foo(void)
{
    pass_nullable(accept_nullable);
    pass_nullable(require_nonnull);
}
```

In contrast, with or without the enhancements to type variance proposed by N3674, use of the `_Optional` qualifier requires the same programmer error to be diagnosed:

```
static void accept_optional(_Optional int *y)
{
    if (y) *y = 1;
}

static void require_int(int *y)
{
    *y = 1;
}

void pass_optional(void (*)(_Optional int *));

void foo(void)
{
    pass_optional(accept_optional);
    pass_optional(require_int);  // constraint violation
}
```

We are aware of no proposal to extend type variance to Clang's nullability attributes.

Qualified function types

N3876 says that

> *Further, the syntactic choice is irregular in that it cannot be applied to all type derivations without resorting to `typedef`/`typeof` or allowing for qualified function types.*

Qualified function types are exactly the intention; their use is not a misfeature. Unlike attributes or annotations, qualified types are amenable to type variance. They are also likely to have uses in other language extensions in future; therefore, there is good reason for WG14 to consider whether existing syntax for expressing them is adequate.

The fact that Clang already spells qualified function types without `typedef` or `typeof` in its diagnostic messages suggests a natural candidate for their eventual direct syntax.

For example, this fragment:

```
typedef int foo_t(void);
_Optional foo_t *foo;
foo_t *foo;
```

yields the following output:

```
<source>:3:8: error: redefinition of 'foo' with a different type: 'foo_t *' (aka
'int (*)()') vs '_Optional foo_t *' (aka 'int (_Optional *)()')
    3 | foo_t *foo;
      |        ^
<source>:2:18: note: previous definition is here
    2 | _Optional foo_t *foo;
      |
```

This is grounds for further investigation of the possible uses and means of expressing qualified function types, not a reason to dismiss the `_Optional` Technical Specification.

N3876 further raises the objection that

> *The proposal says that because _Optional is new, we can standardize its semantics when applied to function types, but never actually attempts to describe what that means or what the impacts are, so it appears to be left in an irregular state.*

That objection has been overtaken by the current draft of the `_Optional` Technical Specification (N3951). Section 6.5.3.1 p7 expressly supersedes the undefined behaviour allowed by C23 and makes optional-qualified function types subject to the constraints, semantics and recommended practice specified in the remainder of the clause. Those general provisions define the treatment of optional-qualified function types without requiring separate rules.

Conclusion

N3876 does not establish that the placement of Clang's annotation is suitable for a type qualifier, nor does it provide complete alternative semantics for conversion, checked access or type variance.

Its anecdotal objections do not outweigh the committee's previous direction, the two working implementations, or the deployment experience gathered since the committee provided strong direction for a language extension along the lines of N3089.

We recommend that WG14 continue development of the `_Optional` Technical Specification using the syntax and semantics already selected by the committee.