Proposal for C2x

WG14 N3089

Title: _Optional: a type qualifier to indicate pointer nullability

Author, affiliation: Christopher Bazley, Arm

Date: 2023-01-26

Proposal category: New features

Target audience: General Developers, Compiler/Tooling Developers

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

Prior art: MyPy, C

_Optional: a type qualifier to indicate pointer nullability

Reply-to: Christopher Bazley (chris.bazley@arm.com)
Document No: N3089
Date: 2020-02-10

Summary of Changes

N3089

• Initial proposal

Philosophical underpinning

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

People criticise C's syntax, but I consider it the foundation of the language. Any experienced C programmer has already acquired the mindset necessary to read and write code using it. Aside from the need to minimize incompatibilities, the syntactic aberrations introduced by C++ can be ignored.

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

Inspiration from Python

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

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

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

Problem statement

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

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

Isn't this a solved problem?

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

C99 extended the syntax for function arguments to allow static within `[]`, which requires the passed array to be at least a specified size:

```
void *my_memcpy(char dest[static 1], const char src[static 1],
size_t len);
void test(void)
{
    char *dest = NULL, *src = NULL;
    my_memcpy(NULL, NULL, 10); // warning: argument 1 to
'char[static 1]' is null where non-null expected
    my_memcpy(dest, src, 10); // no compiler warning
}
```

This trick may generate a warning in cases where a null pointer constant is specified directly as a function argument - but not for any other source of null such as a failed call to `malloc`. It's not a general-purpose solution anyway because arrays of type `void` are illegal, which makes this syntax unusable for declaring functions such as `memcpy`.

A GNU compiler extension [2] (also supported by Clang and the ARM compiler [3]) allows function arguments to be marked as not supposed to be null:

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

I find the `__attribute__` syntax intrusive and verbose. It is also error-prone because attributes only apply to function declarations as a whole: it's easy to accidentally specify wrong argument indices, because the arguments themselves are not annotated. Clang extended the syntax to allow `__attribute__((nonnull))` to be used within an argument list, but the GNU compiler does not support that.

Historically, the semantics of `__attribute__((nonnull))` weren't very useful: it only detected cases where a null pointer constant was specified directly as a function argument. However, version 10 of the GNU compiler introduced a new feature [4], `-fanalyzer`, which uses the same `__attribute__` information during a static analysis pass:

```
<source>:8:3: warning: use of NULL 'dest' where non-null expected
[CWE-476] [-Wanalyzer-null-argument]
    8 |   my_memcpy(dest, src, 10); // no compiler warning
      |   ^~~~~~~~~~~~~~~~~~~~~~~~
  'test': events 1-3
    |
    |    7 |   char *dest = NULL, *src = NULL;
    |      |         ^~~~          ~~~
    |      |         |             |
    |      |         |             (2) 'dest' is NULL
    |      |         (1) 'dest' is NULL
    |    8 |   my_memcpy(dest, src, 10); // no compiler warning
    |      |   ~~~~~~~~~~~~~~~~~~~~~~~~
    |      |   |
    |      |   (3) argument 1 ('dest') NULL where non-null expected
    |
<source>:4:7: note: argument 1 of 'my_memcpy' must be non-null
    4 | void *my_memcpy(void *dest, const void *src, size_t len)
__attribute__((nonnull (1, 2)));
      |       ^~~~~~~~~
```

RFC: Nullability qualifiers (2015) [5] proposed not one but **three** new type annotations: `_Nullable`, `_Nonnull` and `_Null_unspecified`. Support for these was added in version 3.7 of Clang [6], but GCC doesn't recognize them. Like GCC without `-fanalyzer`, Clang itself only detects cases where a null pointer constant is specified directly as a function argument:

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

However, Clang-tidy [7], a standalone tool based on Clang, **can** issue warnings about misuse of pointers that it is able to infer based on annotations and path-sensitive analysis:

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

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

I've seen Clang's nullability qualifiers described as "enormous and useless noise, while providing doubtful value" [8] and the very idea of annotating pointers called a "naive dream". I agree with the first statement, but not the second: other languages have shown that null safety is achievable and useful, whilst C lags with competing partial solutions that are confusing, error-prone, and intrusive.

Clang's annotations provide no value to other compilers, which either ignore (if removed by macros) or reject them. Not all developers to use special build machines costing thousands of pounds: I do a lot of coding on a Raspberry Pi, using a toolchain that dates from the 1980s [9] but is still actively maintained [10] (and recently gained support for C17). For me, having a rapid *edit-compile-run* cycle is paramount.

Even tiny compilers such as cc65 [11] can check that the addresses of objects declared with `const` or `volatile` are not passed to functions that do not accept such pointers, because the rules for type compatibility are simple (for the benefit of machines and people). This is exactly the niche that the C language should be occupying.

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

Syntactic and semantic precedents

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

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

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

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

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

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

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

It's therefore necessary to specify whether or not null is permitted for **every** level of indirection within a *declarator* (e.g. for both `baz[3]` and `(*baz[3])[2])`. The only existing element of C's existing syntax that has such flexibility is a *type-qualifier*.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
void myfunc(const char s[_Nullable]); // s may be a null pointer
```

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

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

Proposed syntax

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

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

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

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

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

```
const int *i; // *i is an int that may be stored in read-only memory
volatile int *j; // *j is an int that may be stored in shared memory
_Optional int *k; // *k is an int for which no storage may be
allocated
```

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

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

Modifying a `const` object only has undefined behaviour if the object was **originally** declared as `const`, which is not always the case when an object is modified by dereferencing a pointer from which a `const` qualifier was cast away. Likewise, accessing an `_Optional` object will only have undefined behaviour if the pointer used to access the object is **actually** null.

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

Unlike assignment to a variable with `const`-qualified type, no error should be reported when compiling code which accesses a variable with `_Optional`-qualified type. Were that my intent, I would have proposed a name like `_None` rather than `_Optional`. Requiring the `_Optional` qualifier to be cast away before accessing a so-qualified object would be tiresome and would sacrifice type safety for null safety. I do not think that is a good trade-off.

Despite this limitation, the new qualifier is useful:

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

Here is some example usage:

```
void foo(int *);

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

    if (i) {
        *i = 5; // okay
    }

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

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

```
   int            bar;

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

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

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

Let's break it down:

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

Note that an 'optional pointer' is **not** a pointer that may have the value null; it's a pointer that may not exist. This is like the existing rule that a 'const pointer' is not a pointer to read-only memory; it's a pointer that may be **stored in** read-only memory.

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

```
void myfunc(_Optional const char s[]); // s may be a null pointer
```

With the above exception, it isn't useful to declare a non-pointed-to object as `_Optional` (although so-qualified types will exist during expression evaluation). Such declarations could be disallowed, like similar abuse of `restrict`, to avoid confusion.

Conversions from maybe-null to not-null

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

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

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

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

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

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

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

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

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

One of my colleagues suggested just such a solution:

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

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

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

The way I envisage this working is:

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

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

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

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

Modification of operator semantics

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

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

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

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

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

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

Moreover, many previously simple expressions became unreadable:

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

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

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

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

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

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

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

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

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

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

Function pointers

C's declaration syntax does not permit type qualifiers to be specified as part of a function declaration:

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

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

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

That doesn't solve the underlying problem, though. GCC does not warn about such declarations, but Clang does:

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

The C language standard currently says:

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

Making this behaviour well-defined (as in C++) would make the language safer, whereas extending the declaration syntax is beyond the scope of my proposal.

Migration of existing code

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

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

Requiring implementations to redefine the constant to which the `NULL` macro expands as `((_Optional void *)0)` is unthinkable because it would invalidate all existing code. However, it might be useful to standardize an alternative macro for use in place of `NULL`. I have not specified such a macro because `NULL` is not a core part of the language.

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

Before

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

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

After

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

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

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

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

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

Proposed language extension

- A new type qualifier, `_Optional`, indicates that a pointer to a so-qualified type may be null. This does not preclude any other pointer type from being null.
- Types other than those of a pointed-to object or pointed-to incomplete type shall not be `_Optional`-qualified in a declaration.
- The semantics of the unary `&` operator are modified so that if its operand has type "*type*" then its result has type "pointer to *type*", with the omission of any `_Optional` qualifier of the pointed-to type.
- If an operand is a pointer to an `_Optional`-qualified type and its value cannot be statically proven never to be null, then implementations may generate a warning of any undefined behaviour that would occur if the value were null.
- A specification of a function type that includes type qualifiers no longer has undefined behaviour. Qualifiers that are not applicable are ignored (as in C++).

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

Considerations for static analysis

Clang's static analyser currently ignores many instances of undefined behaviour. For example, it allows expressions like `&self->super` when `self` is null. This latitude is also required because many commonly used macros such as `offsetof` and `container_of` have undefined behaviour. The simplest definition of `offsetof` incorporates an explicit null pointer dereference:

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

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

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

Possible objections

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

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

Others may agree with Stroustrup [12] that C's syntax and semantics are a "known mess" of "perversities". This paper argues that pointer nullability should be added in a way that conforms to long-established C language idioms rather than violating such norms (as C++ references do) in the hope of satisfying users who will never like C anyway.

Implementations

Currently only a working prototype [13] of the required changes to Clang and Clang-tidy exists. Integration of this prototype into mainline LLVM will require code review and consent from the project maintainers.

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

Acknowledgements

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

References

[0] mypy 0.991 documentation, https://medium.com/r/?url=https%3A%2F%2Fmypy.readthedocs.io%2Fen%2Fstable%2F

[1] TypeScript: JavaScript With Syntax For Types, https://medium.com/r/?url=https%3A%2F%2Fwww.typescriptlang.org%2F

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

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

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

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

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

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

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

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

[10] RISC OS Open: Desktop Development Environment, https://medium.com/r/?url=https%3A%2F%2Fwww.riscosopen.org%2Fcontent%2Fsales%2Fdde

[11] cc65 - a freeware C compiler for 6502 based systems, https://medium.com/r/?url=https%3A%2F%2Fcc65.github.io%2F

[12] Stroustrup, “The Design and Evolution of C++” (1994)

[13] [RFC] _Optional: a type qualifier to indicate pointer nullability – Clang Frontend – LLVM Discussion Forums, https://discourse.llvm.org/t/rfc-optional-a-type-qualifier-to-indicate-pointer-nullability/