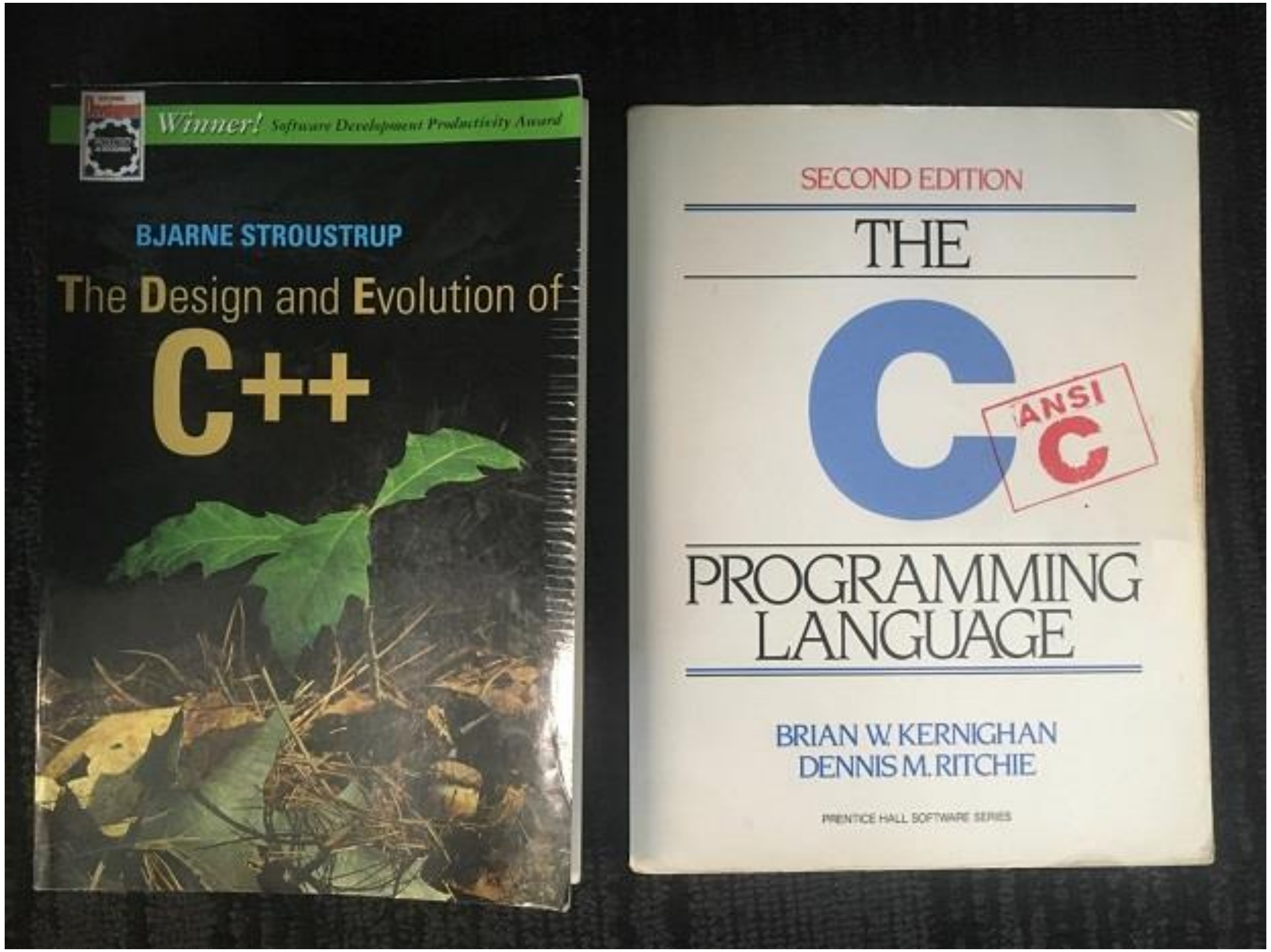
_Optional

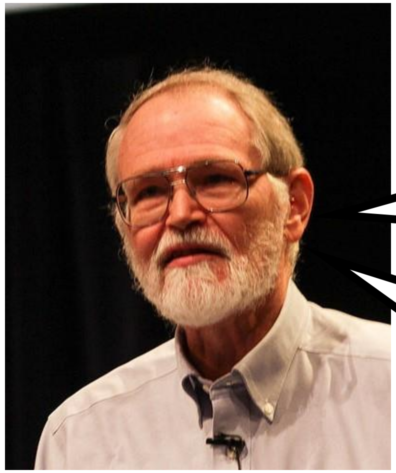
A type qualifier to indicate pointer nullability

Christopher Bazley
24th January 2024

Source of quotes



C has been at peace with itself for a long time



In our experience, C has proven to be a pleasant, expressive, and versatile language for a wide variety of programs. It is easy to learn, and it wears well as one's experience with it grows.

Since C is relatively small, it can be described in a small space, and learned quickly. A programmer can reasonably expect to know and understand and indeed regularly use the entire language.

What do C programmers waste time on?

- Repetitive, longwinded, and unverifiable parameter descriptions:

```
 * @param[in]  wmp           The active WOMUMP context.
 * @param[out] export_list   Non-NULL pointer to a non-NULL pointer to a valid export list.
 */
void womump_context_get_deferred_exports(womump_context *wmp,
                                         uint64_t **export_list);
```

- Assertions to check pointer parameters:

```
void womump_sync_to_cpu(womump_context          *wmp,
                        const womump_mapping    *mapping,
                        void                    *address,
                        const size_t             size)
{
        assert(wmp);
        assert(mapping);
        assert(mapping->hunk);
        assert(address);
```

- Negative tests to verify such assertions:

```
        test_framework_expect_abort();
        womump_sync_to_cpu(wmp, mapping, NULL, 10);
```

Isn’t this a solved problem?

C99 allows `static` within `[]`, which requires a passed array to be at least a given 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 from Clang
}
```

- GCC generates a warning based on path-sensitive analysis (with `-fanalyzer`).
- Clang only generates a warning if a null pointer constant is specified directly.
- Not usable for functions like `memcpy` because arrays of `void` are illegal.
- Not usable for local variables or return values.
- (Arguably) not pleasant or expressive.

Isn’t this a solved problem? (2)

A GCC extension allows parameters to be marked as non-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); // warning: use of NULL 'dest' where non-null expected
}
```

- Generates a warning based on path-sensitive analysis (with `-fanalyzer`).
- Easy to accidentally specify wrong parameter indices.
- Not usable for local variables or return values.
- Not standard C, although also supported by Clang.
- (Arguably) not pleasant or expressive.

Isn’t this a solved problem? (3)

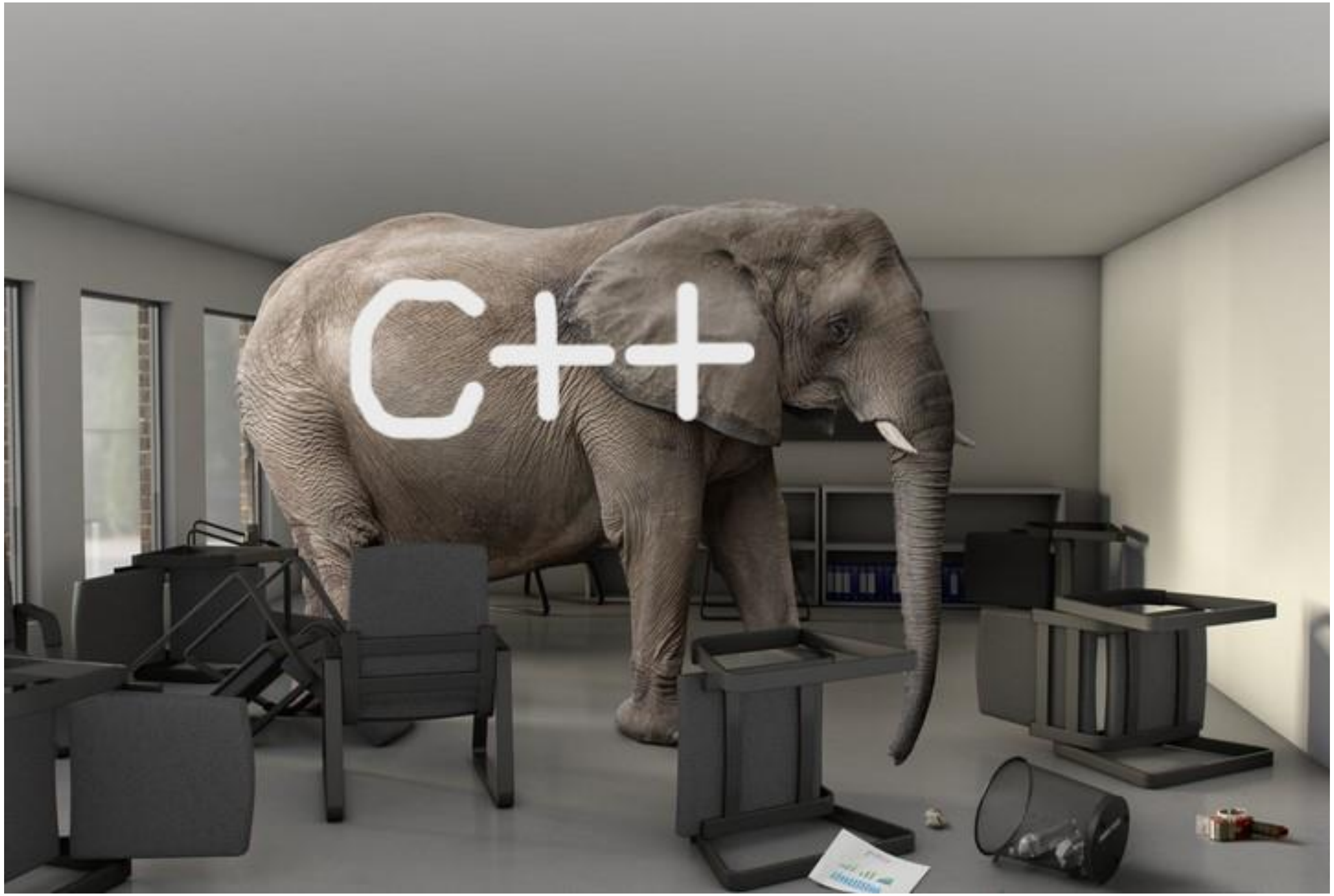
A Clang extension allows `_Nullable`, `_Nonnull` and `_Null_unspecified`:

```
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 argument
  my_memcpy(dest, src, 10); // no warning
}
```

- Generates warnings based on path-sensitive analysis (with `--analyze` or clang-tidy).
- Less verbose and error-prone than the `__attribute__` syntax.
- Usable for local variables and return values.
- Neither standard C, nor supported by GCC.

The elephant in the room



C++ references

- A reference cannot refer to a different object after being initialized.

  Like this pointer:
  ```
  int *const x = &y; // x can only point to y
  ```

- A reference cannot refer to a dereferenced null pointer.

  Like this parameter:
  ```
  void foo(int x[static 1]); // x can't be null
  ```

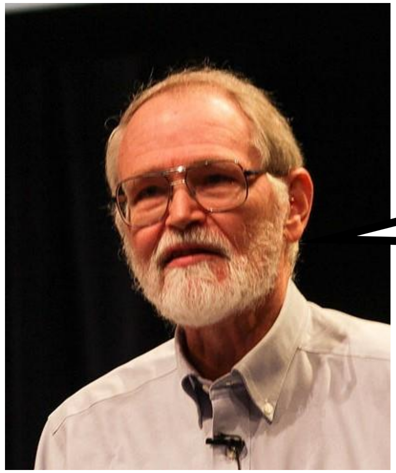
- A reference has the same syntax as an object in expressions.

  Stroustrup thought this desirable to support operator overloading.

- A reference is created implicitly without the address-of operator `&`.

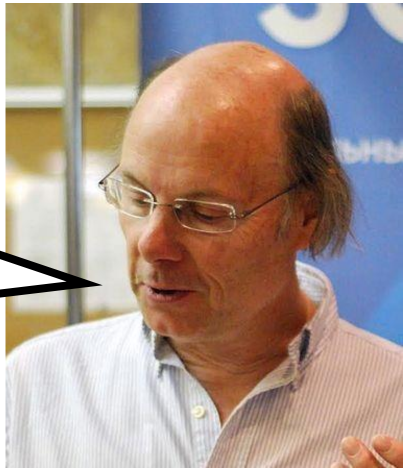
  Call-by-reference is indistinguishable from call-by-value.

C and C++ syntax are irreconcilable



`int *ip` is intended as a mnemonic; it says that the expression `*ip` is an `int`. The syntax of the declaration for a variable mimics the syntax of expressions in which the variable might appear.

The C trick of having the declaration of a name mimic its use leads to declarations that are hard to read and write, and maximises the opportunity for humans and programs to confuse declarations and expressions.



C and C++ syntax are irreconcilable (2)

Consider the following C declaration:

```
int a, // 'a' has type 'int'
    *b, // dereferencing pointer 'b' yields 'int'
    c[3], // elements of array 'c' have type 'int'
    d(float), // value returned by function 'd' has type 'int'
    *e(float); // dereferencing return value of 'e' yields 'int'
```

Now consider the C++ syntax for references:

```
int &f = a, // address of 'f' has type 'int’ ?!?
    *&g = b; // 'g' has type 'int’ ?!?
```

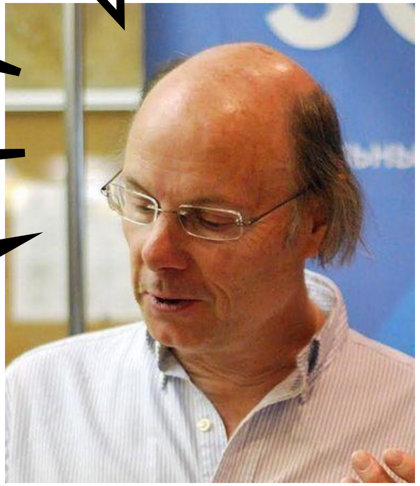
Bjarne Stroustrup kind of hates C

Dealing with **stubborn** old-time C users, **would-be** C experts, and genuine C/C++ compatibility issues has been one of the most **difficult** and **frustrating** aspects of developing C++.

Non-C programmers usually underestimate the value that C programmers attribute to the C syntax.

The **agony** to me and other implementers, documenters, and tool builders caused by the **perversities** of syntax has been significant.

Any new syntax would add complexity to a **known mess**.



A thought experiment

What would references look like if designed by someone who *likes* C?

(The answer isn't `int ip[static const 1]`)

Inspiration from Python

C is Guido's favourite language (after Python).

Python is dynamically typed with annotations.

Mypy is a static type checker for Python.

It makes a strong distinction between values that can be `None` and values that cannot.

```
from typing import Optional

def foo(n: int) -> int:
    return n

def bar(n: Optional[int]) -> int:
    return 0 if n is None else n

foo(None) # error: Argument 1 to "foo" has incompatible type "None"; expected "int"  [arg-type]
bar(None)
```

Coming soon* to C programming

```
_Optional int *ip;
```

A new type qualifier for the purpose of adding pointer nullability information to C programs.

- **Familiar and ergonomic syntax and semantics.**
- **Uses existing type-compatibility rules.**
- **Also useful for path-sensitive analysis.**
- **Makes code self-documenting.**
- **Reduces need for assertions.**
- **Reduces need for negative testing.**

* Subject to approval of paper N3089 by the IST/5/-/24 committee for the C programming language

Proposed C language extension

- `_Optional` indicates that a pointer to a so-qualified type may be null.
- `_Optional` is treated like `const` and `volatile` for lvalue conversion and when determining type compatibility.
- If an operand is a pointer to `_Optional` and its value cannot be proven to be non-null, implementations may generate a warning *as if it were null*.
- Unary `&` is modified to remove any `_Optional` qualifier from its operand.
- Only a pointed-to object or incomplete type may be `_Optional`-qualified in a declaration.

Example usage

```
void foo(int *);
void bar(_Optional int *i)
{
  *i = 10; // path-sensitive 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

  foo(&*i); // path-sensitive warning of unguarded dereference
  foo(&i[10]); // path-sensitive warning of unguarded dereference

  if (i) {
    foo(&*i); // okay
    foo(&i[10]); // okay
  }
}
```

Comparison to Clang’s syntax

```
   int            barley;
// ^ds            ^decl^

   int *_Nullable food[2] = {NULL, &barley};
//     ^^pointer^ ^ddecl^
// ^ds ^^^^declarator^^^

   int *_Nullable (*giraffe[3])[2] = {&food, &food, &food};
//     ^^pointer^ ^^^direct-decl^^
// ^ds ^^^^^^^^declarator^^^^^^^^^

   int *_Nullable (*_Nullable monkey[3])[2] = {&food, NULL, NULL};
//                 ^^pointer^ ^^ddecl^^
//                 ^^^^^^declarator^^^^
//                ^^^direct-declarator^^
//     ^^pointer^ ^^^^^direct-declarator^^^
// ^ds ^^^^^^^^^^^^^declarator^^^^^^^^^^^^^
```

Comparison to Clang’s syntax (2)

```
   int              barley;
// ^ds              ^decl^

   _Optional int *food[2] = {NULL, &barley};
//                ^ddecl^
// ^^decl-spec^^ ^^decl^^

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

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

Why qualify the pointed-to object?

Qualifiers on a pointer **target** must be compatible on assignment, whereas qualifiers on a pointer **value** are discarded.

```
int *const x = getptr();
int *s = x; // no warning
int *volatile y = getptr();
int *t = y; // no warning
int *restrict z = getptr();
int *r = z; // no warning
int *_Nullable v = getptr();
int *u = v; // no warning
```

Wait, what?!

- `_Nullable` isn't really a type qualifier.
- Clang's static analyser tracks whether a pointer may be null **regardless of its type**.
- Impossible to tell what constraints apply to a pointer by referring to its declaration.

Why qualify the pointed-to object? (2)

- Qualifiers on a pointer **target** must be compatible in function declarations, whereas qualifiers on a pointer **value** are ignored.
- Callers don't care what a callee does with its copy of parameter **values**.

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

- Clang ignores differences between rival declarations, except contradictory qualifiers (e.g. `_Nullable` vs `_Nonnull`).
- Impossible to tell what constraints apply to a function simply by referring to its declaration.

Why qualify the pointed-to object? (3)

- Properties conferred by `const`, `volatile`, `restrict` and `_Atomic` relate to how objects are **stored** or how that storage is **accessed**.
- No precedent for restricting the representable **values**.
- Read-only (`const`) objects may be stored in a separate address range so that illegal writes generate SIGSEGV.
- Null pointer values **also** encode a reserved address, typically neither readable nor writable.

```
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
```

Why qualify the pointed-to object? (4)

Clang allows nullability qualifiers to appear between `[]` brackets:

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

Unintuitive but follows 6.7.5.3 in the C language standard:

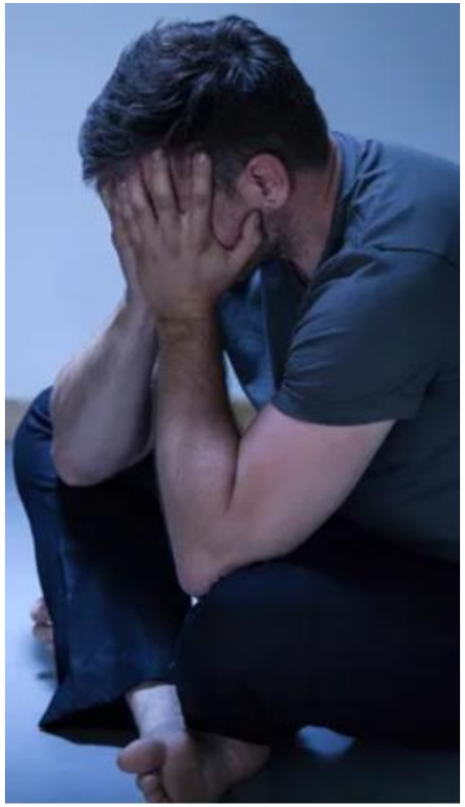
> *A declaration of a parameter as ''array of type'' shall be adjusted to ''qualified pointer to type'', where the type qualifiers (if any) are those specified within the [and] of the array type derivation.*

Declaration can be written more naturally with `_Optional`:

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

(only case where it's useful to declare a non-pointed-to object as `_Optional`)

Function pointers



C does not permit type qualifiers in function declarations:

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

A workaround is to use an intermediate `typedef` name:

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

Clang still complains (unlike GCC):

```
<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
^~~~~~
```

Why _Optional rather than _Mandatory?

Typical interface in a C program:

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

Should we change it to

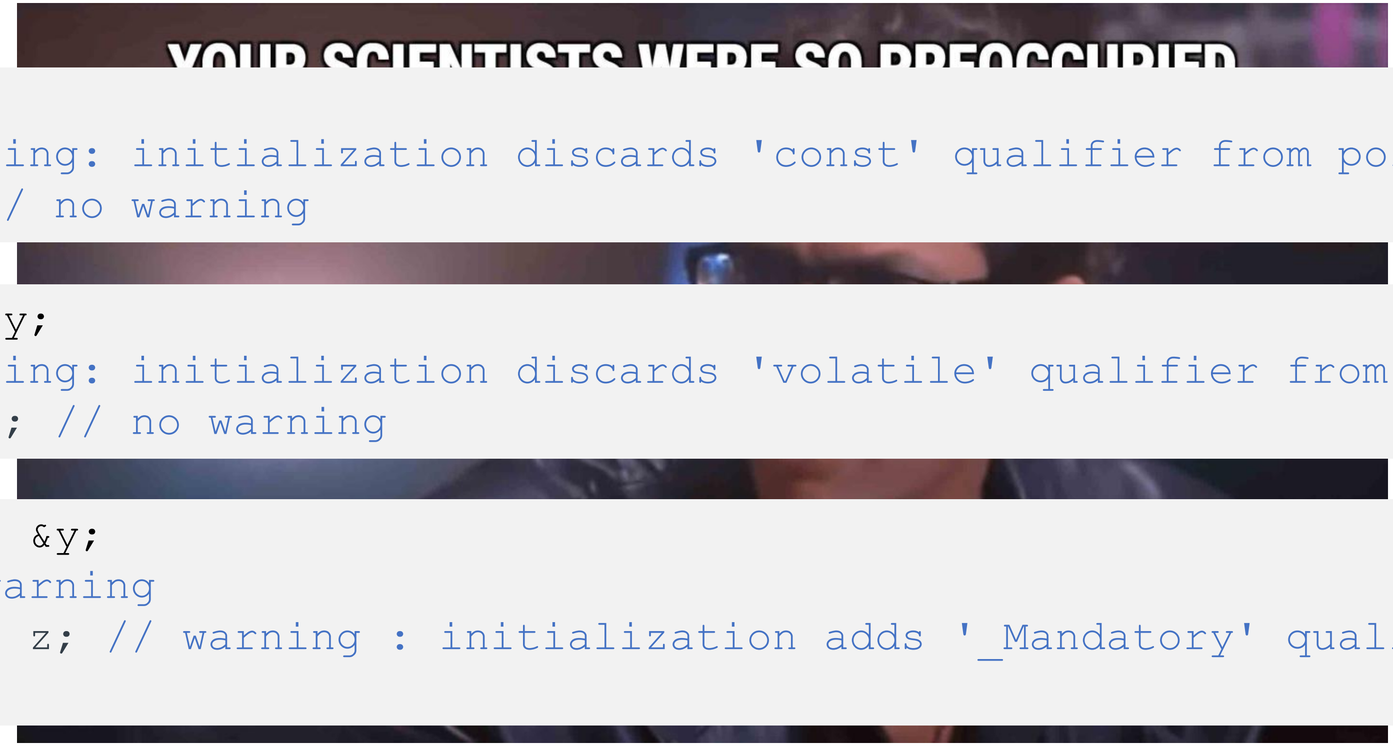
Or this?

What happened to

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

Why _Optional rather than _Mandatory? (2)

- `_Mandatory` is not a restriction on usage of a so-qualified pointer, therefore it should not be contagious.
- Assignment semantics for `_Mandatory` would need to be opposite (warn on acquire) to those for `const` and `volatile` (warn on discard).




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

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

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

Conversions from maybe-null to not-null

```c
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
}
```

Casting is detrimental to readability and type safety:
Safe to use an idiom to remove the qualifier:

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

Conversions from maybe-null to not-null (2)

- Don't want to rely on `#include` to import a conversion function or macro.
- C allows implicit conversions (e.g. from `void *`) where pragmatic.
- `&*s1` is searchable, easy to type, and not too ugly.
- Path-sensitive analysis can check `&*` like a real dereference.
- `optional_cast<char*>(s1)` or equivalent would be safer than a regular cast, but still clutter.
- Reserve casts as a fallback to suppress warnings.

Conversions from maybe-null to not-null (3)

There are many ways to dereference a pointer, but only one way to get the address of an object:

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

- Using `&` to remove `_Optional` from its operand avoids modifying the unary `*`, subscript `[]` **and** member-access `->` operators.
- It is also mnemonic: no object has null as its address.
- Operand of `&` is **already** special, being exempt from lvalue conversion and decay of a function or array into a pointer.

Migration

- `_Optional` can be pre-defined as an empty macro (like `const`).
- Programmers are free to eschew the new qualifier (like `const`).
- Functions which consume pointers can be changed to accept pointer-to-`_Optional`…
- …but not if used as a callback.
- Functions which return pointers can be wrapped or have their result assigned to a pointer to `_Optional`.

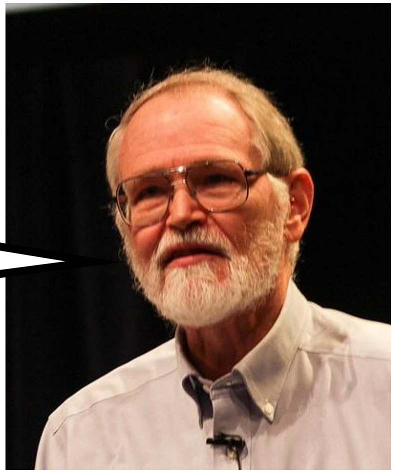
A successful thought experiment?

So, what *do* references look like if designed by someone who likes C?

Can you guess?

```
int *ip;
```

`int *ip` is intended as a mnemonic; it says that the expression `*ip` is an `int`.



Online reaction

"I hope you get N3089 through. You might like to try the C++ route first."

46 claps from seven people.

Score of 64 on Reddit.

"Great idea and I hope it gets itself in to a future standard, but I couldn't wait for something like that to arrive in C, which is why I preferred C++, or at least a sensible subset thereof."

Attributions

- Brian Kernighan (cropped) by Ben Lowe. Licence: CC BY 2.0 DEED.
- Elephant in the Room by mphillips007. Standard licence from iStock.
- Bjarne Stroustrup (cropped) by Julia Kryuchkova. Licence: CC BY-SA 2.5.
- Guido van Rossum (cropped) by Michael Cavotta. Licence: CC BY-NC-ND 4.0.
- Portrait of sorrowful man by Photographee.eu. Standard licence from Adobe Stock.
- Jurassic Park is © Universal City Studios LLC and Amblin Entertainment, Inc.
  The "Your Scientists" meme is believed to be "fair use" in editorial content.
- Flamingos in Ngorongoro Crater, Tanzania by Sburel. Royalty-free licence from Dreamstime.com.
- Reddit and Medium logos are © their respective owners.
  Their inclusion is believed to be "fair use" in editorial content.