Information for WG14
WG14 N3876

Title:	Design concerns with `_Optional`
Author, affiliation:	Aaron Ballman, Intel Corentin Jabot, Open to New Possibilities Erich Keane, Nvidia Shafik Yaghmour, Intel Eli Friedman, Qualcomm
Date:	2026-04-20
Proposal category:	Informative
Target audience:	WG14 and the WG14 `_Optional` study group

Abstract: Provides an explanation as to why the current design proposed for the TS has multi-year, sustained opposition from some implementations.

Design concerns with `_Optional`

Reply-to: Aaron Ballman (aaron@aaronballman.com) , Corentin Jabot (corentin.jabot@gmail.com), Erich Keane (ekeane@nvidia.com), Shafik Yaghmour (shafik.yaghmour@intel.com), Eli Friedman (efriedma@quicinc.com)
Document No: N3876
Date: 2026-04-20

Summary of Changes

N3876

- Initial feedback

Introduction

WG14 N3089 was initially seen in Strasbourg in 2024 and proposed a qualifier in support of null pointer analysis. Aaron Ballman provided feedback from the Clang community on why the feature proposed by that paper was previously rejected by the Clang community for inclusion as an extension, but the feature had strong along-the-lines support (15/0/6) within the committee, including from Clang contributors. The follow-up paper (WG14 N3422) was seen in Graz in 2025 and continued with the same design as the previous paper. The Clang community re-affirmed their opposition to the design presented in the paper. The committee was polled on whether to forward the proposed specific syntax and semantics to a TS, which had less support (12/6/7) than it previously did but still met the consensus requirements the committee had in place at that time. However, the committee has since strengthened their consensus requirements and there is a risk that the planned TS will fail to gain consensus. With hopes of avoiding that outcome, this paper outlines the Clang community concerns which have previously been raised within committee discussions. This paper is the official position of the Clang community as determined by the Clang Area Team, the current members of which are listed as authors on this paper.

Background

Any sufficiently complex program eventually considers the question of how to design APIs involving pointers. Are all pointers nonnull unless otherwise marked? Can any pointer be null unless otherwise marked? Should there be no default assumptions about whether a pointer can or cannot be null? This leads to four general models: (1) mark only pointers which must be nonnull, (2) mark only pointers which can be null, (3) mark all pointers as to whether they can or cannot be null, and (4) mark no pointers with any extra information.

When a program consistently follows a particular model, deviation from that model tends to be uncommon and annotating the source can be viewed as “noise” rather than helpful. However, the biggest challenge in practice is when two bodies of source code following different models

come in contact with each other. For example, when an application assumes model (2) but a library it depends on assumes model (1), it is easy to miss cases where the application passes a potentially null pointer to a library interface which expects a nonnull pointer in the absence of any annotations or whole-program analysis.

Further, a broad class of users want to write code which uses the new standards features as they become available but still fall back to alternative approaches such as compiler extensions when the new standards feature isn't available. This sometimes is because of language standard versions where they want their code to compile in both C17 and C23 modes and sometimes it is because of compiler versions where they want their code to compile in both Clang N and Clang N + 1. Users prefer to incrementally adopt features, so requiring major modifications to the code base to use the new functionality is a barrier to adoption in practice.

Current Design Compared to Prior Art

The current design based on N3422 supports only model (2), however, existing prior art either supports model (1) or model (3). Model (4) is also prior art and is the status quo in C today, but it requires whole program analysis to attempt to support null pointer analysis and thus is inappropriate for standardization due to well-established implementation burdens.

Examples of model (1) are `__attribute__((nonnnull))` and C99's static array extents. Some implementations support the attribute being written on a function to specify which parameters (by index) are nonnull pointers. Some implementations also allow applying the attribute directly to parameters instead of using an index. Static array extents only apply to function parameter declarations. In all cases, the call boundaries are annotated rather than arbitrary pointer declarations and the annotations are not encoded as part of the type system.

Examples of model (3) are `_Nonnull`/`_Nullable` and Source Annotation Language (SAL) attributes. `_Nonnull` and `_Nullable` (along with `_Null_unspecified`) are syntactically qualifiers on pointer types but are not currently part of the type system in terms of their effects. They may be written on arbitrary pointers rather than only function parameters. SAL annotations solve a broader problem than just null pointer analysis, but the relevant markings are written on the declaration of a parameter, such as `_In_` for a non-null pointer and `_In_opt_` for a potentially null pointer. SAL annotations are also not part of the type system.

The rationale for why only model (2) is supported goes into the author's opinions of how C code should be written, with appeals to authority like K&R C about "economy of expression" but is not justified by existing implementation practice or a thorough investigation of how code is written in practice. Model (4) is the most prevalent in the wild because most C implementations do not have any pointer analysis markings. However, there are more implementations supporting model (1) than model (3). Model (2) is only supported by SDCC and only very recently (in an unreleased version of SDCC, as of Apr 2026), making it the least-used practice.

The Clang community does not insist that any particular existing prior art be standardized exactly as-is. However, we strongly encourage WG14 to consider a model that supports the full

generality offered by model (3) in a way that allows for incremental adoption. Standardizing only model (1) would be acceptable given its prevalence in existing practice but limits what C programmers can express and should come with significant justification for those limitations.

Qualifier Position

The design of the proposed feature also places the qualifier in a novel position: despite the feature being about whether the pointer's value might be null, the qualifier is written on the pointee instead of the pointer. N3422 goes to great lengths to explain why this is the correct design approach. Despite that, the most frequent feedback given on initial contact with the feature is "the qualifier is in the wrong position" – this feedback has been given by multiple Clang community members on the initial proposal to our community, multiple WG14 members on the committee reflectors (SC22WG14.26919, SC22WG14.34833, et al), and has been commonly received in informal polls and communications with users which are not publicly available to cite. Regardless of academic debates as to whether that's correct or not, the fact that so many people have the same feedback clearly demonstrates how inappropriate the design is in practice. If around half the people who encounter it *within the committee of C experts* need a lengthy explanation, it's not ready to be inflicted on users. But leaving aside the academic discussion about what a qualifier is or isn't, that design choice creates practical deployment concerns for implementations which already support nullability analysis because all of the prior art places the marking on the pointer rather than the pointee. For users who need portability to older language standards and compiler versions, the best-case scenario is one where the implementation already supports model (3) because that can express the same semantics as `_Optional`, so one might expect it to be the most straightforward case:

```
void func(int * _Nullable ptr);
```

However, you cannot simply drop in `_Optional` as a replacement because that would be a constraint violation due to the novel location of the qualifier. Instead, the user must generate two different declarations depending on which is supported:

```
void func(_Optional int *ptr); // Newer style using standards features
void func(int * _Nullable ptr); // Otherwise, older style w/extensions
```

and the most natural way to generate different declarations depending on external pressures is via macros and the preprocessor. A naïve implementation would be along the lines of:

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

void func(OPTIONAL_PTR(int) ptr);
```

and it would indeed work for this case despite reasonable claims that it makes the declaration of the function less readable. Unfortunately, that implementation is also incorrect; it would fail with an optional pointer to an optional pointer to `int` because `OPTIONAL_PTR(OPTIONAL_PTR(int))` would expand to `_Optional _Optional int * *`. The correct macro requires you to put the qualifier to the right of the type specifier (reigniting “east const” vs “west const” debates):

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

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. 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.

Finally, the rationale in the proposal is that this qualifier position is necessary for correctness due to qualifiers being dropped silently on rvalue conversion. Alternative design approaches exist and appear to be discarded without due consideration. For example, because `_Optional` is new, we can standardize semantics for what happens on rvalue conversion so that the qualifier is not dropped.

In the face of these problems, the Clang community requires the pointer to be annotated rather than the pointee, as in `int * _Optional`.

Shared C and C++ Headers

Null pointer analysis is not unique to C. C++ has similar user needs for distinguishing whether pointers must be nonnull or can be null. While C++ has language features that can help with this (like templates and RAII-based smart pointer types), those features are not present in C and so alternative solutions must be found for the shared header space. Function declarations are one of the most common kinds of declarations found in header files shared between C and C++ translation units and are a common source (or sink) for pointers traveling between components which may have different nullability models, so this shared header space is critical for the success of the feature.

This proposal has not yet been discussed in the C and C++ Compatibility study group, which seems like a procedural oddity that should be corrected before finalizing the contents of the TS. However, the Clang community raised significant concerns about the viability of this feature as a conforming extension to C++. Consider things like partial template deduction, as in:

```
template <typename Ty>
void func(Ty *ptr, Ty val);
```

```
_Optional int *ptr = foo();
func(ptr, 12);
```

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. When the qualifier is correctly associated with the pointer rather than the pointee, the semantics work as a C++ programmer would expect and `Ty` would deduce unambiguously to `int`. 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.

The Clang community would like the C and C++ Compatibility Study Group to be involved in the discussion of this proposal. We don't expect the proposal to get full WG21 committee blessing for standardization in C++, but we believe it is inappropriate to standardize a feature squarely in the shared header space before discussing the impact on the implementation ecosystem outside of WG14. Further, the Clang community feels the current proposal cannot be supported in C++ as a conforming extension and expects compelling justification from WG14 as to why that is the case given how much prior art already exists which does not have this deficiency.

Suitability for a Technical Specification

According to ISO:

A Technical Specification addresses work still under technical development, or where it is believed that there will be a future, but not immediate, possibility of agreement on an International Standard. A Technical Specification is published for immediate use, but it also provides a means to obtain feedback. The aim is that it will eventually be transformed and republished as an International Standard.

If the purpose of a TS is broadly to gather feedback on a proposed design idea that we don't yet believe is ready for the IS but we believe could be ready (perhaps with small adjustments) if we got more deployment experience with the functionality, it can be argued that a TS is inappropriate for this feature. This design space has 40+ years of prior art the committee could be drawing on rather than experimenting with a novel design that was already rejected by one of the major C implementations. Given the Clang community's sustained objections to the design of the feature, it's questionable whether there will ever be agreement on inclusion in an International Standard even if a TS is successfully published. The feature may be plausible to deploy for some other implementations but that does not address the design concerns we have for our implementation's ability to deploy and support the feature.

It may be tempting to view this as a non-issue because Clang is only one C implementation out of many. However, Clang has numerous unique downstream implementations which inherit from upstream Clang and add their own extensions or "secret sauce" specific to the needs of their

users. While those implementations can support things upstream Clang does not, there is little reason to believe that will be the case for this feature.

Conclusion

None of the information presented in this paper should be new to WG14; this feedback was provided to the proposal author when this feature was rejected by the Clang community and was provided again during WG14 meetings. The Clang community continues to support WG14's efforts to standardize a feature in this space but we encourage the committee to consider a significantly modified design that is less likely to receive National Body-level opposition.

Acknowledgements

We would like to recognize the following people for their help in this work: Vlad Serebrennikov and Joshua Cranmer.

References

[N3038]
_Optional: a type qualifier to indicate pointer nullability. Bazley. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3089.pdf

[Clang Community RFC]
[RFC] _Optional: a type qualifier to indicate pointer nullability. Bazley. https://discourse.llvm.org/t/rfc-optional-a-type-qualifier-to-indicate-pointer-nullability/68004

[N3422]
_Optional: a type qualifier to indicate pointer nullability (v2). Bazley. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3422.pdf

[GCC Nonnull Attribute]
Common Attributes (Using the GNU Compiler Collection (GCC)). Unknown. https://gcc.gnu.org/onlinedocs/gcc/Common-Attributes.html#index-nonnull

[Clang Nullability Attributes]
Attributes in Clang. Unknown. https://clang.llvm.org/docs/AttributeReference.html#nullability-attributes

[Source Annotation Language]
Understanding SAL. Microsoft. https://learn.microsoft.com/en-us/cpp/code-quality/understanding-sal?view=msvc-170

[ISO Deliverables]

ISO - Deliverables. International Organization for Standardization.
https://www.iso.org/deliverables-all.html#TS