Proposal for C2y

WG14 N3578

Title: Dogfooding the _Optional qualifier

Author, affiliation: Christopher Bazley, Arm. (WG14 member in individual capacity – GPU expert.)
Date: 2025-06-05

Proposal category: New feature

Target audience: Committee, General developers

Abstract: In this paper, I demonstrate real-world use cases for `_Optional` — a proposed new type qualifier that offers meaningful nullability semantics without turning C programs into a wall of keywords. By solving problems in real programs and libraries, I learned much about how to use the new qualifier to be best advantage, what pitfalls to avoid, and how it compares to Clang's nullability qualifiers. I also uncovered an unintended consequence of my design.

Prior art: N3089, N3422.

Dogfooding the _Optional qualifier

Reply-to: Christopher Bazley (chris.bazley.wg14@gmail.com)
Document No: N3578
Date: 2025-06-05

Table of Contents

Summary of Changes

N3578

- Initial version

Motivation

At the 72nd meeting of WG14 in Graz, the committee decided to create a Technical Specification (TS) based on my proposal, N3422, which formalizes `_Optional` as a type qualifier for nullability, after previously voting overwhelmingly for something along the lines of N3089. Nevertheless, there remains persistent opposition to aspects of my design, notably from Clang's lead maintainer. Since I am not immune to criticism, I've been eating my own dog food this Easter.

Originally, (years ago now) I used my prototype to add pointer nullability information to parts of the user-space Mali GPU driver. This code is not open source; therefore, it is not available for anyone to study, nor is there any evidence to support my claim. However, a lot of my own code is open source on GitHub, and it is that code which I have been updating to use the `_Optional` qualifier.

It's reasonable to ask why I haven't done that before. For one thing, I don't want to take on the task of porting Clang to my preferred development platform. (Even if I had time, it would be appallingly slow and probably run out of memory.) This is not a criticism of Clang in particular, but of complex modern compilers in general. I've hesitated to incorporate `_Optional` into my hobby projects, partly due to how my design was received by Clang's maintainers and partly because I must work and parent my children for most of the time.

However, I cannot expect to be taken seriously if I don't show real-world use cases.

None of the following projects were cherry-picked for suitability, nor did I spend hours refactoring to show my code in a better light or to fit better into my proposed model. For example, I would no longer write parameter declarations with abstract declarators and interleaved comments instead of parameter names (a style that I copied from old system headers).

The following projects are covered by [GPL v2](GPL v2):

- 3dObjLib
- ApocToObj
- CBDebugLib
- CBLibrary, CBLibrary (2), CBLibrary (3), CBLibrary (4)
- CBOSLib, CBOSLib (2)
- CBUtilLib, CBUtilLib (2)
- ChocToObj
- GKeyComp
- GKeyLib
- SF3KtoObj
- SF3KtoProT, SF3KtoProT (2), SF3KtoProT (3)
- SF3KUtils
- strb_t
- StreamLib

Please try to imagine what these commits would look like if every pointer parameter been annotated with either `_Nullable` or `_Nonnull`. Like `restrict`, Clang's nullability attributes lack any mechanism for the compiler to check that they have been used consistently or correctly. Would you want to write or maintain such code? I don't think this is a false dichotomy; one way or another, the days of relying on the diligence of programmers to handle null pointer values safely are numbered.

Excessive verbiage will kill C as a language that programmers *actually want to use*. We are already halfway there. Yes, it may persist for a while for the purpose of describing system headers, but not as a language for writing anything valuable. I'm not yet convinced that those working predominantly on C++ codebases appreciate this, or that they care. Why should they?

Most C programmers do not want to write code like this (real function) definition:

```
void DrawObjs_to_screen(
 const PolyColData *const _Nonnull poly_colours,
 const HillColData *const _Nullable hill_colours,
 const CloudColData *const _Nonnull clouds,
 ObjGfxMeshes *const _Nonnull meshes,
 const View *const _Nonnull view,
 const MapArea *const _Nonnull scr_area,
 DrawObjsReadObjFn *const _Nonnull read_obj,
 DrawObjsReadHillFn *const _Nonnull read_hill,
 void *const cb_arg,
 TriggersData *const _Nullable triggers,
 const ObjEditSelection *const _Nullable restrict selection,
 const Vertex scr_orig,
 const bool is_ghost,
 const ObjEditSelection *const _Nullable restrict occluded)
{
  ...
}
```

At that point, writing C starts to feel more like painstakingly navigating a type system minefield than enjoying a programming language — let alone a hobby. I would rather write C++ than write C in the cluttered style shown in the preceding example. If this style ever becomes widespread, it will risk reinforcing the stereotype of C as an outdated, overly burdensome language — which would be a shame, given how clear and concise it once was.

I really did write all but one of those `const` qualifiers in the preceding example because I am punctilious to the point of mild obsession. On the other hand, I am also clear-eyed about the fact that most programmers have neither the time nor the inclination to write code like this. I did not add the `restrict` qualifiers, and I would never add `_Nonnull` or `_Nullable`.

For me, Clang's nullability attributes were the straw that broke the camel's back. Despite occupying the same syntactic position, they are not qualifiers at all: they are metadata bolted on to types, invisible to the language's core semantics, and incompatible with the clean declarator syntax that C is built around.

Qualifiers have been an important part of C since it was standardized by ANSI, but Denis Ritchie (co-creator of C) was initially sceptical of them:

> *Let me begin by saying that I'm not convinced that even the pre-December qualifiers (`const' and `volatile') carry their weight; I suspect that what they add to the cost of learning and using the language is not repaid in greater expressiveness.*

However, he did not outright reject the idea. He also wrote:

> *Const has two virtues: putting things in read-only memory, and expressing interface restrictions. For example, saying*
>
> *char *strchr(const char *s, int c);*
>
> *is a reasonable way of expressing that the routine cannot change the object referred to by its first argument. I think that minor changes in wording preserve the virtues, yet eliminate the contradictions in the current scheme.*

Indeed, `const` ended up being almost universally accepted. It does a lot to make interfaces self-documenting and to allow verification that read-only objects are not modified. I believe that `_Optional` is in the same tradition: opt-in, minimalist, and useful without imposing too great a burden on compiler authors. By this standard, nullability attributes fall short.

By restraining the desire to over-specify everything, it's possible to reduce the blizzard of clutter; we can then selectively introduce `_Optional` and yet still write mnemonic-style declarators such as `*hill_colours` (in accordance with K&R's design for the language's syntax):

```
void DrawObjs_to_screen(
 const PolyColData                *poly_colours,
 _Optional const HillColData      *hill_colours,
 const CloudColData               *clouds,
 ObjGfxMeshes                     *meshes,
 const View                       *view,
 const MapArea                    *scr_area,
 DrawObjsReadObjFn                *read_obj,
 DrawObjsReadHillFn               *read_hill,
 void                             *cb_arg,
 _Optional TriggersData           *triggers,
 _Optional const ObjEditSelection *selection,
 const Vertex                     scr_orig,
 const bool                       is_ghost,
 _Optional const ObjEditSelection *occluded)
{
  ...
}
```

(Here, `_Optional` qualifies the object being pointed to, not the pointer itself, like most typical usage of `const`.)

Incidentally, Python has distinct [abstract base classes] for read-only collections: e.g., (immutable) `Mapping` vs. (mutable) `dict` and (immutable) `Sequence` vs. (mutable) `list`. Consequently, there is no `Const[dict]` in Python's type annotations, let alone a `Nonconst[dict]`. This is an interesting alternative to qualifiers but a poor fit for a language (like C) that has no generalized and extensible mechanism for subtype polymorphism.

Methodology

It is possible to introduce `_Optional` gradually by reading code and adding the qualifier to the referenced type of any pointer that is only dereferenced after an explicit null check. However, this is not a very thorough method. On the other hand, it is the least disruptive way to start, especially if you restrict yourself to 'leaf' functions whose arguments can be null, but which do not pass those values to other functions or assign them to global variables.

To have confidence that most referenced types that ought to have been qualified as `_Optional` were updated, it is necessary to think about the possible origin of null pointer values:

1. As a result of default initialisation of an object of pointer type (e.g., because it lacks an explicit initialiser or it is initialized with an empty initializer).
2. As a result of the implementation-defined conversion of an integer to a pointer type.
3. As a result of assigning the macro `NULL` or any other null pointer constant.
4. As a result of assigning the return value of a third-party library function that may return null.

Today, there is nothing that can be done about points 1 and 2, although they are interesting avenues for future improvement of tooling. Instead, I chose to focus on points 3 and 4.

In my projects, I only use `NULL` as a null pointer constant (i.e. no `nullptr` from C23, no naked `0` or naked `(void *)0`). I therefore redefined the `NULL` macro to force generation of diagnostics when `NULL` is assigned to objects whose referenced type is not qualified as `_Optional`:

```
#undef NULL
#define NULL ((_Optional void *)0)
```

Strictly speaking, a null pointer constant is not a pointer to a qualified type, therefore my definition of `NULL` is not a null pointer constant:

> *An integer constant expression with the value 0, such an expression cast to type void *, or the predefined constant nullptr is called a null pointer constant.*

(6.3.3.3 Pointers, ISO/IEC 9899:202y (en) — N3550 working draft)

This is a trade-off: I chose practical enforcement over strict standard compliance. In practice, `((_Optional void *)0)` serves as a null pointer constant in every kind of situation but one. In pedantic mode, Clang produces diagnostic messages if `((_Optional void *)0)` is assigned to a function pointer:

```
<source>:9:7: warning: assigning to 'int (*)(int)' from 'const void *' converts
between void pointer and function pointer [-Wpedantic]
    9 |   foo = NULL;
      |       ^ ~~~~
```

Consequently, I had to add casts when assigning `NULL` (as redefined with `_Optional`) to function pointers, since the qualifier is not compatible with function types:

```
  /* Create a temporary sky file */
  EditSky edit_sky;
  (void)edit_sky_init(&edit_sky, NULL, (EditSkyRedrawBandsFn *)NULL,
    (EditSkyRedrawRenderOffsetFn *)NULL, (EditSkyRedrawStarsHeightFn *)NULL);

  Editor tmp;
  editor_init(&tmp, &edit_sky, (EditorRedrawSelectFn *)NULL);
```

Another thing to watch out for when using Clang is that including `<stddef.h>` (or any header file that might include `<stddef.h>`) reinstates the original definition of `NULL` — even if `<stddef.h>` was also included *before* redefining the macro! In practice, this means that a header file that redefines `NULL` must be included last by any file that includes it.

Neither of these issues will exist when compilers gain the ability to diagnose misassignment of null pointer constants (analogous to GCC's `-Wwrite-strings` option to diagnose misuse of string literals), because the `_Optional` qualifier won't need to be explicit in the definition of `NULL` (any more than the `const` qualifier is explicit in `"Hello world"`).

It would be idiomatic to initialise a pointer to an `_Optional` object with the return value of `malloc`, even though the return type of malloc is `void *` rather than `_Optional void *`:

```
_Optional int *ip = malloc(sizeof *ip);
```

Just as assigning the return value of `malloc` to an object of type `int *` adds useful type information, so does assigning it to an object of type `_Optional int *`. However, this technique requires a thorough search for all calls to `malloc` in a program and constant vigilance thereafter.

I therefore defined shims for some third-party library functions. Their purpose is to catch assignment of a return value that can be null to a pointer whose referenced type is not qualified as `_Optional`. They act as a transitional mechanism. I did not do this for functions provided by my own libraries, because it was more useful to invest my time in updating those.

Each shim comprises a substitute function and a macro that replaces a call to the original function with a call to the substitute. Each substitute wraps a call to the original but has a return value with a more qualified type. For example, here is a shim to update the return type of `malloc`:

```
static inline _Optional void *optional_malloc(size_t n)
{
    return malloc(n);
}
#undef malloc
#define malloc(n) optional_malloc(n)
```

The same strategy is used for other functions that can return null, such as `strchr` and `realloc`.

Of course, this is only half of the picture: it would be inconvenient to define a shim to `malloc` that returns a pointer to a qualified type, but not a shim to `free` that accepts the same type. Consequently, a few more shims were needed to adapt parameter types:

```
static inline void optional_free(_Optional void *x)
{
    free((void *)x);
}
#undef free
#define free(x) optional_free(x)
```

`fopen` was the only function operating on a stream that needed a shim. For example, `fclose` does not accept a null pointer.

Known limitations of my approach

- Assignment of `NULL` to function pointers requires explicit casts.
- Redefining `NULL` is fragile across `#include` directives.
- Requires a manual audit of third-party library functions to identify those which may return a null pointer.

Ultimately, if toolchain support for `_Optional` improves, much of this machinery — such as redefining `NULL` or writing wrappers — will become unnecessary. In the meantime, this approach helps enforce correct nullability semantics with my prototype fork of Clang.

Virtue from necessity

Many of the changes that I made whilst dogfooding the `_Optional` qualifier were to add the sigil `&*` to expressions in which the qualifier would not otherwise be removed from a referenced type. For example, in a [commit]() of +1449 -1150 lines, 512 lines added `_Optional` (35%), 300 lines added `&*` (20%), and 2 lines (0.1%) added both. That's closer to parity between use of `_Optional` and `&*` than I expected, but I do not think either significantly impacts the terseness or readability of the code.

I did not make a big effort to minimize the use of `&*` because it is easy to type and transparent to tools that don't understand `_Optional`; the way to do so would be to minimise the number of individual dereferences of maybe-null pointers. I hope everyone can agree that would be a worthy goal, irrespective of anything else.

This idiom may appear ugly, but I have yet to hear anyone propose a practical alternative that doesn't require path-sensitive analysis. In any case, I rather like having something to tell me "Watch out!" when a pointer is assumed to be non-null. It makes it easier to validate code by eye, which is important for code reviews. If a reviewer cannot tell that a pointer argument should not be null without looking at the description of a called function, that is time-consuming.

Anyone can see that a call such as `gkeycomp_compress(&*comp, &params)` should be guarded by control flow statements to ensure that `comp` is never null. Traditionally, it would be necessary to find every definition of the `gkeycomp_compress` function that the translation unit containing this call could conceivably be linked with (assuming they are all open source, and disregarding definitions that haven't been written yet), then analyse every definition to discover whether it is safe to use `comp` as an argument.

`&*` (or any equivalent) can be used as an assertion that a pointer is not null *regardless of whether the referenced object needs to exist*. It is often the case that although a callee could accept null, the programmer does not intend to pass null to that function (either because any error should have been handled earlier, as in the case of `free`, or because null has a special meaning, as in the case of `strtok`). This use-case is enabled by path-sensitive analysis, not made redundant by it.

Using `&*ptr` as a signal —not just as a workaround— enhances the readability of code by making the programmer's intent explicit and allows that intent to be translated into diagnostic messages if appropriate.

_Optional makes code self-documenting

Many large undocumented function parameter lists (the details of which I had long forgotten) immediately became at least somewhat self-documenting as a result of selectively adding the `_Optional` qualifier.

There's no chance that I would have bothered to go back and document functions in my hobby projects after all this time; adding the `_Optional` qualifier was a lot easier, more useful and more fun.

For example, this:

```
static bool process_file(const char * const model_file,
                         const char * const index_file,
                         const char * const output_file,
                         const int first, const int last,
                         const char * const name,
                         const long int data_start,
                         const char * const mtl_file,
                         double const thick,
                         const unsigned int flags, const bool time,
                         const bool raw)
```

Became this:

```
static bool process_file(const char * const model_file,
                         _Optional const char * const index_file,
                         _Optional const char * const output_file,
                         const int first, const int last,
                         _Optional const char * const name,
                         const long int data_start,
                         const char * const mtl_file,
                         double const thick,
                         const unsigned int flags, const bool time,
                         const bool raw)
```

The reference parameters marked `_Optional` immediately stand out because references to non-optional objects are *not* marked with `_Nonnull` (or equivalent). Now, someone can instantly see that they don't need to pass an object name, index file name or output file name to `process_file`. This might be sufficient to jog their memory or even to guess correctly why not. (Clue: it has something to do with `stdin` and `stdout`.)

Dealing with interface mismatches

The chief proponent of Clang's nullability attributes has said that, for his users, "not sure whether it can be null" (aka `_Null_unspecified`, but usually implicit) is the most important category of pointer.

I did not require a third state when dogfooding the `_Optional` qualifier, nor do I think it would have meaningful semantics for my programs. We do not talk about whether data is immutable, mutable or not-sure; nor is that distinction part of the language. Either an lvalue is assumed to be modifiable or it is not; either an lvalue is assumed to be valid or it is not.

Does anyone seriously wish that the ANSI C committee had mandated `_Const`, `_Nonconst` and `_Const_unspecified` attributes instead of a single `const` qualifier? Would the state of C programming be better today if every mutable type were qualified as `_Nonconst`? I do not think so.

However, nor do I not dismiss the idea that it is sometimes necessary to update code separately from libraries that it depends upon, or update a library separately from code that depends upon it. My original dogfooding exercise in 2022 modified a codebase that had few external dependencies; many real projects are not like that.

There are two main scenarios of interest, when it comes to compatibility. I'll examine them in turn, using real code examples, then compare use of `_Optional` with use of Clang's nullability attributes.

Outdated callers that consume potentially-null pointers

A library function that can return null might be updated to output a pointer to an `_Optional`-qualified type before all code that depends on that library has been updated to handle `_Optional`-qualified types:

```
_Optional GKeyComp *gkeycomp_make(unsigned int /*history_log_2*/);
```

Unless the `_Optional` qualifier is used in `_Generic` selection, it can be defined for compatibility purposes as a macro that expands to nothing. This makes it relatively easy to avoid the need to update outdated code that consumes a potentially-null pointer produced by a function.

I went further than that: every library header that I updated to specify an interface using `_Optional` also defines it as a macro that expands to nothing unless explicitly overridden:

```
#if !defined(USE_OPTIONAL) && !defined(_Optional)
#define _Optional
#endif
```

To enable the `_Optional` qualifier, `USE_OPTIONAL` must be defined as part of the command line used to invoke the compiler. This avoids imposing any requirement on outdated code that might depend on such headers.

Once a decision has been made to update an outdated caller of a function that can return null, the referenced type of any variable to which the return value is assigned must be updated to add the `_Optional` qualifier:

```
_Optional GKeyComp *comp = NULL;
// ...
comp = gkeycomp_make(history_log_2);
```

This makes it easier to reason about the calling code, since the imported constraint on the variable is explicitly stated. However, subsequent assignments of the variable's value (including in function calls) may then cause constraint violations that require further updates to the calling code.

For example, a constraint violation may be diagnosed in a subsequent call to a function that does not accept the address of an `_Optional` object:

```
gkcomp.c:171:32: warning: passing '_Optional GKeyComp *' (aka '_Optional struct
GKeyComp *') to parameter of type 'GKeyComp *' (aka 'struct GKeyComp *') discards
qualifiers [-Wincompatible-pointer-types-discards-qualifiers]
  171 |     status = gkeycomp_compress(comp, &params);
      |                                ^~~~
/work/GKeyLib/GKeyComp.h:68:54: note: passing argument to parameter here
   68 | GKeyStatus gkeycomp_compress(GKeyComp      */*comp*/,
      |                                                     ^
```

Having checked that the address of the `_Optional` object cannot be assigned to an argument or variable that discards the qualifier if the pointer is actually null, those expressions will need [updating to explicitly remove the `_Optional` qualifier] (e.g., using the `&*` idiom):

```
status = gkeycomp_compress(&*comp, &params);
```

If available, path-sensitive analysis can then be used to verify that the updated expressions really are guarded by control flow statement.

In contrast, had the `gkeycomp_make` function been updated to output a `_Nullable` pointer then no changes would have been required to the calling code. This is because the `_Nullable` attribute is not really part of the type; it has no meaningful semantics without path-sensitive analysis.

C claims to be a language that allows "programmers and tools to reason about code, allows for diverse implementations, keeps compilation times short". Programmers cannot reason about code into which constraints are imported invisibly. Relying solely on path-sensitive analysis has advantages but it is conducive neither to diverse implementations nor to short compilation times.

Outdated callees that consume potentially-null pointers

Some code that calls a library function that can accept null might be [updated to pass a pointer to an `_Optional`-qualified type] before the library they depend on has been [updated to handle `_Optional`-qualified types]:

```
_Optional CONST _kernel_oserror *canonicalise(_Optional char **b,
                                              _Optional const char *pv,
                                              _Optional const char *ps,
                                              const char *f)
{
  assert(b != NULL);
  assert(f != NULL);
  DEBUGF("Canonical: About to do path '%s' with variable '%s' and string '%s'\n",
         f, STRING_OR_NULL(pv), STRING_OR_NULL(ps));

  /* First pass - determine buffer size needed */
  size_t nbytes;
  _Optional CONST _kernel_oserror *e = os_fscontrol_canonicalise(
                                 NULL, 0, pv, ps, f, &nbytes);
```

The preceding function serves as a memory-allocation veneer for a function in another library and passes several potentially-null pointers (`pv` and `ps`) straight through. This situation is rare, in my experience. Usually, potentially-null arguments are either pointers being passed back into a library whence they originated (as in the case of `malloc` and `free`), or they are null pointer constants.

There is no way of preprocessing an old library header such that `_Optional` qualifiers magically appear in the parameter types of functions that it declares, therefore the only place to address the problem is in the calling code.

Casts can be used to remove the `_Optional` qualifier from the referenced type of arguments passed to an outdated function:

```
_Optional CONST _kernel_oserror *e = os_fscontrol_canonicalise(
        (char *)NULL, 0, (const char *)pv, (const char *)ps, f, &nbytes);
```

Casts are the solution that C programmers have always used when calling functions that do not accept pointer-to-`const`, but they are not type-safe. A macro to remove the `_Optional` qualifier from a referenced type more safely can be created by making use of the fact that the operand of `typeof` is not evaluated:

```
#define optional_cast(p) ((typeof(&*(p)))(p))

_Optional CONST _kernel_oserror *e = os_fscontrol_canonicalise(
        optional_cast(NULL), 0, optional_cast(pv), optional_cast(ps), f, &nbytes);
```

This has another advantage which is that macro-style casts are easier to find, and their purpose is immediately obvious. When passing `NULL` as an argument, less information is lost by using an ordinary cast, and `(void *)NULL` is as easy to spot or search for as `optional_cast(NULL)`.

An alternative might be to use a null pointer constant such as `(void *)0` in place of my modified definition of `NULL`, but that would not be future proof should compilers gain the ability to diagnose misassignment of null. The other alternative of `0` is worse for readability and would provoke a diagnostic message if `-Wzero-as-null-pointer-constant` were enabled.

In contrast, had the `canonicalise` function instead tried to pass `_Nullable` pointer arguments `pv` and `ps` to an outdated version of `os_fscontrol_canonicalise` then no changes would have been required to either function. This is because the `_Nullable` attribute is not really part of the type; it has no meaningful semantics without path-sensitive analysis.

Neither Clang nor its static analyser complain when the `_Nullable` attribute is implicitly discarded, nor when a null pointer constant is an argument unless the callee's parameters have explicitly been annotated as `_Nonnull`. Effectively, the default assumption is that any function can accept null or a `_Nullable` pointer. This requires all parameters of pointer type to be written with the `_Nonnull` attribute; a bigger change to use of the language than selective use of `_Optional`.

Casts in loop macros

In general, casts are discouraged because they discard useful type information, but in some cases — such as macro-generated `for` loops— they remain the most practical tool.

I found them useful in the specific use-case of a loop macro that traverses an intrusive linked list:

```
#define LINKEDLIST_FOR_EACH_SAFE(list, item, tmp) \
  for (LinkedListItem *(item) = (LinkedListItem *)linkedlist_get_head(list), \
                      *(tmp); \
       (tmp) = (item) ? \
                  (LinkedListItem *)linkedlist_get_next(item) : \
                  (LinkedListItem *)NULL, \
       (item) != NULL; \
       (item) = (tmp))
```

(Reformatted for greater clarity.)

Both `linkedlist_get_head` and `linkedlist_get_next` return a pointer to an `_Optional LinkedListItem`. I chose to cast away the `_Optional` qualifier rather than qualify the referenced type of `item` to match the return type of these functions, because users of the macro expect a non-null pointer to the current item within the loop body. Given the constraints of `for` statements, I couldn't think of a better way to express that.

In practice, this type information is often discarded immediately by macros like `CONTAINER_OF`, so the precise type of `item` rarely matters outside the loop header:

```
LINKEDLIST_FOR_EACH_SAFE(&path->waypoints, item, tmp)
{
  Waypoint *const waypoint = CONTAINER_OF(item, Waypoint, link);
  waypoint_delete(waypoint);
}
```

In other cases, it would not be too onerous to use the `&*` idiom within the loop body — merely an unwelcome surprise:

```
  LINKEDLIST_FOR_EACH_SAFE(&list, item, tmp)
  {
    if (j++ % KeepInterval)
    {
      linkedlist_remove(&list, &*item);
    }
  }
```

While using `&*item` is safe and amenable to static analysis, I feel that casting in the macro definition produces cleaner and more intuitive code.

Callback context conundrum

In C, null is used for two semantically distinct purposes: to indicate the absence of a referenced object, or as a placeholder where a pointer is required but its value is irrelevant. The latter usage is common in callbacks, where null is often passed simply because the callback function signature requires a pointer argument.

An example is the `void *` parameter of the standard library function `bsearch_s`, whose value is passed to a user-specified comparison function. If the comparison function requires no context, then there is no point in requiring a non-null pointer to be passed.

Precedent for use of existing qualifiers

Let us consider the precedent set by `const` and `volatile`. It is plausible for callback functions to use immutable or volatile contexts, yet it is rare for C libraries to allow the address of such contexts to be passed without casting.

For example, [this function]() allows an event handler to be registered:

```
typedef int (WimpEventHandler) (int event_code,
                                WimpPollBlock *event,
                                IdBlock       *id_block,
                                void          *handle);

_kernel_oserror *event_register_wimp_handler (ObjectId object_id, int event_code,
                                              WimpEventHandler *handler,
                                              void *handle);
```

For a function such as `event_register_wimp_handler` to accept a context pointer of type `const void *`, `volatile void *` or `const volatile void *`, the two declarations would need to have been written like this:

```
typedef int (WimpEventHandler) (int event_code,
                                WimpPollBlock       *event,
                                IdBlock             *id_block,
                                const volatile void *handle);

_kernel_oserror *event_register_wimp_handler (ObjectId object_id, int event_code,
                                              WimpEventHandler *handler,
                                              const volatile void *handle);
```

This is only a small inconvenience for the author of the library that provides `event_register_wimp_handler`, but a huge inconvenience for users who implement `WimpEventHandler` functions. Accepting a pointer to a qualified type in the registration function forces all event handlers to accept it too!

Nowadays, it would be possible to solve this using `_Generic`, by selecting the type of the registration function according to the type of the callback function:

```
typedef int (WimpEventHandler) (int event_code,
                                WimpPollBlock *event,
                                IdBlock       *id_block,
                                void          *handle);

typedef int (WimpEventHandlerC) (int event_code,
                                 WimpPollBlock *event,
                                 IdBlock       *id_block,
                                 const void    *handle);

typedef int (WimpEventHandlerCV) (int event_code,
                                  WimpPollBlock       *event,
                                  IdBlock             *id_block,
                                  const volatile void *handle);

typedef int (WimpEventHandlerV) (int event_code,
                                 WimpPollBlock *event,
                                 IdBlock       *id_block,
                                 volatile void *handle);

_kernel_oserror *event_register_wimp_handler (ObjectId object_id, int event_code,
                                              WimpEventHandler *handler,
                                              void *handle);

_kernel_oserror *event_register_wimp_handler_c (ObjectId object_id, int event_code,
                                                WimpEventHandlerC *handler,
                                                const void *handle);

_kernel_oserror *event_register_wimp_handler_cv (ObjectId object_id,
                                                 int event_code,
                                                 WimpEventHandlerCV *handler,
                                                 const volatile void *handle);

_kernel_oserror *event_register_wimp_handler_v (ObjectId object_id, int event_code,
                                                WimpEventHandlerV *handler,
                                                volatile void *handle);

#define event_register_wimp_handler(object_id, event_code, handler, handle) \
  _Generic(handler, \
           WimpEventHandler *: event_register_wimp_handler, \
           WimpEventHandlerC *: event_register_wimp_handler_c, \
           WimpEventHandlerCV *: event_register_wimp_handler_cv, \
           WimpEventHandlerV *: event_register_wimp_handler_v) \
   (object_id, event_code, handler, handle)
```

However, there is a combinatorial explosion as support for more qualifiers is added: all combinations of `const`, `volatile`, `_Optional` and `_Atomic` would require 16 variants of `event_register_wimp_handler`. Supporting all the desired combinations with polymorphism may be impractical.

One goal of dogfooding was to evaluate how well `_Optional` integrates with existing libraries. I therefore avoided 'cheating' by assuming that the types of `event_register_wimp_handler` and `WimpEventHandler` could be redefined arbitrarily.

Instead of polymorphic solutions such as that shown in the preceding example, there are two common ways of dealing with existing qualifiers:

1. Cast the qualifier away when converting a pointer to type `void *`, or
2. Use an extra level of indirection (i.e. pass the address of an unqualified pointer to the qualified type, instead of passing the pointer itself).

Either solution also works for the `_Optional` qualifier, but `_Optional` is a bit different: if a null pointer is passed to a callback function, then that argument cannot be used (by definition) except for its capacity to encode one bit of information: pointer-to-object vs. pointer-to-nothing.

This suggests a third option: **require the address of a callback context object.**

Substitution of a non-null value

The `writer_internal_init` function is used to [initialise an instance of a struct type]() that has a member of type `void *`. The role of this member is similar to the `context` argument passed to `bsearch_s` except that instead of being passed directly to callbacks, it can be read from a struct. It could be thought of as a pointer to instance variables of subclasses of `Writer`.

Previously, not all callers of `writer_internal_init` supplied a `data` pointer; some (having no instance variables) [instead passed]() `NULL`:

```
void writer_null_init(Writer * const writer)
{
  assert(writer != NULL);
  static WriterFns const fns = {writer_null_fwrite, writer_null_destroy};
  writer_internal_init(writer, &fns, NULL);
}
```

I did not want to allow the `data` member of the struct to be `NULL`, because that would have made work everywhere that member is used. (In general, it's a bad idea to use unnecessarily permissive types.)

Instead, I considered modifying `writer_internal_init` to automatically substitute a default pointer value for null:

```
void writer_internal_init(Writer *const writer, WriterFns const *const fns,
                          _Optional void *data)
{
  assert(writer != NULL);
  DEBUGF("Initializing writer %p with data %p\n", (void *)writer, data);
  assert(fns != NULL);

  *writer = (Writer){
    .fns = *fns,
    .data = data ? &*data : writer, // substitute a default non-null value
    // ...
  };
}
```

There is a problem with this approach though: null does not merely indicate the absence of a referenced object; it is also commonly used to indicate an error. For example, the `writer_gkey_init_from` function [allocates storage for an object] whose address is passed as 'data':

```
  WriterGKeyData *const data = malloc(sizeof(*data));
  if (data == NULL) {
    DEBUGF("Failed to allocate writer data\n");
    return false;
  }

  // ...

  static WriterFns const fns = {writer_gkey_fwrite, writer_gkey_destroy};
  writer_internal_init(writer, &fns, data);
```

If this, or any other caller of `writer_internal_init`, neglected to check for data `== NULL` then a default non-null value would have been substituted. Callback functions ([such as] `writer_gkey_destroy`) could later misinterpret the default value as the address of something else (in this case, a `WriterGKeyData` object instead of a `Writer` object):

```
static bool writer_gkey_destroy(Writer * const writer)
{
  assert(writer != NULL);
  WriterGKeyData *const data = writer->data;
```

Instead, I decided to modify callers that previously passed `NULL` to [pass a (dummy) non-null value] instead:

```
void writer_null_init(Writer * const writer)
{
  assert(writer != NULL);
  static WriterFns const fns = {writer_null_fwrite, writer_null_destroy};
  writer_internal_init(writer, &fns, writer);
}
```

The callers of `writer_internal_init` are now responsible for any consequences of passing the address of the wrong object, instead of providing a footgun by hiding this substitution.

The easiest way of conjuring up a non-null pointer compatible with `void *` is to use a string literal (e.g. `""` or `"none"`):

```
void writer_null_init(Writer * const writer)
{
  assert(writer != NULL);
  static WriterFns const fns = {writer_null_fwrite, writer_null_destroy};
  writer_internal_init(writer, &fns, "");
}
```

Although tempting, this is probably a bad idea because it would generate a diagnostic message if the code were compiled by GCC with the `-Wwrite-strings` option enabled.

In the preceding example, there is no reason for `writer_gkey_destroy` to use a `Writer` address stored in `writer->data` instead of using `writer` directly; in other cases, I was able to substitute a callback context that could plausibly be useful.

For example, I replaced this code:

```
bool IO_copy(EditWin *const edit_win)
{
  // ...
  if (E(entity2_claim(Wimp_MClaimEntity_Clipboard, export_file_types,
           estimate_cb, cb_write, cb_lost, NULL)))
  {
    return false;
  }
```

With this:

```
bool IO_copy(EditWin *const edit_win)
{
  // ...
  if (E(entity2_claim(Wimp_MClaimEntity_Clipboard, export_file_types,
           estimate_cb, cb_write, cb_lost, edit_win)))
  {
    return false;
  }
```

As it happens, the callback functions `estimate_cb`, `cb_write` and `cb_lost` do not currently need a reference to the `EditWin`, but the change is harmless and unobtrusive.

Using indirection for type compatibility

An interesting use-case that I came across concerned use of a Boolean variable as a callback context:

```
bool is_safe = true;
success = loader3_load_file(canonical_path,
                            message->data.data_load.file_type,
                            read_file, load_failed, &is_safe);
```

It would be reasonable to assume that the callback function `read_file` dereferences the callback context pointer to get a value of true or false, but it does not!

Instead, the value of the pointer is used directly, as an optimisation:

```
static bool read_file(Reader *const reader, int const estimated_size,
  int const file_type, char const *const filename, void *const client_handle)
{
  bool const is_safe = client_handle != NULL;
```

A different call to register `read_file` as a callback function does not even specify a callback context:

```
ON_ERR_RPT(loader3_receive_data(message, read_file, load_fail, NULL));
```

The type of the context and its value of `true` are irrelevant because the value of `is_safe` is unused! This illustrates that even when the pointed-to value is irrelevant, every pointer can encode one bit of information that can be extracted by comparing it with null.

Instead of changing the signature of `loader3_receive_data` to allow null as a callback context pointer, I refactored the calling code so that a callback context is always specified:

```
static bool is_safe = false;
ON_ERR_RPT(loader3_receive_data(message, read_file, load_fail, &is_safe));
```

Consequently, the callback function no longer needs to handle null:

```
static bool read_file(Reader *const reader, int const estimated_size,
  int const file_type, char const *const filename, void *const client_handle)
{
  bool const *const is_safe = client_handle;
```

Suggested best practices

- Do not permit null as a callback context simply as a placeholder.
- Avoid using a null/non-null as a substitute for a Boolean variable.
- Prefer credible (even redundant) callback contexts over dummy values.
- Prefer explicitly passing a dummy value as a callback context rather than letting the callee substitute one implicitly.

Pointer property predicament

In object-oriented programming, getter and setter functions are commonly used to access properties of an object. Some properties may be pointers. For example, the following pair of functions allow the address of some data to be associated with a window, icon or menu:

```
extern _kernel_oserror *toolbox_set_client_handle ( unsigned int flags,
                                                     ObjectId id,
                                                     void *client_handle
                                                   );

extern _kernel_oserror *toolbox_get_client_handle ( unsigned int flags,
                                                     ObjectId id,
                                                     void **client_handle
                                                   );
```

It is possible to set null as the client handle of an object, but there is rarely a use-case for doing so. Users of this library are not obliged to call `toolbox_set_client_handle`, so its argument is never just a placeholder; if the function is called at all, then the passed-in value is always meaningful — even if it is null.

Since none of my programs pass `NULL` to `toolbox_set_client_handle`, I had no incentive to create a shim to allow it. On the other hand, null is the default value of an object's client handle, therefore there is an argument for requiring callers of `toolbox_get_client_handle` to pass the address of a pointer to `_Optional void` in case the setter has not been called yet.

The declaration of the getter could be updated as follows:

```
extern _kernel_oserror *toolbox_get_client_handle ( unsigned int flags,
                                                     ObjectId id,
                                                     _Optional void **client_handle
                                                   );
```

Consequently, existing event handler code like this:

```
  void *client_handle;
  if (!E(toolbox_get_client_handle(0, id_block->ancestor_id, &client_handle)))
  {
    EditWin * const edit_win = client_handle;
```

Would need to be rewritten defensively:

```
  _Optional void *client_handle;
  if (!E(toolbox_get_client_handle(0, id_block->ancestor_id, &client_handle)) &&
      client_handle)
  {
    EditWin * const edit_win = &*client_handle;
```

I decided not to make the preceding changes for reasons of expediency: to avoid modifying a third-party interface (or creating shims for it), and to avoid updating every call to `toolbox_get_client_handle` in my own projects. Ultimately, the benefits just weren't worth the effort.

Improved code clarity

I believe that some of my changes improved code clarity because of the rule that `&s[n]` removes any `_Optional` qualifier from the referenced type of `s`, whereas `s + n` does not.

> *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.*

(N3422, _Optional: a type qualifier to indicate pointer nullability (v2))

Consequently, I was 'forced' to replace this code:

```
for (int pt_sample_no = 0;
     pt_sample_no < pt_samples->count;
     pt_sample_no++) {
  Fortify_CheckAllMemory();

  const PTSampleInfo * const ptsi = pt_samples->sample_info + pt_sample_no;
  const SampleInfo * const sample = sf_samples->sample_info + ptsi->sample_num;
```

with this:

```
_Optional const PTSampleInfo * const ptsi_array = pt_samples->sample_info;
_Optional const SampleInfo * const sample_array = sf_samples->sample_info;
if (!ptsi_array || !sample_array) {
  return false;
}

for (int pt_sample_no = 0;
     pt_sample_no < pt_samples->count;
     pt_sample_no++) {
  Fortify_CheckAllMemory();

  const PTSampleInfo * const ptsi = &ptsi_array[pt_sample_no];
  const SampleInfo * const sample = &sample_array[ptsi->sample_num];
```

I consider the latter to be an improvement in readability and efficiency as well as robustness:

- The array element syntax is clearer than use of pointer arithmetic.
- It is easier for the optimiser to see that neither `pt_samples->sample_info` nor `sf_samples->sample_info` are modified within the loop.
- It is more robust and self-evidently correct to check for `pt_samples->sample_info` or `sf_samples->sample_info` being null instead of relying on the value of `pt_samples->count` being zero in those circumstances.

Defensive control flow

In code with complex control flow that I knew would not be performance sensitive, if there was any possible ambiguity about whether a pointer could be null or not, I often chose to add explicit checks for null pointer values using an `if` statement.

For example, I chose to replace this:

```
if (success) {
  const clock_t start_time = time ? clock() : 0;

  success = processor(in, tmp != NULL ? tmp : out, history_log_2, verbose);

  if (success && time)
  {
    printf("Time taken: %.2f seconds\n",
           (double)(clock_t)(clock() - start_time) / CLOCKS_PER_SEC);
  }
}
```

With this:

```
if (success && in && out) {
  const clock_t start_time = time ? clock() : 0;

  success = processor(&*in, tmp != NULL ? &*tmp : &*out, history_log_2, verbose);

  if (success && time)
  {
    printf("Time taken: %.2f seconds\n",
           (double)(clock_t)(clock() - start_time) / CLOCKS_PER_SEC);
  }
}
```

Could I have proved that neither `in` nor `out` could be null if success were true? Yes. But defensive programming makes it easier for people and tools that analyse the code to see that it is safe, especially when there is complex conditional logic leading up to the point where a pointer must not be null. If a compiler can optimise away such checks, then it will.

There are also cases where I did not follow that rule but perhaps, I should have done so. In the following code, `input_file` is never null if `tmp` is not null, but that is not evident without analysing the entire function:

```
if (success) {
  if (output_file != NULL) {
    if (input_file != NULL && strcmp(&*output_file, &*input_file) == 0) {
      /* Can't overwrite the input file whilst reading from it, so direct
         output to a temporary file instead */
      if (verbose)
        puts("Opening temporary output file");

      tmp = tmpfile();
      if (tmp == NULL) {
        fprintf(stderr, "Failed to create temporary output file: %s\n",
                strerror(errno));
        success = false;
      }
    }
    // ...
  }
}

// ...

if (tmp != NULL) {
  if (success) {
    if (output_file != NULL) {
      /* Open the real output file */
      if (verbose)
        printf("Opening output file '%s'\n", output_file);

      out = fopen(&*input_file, "wb");
```

Defensive provision of objects

Where a function requires its caller to pass a pointer to an object of a specific type (as opposed to a pointer to `void`), I often found it preferable to pass the address of an object of the expected type instead of passing `NULL` based on assumptions about the definition of the function.

For example, the following function doesn't use the `WimpMessage` object passed to it:

```
static int mode_change_msg(WimpMessage *const message, void *const handle)
{
  // ...
  NOT_USED(handle);
  NOT_USED(message);
```

This function is not only called on receipt of a ModeChange message; it is also called when the program starts up. Knowing that it uses neither of its arguments, I originally passed `NULL` as the value of both:

```
  /* Read variables for current screen mode */
  mode_change_msg(NULL, NULL);
```

However, this is not how a `WimpMessageHandler` is called by the event library, therefore it seemed preferable to pass the address of a real `WimpMessage` object instead of casting `NULL` to remove the `_Optional` qualifier from the referenced type:

```
/* Read variables for current screen mode */
  mode_change_msg(&(WimpMessage){0}, &(int){0});
```

A more efficient alternative might have been to create a new function with no parameters, called both by `mode_change_msg` and during start up.

Another example concerns unit tests for one of my editor programs.
The `editor_redo` [function](#) requires its caller to pass the address of an array of palette entries, but only uses that array when redoing a subset of actions (including `EditRecordType_Interpolate` but not `EditRecordType_Move`):

```
bool editor_redo(Editor *const editor, PaletteEntry const palette[])
{
  // ...
  EditSky *const edit_sky = editor->edit_sky;
  assert(edit_sky != NULL);
  LinkedListItem *const redo_item = get_redo_item(edit_sky);
  assert(redo_item != NULL);
  EditRecord *const rec = CONTAINER_OF(redo_item, EditRecord, link);
  edit_sky->next_undo = redo_item;

  bool changed = false;
  DEBUGF("Redo of type %d\n", (int)rec->type);
  switch (rec->type)
  {
  // ...
  case EditRecordType_Interpolate:// Requires palette
    if (s_interpolate(&edit_sky->sky, palette,
      rec->data.edit.dst_start, rec->data.edit.old_dst_end,
      rec->data.edit.fill, NULL, 0))
    {
      redraw_bands(edit_sky, rec->data.edit.dst_start,
                   rec->data.edit.old_dst_end);
      changed = true;
    }
    break;
  case EditRecordType_Move: // Does not require palette
    changed = redo_move(editor, rec);
    if (changed)
    {
      redraw_move(edit_sky, rec);
    }
    break;
```

In normal usage, it is impossible to predict which action will be redone by this function. In unit tests, however, the action type is entirely predictable. I had made use of that knowledge to pass `NULL` when calling `editor_redo` in many tests; for example, in a test for `EditRecordType_SetRenderOffset`:

```
  assert(editor_redo(&editor, NULL));
  assert(sky_get_render_offset(edit_sky_get_sky(&edit_sky)) == RenderOffset);
  check_redraw_render_offset(i++, &edit_sky);
  assert(render_offset_count == i);
```

Changing the signature of `editor_redo` to allow null to be passed would have required new control flow inside that function, but there is no real-world use case for passing null. Instead, I updated tests to always pass the address of an array regardless of whether they expected it to be used. The compound literal that I substituted simply maps every palette entry to black:

```
  assert(editor_redo(&editor, (PaletteEntry [NumColours]){0}));
  assert(sky_get_render_offset(edit_sky_get_sky(&edit_sky)) == RenderOffset);
  check_redraw_render_offset(i++, &edit_sky);
  assert(render_offset_count == i);
```

Another example where *not* passing null seemed the right thing to do was when it had been used as a stand-in for a string:

```
static CONST _kernel_oserror *lookup_error(const char *const token,
  const char *const param)
{
  /* Look up error message from the token, outputting to an internal buffer */
  return messagetrans_error_lookup(desc, DUMMY_ERRNO, token, 1, param);
}

/* ---------------------------------------------------------------------- */

static CONST _kernel_oserror *no_mem(void)
{
  return lookup_error("NoMem", NULL);
}
```

`messagetrans_error_lookup` is a variadic function, so its trailing parameter types are not explicit and cannot be qualified. Nevertheless, it does handle null arguments.

Instead of qualifying the `param` parameter of `lookup_error` as `_Optional`, I modified its callers to pass `""` instead of `NULL`. One could argue that `NULL` is more efficient, but efficient error reporting is rarely important.

The result is terser and more explicit:

```
static CONST _kernel_oserror *no_mem(void)
{
  return lookup_error("NoMem", "");
}
```

I took the same approach to third-party interfaces where the benefit —or even the validity— of allowing `NULL` to be passed was in doubt.

For example, the following library function cannot be passed a pointer to an `_Optional` buffer, even though it seems logical for its `buffer` argument to be ignored if `bytes_written` is zero:

```
extern _kernel_oserror *saveas_buffer_filled ( unsigned int flags,
                                               ObjectId saveas,
                                               void *buffer,
                                               int bytes_written
                                             );
```

Previously, `saveas_buffer_filled` could be called with `NULL` by the following code in one of my programs:

```
  void *const buffer = *dst ? (char *)*dst + safbe->no_bytes : NULL;
  DEBUGF("Saved %d bytes to buffer %p for object 0x%x\n",
         chunk_size, buffer, saveas_id);

  ON_ERR_RPT(saveas_buffer_filled(0, saveas_id, buffer, chunk_size));
```

I modified the calling code to ensure that `saveas_buffer_filled` is never called with `NULL` by passing the address of a tiny dummy object instead:

```
  static char dummy;
  void *const buffer = *dst ? (char *)*dst + safbe->no_bytes : &dummy;
  DEBUGF("Saved %d bytes to buffer %p for object 0x%x\n",
         chunk_size, buffer, id_block->self_id);

  ON_ERR_RPT(saveas_buffer_filled(0, id_block->self_id, buffer, chunk_size));
```

Allowing null arguments — the right way

In contrast to the preceding examples, I relaxed the constraints on passing null to some of my own functions to make them more robust and simplify usage. I consider such cases distinct from defensive programming because they involve a new guarantee that null is handled rather than runtime checks added merely to clarify existing code or as a precaution.

For example, the following function relied on assertions to ensure that its callers never passed a null pointer as the value of `buffer` unless the `buff_size` was zero:

```
_kernel_oserror *colourtrans_read_palette(unsigned int              flags,
                                          const ColourTransContext *source,
                                          PaletteEntry             *buffer,
                                          size_t                    buff_size,
                                          size_t                   *nbytes)
{
  _kernel_oserror *e = NULL;
  _kernel_swi_regs regs;

  assert(source != NULL);
  assert(buffer != NULL || buff_size == 0);

  assign_regs(&regs.r[0], source);

  /* Find buffer size and/or read palette into caller's buffer */
  regs.r[2] = (int)buffer;
  regs.r[3] = buff_size;
  regs.r[4] = flags;
  DEBUGF("ClrTrans: Calling ColourTrans_ReadPalette with "
         "0x%x,0x%x,0x%x,0x%x,0x%x\n",
         regs.r[0], regs.r[1], regs.r[2], regs.r[3], regs.r[4]);

  e = _kernel_swi(ColourTrans_ReadPalette, &regs, &regs);
```

I modified the function to ignore the value of `buff_size` if a null pointer is passed as the value of `buffer`:

```
_Optional _kernel_oserror *colourtrans_read_palette(
                                          unsigned int              flags,
                                          const ColourTransContext *source,
                                          _Optional PaletteEntry   *buffer,
                                          size_t                    buff_size,
                                          _Optional size_t         *nbytes)
{
  _Optional _kernel_oserror *e = NULL;
  _kernel_swi_regs regs;

  assert(source != NULL);

  if (!buffer)
  {
    buff_size = 0;
  }

  assign_regs(&regs.r[0], source);

  /* Find buffer size and/or read palette into caller's buffer */
  regs.r[2] = buffer ? (int)buffer : 0;
  regs.r[3] = buff_size;
  regs.r[4] = flags;
  DEBUGF("ClrTrans: Calling ColourTrans_ReadPalette with "
         "0x%x,0x%x,0x%x,0x%x,0x%x\n",
         regs.r[0], regs.r[1], regs.r[2], regs.r[3], regs.r[4]);

  e = _kernel_swi(ColourTrans_ReadPalette, &regs, &regs);
```

(The return value was also updated to `_Optional _kernel_oserror *` to indicate that the function may return null, which it does if no error occurred.)

In contrast to N3322, which permits null only when the length is zero (requiring tools and programmers to correlate argument values) this change makes acceptance of null explicit and unconditional. Consequently, such interfaces are both safer and more amenable to static analysis, since nullability is part of the type and does not depend on external conditions.

Casts harm analysis

Most existing code uses at least some casts. In rare cases, this hindered my efforts to ensure that all pointers that can be null are declared with their referenced type qualified by `_Optional`.

I initially overlooked [the following code](), which uses an intrusive linked list. It casts the return value of `linkedlist_for_each` (a pointer to a `_Optional LinkedListItem`) to a `LoadOpData *`, thereby discarding the `_Optional` qualifier from the referenced type:

```
static LoadOpData *_ldr2_find_record(int msg_ref)
{
  LoadOpData *load_op_data;

  DEBUGF("Loader2: Searching for operation awaiting reply to %d\n", msg_ref);
  load_op_data = (LoadOpData *)linkedlist_for_each(
                &load_op_data_list, _ldr2_op_has_ref, &msg_ref);

  if (load_op_data == NULL)
  {
    DEBUGF("Loader2: End of linked list (no match)\n");
  }
  else
  {
    DEBUGF("Loader2: Record %p has matching message ID\n", (void *)load_op_data);
  }
  return load_op_data;
}
```

The cast hides the nullability of the result, preventing both type-based diagnostics and static analysis from catching potential misuses. It was written before `linkedlist_for_each` was updated to return a pointer to a `_Optional LinkedListItem`, but its presence now discards valuable information.

Had `linkedlist_for_each` instead been declared as returning a `_Nullable` pointer, Clang's static analyser would not have produced a diagnostic either — unless `load_op_data` were declared as `_Nonnull` *with an exactly matching type*. This illustrates that nullability attributes suffer the same issue and don't offer a clear advantage in such cases.

In situations like this, only the diligence of programmers can ensure that pointer nullability information is preserved across type conversions.

Pointers as output parameters

In current practice, it is common to initialise an object (including one of pointer type) whose address will be passed to another function for use as an output parameter. The obvious initialiser for pointers is `NULL`.

An example is the character pointer `endp` in [the following code](#):

```
    char *endp = NULL;
    tile_num = strtol(name + sizeof(TILE_SPR_NAME)-1, &endp, 10);

    if (tile_num > MapTileMax || *endp != '\0')
    {
      tile_num = -1;
    }
```

The called function, `strtol`, is declared in `<stdlib.h>` as:

```
long int strtol(const char * restrict nptr, char ** restrict endptr, int base);
```

In the description of `strtol`, there is no mention of the input value of the object pointed to by `endptr` having any significance; only the value of the `endptr` argument itself:

> *A pointer to the final string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.*

(7.25.2.8 The strtol, strtoll, strtoul, and strtoull functions, ISO/IEC 9899:202y (en) — N3550 working draft)

In the preceding example, `endptr` is `&endp`, so the 'object pointed to by `endptr`' is `endp` itself.

Given that the initial value (here, `NULL`) is unused, the initialisation is redundant. Such initialisations are typically added to appease static analysers, which may otherwise complain that an object passed by address is uninitialised. They are also used to guard against future changes, or when a called function does not always assign a value to its output parameters.

With my redefinition of `NULL` as `((_Optional void *)0)`, the compiler correctly diagnoses `*endp = NULL` as a constraint violation. A naive attempt to fix this might involve qualifying the referenced type of `endp`:

```
_Optional char *endp = NULL;
```

However, this causes another problem: the `strtol` function does not accept the address of a pointer to `_Optional`. Should it?

Qualifying the type of `endptr` as `_Optional char **` would imply that the value written by `strtol` might be null. This is not the case — even on failure:

> *If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.*

(7.25.2.8 The strtol, strtoll, strtoul, and strtoull functions, ISO/IEC 9899:202y (en) — N3550 working draft)

The only reason `*endptr` might be null is if it was initialised that way and `strtol` has not yet overwritten it with `nptr`or a pointer derived from `nptr`.

My [final version](#) of the calling code was simply to remove the redundant initializer:

```
    char *endp;
    tile_num = strtol(name + sizeof(TILE_SPR_NAME)-1, &endp, 10);

    if (tile_num > MapTileMax || *endp != '\0')
    {
      tile_num = -1;
    }
```

Since `endp` has automatic storage duration, it is not subject to default initialisation. Its value is indeterminate until `strtol` writes to it — which is guaranteed because `endptr` is not null. However, this raises an important question: how should tools reason about pointer values that are null only because of default initialisation?

Global variables

Global variables (aka external object definitions) of pointer type present certain unique challenges.

Global pointers to dynamically allocated storage are typically initialised to null in their declarations; even when not explicitly initialised, default initialisation gives them a null value. They are usually assigned non-null values later, during program startup, often by an explicit initialisation function.

For example, one of my programs declared the following global variables:

```
static bool translate_cols = true;
static void *col_trans_table = NULL; /* table of colour numbers for drawing
                                        sprite in desktop (flex anchor) */
static TrigTable *trig_table = NULL; /* table of (co)sine values */
static bool def_toolbars = true; /* default toolbar show state */
static int def_scale = Scale_Default; /* default percentage scale */
static int *persp_table = NULL; /* table of reciprocal values for perspective
                                   projection (flex anchor) */
```

With my redefinition of `NULL` as `((_Optional void *)0)`, the compiler correctly diagnosed a constraint violation in declarations such as `static void *col_trans_table = NULL`.

I chose to qualify the referenced type of such pointers as `_Optional` instead of removing their explicit `NULL` initializers because I wanted the additional rigour that `_Optional` brings. As before, the `NULL` initializers are strictly redundant, but I kept them for clarity:

```
static bool translate_cols = true;
static _Optional void *col_trans_table = NULL; /* table of colour numbers for
                                                  drawing sprite in desktop */
static _Optional TrigTable *trig_table = NULL; /* table of (co)sine values */
static bool def_toolbars = true; /* default toolbar show state */
static int def_scale = Scale_Default; /* default percentage scale */
static _Optional int *persp_table = NULL; /* table of reciprocal values for
                                             perspective projection */
```

The `trig_table` and `persp_table` pointers are only null until `Preview_initialise` is called on start-up; if `Preview_initialise` fails then the program exits. However, there is no way to indicate that global pointers cannot be null after being assigned a value, therefore every function that uses such pointers must assume they could be null.

The [following function](#) uses `trig_table` to rotate coordinates:

```
static void cam_rotate(Point3D *const p, int const x_angle, int const y_angle)
{
  // ...
  int const x_in = p->x;
  int y_in = p->y;
  int const z_in = p->z;

  /* Apply X rotation */
  int cos = TrigTable_look_up_cosine(trig_table, x_angle),
      sin = TrigTable_look_up_sine(trig_table, x_angle);

  p->x = (x_in * cos) / (SineMultiplier / PostRotateScaler) -
         (y_in * sin) / (SineMultiplier / PostRotateScaler);

  y_in = (x_in * sin) / SineMultiplier +
         (y_in * cos) / SineMultiplier;

  /* Apply Y rotation */
  cos = TrigTable_look_up_cosine(trig_table, y_angle);
  sin = TrigTable_look_up_sine(trig_table, y_angle);
  // ...
```

The compiler diagnoses constraint violations when compiling this function with the modified declaration of `trig_table`, because the possibly-null pointer is passed to `TrigTable_look_up_cosine` and `TrigTable_look_up_sine`:

```
Preview.c:204:38: warning: passing '_Optional TrigTable *' (aka '_Optional struct
TrigTable *') to parameter of type 'const TrigTable *' (aka 'const struct TrigTable
*') discards qualifiers [-Wincompatible-pointer-types-discards-qualifiers]
  204 |   int cos = TrigTable_look_up_cosine(trig_table, x_angle),
      |                                      ^~~~~~~~~~
/work/CBUtilLib/TrigTable.h:60:47: note: passing argument to parameter 'table' here
   60 | int TrigTable_look_up_cosine(const TrigTable *table, int angle);
      |                                               ^
```

A naive solution would be to add a defensive check for `trig_table` being null upon entry to the function and then substitute `&*trig_table` to remove `_Optional` from the referenced type of the pointer, just as one would for an argument:

```
static void cam_rotate(Point3D *const p, int const x_angle, int const y_angle)
{
  // ...
  int const x_in = p->x;
  int y_in = p->y;
  int const z_in = p->z;

  if (!trig_table) return;

  /* Apply X rotation */
  int cos = TrigTable_look_up_cosine(&*trig_table, x_angle),
      sin = TrigTable_look_up_sine(&*trig_table, x_angle);

  p->x = (x_in * cos) / (SineMultiplier / PostRotateScaler) -
         (y_in * sin) / (SineMultiplier / PostRotateScaler);

  y_in = (x_in * sin) / SineMultiplier +
         (y_in * cos) / SineMultiplier;

  /* Apply Y rotation */
  cos = TrigTable_look_up_cosine(&*trig_table, y_angle);
  sin = TrigTable_look_up_sine(&*trig_table, y_angle);
  // ...
```

This prevents constraint violations, but the static analyser still reports a warning — and the explanation is subtle:

```
Preview.c:207:37: warning: Pointer to _Optional object is dereferenced without a
preceding check for null [optionality.OptionalityChecker]
  207 |       sin = TrigTable_look_up_sine(&*trig_table, x_angle);
      |                                    ^~~~~~~~~~~
1 warning generated.
```

Strangely, this warning relates to the second use of `trig_table`, not the first. It might not be immediately obvious why. The answer is that the analyser must assume that any function call might modify the value of global variables. So, from the point of view of the analyser, the call to `TrigTable_look_up_cosine` doesn't just use the value of `trig_table`; it also potentially invalidates it!

These issues are very hard for programmers to spot, because it is rarely clear whether a variable is global or local. The correct solution is to assign the value of a global variable to a local variable before checking whether it is null. The analyser knows the local variable can't change between uses, so the null check remains valid throughout the block.

This also allows the checked pointer to be given a more restrictive type, thereby all but eliminating use of the `&*` sigil (used to remove `_Optional`) and simplifying the programmer's mental model:

```
static void cam_rotate(Point3D *const p, int const x_angle, int const y_angle)
{
  // ...
  int const x_in = p->x;
  int y_in = p->y;
  int const z_in = p->z;

  if (!trig_table) {
    return;
  }
  const TrigTable *const tt = &*trig_table;

  /* Apply X rotation */
  int cos = TrigTable_look_up_cosine(tt, x_angle),
      sin = TrigTable_look_up_sine(tt, x_angle);

  p->x = (x_in * cos) / (SineMultiplier / PostRotateScaler) -
         (y_in * sin) / (SineMultiplier / PostRotateScaler);

  y_in = (x_in * sin) / SineMultiplier +
         (y_in * cos) / SineMultiplier;

  /* Apply Y rotation */
  cos = TrigTable_look_up_cosine(tt, y_angle);
  sin = TrigTable_look_up_sine(tt, y_angle);
  // ...
```

The preceding example is of the most trivial kind, in which it is 'obvious' that none of the callees can have modified the value of `trig_table`. In more complex cases, confirming that the value of a global variable does not change can require detailed analysis.

For instance, I had to convince myself that the `next_client` pointer of a round-robin scheduler could not be modified by any of the functions that it schedules. The payoff for this analysis was improved clarity: renaming the global variable as `global_next` and assigning its value to a local `next_client` made [the code](#) easier to reason about:

```
_Optional SchedulerClient *last_called = NULL, *next_client = global_next;
while (clients_count)
{
  if (next_client == NULL)
  {
    /* We have lost our place in the list, or reached the end */
    DEBUG_VERBOSEF("Scheduler: returning to head of client list\n");
    _Optional LinkedListItem *const head = linkedlist_get_head(&clients_list);
    if (head == NULL)
    {
      DEBUGF("Scheduler: client list is empty!\n");
      break; /* paranoia */
    }
    next_client = CONTAINER_OF(head, SchedulerClient, list_item);
  }
```

Once again, using `_Optional` had made my code less ambiguous. This pattern of assigning global state to a local variable can reduce both false positives in analysis and mental burden for readers and reviewers alike.

When pointers cannot be copied safely

Several of my programs use a shifting heap provided by a third-party library named Flex. It is built on the following `typedef`, which represents a pointer to an 'anchor' that both uniquely identifies a heap block and stores its current address:

```
typedef void **flex_ptr;
```

Hiding pointer types usually harms the clarity of code. The Linux kernel coding style guide goes further, by also arguing against hiding `struct` types using `typedef`. Nevertheless, both practices are common.

Type aliases that hide pointers have another drawback: the referenced type cannot easily be qualified as `const` or `_Optional`. For example, `const flex_ptr` means a constant pointer to a `void *` — not a pointer to a `const void *`, which may have been the intent.

`flex_ptr` is not an opaque type because no functions are provided to allocate an 'anchor' or get the address stored in it. Instead, users are expected to allocate their own 'anchor' and pass its address to functions that expect a `flex_ptr`. This results in weak type-safety, because the only type of pointer that matches without casting is `void *`.

Consider the following example. There's nothing in the type of the `records` member that signals its intended use, so a comment is needed to explain that its address is intended to be a `flex_ptr`:

```
typedef struct
{
  int num_cols;
  void *records; /* flex anchor */
}
ExpColFile;
```

Elsewhere in the program, whether `records` is null determines whether `flex_free` is called to free the associated heap block:

```
void ExpColFile_destroy(ExpColFile *const file)
{
  if (file->records)
  {
    flex_free(&file->records);
  }
}
```

Theoretically, this check would allow `ExpColFile_destroy` to be called after a failed call to `ExpColFile_init`, or twice for the same `ExpColFile`. This is because `flex_alloc` sets `records` to null if it fails (although undocumented), and `flex_free` sets `records` to null (as documented).

It might seem that the referenced type of `records` ought to be qualified as `_Optional` since it can be null. However, the program does not rely on that: if `flex_alloc` succeeds, then the value of `records` is assumed to be non-null; if `flex_alloc` fails or `flex_free` has been called, `records` is not used again. As such, the array pointed to by `records` is not really treated as optional.

In any case, it would be impossible to change the type of `records` to `_Optional void *` without requiring a cast whenever `&records` is passed to a function that expects `flex_ptr` (aka `void **`) instead of `_Optional void **`. (Some existing users already cast for a different reason, having judged that declaring 'anchors' with a specific type is preferable to using `void *` where there is no actual requirement for polymorphism.)

Conversely, redefining `flex_ptr` as `_Optional void **` would break all existing code that passes `void **`, since `void **` cannot be implicitly converted to that type. (In contrast, redefining `flex_ptr` as an alias for `void * _Optional *`would be fine, but null *arguments* are not what is needed.)

Nor would the enhanced type variance proposed by N3510 avoid incompatibilities: passing `void **` to a function that treats it as `_Optional void **` would permit the callee to set the caller's pointer to null — a transformation that must be assumed unsafe. (An implicit conversion to `_Optional void *const *` would be allowed, but that contradicts the actual behaviour of the library.)

Even if redefining `flex_ptr` as `_Optional void **` were practical, the semantics would not be a good fit for my program: casts (to remove the `_Optional` qualifier) or null checks would be required every time `records` is used. The usual way to avoid repetitive null checks is to assign an `_Optional void *` to a variable of unqualified type (in this case, `ExportColFileRecord *`) and use that instead. That specifically does not work with Flex, because it requires a unique pointer to each heap block — otherwise, it wouldn't know which pointer to update when shifting blocks.

Ultimately, I left the definition of `flex_ptr` —and my usage of it— unchanged. Not every pointer that can be null must be qualified as a pointer to `_Optional`. Still, the issues raised here should concern anyone designing a similar library: forbidding pointer copying severely limits the ability of the type system to enforce safety.

Array to pointer decay and its consequences

Many of my projects are RISC OS applications. RISC OS is a venerable operating system that represents errors using the following struct type:

```
typedef struct
{
  int errnum;
  char errmess[252];
} _kernel_oserror;
```

Conventionally, "no error" (i.e. success) is represented by a null pointer of type `_kernel_oserror *`.

A curious quirk of C is that use of the `errmess` member of this struct is —under most circumstances— indistinguishable from use of the `errmess` member of the following (hypothetical) struct:

```
typedef struct
{
  int errnum;
  char *errmess;
} _kernel_oserror;
```

This is because objects of array type 'decay' automatically into pointers to the first array element (with a few exceptions, such as when used as the operand of the address-of operator).

When compiling some obsolete code after an incomplete effort to add the `_Optional` qualifier to most uses of `_kernel_oserror`, I encountered an unexpected constraint violation:

```
Loader.c:866:40: warning: passing '_Optional char[256]' to parameter of type 'const
char *' discards qualifiers [-Wincompatible-pointer-types-discards-qualifiers]
  866 |            err_complain(errptr->errnum, errptr->errmess);
      |                                         ^~~~~~~~~~~~~~~
./Err.h:152:53: note: passing argument to parameter here
  152 | void err_complain(int /*num*/, const char * /*mess*/);   /* Cancel & OK
buttons */
      |                                                     ^
```

This was easily cured by using the `&*` idiom to remove the `_Optional` qualifier from the referenced type of `errptr->errmess`:

```
      _Optional _kernel_oserror *errptr;
      errptr=toolbox_get_object_class(0, object, &objclass);
      if (errptr != NULL) {
        if (errptr->errnum != ERR_BAD_OBJECT_ID)
          err_complain(errptr->errnum, &*errptr->errmess);
        return false; /* ignore listener if bad object ID */
      }
```

However, the resultant code looks noisy and puzzling — neither of which I intended when designing `_Optional`. What does a null check on `errptr` have to do with the nullability of the pointer `errptr->errmess`? Nothing — yet the code appears to treat them as related.

The `errmess` member of `_kernel_oserror` is treated as `_Optional char[252]` because the expression `errptr->errmess` is an lvalue derived from a pointer to an `_Optional` type. This array type 'decays' into a pointer to the first character of the error message, which is also `_Optional`. The `err_complain` function does not accept `_Optional char *` —only `char *`— therefore the qualifier must be removed again.

It makes sense that the error message is treated as optional, since it belongs to an optional object. However, it is not normally possible to construct a pointer to such a subobject without first removing the `_Optional` qualifier, because the address of an object is never null — only the pointer to it can be.

At the heart of this problem is the fact that one aspect of my design for `_Optional` is based on a false assumption:

> *There is only one way to get the address of an object (excepting arithmetic), whereas there are many ways to dereference a pointer.*

(N3422, _Optional: a type qualifier to indicate pointer nullability (v2))

Thanks to array-to-pointer 'decay', that is not true: an array type can 'decay' into the corresponding pointer type without explicit use of the unary `&` operator. This suggests that my design decision to remove the `_Optional` qualifier using the `&` operator instead of the dereferencing operators (`*`, `->` and `[]`) should be revisited.

_Optional really does find bugs

Pointer arithmetic and array indexing

I discovered several latent bugs in the following code:

```
        const SampleInfo * const sample = sf_samples->sample_info + sample_num;
        if ((sample_num >= sf_samples->count) ||
            (sample->type == SampleInfo_Type_Unused)) {
          printf("%d %d %d\n", sf_samples->count, sample_num, sample->type);
          fprintf(stderr, "Warning: Sample number %d is not defined!\n",
                          sample_num);
          continue;
        }
```

Let us assume that `sf_samples->sample_info` is an array of at least `sf_samples->count` elements, which prevents `(sample->type == SampleInfo_Type_Unused)` from being evaluated when `sample` is an invalid pointer.

Nevertheless:

1. The expression `sf_samples->sample_info + sample_num` has undefined behaviour if `sf_samples->sample_info` is null pointer regardless of the value of `sample_num`, according to section 6.5.7 of the ISO C23 standard. (The latest draft of C2Y has relaxed this rule, allowing zero-length operations on null pointers.)
2. `sf_samples->sample_info + sample_num` is using an array element index of unknown magnitude. This has undefined behaviour if `sample_num` is greater than the number of elements in the array.
3. The resultant indeterminate pointer is dereferenced by `printf("%d %d %d\n", sf_samples->count, sample_num, sample->type)`.

My first step was to qualify the referenced type of the `sample_info` member as `_Optional`:

```
typedef struct {
  int count;
  int alloc;
  _Optional PTSampleInfo *sample_info;
} PTSampleArray;
```

The compiler then produced a diagnostic of a constraint violation in the declaration that contains the first bug, without path-sensitive analysis:

```
protracker.c:925:34: warning: initializing 'const SampleInfo *const' with an
expression of type '_Optional SampleInfo *' discards qualifiers [-Wincompatible-
pointer-types-discards-qualifiers]
  925 |         const SampleInfo * const sample = sf_samples->sample_info +
sample_num;
      |                                   ^
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Although the diagnostic message does not pertain to the use of the additive operator, it seems unlikely that `sf_samples->sample_info + sample_num` would be evaluated without assigning the result to an argument or variable.

The simplistic fix of adding an `_Optional` qualifier to the declaration specifiers of `sample` (to avoid the constraint violation) is wrong, but a diagnostic message is only produced when the amended code is subjected to path-sensitive analysis:

```
protracker.c:925:77: warning: Pointer to _Optional object is dereferenced without a
preceding check for null [optionality.OptionalityChecker]
  925 |         _Optional const SampleInfo * const sample = sf_samples->sample_info
+ sample_num;
      |
~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~
```

Had the same declaration been written as

```
const SampleInfo * const sample = &sf_samples->sample_info[sample_num];
```

then no constraint would have been violated, whereas the static analyser produces a diagnosis for both variants:

```
protracker.c:925:44: warning: Pointer to _Optional object is dereferenced without a
preceding check for null [optionality.OptionalityChecker]
  925 |         const SampleInfo * const sample =
&sf_samples->sample_info[sample_num];
      |
^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
```

and

```
protracker.c:925:67: warning: Pointer to _Optional object is dereferenced without a
preceding check for null [optionality.OptionalityChecker]
  925 |         const SampleInfo * const sample = sf_samples->sample_info +
sample_num;
      |
~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~
```

It is not clear to me that static analysers will continue to be able to diagnose arithmetic on null pointers, since N3322was accepted by WG14. Thus, acceptance of a corner case undermines detection of the common case.

My initial attempted solution satisfied both the compiler and the static analyser, but was incomplete:

```
        _Optional const SampleInfo * const sample = sf_samples->sample_info ?
          &sf_samples->sample_info[sample_num] :
          NULL;

        if (!sample ||
            (sample_num >= sf_samples->count) ||
            (sample->type == SampleInfo_Type_Unused)) {
          printf("%d %d %d\n", sf_samples->count, sample_num,
                 sample ? sample->type : SampleInfo_Type_Unused);
          fprintf(stderr, "Warning: Sample number %d is not defined!\n",
                  sample_num);
          continue;
        }
```

The first bug (arithmetic on a null pointer) is fixed, but the second (possible out-of-bounds access) and third (dereferencing an indeterminate pointer) still exist. The `_Optional` qualifier cannot solve array out-of-bounds issues like this, although it may coincidentally draw attention to them.

My eventual solution was to check the value of `sf_samples->sample_info` and `sf_samples->count` (separately, instead of relying on `count` to be 0 when `sample_info` is null) before computing the address of the relevant `SampleInfo` element:

```
        _Optional const SampleInfo * const sample =
          sf_samples->sample_info && sample_num < sf_samples->count ?
            &sf_samples->sample_info[sample_num] :
            NULL;

        if (!sample || (sample->type == SampleInfo_Type_Unused)) {
          printf("%d %d %d\n", sf_samples->count, sample_num,
                 sample ? sample->type : SampleInfo_Type_Unused);
          fprintf(stderr, "Warning: Sample number %d is not defined!\n",
                           sample_num);
          continue;
        }
```

Checking fopen return values: a missed branch

Another bug that I found was in the following code, which passed a possibly-null pointer to a `FILE` (the value of `out`) into the `fcopy` function. If `fopen` failed, then its caller set a variable named `success` to false but subsequent code did not check the value of that variable before calling `fcopy`:

```
    if (success) {
      if (output_file != NULL) {
        /* Open the real output file */
        if (verbose)
          printf("Opening output file '%s'\n", output_file);

        out = fopen(input_file, "wb");
        if (out == NULL) {
          fprintf(stderr,
                  "Failed to open output file: %s\n",
                  strerror(errno));
          success = false;
        }
      } else {
        /* Default output is to standard output stream */
        out = stdout;
      }

      if (verbose)
        puts("Copying from temporary to final output");

      if (fseek(tmp, 0L, SEEK_SET)) {
        fprintf(stderr, "Failed to seek start of temporary file\n");
        success = false;
      } else if (!fcopy(tmp, out)) {
        success = false;
      }
    }
```

When I changed the type of `out` from `FILE *` to `_Optional FILE *` (for compatibility with the return type of my hidden shim, `optional_fopen`), the compiler correctly reported a constraint violation:

```
gkcommon.c:184:32: warning: passing '_Optional FILE *' (aka '_Optional struct
_IO_FILE *') to parameter of type 'FILE *' (aka 'struct _IO_FILE *') discards
qualifiers [-Wincompatible-pointer-types-discards-qualifiers]
 184 | } else if (!fcopy(&*tmp, out)) {
     |                          ^~~
 gkcommon.c:48:35: note: passing argument to parameter 'out' here
 48 | static bool fcopy(FILE *in, FILE *out)
    |                                   ^
```

(The `&*tmp` idiom removes `_Optional` from the referenced type safely, without requiring a cast; so far, this has only been applied to one of the arguments to `fcopy`.)

Having to *explicitly* remove `_Optional` from referenced types encourages the programmer to think about whether it is safe to do so. Thus, I noticed the flow-control bug when changing `out` to `&*out` in the argument list of the call to `fcopy`, but I deliberately left the bug in the modified code to verify that the path-sensitive analyser also found it:

```
gkcommon.c:184:33: warning: Pointer to _Optional object is dereferenced without a
preceding check for null [optionality.OptionalityChecker]
 184 | } else if (!fcopy(&*tmp, &*out)) {
     |                           ^~~~
 1 warning generated.
```

This illustrates that `_Optional` provides value to a careful programmer even without path-sensitive analysis, although such tools are still useful to validate the final version of some code.

The real solution was to make the call to `fcopy` conditional on the value of `success` (although it would arguably have been more straightforward to check the value of `out`):

```
    if (success) {
      if (output_file != NULL) {
        /* Open the real output file */
        if (verbose)
          printf("Opening output file '%s'\n", output_file);

        out = fopen(&*input_file, "wb");
        if (out == NULL) {
          fprintf(stderr,
                  "Failed to open output file: %s\n",
                  strerror(errno));
          success = false;
        }
      } else {
        /* Default output is to standard output stream */
        out = stdout;
      }
    }

    if (success) {
      if (verbose)
        puts("Copying from temporary to final output");

      if (fseek(&*tmp, 0L, SEEK_SET)) {
        fprintf(stderr, "Failed to seek start of temporary file\n");
        success = false;
      } else if (!fcopy(&*tmp, &*out)) {
        success = false;
      }
    }
```

With this fix, all paths that could pass a potentially null value to a non-nullable parameter are guarded by explicit checks.

Conclusion

`_Optional` is not magic. It doesn't solve array bounds checking or replace the need for careful control flow, but it does makes problems visible earlier and forces programmers to reflect on their code.

Dogfooding also found a bug in my fork of Clang

Years ago, I designed a 3D object library to use pointer-to-array types throughout, as an experiment in unusual use of C's type system. That made it perfect for finding a bug in my fork of Clang: The code that I had written to remove the `_Optional` qualifier from the type of the operand of the address-of operator did not remove the qualifier from array types, resulting in incorrect qualifier mismatch diagnostics.

For example:

```
void fred(int (*x)[10]);

void jim(void)
{
  _Optional int (*y)[10];
  fred(&*y); // warning: passing '_Optional int (*)[10]' to parameter of type 'int
(*)[10]' discards qualifiers
}
```

Similar spurious messages prevented me from doing a thorough job of updating my 3D object library. In the following example, the `vector_x` and `vector_y` functions both require the address of an array of coordinates, but only the first call has been updated to explicitly remove the `_Optional` qualifier from the referenced type of `point`:

```
  _Optional Coord (* const point)[3] = vertex_array_get_coords(varray, v);
  if (!point) {
    return false;
  }
  const Coord px = *vector_x(&*point, plane);
  Coord py = *vector_y(point, plane);
```

I don't think this is indicative of any long-term problem though, because using `point` instead of `&*point` passes a pointer to an `_Optional`-qualified referenced type to a function that cannot accept it — a constraint violation that must be diagnosed.

I have since pushed a fix to my fork of Clang (+23 -1 lines changed). Matt Godbolt has also kindly updated Compiler Explorer to include my fix. Since the wording that I proposed in my most recent paper, N3422, did not include any example usage of arrays, I will ensure that the next version does so.

This bug illustrates the importance of exercising even the more obscure corners of the type system.

In conclusion

If you have the time and inclination, then please experiment with using `_Optional` in your own projects. It's relatively low risk: the qualifier can easily be deleted again with `grep` and `sed`, as can the `&*` pattern used to remove it from referenced types. In particular, I am addressing those who support standardisation of `_Optional`. I'd rather that support for such an important new feature of the language be based on real-world usage rather than purely on trust.

Acknowledgements

Alex Celeste for suggesting the `optional_cast` macro (by a different name).