Proposal for C2y

WG14 N3422

Title: _Optional: a type qualifier to indicate pointer nullability (v2)

Author, affiliation: Christopher Bazley, Arm. (WG14 member in individual capacity – GPU expert.)

Date: 2024-12-16

Proposal category: New features

Target audience: General Developers, Compiler/Tooling Developers

Abstract: This paper proposes a new type qualifier for the purpose of adding pointer nullability information to C programs. Its goal is to provide value not only for static analysis and documentation, but also for translators which report errors based only on existing type-compatibility rules. The syntax and semantics are designed to be as familiar (to C programmers) and ergonomic as possible. In contrast, existing solutions are incompatible, confusing, error-prone, and intrusive.

Prior art: MyPy, C, Clang, GCC

_Optional: a type qualifier to indicate pointer nullability (v2)

Reply-to: Christopher Bazley (chris.bazley@arm.com)
Document No: N3422
Date: 2024-12-16

Summary of Changes

N3089

- Initial proposal

N3422

- Added 'What is a null pointer?'
- Added examples of spooky action at a distance and where annotations fail to diagnose errors that are detected by type compatibility rules.
- Added 'Why _Optional rather than _Mandatory?'
- Added 'But any pointer can be null'.
- Added 'Treatment of null pointer constants'.
- Rewrote the section on 'Function pointers' to include the possibility of using `typeof` and explain the potential benefits of using `typedef`.
- Expanded 'Considerations for static analysis' section with effects of conversions.
- Tried to clarify the distinction between an optional pointer and a pointer-to-optional.
- Added proposed wording changes to the standard.
- Edited to take account of the changes in N3342.
- Mentioned C23's standard attribute syntax.
- Tidied up web links to put them in line and remove indirection via Medium.

Philosophical underpinning

The single most important (and redeeming) feature of C is its simplicity. It should be (relatively) quick to learn every aspect of the language, (relatively) easy to create a translator for it, and the language's semantics should follow (more-or-less) directly from its syntax.

People criticise C's syntax, but I consider it the foundation of the language. Any experienced C programmer has already acquired the mindset necessary to read and write code using it. Aside from the need to minimize incompatibilities, the syntactic aberrations introduced by C++ can be ignored. There are significant differences between the design philosophies of C and C++; most relevantly, the designer of C++ did not approve of C's mnemonic syntax for declarations.

"Pythonic" is sometimes used as an adjective to praise code for its use of Python-specific language idioms. I believe that an equivalent word "scenic" could be used to describe C language idioms, meaning that they conform to a mode of expression characteristic of C. I've tried to keep that in mind when evaluating syntax and semantics.

Inspiration from Python

For the past twenty years, I've mostly been coding in C. I had always considered C to be a strongly typed language: it allows implicit conversions between `void *` pointers and other pointer types, and between `enum` and integer types, but those aren't serious shortcomings so long as the programmer is aware of them.

Recently, I switched to a team that writes code in a mixture of languages (including C++, Python, and JavaScript). Writing code in languages that are dynamically typed but with statically checked type annotations was a revelation to me. Our project uses MyPy [0] and Typescript [1] for static type checking.

The main thing that I grew to appreciate was the strong distinction that MyPy makes between values that can be `None` and values that cannot. Such values are annotated as `Optional[int]`, for example. Any attempt to pass an `Optional` value to a function that isn't annotated to accept `None` is faulted, as is any attempt to do unguarded operations on `Optional` values (i.e., without first checking for the value being `None`).

Problem statement

In contrast to Python, C's type system makes no distinction between pointer values that can be null, and those that cannot. Effectively, any pointer in a C program can be null, which leads to repetitive, longwinded, and unverifiable parameter descriptions such as "Non-null pointer to…" or "Address of X … (must not be null)".

Such invariants are not usually documented within a function except by assertions, which clutter the source code and are ineffective without testing. Some programmers even write tests to verify that assertions fail when null is passed to a function, although the same stimulus would provoke undefined behaviour in release builds. The amount of time and effort that could be saved if such misuse were instead caught during translation is huge.

What is a null pointer?

A null pointer is defined by 6.3.3.3 p3 of the ISO C standard:

> *If a null pointer constant or a value of the type nullptr_t (which is necessarily the value nullptr) is converted to a pointer type, the resulting pointer, called **a null pointer, is guaranteed to compare unequal to a pointer to any object or function**. Conversion of a null pointer to another pointer type yields **a null pointer of that type**. **Any two null pointers shall compare equal**.*

Like any other pointer, a null pointer has a referenced type which may be qualified. Although all null pointers compare equal, it is a constraint violation to compare pointers of incompatible types (as per 6.5.10). Translators can reason about the referenced type of a pointer that points to no object or function, e.g. to diagnose a constraint violation upon assignment to a `const`-qualified lvalue.

Pointers can be dereferenced using the indirection operator. The only constraint is that the operand shall have pointer type, therefore no diagnostic message is required when the operand is a null pointer. (This is generally impossible to determine at translation time anyway.)

The semantics of indirection are defined by 6.5.4.3 p4 of the ISO C standard:

> *If the operand points to a function, the result is a function designator; if it points to an object, the result is an lvalue designating the object. If the operand has type "pointer to type", the result has type "type". If an invalid value has been assigned to the pointer, the behavior of the unary * operator is undefined. (**Among the invalid values for dereferencing a pointer by the unary * operator are a null pointer**…)*

An lvalue is defined by 6.3.3.1 p1 of the ISO C standard:

> *An lvalue is an expression with an array type or a complete object type that potentially designates an object; **if an lvalue does not designate an object when it is evaluated, the behavior is undefined**.*

Many programs contain lvalues that do not always designate an object when evaluated, including as a result of applying the indirection operator to a null pointer. C lacks mechanisms to help programmers avoid creating such lvalues.

Pointer arithmetic is limited by 6.5.7 p9 and p10 of the ISO C standard:

> *When an expression that has integer type is added to or subtracted from a pointer, the result has the type of the pointer operand. **If the pointer operand points to an element of an array object**, and the array is large enough, the result points to an element offset from the original element such that the difference of the subscripts of the resulting and original array elements equals the integer expression*
>
> *When two pointers are subtracted, **both shall point to elements of the same array object**, or one past the last element of the array object; the result is the difference of the subscripts of the two array elements*

Since a null pointer does not point to any object, arithmetic on null pointers has undefined behaviour. C lacks mechanisms to help programmers avoid performing arithmetic on null pointers.

Isn't this a solved problem?

Given that the issue of undefined behaviour caused by null pointers has been present in C since its inception, many solutions have already been attempted.

Static in parameter declarations

C99 extended the syntax for function parameter declarations to allow the `static` keyword to appear within `[]`, which requires the passed-in array to be at least a specified size:

```
#include <stddef.h>
#include <stdlib.h>

void *my_memcpy(char dest[static 1], const char src[static 1],
size_t len);

void test(void)
{
    my_memcpy(NULL, NULL, 10); // warning: argument 1 to
'char[static 1]' is null where non-null expected
}

void test2(void)
{
    char *dest = malloc(10), *src = NULL;
    my_memcpy(dest, src, 10);
    free(dest);
}
```

This syntax is not a general-purpose solution because it can only be used for function parameters. Even in parameter declarations, it has limited applicability because arrays of type `void` are illegal. This makes it unusable for declaring functions such as `memcpy`.

Historically, support for checking parameters specified using `[static 1]` was very limited. GCC 11.1.0 only produces a diagnostic message when a null pointer constant is specified directly as a function argument [2]:

```
<source>: In function 'test':
<source>:8:5: warning: argument 1 to 'char[static 1]' is null where
non-null expected [-Wnonnull]
    8 |     my_memcpy(NULL, NULL, 10); // warning: argument 1 to
'char[static 1]' is null where non-null expected
      |     ^~~~~~~~~~~~~~~~~~~~~~~~~
<source>:8:5: warning: argument 2 to 'char[static 1]' is null where
non-null expected [-Wnonnull]
<source>:4:7: note: in a call to function 'my_memcpy'
    4 | void *my_memcpy(char dest[static 1], const char src[static
1], size_t len);
      |       ^~~~~~~~~
```

Later versions of the GNU compiler introduced a new static analysis pass enabled by the command line argument `-fanalyzer` [3]. This allows GCC 14.1.0 to diagnose use of null pointer values that have been stored in intermediate variables, including potential null pointer values originating from `malloc` [4]:

```
<source>: In function 'test2':
<source>:9:5: warning: use of NULL 'src' where non-null expected
[CWE-476] [-Wanalyzer-null-argument]
    9 |     my_memcpy(dest, src, 10);
      |     ^~~~~~~~~~~~~~~~~~~~~~~~
  'test2': events 1-3
    |
    |    8 |     char *dest = malloc(10), *src = NULL;
    |      |                               ^~~
    |      |                               |
    |      |                               (1) 'src' is NULL
    |      |                               (2) 'src' is NULL
    |    9 |     my_memcpy(dest, src, 10);
    |      |     ~~~~~~~~~~~~~~~~~~~~~~~~
    |      |     |
    |      |     (3) argument 2 ('src') NULL where non-null expected
    |
<source>:4:7: note: argument 2 of 'my_memcpy' must be non-null
    4 | void *my_memcpy(char dest[static 1], const char src[static
1], size_t len);
      |       ^~~~~~~~~
<source>:9:5: warning: use of possibly-NULL 'dest' where non-null
expected [CWE-690] [-Wanalyzer-possible-null-argument]
    9 |     my_memcpy(dest, src, 10);
      |     ^~~~~~~~~~~~~~~~~~~~~~~~
  'test2': events 1-2
    |
    |    8 |     char *dest = malloc(10), *src = NULL;
    |      |                  ^~~~~~~~~~
    |      |                  |
    |      |                  (1) this call could return NULL
    |    9 |     my_memcpy(dest, src, 10);
    |      |     ~~~~~~~~~~~~~~~~~~~~~~~~
    |      |     |
    |      |     (2) argument 1 ('dest') from (1) could be NULL
where non-null expected
    |
<source>:4:7: note: argument 1 of 'my_memcpy' must be non-null
    4 | void *my_memcpy(char dest[static 1], const char src[static
1], size_t len);
      |       ^~~~~~~~~
```

GNU attribute

A GNU compiler extension [5] (also supported by Clang and the ARM compiler [6]) allows function arguments to be marked as not supposed to be null:

```
void *my_memcpy(void *dest, const void *src, size_t len)
__attribute__((nonnull (1, 2)));
void test(void)
{
    char *dest = NULL, *src = NULL;
    my_memcpy(NULL, NULL, 10); // warning: argument 1 null where
non-null expected
    my_memcpy(dest, src, 10); // no diagnostic message
}
```

Since GCC began to support C23's attribute specifier syntax, an alternative spelling of the same attribute is available. For example, `[[gnu::nonnull(1, 2)]]` could be substituted for `__attribute__((nonnull (1, 2)))`.

Regardless of how it is spelt, I find the `__attribute__` syntax intrusive and verbose. It is also error-prone because attributes only apply to function declarations as a whole: it's easy to accidentally specify wrong argument indices, because the arguments themselves are not annotated.

Clang extended the syntax to allow `__attribute__((nonnull))` to be used within an argument list, but the GNU compiler does not support that.

Clang annotations

RFC: Nullability qualifiers (2015) [7] proposed not one but **three** new type annotations: `_Nullable`, `_Nonnull` and `_Null_unspecified`. Support for these was added in version 3.7 of Clang [8], but GCC doesn't recognize them. Like GCC without `-fanalyzer`, Clang itself only detects cases where a null pointer constant is specified directly as a function argument:

```
void *my_memcpy(void *_Nonnull dest, const void *_Nonnull src,
size_t len);
void test(void)
{
    char *dest = NULL, *src = NULL;
    my_memcpy(NULL, NULL, 10); // warning: Null passed to a callee
that requires a non-null 1st parameter
    my_memcpy(dest, src, 10); // no diagnostic message
}
```

However, Clang-tidy [9], a standalone tool based on Clang, **can** issue diagnostic messages about misuse of pointers that it is able to infer based on annotations and path-sensitive analysis:

```
<source>:9:3: warning: Null pointer passed to 1st parameter
expecting 'nonnull' [clang-analyzer-core.NonNullParamChecker]
  my_memcpy(dest, src, 10); // no diagnostic message
  ^         ~~~~
<source>:7:9: note: 'dest' initialized to a null pointer value
  char *dest = NULL, *src = NULL;
        ^~~~
<source>:9:3: note: Null pointer passed to 1st parameter expecting
'nonnull'
  my_memcpy(dest, src, 10); // no diagnostic message
  ^         ~~~~
```

Clang's syntax is less verbose and error-prone than `__attribute__`, but the requirement to annotate all pointers as either `_Nullable` or `_Nonnull` makes code harder to read and write. Most pointers should not be null: consider the instance pointer passed to every method of a class. It's no longer safe to write such declarations in traditional style with economy of effort. I also think the semantics of these annotations (discussed later) are far more complex than befits a simple language like C, and likely to cause confusion.

Because Clang's annotations are not type qualifiers, neither the compiler nor the static analyser diagnose some programmer errors that would be detected by existing type compatibility rules [10]:

```
void foo(int *_Nullable x)
{
    void baz(int *_Nonnull *l);
    int *_Nonnull *k;
    k = &x; // no constraint violation or diagnostic
    baz(&x); // no constraint violation or diagnostic
}
```

Another confounding aspects of Clang's nullability attributes is that they appear to exert a kind of spooky action at a distance over pointers that either lack any explicit nullability attribute or from which it **appears to have been explicitly cast away**:

```
void bar(int *_Nonnull y);
void foo(int *_Nullable i)
{
    int *j;
    j = (int *)i;
    bar(j); /* warning: Nullable pointer is passed to a callee
               that requires a non-null 1st parameter */
}
```

Clang's static analyser generates a diagnostic for the above snippet [11] because it treats the cast `(int *)` as equivalent to `(int *_Null_unspecified)`, which is useless in this scenario. The "assumed null unless disproven" quality of the pointer `i` is preserved upon assigning it to `j` (with or without a cast), which appears to contradict the declared type of `j`.

A solution [12] is to modify the cast to be more explicit:

```
void bar(int *_Nonnull y);
void foo(int *_Nullable i)
{
    int *j;
    j = (int *_Nonnull)i;
    bar(j);
}
```

Conclusion

I’ve seen Clang’s nullability qualifiers described as “enormous and useless noise, while providing doubtful value” [13] and the very idea of annotating pointers called a “naive dream”. I agree with the first statement, but not the second: other languages have shown that null safety is achievable and useful, whilst C lags with competing partial solutions that are confusing, error-prone, and intrusive.

Attributes and annotations must be hidden behind macros for compatibility between tools. They provide no value (other than documentation) to developers using translators which do not support path-sensitive analysis. Not all developers use special build machines costing thousands of pounds: I do a lot of coding on a Raspberry Pi, using a toolchain that dates from the 1980s [14] but is still actively maintained [15] (and recently gained support for C17). For me, having a rapid *edit-compile-run* cycle is paramount.

Even tiny translators such as cc65 [16] can check that the addresses of objects declared with `const` or `volatile` are not passed to functions that do not accept such pointers, because the rules for type compatibility are simple (for the benefit of machines and people). This is exactly the niche that the C language should be occupying.

I postulate that improved null safety does not require path-sensitive analysis.

Syntactic and semantic precedents

Type qualifiers (as we understand them today) didn't exist in pre-ANSI C, which consequently had a stronger similarity between declarations and expressions, since qualifiers can't appear in expressions (except as part of a cast).

The second edition of 'The C Programming Language' (K&R, 1988) says only that:

> *Types may also be qualified, to indicate special properties of the objects being declared.*

Notably, the special properties conferred by `const`, `volatile`, `restrict` and `_Atomic` all relate to how objects are **stored** or how that storage is **accessed** - not the range of values representable by an object of the qualified type.

Is the property of being able to represent a null pointer value the kind of property that **should** be indicated by a *type-qualifier*? Restrictions on the range of values representable by an object are usually implied by its *type-specifiers* (although `long`, `short`, `signed`, and `unsigned` are intriguingly also called "qualifiers" by K&R, presumably because their text predates ANSI C).

Pointers are a special type of object though. Multiple levels of indirection can be nested within a single declaration, as in the following declaration of `baz` (an array of pointers to arrays of pointers to `int`):

```
int bar;
int *foo[2] = {NULL, &bar};
int *(*baz[3])[2] = {&foo, NULL, NULL};
```

It's therefore necessary to specify whether null is permitted for **every** level of indirection within a *declarator* (e.g. for both `baz[3]` and `(*baz[3])[2]`). The only existing element of C's existing syntax that has such flexibility is a *type-qualifier*.

It's not meaningful to specify whether null is permitted as part of the *declaration-specifiers* (e.g. `static int`) on the lefthand side of a *declaration*, because this property only applies to pointers. The `restrict` qualifier already has this limitation.

Here's an example of how the above declaration might look with Clang's nullability qualifiers:

```
int bar;
int *_Nullable foo[2] = {NULL, &bar};
int *_Nullable (*_Nullable baz[3])[2] = {&foo, NULL, NULL};
```

Syntactically, this may look like a perfect solution; semantically, this paper will argue that it is not!

A variable of type `char *const` (`const` pointer to `char`) can be assigned to a variable of type `char *` (pointer to `char`), but a variable of type `const char *` (pointer to `const char`) cannot. After a learner internalizes the knowledge that qualifiers on a pointer **target** must be compatible, whereas qualifiers on a pointer **value** are discarded, this rule can be applied to any assignment or initialization:

```
int *const x = NULL;
int *s = x; // no warning
int *volatile y = NULL;
int *t = y; // no warning
int *restrict z = NULL;
int *r = z; // no warning
```

One might not expect the same laxity to apply to the `_Nullable` and `_Nonnull` qualifiers, because they relate to the assigned value, not the storage access properties of a particular copy of it. Despite that, Clang-tidy allows an assigned value to be `_Nullable` unless the type of the assigned-to-object is qualified as `_Nonnull`:

```
extern int *_Nullable getptr(void);
int *_Nullable z = getptr();
int *q = z; // no warning
int *_Nonnull p = z; // warning: Nullable pointer is assigned to a
pointer which is expected to have non-null value
*q = 10; // warning: Nullable pointer is dereferenced
```

This compromise between the traditional semantics of assignment (discard top-level qualifiers) and the semantics needed to track nullability (ensure compatible qualifiers) looks like a weak basis for null safety; however, it is mitigated by the fact that the static analyser tracks whether a pointer value may be null **regardless of its type**. In turn, that makes it **impossible to tell what constraints apply to a pointer value by referring to its declaration**.

A related issue is that top-level qualifiers on arguments are redundant in a function declaration (as opposed to definition) because arguments are passed by value. Callers don't care what the callee does with its copy of a pointer argument - only what it does with the pointed-to object.

Consequently, such qualifiers are ignored when determining compatibility between declarations and definitions of the same function. The normative part of an argument declaration is to the left of the asterisk:

```
void myfunc(const char *const s);
//          ^^^^^^^^^^  ^^^^^
//          Normative   Not normative
//          vvvvvvvvvv  vvvvvvvv
void myfunc(const char *restrict s)
{
}
```

Notably, this rule also applies to `restrict`-qualified arguments, despite an apparent conflict with a principle stated in WG14's charter:

> *Application Programming Interfaces (APIs) should be self-documenting when possible*

Interfaces cannot be self-documenting when qualifiers (such as `restrict`) that are part of the contract may differ between the function definition and the function declaration used by callers.

The same laxity should **not** apply to the `_Nullable` and `_Nonnull` qualifiers, because they relate to the passed value, not its storage access properties. Despite that, Clang ignores any differences between rival declarations of a function, except in cases where contradictory qualifiers were used.

It is permissible to write `[]` instead of `*` in a parameter declaration, to hint that an array is passed (by reference) to a function. One might expect this `[]` syntax to be incompatible with qualifying the type of the pointer (as opposed to the type of array elements). On the contrary, Clang allows nullability qualifiers to appear between the brackets:

```
void myfunc(const char s[_Nullable]); // s "may store a null value
                                      // at runtime"
```

This syntax is not intuitive to me (usually `[]` indicates an index or size) but it does follow 6.7.5.3 of the C language standard:

> *A declaration of a parameter as ''array of type'' shall be adjusted to ''qualified pointer to type'', where the type qualifiers (if any) are those specified within the [and] of the array type derivation.*

Proposed syntax

An essential feature of a new type qualifier expressing 'may be null'[1] is that this property **must not** be lost when a qualified pointer is copied (including when it is passed as a function argument).

Qualifiers on a pointed-to type must be compatible in assignments, initializations, and function calls, whereas qualifiers on a pointer type need not be. The fact that every programmer has internalized this rule makes me reluctant to propose (or embrace) any change to it for nullability qualifiers on a pointer type.

I'm tempted to say that both `restrict` and the Clang annotations `_Nullable` and `_Nonnull` are in the wrong place. The `restrict` qualifier frees an optimizer to generate more efficient code, almost like the opposite of `volatile`. Isn't the quality of being aliased a property of an object, rather than any single pointer to it?

At the heart of C's syntax is the primacy of fundamental types such as `int`. Every declaration is a description of how a chain of indirections leads to such a type.

Let's reframe the 'may be null' property as a quality of the pointed-to object, rather than the pointer:

```
const int *i; // *i is an int that may be stored in read-only memory
volatile int *j; // *j is an int that may be stored in shared memory
_Optional int *k; // *k is an int that may not be stored at all
```

I chose the name `_Optional` to bootstrap existing knowledge of Python and make a clear distinction between this qualifier and `_Nullable`. I also like the idea of Python giving something back to C.

`_Optional` is the same length as `_Nullable` and only one character longer than `volatile`. C's syntax isn't known for its brevity, anyway. (Think not of functions such as `strcpy`, but of declarations such as `const volatile unsigned long int`.)

Read-only objects are often stored in a separate address range so that illegal write accesses generate a segmentation fault (on machines with an MMU). Likewise, null pointer values encode a reserved address, which is typically neither readable nor writable by user programs. In both cases (`const` and `_Optional`), a qualifier on the referenced type of a pointer may reflect something about the address of an object of that type.

Modifying an object designated by an lvalue that has `const`-qualified type only has undefined behaviour if the object was actually defined as `const`; otherwise, it is merely a constraint violation. Likewise, an lvalue that does not designate an object because it is the result of dereferencing a null pointer only has undefined behaviour if evaluated.

Could potential null pointer dereferences also be moved to the domain of constraint violations?

Unlike assignment to an lvalue with `const`-qualified type, no diagnostic should be produced for accesses to an object designed by an lvalue with `_Optional`-qualified type. Were that my intent, I would have proposed a name like `_None` rather than `_Optional`, with constraints against use of lvalues that have `_None`-qualified types as operands.

[1] This wording is sufficiently controversial that a separate section has been devoted to it.

Requiring the `_Optional` qualifier to be cast away before accessing a so-qualified object would be tiresome and would sacrifice type safety for null safety. I do not think that is a good trade-off.

Despite this limitation, the new qualifier is useful:

- It allows interfaces to be self-documenting. (Function declarations must match their definition.)
- It allows the translator to report errors on initialization or assignment, if implicitly converting a pointer to `_Optional` into a pointer to an unqualified type.
- It provides information to static analysis tools, which can warn about dereferences of a pointer to `_Optional` if path-sensitive analysis does not reveal a guarding check for null in the preceding code.

Example usage

```
void foo(int *);

void bar(_Optional int *i)
{
    *i = 10; // path-sensitive warning of unguarded dereference

    if (i) {
        *i = 5; // no path-sensitive warning
    }

    int *j = i; // warning: initializing discard qualifiers
    j = i; // warning: assignment discards qualifiers
    foo(i); // warning: passing parameter discards qualifiers
}
```

Here's an example of complex declarations that I used earlier, updated to use the proposed qualifier:

```
   int              bar;

   _Optional int    *foo[2] = {NULL, &bar};
// ^^decl-spec^^    ^^decl^

   _Optional int    *(*qux[3])[2] = {&foo, &foo, &foo};
// ^^decl-spec^^    ^^declarator^

   _Optional int    *_Optional (*baz[3])[2] = {&foo, NULL, NULL};
//                              ^^decl^
// ^^decl-spec^^    ^^pointer^ ^^dir-decl^^
//                  ^^^^^^declarator^^^^^^^
```

Let's break it down:

- Storage is allocated for an object, `bar`, of type `int`. This will be used as the target of a pointer to `_Optional int` but doesn't need to be qualified as such (any more than a `const` array **must** be passed to `strlen`).
- Storage is allocated for an array, `foo`, of two pointers to `_Optional int`. `_Optional` in the *declaration-specifiers* indicates that elements of `foo` may be null; an expression resembling the *declarator* (e.g. `*foo[0]`) may have undefined behaviour.
- Storage is allocated for an array, `qux`, of three pointers to arrays of pointers to `_Optional int`. `_Optional` in the *declaration-specifiers* indicates that elements of the pointed-to arrays may be null; an expression resembling the *declarator* (e.g. `*(*qux[0])[0]`) may have undefined behaviour.
- Storage is allocated for an array, `baz`, of three pointers to `_Optional` arrays of pointers to `_Optional int`. `_Optional` in the *pointer(opt)* of the top-level *declarator* indicates that elements of `baz` may be null; an expression resembling the inner *declarator* (e.g. `*baz[0]`) may have undefined behaviour. `_Optional` in the *declaration-specifiers* has the same meaning as for `qux`.

Qualification of types other than referenced types

Parameter declarations using `[]` syntax can be written more naturally using an `_Optional` qualifier than using Clang's `_Nullable` qualifier:

```
void myfunc(_Optional const char s[]);
```

With the above exception, it isn't useful to qualify a type other than a referenced type as `_Optional` (although so-qualified types exist during expression evaluation). Such declarations are therefore disallowed, like similar abuse of `restrict`, to avoid confusion.

Consequently, there is no need to create another class of undefined behaviour when an attempt is made to access an object defined with an optional-qualified type through use of an lvalue with non-optional-qualified type. (The equivalent semantics for existing qualifiers are given by 6.7.4.1p7.)

Optional pointer versus pointer-to-optional

The following code declares `p`, a pointer to an **optional pointer**, and `q`, a pointer to a **const pointer**:

```
int *_Optional *p, *const *q;
```

(It is not syntactically valid to declare an optional pointer directly.)

Note that an 'optional pointer' is **not** a pointer whose value may be null; it's a pointer obtained by **dereferencing** a pointer that may be null[2]. If `*p` is used as an lvalue and it does not designate an object because `p` is null, then the object (of type `int *_Optional`) designated by `*p` does not exist.

This is like the existing rule that a 'const pointer' is not a pointer that may address an immutable object; it's a pointer that is not itself modifiable. If `*q` is used as an lvalue then the object (of type `int *const`) designated by `*q` cannot be modified.

Attempting to police this distinction is probably fruitless, since 'const pointer' and 'pointer to const' are already used interchangeably, causing considerable confusion.

A secondary reason that it does not make sense to describe `_Optional int *p` as an 'optional pointer' is that **a null pointer is a pointer** – it's just a pointer that does not point to any object or function. The pointer is not optional; the referenced object is.

[2] By 'may be null', please understand the meaning to be that generation of relevant diagnostics is enabled – not that only pointers to so-qualified referenced types can be null.

Why _Optional rather than _Mandatory?

As mentioned earlier, function parameters that can legitimately be null are outnumbered by parameters that cannot be null without provoking undefined behaviour.

The very first sentence of K&R's book "The C programming language" is

> *C is a general-purpose programming language which features economy of expression…*

Neither static array extents nor any alternative method of annotating function parameters as non-null (including a future `_Mandatory` qualifier) resemble "economy of expression".

The goal of keeping C "pleasant, expressive, and versatile" (as K&R described it) is incompatible with declaring interfaces like this:

```
bool coord_stack_init(coord_stack stack[static 1], size_t limit);
void coord_stack_term(coord_stack stack[static 1]);
bool coord_stack_push(coord_stack stack[static 1], coord item);
coord coord_stack_pop(coord_stack stack[static 1]);
bool coord_stack_is_empty(coord_stack stack[static 1]);
```

Or this:

```
bool coord_stack_init(coord_stack *_Nonnull stack, size_t limit);
void coord_stack_term(coord_stack *_Nonnull stack);
bool coord_stack_push(coord_stack *_Nonnull stack, coord item);
coord coord_stack_pop(coord_stack *_Nonnull stack);
bool coord_stack_is_empty(coord_stack *_Nonnull stack);
```

Or this:

```
bool coord_stack_init(coord_stack *stack, size_t limit)
                      __attribute__((nonnull (1, 1)));

void coord_stack_term(coord_stack *stack)
                      __attribute__((nonnull (1, 1)));

bool coord_stack_push(coord_stack *stack, coord item)
                      __attribute__((nonnull (1, 1)));

coord coord_stack_pop(coord_stack *stack)
                      __attribute__((nonnull (1, 1)));

bool coord_stack_is_empty(coord_stack *stack)
                          __attribute__((nonnull (1, 1)));
```

Instead of like this:

```
bool coord_stack_init(coord_stack *stack, size_t limit);
void coord_stack_term(coord_stack *stack);
bool coord_stack_push(coord_stack *stack, coord item);
coord coord_stack_pop(coord_stack *stack);
bool coord_stack_is_empty(coord_stack *stack);
```

An equally important reason to add _Optional instead of _Mandatory is that the assignment semantics for _Mandatory would need to be opposite to those for const and volatile (because _Mandatory is not a restriction on usage of a so-qualified pointer, therefore it should not be contagious).

Implicitly discarding a const qualifier provokes a diagnostic message:

```
const int *x = &y;
int *z = x; // warning: initialization discards 'const' qualifier
            // from pointer target type
const int *q = z; // no warning
```

As does implicitly discarding a volatile qualifier:

```
volatile int *x = &y;
int *z = x; // warning: initialization discards 'volatile' qualifier
            // from pointer target type
volatile int *q = z; // no warning
```

Implicitly discarding a _Mandatory qualifier would not provoke a diagnostic message; instead, a new type of diagnostic message might be generated upon implicitly acquiring the qualifier:

```
_Mandatory int *x = &y;
int *z = x; // no warning
_Mandatory int *q = z; // warning : initialization adds '_Mandatory'
                       // qualifier to pointer target type
```

Choosing defaults wisely so that they do not need to be constantly overridden is an important aspect of usability. Limiting the number of distinct rules and patterns makes regular languages easier to learn than irregular ones.

The implementation cost of adding a _Mandatory qualifier might not differ much from the cost of adding an _Optional qualifier, but I believe that its effect on those learning and using the language would be detrimental to the extent that it might not be used at all. The limited uptake of static array size expressions, despite their potential for annotating function parameters as non-null, seems to point in this direction.

A new qualifier indicating that a pointer may be null can only have been explicitly added to an existing program by its maintainer. Consequently, implementations can immediately begin to generate diagnostic messages about misuse of pointers to so-qualified objects. In contrast, implementations cannot reasonably begin to generate diagnostic messages about usage of pointers to unqualified objects when they introduce support for a qualifier indicating that a pointer cannot be null.

But any pointer can be null

A common objection to the idea of a qualifier expressing the property "this pointer can be null" is that any pointer can be null, therefore a creating new category of pointers that can be null would be meaningless.

This is based on a misconception about the meaning of the proposed `_Optional` qualifier: it does not mean "a pointer to a so-qualified type is **permitted** to be null"; it means "a pointer to a so-qualified type is **assumed to be null in the absence of other information**, for the purpose of generating diagnostics."

Calling a function, `foo`, with a null pointer argument in the following example illustrates that `foo` may receive a null pointer regardless of its parameter type:

```
void foo(char *i)
{
    if (!i) {
        puts("i is not a pointer to any object or function");
    }
    char k;
    _Optional char *j = i;
    k = *j; // diagnostic message because j is assumed to be null
    if (j) { // j could now be a pointer to an object or function
        k = *j; // no diagnostic message because j is constrained
    }
}

// Another translation unit:
int main(void)
{
    foo(NULL); // i becomes null although it is unqualified
    return 0;
}
```

If the above program is executed, then "i is not a pointer to any object or function" is sent to the standard output stream before `*j` is evaluated (with undefined behaviour).

It is not necessarily redundant for programs to check for the value of a pointer to an unqualified referenced type (such as `i`) being null, and compilers **must not** make optimisations based on false inferences that pointers to unqualified referenced types cannot be null.

Whether to employ defensive programming techniques depends on the type of program or library being written, the security environment it operates within, and the trustworthiness of calling code (which may even be written in another language). It is also possible to create null pointers using `memset`. No language solution can fully guarantee that a program is free of null pointers whose nullability is not expressed though the type system.

A similar situation exists with the `const` qualifier: some programmers use it sparingly, and others not at all. Modifying an object through an lvalue of unqualified type is often an encapsulation violation whether it has undefined behaviour or not. There is no way of guaranteeing that a program is free of modifiable lvalues that designate immutable objects.

Modifiable lvalues are defined by 6.3.3.1 p1 of the ISO C standard:

> *A modifiable lvalue is an lvalue that does not have array type, does not have an incomplete type, does not have a const-qualified type, and if it is a structure or union, does not have any member (including, recursively, any member or element of all contained aggregates or unions) with a const-qualified type.*

The quoted text does not guarantee that an lvalue with `const`-qualified type designates an immutable object; nor does it prevent an immutable object from being designated by a modifiable lvalue that does not have a `const`-qualified type.

Analogous to the given meaning of the proposed `_Optional` qualifier, one could say that `const` does not mean "an object designated by an lvalue of this type is **permitted** to be immutable"; it means "an object designated by an lvalue of this type is **assumed** to be immutable".

Consider the declaration of `strstr` in 7.21.5.7 of the C99 standard:

> *char *strstr(const char *s1, const char *s2);*

Dereferencing the returned pointer produces a modifiable lvalue even if the `s1` argument does not designate a mutable object!

Modifying an object defined with a `const`-qualified type through use of an lvalue with non-`const`-qualified type has undefined behaviour, but it is not a constraint violation:

```
const char s1[] = "carwash"; // immutable object
*strstr(s1, "wash") = 'm'; // undefined behaviour
```

A static array generated from a string literal can be stored in shared or read-only memory as per 6.4.6 p7 of the ISO C standard:

> *It is unspecified whether these arrays are distinct provided their elements have the appropriate values. If the program attempts to modify such an array, the behavior is undefined.*

The type of such arrays is `char`, which means that that string literals are another potential source of modifiable lvalues that designate immutable objects.

Assigning to a static array generated from a string literal has undefined behaviour, but it is not a constraint violation:

```
char *s = "carwash"; // pointer to an immutable object
*s = 'm'; // undefined behaviour
```

Nevertheless, `const` qualifiers have considerable practical value. Nobody would argue that since assigning to an lvalue has undefined behaviour if the designated object is immutable, creating a subcategory of modifiable lvalues is pointless.

Calling a function, `bar`, with a string literal argument in the following example [17] illustrates that `bar` may receive a reference to an immutable object regardless of its parameter type:

```
char h;
void bar(char *i)
{
    if (i != &h) {
        puts("i is not known to designate a mutable object");
    }
    const char *j = i;
    *j = 4; // constraint violation because *j is assumed to
            // designate an immutable object
    if (j == &h) { // *j must now designate a mutable object
        *j = 5; // still a constraint violation
    }
}

// Another translation unit:
int main(void)
{
    bar("cuppa");
    return 0;
}
```

It is not possible for `bar` to check that `*i` is mutable in the general case, but `bar` could reject references other than those of a set of known mutable objects (such as `h`).

The fact that `*i` is a modifiable lvalue whereas `*j` is not (even when `i == j` and `j == &h`) illustrates a key difference between `_Optional` and `const`: like nullability, immutability can be disproven, but translators do not use such inferences to modify their generation of diagnostics. The standard requires diagnostics for constraint violations, and constraints must be simple enough to be enforced by all implementations.

Treatment of null pointer constants

It would be invaluable for a diagnostic message to be produced whenever a null pointer constant is assigned to a pointer that does not have a `_Optional`-qualified referenced type.

The simplest way for users to implement the desired behaviour would be to define `nullptr` (or `NULL`) as a macro that expands to `((_Optional void *)0)`. The following example illustrates that by invoking GCC 14.2.0 with the command line argument `-Dnullptr='((const void *)0)'` [18]:

```
int main(void)
{
    void baz(int *);
    int *i = nullptr; // discards qualifier from pointer target type
    i = nullptr; // discards qualifier from pointer target type
    baz(nullptr); // discards qualifier from pointer target type
    return 0;
}
```

A better long-term direction for the language would be to follow the path established by GCC for treatment of string literals.

GCC provides a compiler option to change the type of the static array generated from string literals [19]:

> *-Wwrite-strings*
>
> *When compiling C, give string constants the type const char[length] so that copying the address of one into a non-const char * pointer will get a warning; when compiling C++, warn about the deprecated conversion from string literals to char *.*

An analogous compiler option could be provided to change the type of null pointers generated from null pointer constants, and those generated when an object is subject to default initialisation. This would be more powerful than a macro solution.

It is not easy to determine whether a given program can be translated with `-Wwrite-strings` (or a new option such as `-Wnull-pointers`) without generating diagnostics. Portability issues such as this are covered by Annex J of the ISO C standard. Writable string literals are described therein both as an example of undefined behaviour (J.2) and as a common extension (J.5.6).

Conversions from maybe-null to not-null

I presented the idea of diagnostic messages when a pointer-to-`_Optional` is passed to a function with incompatible argument types as an unalloyed good. In fact, such usage has legitimate applications.

Consider the following veneer for the `strcmp` function which safely handles null pointer values by substituting the empty string:

```
int safe_strcmp(_Optional const char *s1, _Optional const char *s2)
{
    if (!s1) s1 = "";
    if (!s2) s2 = "";
    return strcmp(s1, s2); // warning: passing parameter discards
qualifiers
}
```

In the above situation, both `s1` and `s2` would both need to be cast before calling `strcmp`:

```
int safe_strcmp(_Optional const char *s1, _Optional const char *s2)
{
    if (!s1) s1 = "";
    if (!s2) s2 = "";
    return strcmp((const char *)s1, (const char *)s2);
}
```

The above solution would be detrimental to readability and type safety.

It could be argued that **any** mechanism to remove `_Optional` from the target of a pointer without first checking its value (at runtime) fatally compromises null safety. I disagree: C provides tools to write type-safe code, whilst allowing leniency where it is pragmatic to do so.

It might be possible to use some combination of `_Generic` and `unqual_typeof` to remove only a specific qualifier from a type (like `const_cast` in C++) but such casts would still clutter the code and therefore seem likely be rejected by programmers who prefer to rely solely on path-sensitive analysis.

What is required is a solution that accommodates **both** advanced translators and translators which report errors based only on simple type-compatibility rules. Translators capable of doing so must be able to validate conversions from maybe-null to not-null in the same way as they would validate a real pointer dereference.

One of my colleagues suggested just such a solution:

```
int safe_strcmp(_Optional const char *s1, _Optional const char *s2)
{
  if (!s1) s1 = "";
  if (!s2) s2 = "";
  return strcmp(&*s1, &*s2);
}
```

This idiom has the benefit that it is already 'on the radar' of implementers (and some programmers) because of an existing rule that neither operator of `&*` is evaluated. It's searchable, easy to type (`&` and `*` are on adjacent keys), and not too ugly.

Do not underestimate the importance of `&*` being easy to type! I must have written it thousands of times by now. The alternatives that I considered would have made updating a large existing codebase unbearable.

The way I envisage this working is:

- All translators implicitly remove the `_Optional` qualifier from the type of the pointed-to object in the result of the expressions `&*s1` and `&*s2`.
- A translator that does not attempt path-sensitive analysis will not warn about the expressions `&*s1` and `&*s2`, since it cannot tell whether s1 and s2 are null pointers.
- A translator that warns about dereferences of pointers to `_Optional`, in cases where such pointers cannot be proven to be non-null, may warn about the expressions `&*s1` and `&*s2` if the guarding `if` statements are removed.

However, this proposal might entail a modification to the description of the address and indirection operators in 6.5.4.3 p3 of the ISO C standard:

> *If the operand [of the unary & operator] is the result of a unary * operator, neither that operator nor the & operator is evaluated and the result is as if both were omitted, except that the constraints on the operators still apply and the result is not an lvalue.*

C++ does not currently allow indirection on an operand of type `void *`. This rule would either need to be aligned with C, or else C++ programmers would need to cast away the qualifier from `_Optional void *` in some circumstances, rather than using an idiom such as `&*`.

Modification of operator semantics

I haven't yet explained how such an expression such as `&*s` would remove the `_Optional` qualifier from the type of a pointed-to object.

Whereas a qualifier that applies to a pointer type is naturally removed by dereferencing that pointer, a qualifier (such as `_Optional`) that applies to a pointed-to object is not:

```
int *const x;
typeof(&*x) y; // y has type 'int *' not 'int *const'
y = 0;

int b;
int const *a = &b;
typeof(&*a) c; // c has type 'int const *'
*c = 0; // error: read-only variable is not assignable
```

Consequently, modified semantics are required for the unary `*` operator, the unary `&` operator, or both.

It's tempting to think that the appropriate time to remove a maybe-null qualifier from a pointer is the same moment at which undefined behaviour would ensue if the pointer were null. I prototyped a change to remove the `_Optional` qualifier from the result of unary `*` but found it onerous to add `&*` everywhere it was necessary to remove the `_Optional` qualifier from a pointer.

Moreover, many previously simple expressions became unreadable:

- `&(&*s)[index]` (instead of `&s[index]`)
- `&(&*s)->member` (instead of `&s->member`)

Whilst it would have been possible to improve readability by using more intermediate variables, that isn't the frictionless experience I look for in a programming language. (The same consideration applies to reliance on casts in the absence of modified operator semantics.)

The proposed idiom `&*s` is merely the simplest expression that incorporates a semantic dereference without accessing the pointed-to object. A whole class of similar expressions exist, all of which are typically translated to a machine-level instruction to move or add to a register value (rather than a load from memory):

- `&s[0]`
- `&0[s]` (by definition, `E1[E2]` is equivalent to `(*((E1)+(E2)))`)
- `&(*s).member`
- `&s->member`

There is only one way to get the address of an object (excepting arithmetic), whereas there are many ways to dereference a pointer. Therefore, I propose that any `_Optional` qualifier be implicitly removed from the operand of the unary `&` operator, rather than modifying the semantics of the unary `*`, subscript `[]` and member-access `->` operators.

The operand of `&` is already treated specially, being exempt from conversion from an *lvalue* to the value stored in the designated object, and from implicit conversion of an array or function type into a pointer. It therefore seems less surprising to add new semantics for `&` than `*`.

Another class of expressions that generate an address from a pointer without accessing the pointed-to object are arithmetic expressions in which one operand is a pointer:

- `1 + s`
- `s - 1`
- `++s`

None of the above expressions affect the qualifiers of a pointed-to object in the result type: if the type of `s` is a pointer-to-`const` then so is the type of `s + 1`.

Although `s + n` is equivalent to `&s[n]` in current code, it does not occur often enough to justify modifying arithmetic operators to remove any `_Optional` qualifier from a pointed-to object. This also avoids the question of changes to prefix/postfix operators such as ++ and compound assignments such as +=. The alternative substitution of `&*s + n` is tolerably readable.

Function pointers

Traditionally, C's declaration syntax did not permit qualifiers on the referenced type to be specified as part of a function pointer declaration:

```
<source>:4:6: error: expected ')' [clang-diagnostic-error]
int (const *f)(int); // pointer to const-qualified function
     ^
```

A syntactic way around this limitation is to use an intermediate `typedef` name:

```
typedef int func_t(int);
const func_t *f; // pointer to const-qualified function
```

It is easier to document the parameters and return value of a function type when it is named by a separate `typedef` declaration.

The resultant `typedef` name can also be used as a convenient way of declaring functions of the given type without repetition:

```
func_t inches_to_mm, feet_to_mm, metres_to_mm;
```

Since the publication of C23, the `typeof` operator can alternatively be used to specify qualifiers for the referenced type in a function pointer declaration:

```
const typeof(int (int)) *f; // pointer to const-qualified function
```

Some programmers may find a declaration that uses `typeof` easier to read than a traditional-style declaration, especially when the function pointer type is a return type. Others may prefer to adhere to K&R's principle that declarations should resemble usage.

Further extending the declaration syntax is beyond the scope of my proposal.

None of this solves the underlying semantic issue. GCC 14.2.0 does not produce a diagnostic message about qualified function types unless `-pedantic` is passed as a command line argument, but Clang 18.1.0 does [20]:

```
<source>:5:1: warning: 'const' qualifier on function type 'func_t'
(aka 'int (int)') has unspecified behavior [clang-diagnostic-
warning]
const func_t *f; // pointer to const-qualified function
^~~~~~
```

This diagnostic message can be justified by 6.7.4.1 p10 of the ISO C23 standard:

> *If the specification of a function type includes any type qualifiers, the behavior is undefined.*

N3342 [24] (which has been accepted by WG14) modified this text as follows:

> *If the specification of a function type includes any type qualifiers, the behavior is implementation-defined.*

This change moved qualified function types out of a category of behaviour that could be unpredictable and into a category of behaviour for which implementors are expected to document their choices. Such use of `const` and `volatile` is still a possible source of portability issues.

Since `_Optional` is new, its semantics when applied to function types can instead be standardized.

Migration of existing code

Functions which consume pointers that can legitimately be null can usually be changed with no effect on compatibility. For example, `void free(_Optional void *)` can consume a pointer to an `_Optional`-qualified type, or a pointer to an unqualified type, without casting.

Such changes are not seamless when functions are used as callback functions (i.e. called through function pointers), because any extra qualifiers cause a type mismatch. It would be incredibly useful for C to support variance in assignments of function pointers to the same degree that it supports variance for function calls (not only for `_Optional`, but for existing qualifiers). Such a proposal lies outside the scope of this paper.

'Safe' wrappers for existing functions that produce null pointers can also be written, for example `_Optional FILE *safe_fopen(const char *, const char *)` would produce a pointer that can only be passed to functions which accept pointers to `_Optional`-qualified types.

Here is an example of one type of change that I made to an existing codebase:

Before

```
entry_t *old_entries = d->entries;
d->entries = mem_alloc(sizeof(entry_t) * new_size);

if (NULL == d->entries)
{
    d->entries = old_entries;
    return ERROR_OOM;
}
```

After

```
_Optional entry_t *new_entries = mem_alloc(sizeof(entry_t) *
new_size);

if (NULL == new_entries)
{
    return ERROR_OOM;
}

d->entries = &*new_entries;
```

This pattern avoids the need to qualify the array pointed to by `struct` member `entries` as `_Optional`, thereby simplifying all other code which uses it. When nullability is part of the type system, more discipline and less constructive ambiguity is required. General-purpose `struct` types for which pointer nullability depends on specific usage become a liability.

Of course, programmers are free to eschew the new qualifier, just as many do not consider `const` correctness to be worth their time.

Proposed language extension

- A new type qualifier, `_Optional`, indicates that a pointer to a so-qualified type should be assumed to be null (for the purpose of issuing diagnostic messages only) unless proven otherwise. This does not preclude any other pointer type from being null.
- Types other than those of a pointed-to object or pointed-to incomplete type shall not be `_Optional`-qualified in a declaration.
- The semantics of the unary `&` operator are modified so that if its operand has type "*type*" then its result has type "pointer to *type*", with the omission of any `_Optional` qualifier of the pointed-to type.
- If an operand is a pointer to an `_Optional`-qualified type and its value cannot be statically proven never to be null, then implementations may generate a diagnostic message of any undefined behaviour that would occur if the value were null.
- A specification of a function pointer type that has an `_Optional`-qualified referenced type does not have implementation-defined behaviour, unlike other qualifiers.

The `_Optional` qualifier is treated like existing qualifiers when determining compatibility between types, and when determining whether a pointer may be implicitly converted to a pointer to a differently qualified type.

Considerations for static analysis

Dereferences that do not access storage

Clang's static analyser currently ignores many instances of undefined behaviour. For example, it allows expressions like `&self->super` (equivalent to `&(*self).super`) when `self` is null.

This expression is not exempted by the rule that

> *If the operand [of the unary & operator] is the result of a unary * operator, neither that operator nor the & operator is evaluated and the result is as if both were omitted*

because the operand of the address-of operator is `(*self).super`, not `*self`.

This latitude is also required because many common user-defined macros such as `container_of` have undefined behaviour.

`offsetof` is a standard macro, and therefore it can be assumed to be free of undefined behaviour. However, the simplest definition of `offsetof` incorporates an explicit null pointer dereference:

```
#define offsetof(st, m) \
 ((size_t)&(((st *)0)->m))
```

Such expressions must be rejected when applied to pointers to `_Optional` values, otherwise it would not be safe to remove `_Optional` from a pointer target by use of the proposed `&*` idiom (or any equivalent). Effectively, qualifying a type as `_Optional` enables an **enhanced level of checking for undefined behaviour**, which operates partly at a syntactic level rather than solely at the level of simulated memory accesses.

This paper argues that unlocking improved checking for UB is a powerful and desirable side-effect of adding a new type qualifier.

Conversions from qualified to unqualified type

Explicitly casting an expression to remove `_Optional` from a referenced type instead of using the proposed `&*` idiom could be considered bad style because it prevents effective static analysis. However, part of the traditional spirit of C is that programmers should be able to override checks when necessary. This philosophy is sometimes expressed in phrases such as “Don't prevent the programmer from doing what needs to be done."

It is also necessary to consider the case where an expression is **not** explicitly cast to remove `_Optional` from a pointer target before using it as the righthand operand of an assignment. Such code violates one of the constraints of section 6.5.16.1 in the C standard, specifically:

> *the type pointed to by the left operand has all the qualifiers of the type pointed to by the right operand*

Section 5.2.1.3 does not require that translation of a program must terminate when a constraint violation is detected, merely that a diagnostic message must be produced:

> *Of course, an implementation is free to produce any number of diagnostic messages, often referred to as warnings, as long as a valid program is still correctly translated. It can also successfully translate an invalid program.*

It is the default behaviour of many implementations to continue translation when certain constraint violations are detected.

For example:

```
int main(void)
{
    const int *i = &(int){0};
    int *j;
    j = i; // constraint violation: j lacks a qualifier of i
    static_assert(_Generic(j, int *:1, default:0)); // passes
    *j = 2;
    *i = 1; // constraint violation: not a modifiable lvalue
    return 0;
}
```

Clang 19.1.0 and GCC 14.1 produce a diagnostic message for both constraint violations but only terminate translation if the second is present [21]. If the second constraint violation is removed, then the modified program is translated successfully. It is therefore necessary to think about recommended practice for situations where a qualifier is implicitly removed by assignment, as well as when it is explicitly cast away.

As evidenced by the static assertion, the referenced type of `j` does not mysteriously acquire the qualifier of the referenced type of `i` when `j` is assigned the value of `i`. Consequently, the subsequent assignment to `*j` is not a constraint violation and does not produce a diagnostic. The object referenced by `j` was not declared as `const`, therefore the assignment `*j = 2` succeeds.

This paper considers it axiomatic that **constraints expressed through the type system must also be removable through the type system.** One aspect is that such constraints must be removable by cast or assignment; another is that they must not reappear after being discarded.

It follows that implementations should not produce a diagnostic for operators such as unary `*` when the referenced type of their operand is not `_Optional`-qualified, if the **only** reason for doing so is that static analysis can trace the provenance of that operand's value back to a pointer to an `_Optional`-qualified referenced type (on at least one execution path).

For example:

```
void foo(_Optional int *i)
{
    int *j;
    j = i; // violates type constraints for =
    *j = 5; // no diagnostic recommended although
            // *i is optional and j == i
}
```

Nor should implementations produce a diagnostic for an assignment to an lvalue (or parameter) when neither the lvalue (or parameter) nor the assigned expression have a referenced type which is _Optional-qualified, if the **only** reason for doing so is that static analysis can trace the provenance of the expected value of the assigned expression back to an object declared with (or value cast to) an _Optional-qualified referenced type.

For example:

```
void bar(int *y);
void foo(_Optional int *i)
{
    int *j;
    j = i; // violates type constraints for =
    bar(j); // no diagnostic recommended although
            // *i is optional and j == i
}
```

The proposed recommended practice resembles the behaviour of MyPy for equivalent Python code:

```
from typing import Optional

def bar(y:int) -> None:
    pass

def foo(i:Optional[int]) -> None:
    j:int = 0
    j = i # incompatible types in assignment
    bar(j) # no diagnostic although j == i
```

When prototyping _Optional in Clang's static analyser, I chose to mutate the state of tracked values from 'Nullable' to 'Unspecified' upon encountering conversions of references from an _Optional-qualified referenced type to a non-_Optional-qualified referenced type. This is only a change to the semantics of Nullable when used in conjunction with _Optional, in which case the qualifier overrules the attribute.

For example:

```
void foo(_Optional int *_Nullable i)
{
    int *j;
    j = (int *)i; // cast to avoid constraint violation
    *j = 5; // no diagnostic because *i is _Optional
}
```

An alternative solution would have been to store optionality state separately from nullability state.

Conversions from unqualified to qualified type

This paper argues that conversions from a non-`_Optional`-qualified referenced type to an `_Optional`-qualified referenced type should not discard any “proven not null” property of the converted pointer value. That property is a product of path-sensitive analysis; it was not introduced by and is not visible through the type system[3], therefore it need not be removable through the type system.

For example:

```
void foo(char *i)
{
    if (i) { // constrains i to be non-null
        _Optional char *j;
        char k;
        j = i; // j inherits the non-null constraint of i...
        k = *j; // ...therefore no diagnostic message is recommended
    }
}
```

Discarding the “proven not null” property of a pointer value upon conversion would prevent common patterns such as:

```
char *first = "one"; // constrains first to be non-null
list_t list = {};
_Optional char *buf = first; // *buf is assumed to be UB as per type
add_item(&list, &*buf); // unwanted diagnostic message!
if (more) {
    buf = strdup("two"); // *buf is assumed to be UB as per type
    if (buf) // constrains buf to be non-null
        add_item(&list, &*buf); // no diagnostic message
}
```

[3] For example, a controlling expression such as `i != NULL` within an `if` statement does not implicitly remove `_Optional` from the referenced type of `*i` within the secondary block (if `i` has a pointer type).

Possible objections

The need to define a `typedef` name before declaring a pointer to an `_Optional` function, or alternatively use `typeof` to specify the function type, is a drawback of qualifying the referenced type rather than the pointer type. This paper argues that code clarity and documentation is often improved by composing complex declarations from simpler declarations, and that this limitation of the declaration syntax is outweighed by the benefit of regular semantics in actual usage.

Existing standard library functions do not accept a null pointer instead of any function pointer parameters. Nevertheless, it might be worth considering an extension to C's declaration syntax to remove the need to use `typedef` or `typeof` for such declarations, especially if other applications of qualified function types arise.

Some may struggle to accept a novel syntax for adding nullability information to pointers, given the existence of more prosaic solutions. This paper urges them to consider whether a solution inspired by pointer-to-`const` is really such a novelty - especially in comparison to the irregular new semantics required when the pointer type itself is qualified.

Others may agree with Stroustrup [22] that C's syntax and semantics are a "known mess" of "perversities". This paper argues that pointer nullability should be added in a way that conforms to long-established C language idioms rather than violating such norms (as C++ references do) in the hope of satisfying users who will never like C anyway.

Implementations

A working prototype [23] of the required changes to Clang and Clang-tidy exists.

The prototype has been used successfully at Arm to add pointer nullability information to parts of the user-space Mali GPU driver. I found the new qualifier useful for finding issues caused by not handling null values defensively even before having updated Clang-tidy. This is what I had hoped because a new qualifier cannot be justified unless it provides value in the absence of static analysis.

Proposed wording changes

The wording proposed is a diff from the N3054 working draft — September 3, 2022 ISO/IEC 9899:2023 with changes from N3342 [24] incorporated. Green text is new text, while ~~red~~ text is deleted text.

6.2.5 Types

31 Any type so far mentioned is an unqualified type. Each unqualified type has several qualified versions of its type[52], corresponding to the combinations of one, two, three, or all ~~three~~four of the `const`, `volatile`, ~~and~~ `restrict`, and `_Optional` qualifiers. The qualified or unqualified versions of a type are distinct types that belong to the same type category and have the same representation and alignment requirements.[53] An array and its element type are always considered to be identically qualified.[54] Any other derived type is not qualified by the qualifiers (if any) of the type from which it is derived.

6.4.1 Keywords

Syntax

1 keyword: one of

`alignas`	`enum`	`signed`	`while`
`alignof`	`extern`	`sizeof`	`_Atomic`
`auto`	`false`	`static`	`_BitInt`
`bool`	`float`	`static_assert`	`_Complex`
`break`	`for`	`struct`	`_Decimal128`
`case`	`goto`	`switch`	`_Decimal32`
`char`	`if`	`thread_local`	`_Decimal64`
`const`	`inline`	`true`	`_Generic`
`constexpr`	`int`	`typedef`	`_Imaginary`
`continue`	`long`	`typeof`	`_Noreturn`
`default`	`nullptr`	`typeof_unqual`	`_Optional`
`do`	`register`	`union`	
`double`	`restrict`	`unsigned`	
`else`	`return`	`void`	
	`short`	`volatile`	

6.5.3.2 Address and indirection operators

Semantics

3 The unary `&` operator yields the address of its operand. If the operand has type "*type*", the result has type "pointer to *type*", preserving all qualifiers except any `_Optional` qualifier that previously applied to the type category of the operand. If the operand is the result of a unary `*` operator, neither that operator nor the `&` operator is evaluated and the result is as if both were omitted, except that the constraints on the operators still apply, any `_Optional` qualifier is still removed from the referenced type, and the result is not an lvalue. Similarly, if the operand is the result of a `[]` operator, neither the `&` operator nor the unary `*` that is implied by the `[]` is evaluated and the result is as if the `&` operator were removed and the `[]` operator were changed to a `+` operator except that any `_Optional` qualifier is removed from the referenced type. Otherwise, the result is a pointer to the object or function designated by its operand.

6.5.15 Conditional operator

9 Given the declarations

```
const void *c_vp;
void *vp;
const int *c_ip;
volatile int *v_ip;
int *ip;
const char *c_cp;
_Optional int *o_ip;
```

the third column in the following table is the common type that is the result of a conditional expression in which the first two columns are the second and third operands (in either order):

`c_vp`	`c_ip`	`const void *`
`v_ip`	`0`	`volatile int *`
`c_ip`	`v_ip`	`const volatile int *`
`vp`	`c_cp`	`const void *`
`ip`	`c_ip`	`const int *`
`vp`	`ip`	`void *`
`c_ip`	`o_ip`	`const _Optional int *`
`c_vp`	`o_ip`	`const _Optional void *`

6.7.3 Type qualifiers

Syntax

1 *type-qualifier:*

```
const
restrict
volatile
_Atomic
_Optional
```

Constraints

2 Types other than pointer types whose referenced type is an object type and (possibly multi-dimensional) array types with such pointer types as element type shall not be restrict-qualified.

3 The `_Atomic` qualifier shall not be used if the implementation does not support atomic types (see 6.10.9.3).

4 The type modified by the `_Atomic` qualifier shall not be an array type or a function type.

5 Types other than the referenced type of a pointer type shall not be optional-qualified. This rule is applied recursively (see 6.2.5).

Semantics

10 The intended use of the `_Optional` qualifier is to promote analysis that diagnoses potential undefined behavior during translation. The qualifier indicates that a pointer to a so-qualified type is potentially null. Deleting all instances of the qualifier from all preprocessing translation units composing a conforming program does not change its meaning (i.e., observable behavior).

11 NOTE Pointers to non-optional-qualified types can also be null pointers. Implementations are free to detect and diagnose undefined behavior resulting from use of null pointers regardless of their type. XX) The absence of an `Optional` qualifier is not an optimization opportunity.

XX) An attempt to access an object through an lvalue that does not designate an object because evaluating it entails dereferencing a null pointer could be considered analogous to an attempt to modify an object defined with a const-qualified type through use of a modifiable lvalue. Implementations are free to diagnose both kinds of undefined behavior regardless of whether the lvalue used for the access has an optional or const qualified type.

~~10~~12 If the specification of an array type includes any type qualifiers, both the array and the element type are so-qualified. If the specification of a function type includes any type qualifiers other than `_Optional`, the behavior is implementation-defined. 151)

Recommended practice

17 Implementations that do not perform data-flow analysis should not produce diagnostic messages about use of pointers to optional-qualified types except to indicate violation of a syntax rule or constraint.

18 Many operators result in undefined behavior if one or more of their operands is a null pointer. If such an operand has the type of a pointer to an optional-qualified type, then an implementation that performs data-flow analysis may produce a diagnostic message. A diagnostic message is only recommended if the implementation is unable to prove that no path exists which results in the operand being a null pointer.

19 Interprocedural analysis is not required to correctly diagnose use of pointers to optional-qualified referenced types as potentially invalid operands. Context-insensitive analysis is recommended for this purpose, because it promotes stability in the set of diagnostic messages produced for each function definition regardless of any changes to its callers.

20 The value of a pointer whose referenced type is optional-qualified may be assumed to be potentially null until its set of potential values has been constrained by a conditional branch or assignment of a value (other than null) on the path being followed during data-flow analysis.

EXAMPLE 4 The snippet below illustrates valid and invalid use of the `_Optional` type qualifier in declarations:

```
_Optional int *poi; // valid
typeof(&*poi) pi; // pi has type "int *" because of &
static_assert(_Generic(typeof(pi), int *: 1, default: 0));

const int *pci; // valid
typeof(&*pci) pci2; // pci2 has type "const int *" despite &
static_assert(_Generic(typeof(pci2), const int *: 1, default: 0));

int *_Optional opi; // invalid: "int *" is not a referenced type
_Optional int oi; // invalid: "int" is not a referenced type
_Optional struct s; // invalid: "struct s" is not a referenced type
_Optional int oa[2][3]; // invalid: int is not a referenced type
_Optional int (*poa)[2][3]; // valid

typeof(*poi) k; // invalid: "int" is not a referenced type
typeof(*pci) m; // valid
static_assert(_Generic(typeof(m), const int: 1, default: 0));

typedef int U[15];
_Optional U oat; // invalid: oat has type "_Optional int [15]"
_Optional U *poat; // valid: poat has type "_Optional int (*)[15]"

_Optional int *f(float); // valid
_Optional int f2(float); // invalid: int is not a referenced type
_Optional int (*fp)(float); // invalid: int is not a referenced type

typedef int F(float);
_Optional F *fp2; // valid: fp2 has type "int (*)(float) _Optional"

void h(_Optional int); // invalid: "int" is not a referenced type
void l(_Optional int param[2][3]); // valid: param is a pointer
```

EXAMPLE 5 The snippet below illustrates valid and invalid use of the _Optional type qualifier in assignments and function calls:

```
void fred(_Optional int *i)
{
    void foo(int *);
    void bar(_Optional int *);
    int *j, k;
    _Optional int *m;

    j = i; // violates type constraints for =
    foo(i); // violates type constraints for function call

    m = i; // valid
    i = j; // valid
    bar(j); // valid

    j = (int *)i; // valid
    foo((int *)i); // valid

    // type constraints aren't lifted by path-sensitive analysis
    if (i) {
        j = i; // violates type constraints for =
        foo(i); // violates type constraints for function call
    }

    i = &k;
    j = i; // violates type constraints for =
    foo(i); // violates type constraints for function call
}
```

EXAMPLE 6 The snippet below illustrates the recommended effect of the `_Optional` type qualifier on production of path-sensitive diagnostics:

```
void jim(_Optional int *i)
{
    void foo(int *);
    int *j, k;

    // A diagnostic is recommended for the following statements,
    // because of the type and unconstrained value of i
    *i = 10;
    k = *i;
    j = &*i;
    foo(&*i);
    foo(&i[15]);

    // No diagnostic is recommended for the following
    // statements because the value of i is constrained
    // to non-null
    if (i) {
        *i = 5;
        foo(&*i);
        foo(&i[15]);
    }

    for (; i;) {
        *i = 6;
        foo(&*i);
        foo(&i[15]);
    }

    while (i) {
        *i = 7;
        foo(&*i);
        foo(&i[15]);
    }

    if (!i) {
    } else {
        *i = 8;
        foo(&*i);
        foo(&i[15]);
    }

    k = i ? *i : 0;
}
```

EXAMPLE 7 The snippet below illustrates the recommended effect of conversions of constrained values to `_Optional`-qualified referenced types on production of diagnostics:

```
void sheila(int *j)
{
    int k;
    _Optional int *i, *m;

    if (!j) return;

    // Conversion preserves non-null constraint on value
    i = j;

    // No diagnostic is recommended for the following
    // statements, because the value of i is constrained
    // to non-null
    *i = 10;
    k = *i;

    // Cast preserves non-null constraint on value
    *(_Optional int *)i = 10;
    k = *(_Optional int *)i;

    // Assignment preserves non-null constraint on value
    m = i;

    // No diagnostic is recommended for the following statements,
    // because the value of m is constrained to non-null
    *m = 10;
    k = *m;
}
```

EXAMPLE 8 The snippet below illustrates the recommended effect of conversions of unconstrained values from `_Optional`-qualified referenced types on production of diagnostics:

```
void andy(_Optional int *i)
{
    int *j, k;

    // A diagnostic is recommended for the following statements,
    // because of the unconstrained value of i
    *i = 10;
    k = *i;

    // No diagnostic is recommended for the following statements,
    // despite the unconstrained value of i
    *(int *)i = 10;
    k = *(int *)i;

    // A diagnostic is recommended for the following statements,
    // because of the unconstrained value of i
    *(_Optional int *)(int *)i = 10;
    k = *(_Optional int *)(int *)i;

    // Cast does not constrain value to non-null
    j = (int *)i;

    // No diagnostic is recommended for the following statements,
    // despite the unconstrained value of j
    *j = 1;
    k = *j;

    // A diagnostic is recommended for the following statements,
    // because of the unconstrained value of j
    *(_Optional int *)j = 10;
    k = *(_Optional int *)j;

    // Conversion does not constrain value to non-null
    j = i; // violates type constraints for =

    // No diagnostic is recommended for the following statements,
    // despite the unconstrained value of j
    *j = 2;
    k = *j;
}
```

EXAMPLE 9 The snippet below illustrates the recommended effect of conversions of unconstrained values to `_Optional`-qualified referenced types on production of diagnostics:

```
void hazel(int *i)
{
    _Optional int *j;
    int k;

    // A diagnostic is recommended for the following statements,
    // because of the unconstrained value of i
    *(_Optional int *)i = 10;
    k = *(_Optional int *)i;

    // No diagnostic is recommended for the following statements,
    // despite the unconstrained value of i
    *(int *)(_Optional int *)i = 10;
    k = *(int *)(_Optional int *)i;

    // Conversion does not constrain value to non-null
    j = i;

    // A diagnostic is recommended for the following statements,
    // because of the unconstrained value of j
    *j = 10;
    k = *j;

    // No diagnostic is recommended for the following statements,
    // despite the unconstrained value of j
    *(int *)j = 10;
    k = *(int *)j;
}
```

Annex I (informative) Common warnings

I.2 Common situations

1 The following are a few of the common situations where an implementation may generate a warning:

— An unrecognized #pragma directive is encountered (6.10.8).

— An implicit conversion of a null pointer constant to a pointer type whose referenced type is not optional-qualified is encountered (6.3.3.3, 6.7.3).

— An object of a pointer type whose referenced type is not optional-qualified is default-initialized (6.7.11, 6.7.3).

Acknowledgements

I would like to recognize the following people for their ideas, help, feedback, and encouragement: Mihail Atanassov, Nikunj Patel, Peter Smith, Matthew Clarkson, Mats Petersson, Anastasia Stulova, Raffaele Aquilone, Jonathan Ely, Jim Chaney, Aaron Ballman, Alejandro Colomar, Elizabeth Bazley, Jakub Łukasiewicz, Martin Uecker and Jens Gustedt.

References

[0] mypy 0.991 documentation
https://mypy.readthedocs.io/en/stable/

[1] TypeScript: JavaScript With Syntax For Types
https://www.typescriptlang.org/

[2] Compiler Explorer
https://godbolt.org/z/GvjdxbjTW

[3] Static analysis in GCC 10 | Red Hat Developer
https://developers.redhat.com/blog/2020/03/26/static-analysis-in-gcc-10

[4] Compiler Explorer
https://godbolt.org/z/s87vrz4n1

[5] Common Function Attributes (Using the GNU Compiler Collection (GCC))
https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#Common-Function-Attributes

[6] ARM Compiler toolchain Compiler Reference Version 5.03
https://developer.arm.com/documentation/dui0491/i/Compiler-specific-Features/--attribute----nonnull---function-attribute

[7] RFC: Nullability qualifiers – Clang Frontend – LLVM Discussion Forums
https://discourse.llvm.org/t/rfc-nullability-qualifiers/35672

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

[9] Clang-Tidy - Extra Clang Tools 17.0.0git documentation
https://clang.llvm.org/extra/clang-tidy/

[10] Compiler Explorer
https://godbolt.org/z/7TTa9ehWs

[11] Compiler Explorer
https://godbolt.org/z/7cTWETePW

[12] Compiler Explorer
https://godbolt.org/z/dsE67Mdq1

[13] D9004 Addition of clang nullability attributes
https://reviews.freebsd.org/D9004

[14] Norcroft C compiler – Wikipedia
https://en.wikipedia.org/wiki/Norcroft_C_compiler

[15] RISC OS Open: Desktop Development Environment
https://www.riscosopen.org/content/sales/dde

[16] cc65 - a freeware C compiler for 6502 based systems
https://cc65.github.io/

[17] Compiler Explorer
https://godbolt.org/z/xad86788P

[18] Compiler Explorer
https://godbolt.org/z/9xn3Esqvf

[19] Warning Options – Using the GNU Compiler Collection (GCC)
https://gcc.gnu.org/onlinedocs/gcc-4.2.4/gcc/Warning-Options.html

[20] Compiler Explorer
https://godbolt.org/z/8bqMdPeK4

[21] Compiler Explorer
https://godbolt.org/z/6jKxdfKb4

[22] Stroustrup, "The Design and Evolution of C++" (1994)

[23] The author's fork of the LLVM Project
https://github.com/chrisbazley/llvm-project

[24] Slay Some Earthly Demons IV, Martin Uecker
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3342.pdf