Proposal for C2x
WG14 N3656

Title: Dependent Attributes
Date: July 24th, 2025
Authors: Yeoul Na (Apple), John McCall (Apple)
Proposal Category: New Features
Target Audience: Developers and compiler/tooling engineers working on memory safety features in C/C++

Abstract: This paper defines a new attribute category, called dependent attributes, that can express relationships between values. Dependent attributes are increasingly needed in C, particularly for memory safety extensions, static analysis, and security mitigations. However, C's current lookup rules make it difficult for developers to implement dependent attributes, due to the lack of support for forward references and for referencing fields within structures. This paper proposes solutions to these problems, enabling further progress with memory safety extensions, static analysis, and security mitigations for C that rely on dependent attributes.

Dependent Attributes

Reply-to: Yeoul Na and John McCall ({yeoul_na,rjmccall}@apple.com)
Document No: N3656
Date: July 24th, 2025

Introduction and rationale

There is an increasing need for annotations in C that can express various relationships between values. For example:

- A thread-safety attribute might need to express that a particular field should only be accessed while holding a lock held in another field.
- A bounds-safety attribute might need to express that a parameter points to an array whose length can be derived from another parameter.
- A lifetime-safety attribute might need to express that the lifetime of a parameter is derived from another parameter.

Attributes like these that semantically express an expected relationship at a high level are very powerful. A single attribute can be used for many different purposes, including static analysis, dynamic instrumentation, security mitigations, debugging improvements, and language interoperation.

Because these annotations express a sort of value dependency, akin to the value dependencies in dependent type systems, we will call this new category of attributes "dependent attributes".

The developers of dependent attributes generally want them to be easy to adopt in existing code. Programmers should be able to conveniently capture the relationships in an attribute without needing to make more complex structural changes to their source code. For example, a programmer might not have the option of re-ordering the parameters of a function to satisfy the limitations of the attribute: doing so might break the Application Boundary Interface (ABI), or it might just require large-scale changes across the codebase. Even when it is possible, programmers adopting these features are often trying to add many attributes at once, and needing to make correlated changes to the actual code for each attribute would greatly increase the cost of adoption. Making the attribute flexible enough to express relationships in existing source code is therefore critical to having an attribute that is usable in practice.

Unfortunately, C's lookup rules frequently cause problems for developers of dependent attributes. There are a few common needs that do not work with existing lookup rules:

- A dependent attribute in a function return type must be able to refer to the parameters of the function. In C, function return types must be written in a grammatical position where the parameters are not yet in scope. (C++11 introduced trailing return types specifically to address similar needs, although WG21 was mostly thinking about the decltype feature rather than dependent attributes.)
- A dependent attribute in a function parameter must also be able to refer to the parameters of the function. In C, function parameters come into scope immediately after the parameter declarator is complete, allowing references to earlier parameters, but forward- and self- references are not allowed. Some dependent attribute features are affected more by this problem than others. For example, a bounds safety feature is significantly affected by parameter order because it is a common pattern to pass the bounds of a pointer after the pointer; every function in the C standard library follows this pattern.
- A dependent attribute on a struct member must be able to refer to another member of the same struct. In C, struct fields are never in scope for ordinary lookup, and so all references to other members are impossible. (In C++, struct fields do come into scope immediately after their declarator is complete, but forward- and self-references are still impossible. To deal with this, C++ delays lookup in certain contexts until the enclosing struct is complete. Again, this was not done specifically with dependent attributes in mind, but it is convenient for implementors to take advantage of.)

All of these dependent attributes are language extensions, so implementors are not technically beholden to the C committee, and the C committee does not technically need to take a position on them. However, the lack of guidance from the committee has been causing problems for implementations. Discussions about accepting features into open-source compilers, or porting them to new compilers, have tended to get side-tracked by (necessary) design arguments about how best to express these dependent relationships given the constraints of C. This in turn tends to

keep these features siloed to research platforms or to individual vendors, impending the virtuous cycle of broader adoption.

The lack of guidance also inhibits any potential standardization of these dependent attribute features, as, again, any proposal introducing one would have to spend disproportionate effort on solving the general dependency expression problem rather than the specific application of that to the goals of the proposal.

The challenge is especially significant as the C language faces increasing pressure to address memory safety. In particular, dependent attributes will be essential for enabling memory safety modes in C, especially for large existing codebases. By "memory safety modes", we refer to a potential future in which C adopts optional modes or configurations that guarantee memory safety at compile time and/or run time [2]. It is not realistic to expect the vast body of existing code to be entirely rewritten or refactored. Instead of requiring code to be rewritten with new safe types, using attributes can preserve both source compatibility and ABI, enabling incremental adoption. This presents a practical path forward.

Additionally, because different implementation efforts come to different decisions about how far to take their solution to this problem, features are more likely to have inconsistent designs with different, often artificial limitations, rather than being able to take advantage of common infrastructure within the compiler, as they would if a standard feature needed to solve the problem.

This paper will discuss several potential solutions to the dependent attributes problem, then argue that the committee should give its blessing to one in particular. This is a somewhat unusual proposal because we are not proposing standardizing any specific dependent attributes, so accepting this proposal will not entail making any immediate changes to the standard document. However, accepting it would mean agreeing in principle that future standardized dependent attributes should use the proposed solution for expressing dependent attributes, which would provide strong guidance to implementors of such attributes.

Solving these problems could also help address longstanding challenges in C, such as forward-referencing parameters for array lengths. It could also address the more recent issue of specifing a `constexpr` member and using it as the length of an array member. Later in this paper, we discuss how this proposal interacts with N3433 [3] (a proposal to standardize forward parameter declarations), including whether this proposed approach would serve as an alternative solution.

Dependent attributes

This paper was drafted in the context of the design discussions around several different memory safety features in Clang and GCC. Chief among these is the bounds safety extension for C (sometimes known as `-fbounds-safety`), which was originally developed as a vendor extension by Apple [1], but we have also looked at attributes for thread safety [4] and lifetime safety [5]. We believe that the following description is reasonably representative of the entire class of dependent attributes, the range of what they need to be able to express, and the cases that are important for them to prioritize. We have included a Historical Context section at the end of this document that describes several of these attributes in more detail.

In general, a dependent attribute either:

- abstractly refers to a peer of the declaration that it is part of, as an l-value; or
- computes a value in terms of one or more peers of the declaration it's part of, as an r-value.

An example of the former is the `__lifetime_capture_by(X)` attribute, added for lifetime safety checks. Here, `X` specifies a peer parameter that captures the lifetime of the annotated parameter. At present, `X` is restricted to a peer parameter, though in principle, struct fields, member accesses, or other variables could be supported in the future. There is generally no need or desire for a more complex expression.

An example of the latter is the `__counted_by(N)` bounds safety attribute, introduced as part of the bounds safety extension. Here, `N` is an expression of integer type that represents the length of an array. In our experience, the most common form of this attribute—by a substantial margin—is for `N` to be a simple reference to a peer that directly stores the desired value, such as:

```
struct my_vector {
  const double * __counted_by(length) data;
  size_t length;
};
```

`N` being a constant expression with no declaration references at all is a respectable second place. More complex cases (involving arithmetic or other expressions) are common enough to be worthwhile to allow, but still tend to be distant runners-up.

Another example is the `__ended_by(P)` bounds safety attribute, where `P` is an expression of pointer type that represents the upper bound of an array. Currently, `P` is limited to referencing a peer, but in theory, the upper bound could also be specified using a constant expression or an arithmetic expression.

The `__guarded_by(L)` attribute further demonstrates that supporting more complex expressions can be worthwhile, even though references to peers remain the predominant use case. The attribute is used for `-Wthread-safety`, which is Clang's thread safety analysis. While `L` is almost always a simple identifier representing a peer that contains the lock, it can also be a more complex expression, such as a member access, address-of, dereference, or a function call that returns a lockable object.

In summary, all of these dependent attributes may or may not allow complex expressions, but a reference to a peer is by far the predominant use case.

Therefore, a good design for either kind of attribute would ideally prioritize referring to peers.

By "peer", we mean:

- a parameter, for an attribute in a function prototype;
- a member of a struct, union, or enumeration, for an attribute in a member of a struct, union, or enumeration*;
- a global variable, for an attribute in a global variable; or
- a local variable of the same scope, for an attribute in a local variable.

(*This proposal does not address mechanisms for attributes on enumeration members to forward-reference peers, but such mechanisms are possible in theory.)

With that established, we can now consider several different options for how to write the operand of the dependent attribute. The most obvious option would be to use an expression. As discussed in the introduction, this has two basic problems when applying the ordinary rules of C:

- The forward-reference problem: lookup can only find names that have previously been declared.
- The field-reference problem: lookup within a struct or union does not find the members of the struct or union.

The current design for the dependent attributes in the bounds safety extension solves these problems as follows:

1. It solves the forward-reference problem using late parsing, so that all peers are findable by lookups performed within these attributes.
2. It solves the field-reference problem by changing the lookup rules for dependent attributes in the lexical context of a struct or union, so that fields are findable by lookups performed within these attributes.

That is what we are proposing that the C committee accepts in principle as the solution for dependent attributes in C. The next two sections will describe various alternatives and explain why we are in favor of this approach. The following section will then spell out the specific details of our proposal.

Possible solutions to the forward-reference problem

Forward declarations

C's traditional solution to forward-reference problems is forward declarations. It is not possible currently to forward-declare a parameter or field, but that could be allowed.

This would require multiple independent additions to the language to address each of the contexts in which a forward-reference may be required. For forward references to parameters within the parameter clause, a series of papers (most recently N3433 [3]) have proposed standardizing something like the existing GCC C extension of forward declarations of parameters. There are multiple possible reasons to do this; one is that it would allow dependent attributes in the parameter clause to refer to arbitrary parameters. For forward references to parameters within the return type, that would be not enough because the return type precedes the entire parameter clause, which is where parameter forward declarations are written. To solve that, C would need to also introduce something like C++'s trailing return type feature, e.g. `auto strdup(const char *) -> char *`. For forward references to fields within structs and unions, C would require some ability to forward-declare a field. While we are not aware of

any current extensions that allow field forward declarations, there is no technical reason this could not be done, perhaps using a specifier like extern that is not currently allowed on fields.

There may be good reasons to make each of these changes to C, but we believe they have serious problems as a solution for forward references in dependent attributes. Some of these problems are feature-specific, but many of them are common to the idea:

- **Increased burden for common cases:** The need to also introduce forward declarations adds a significant extra burden on a common case of attribute adoption. This becomes particularly onerous when the declaration has a lot of details that would need to be repeated, such as a bit-field width or a complicated type. The expected impact will vary by attribute. `__counted_by`, for example, will typically only refer to values of integer type, and integer types tend to be short, but this also means it can refer to bit-fields. An attribute like `__ended_by`, meanwhile, is likely to refer to a declaration of pointer type, which presumably must be repeated exactly in its forward declaration. (`__ended_by` is likely to always require a forward declaration, since it's natural to pass the start pointer first.)
- **Header portability:** Since dependent attributes often must be written in headers, and many headers are expected to be parseable by multiple compilers and in many language modes, programmers cannot use forward declarations in those headers without guarding them with the preprocessor. This further increases the burden of introducing the forward declaration, and thus of adding the dependent attribute. This would be especially awkward for some of the features above; imagine telling a C programmer that they first need to conditionally rewrite their thirty-year-old function prototype to use a trailing return type so that it can have a dependent attribute in it.
- **C++ compatibility:** Many C headers are also expected to be parseable in C++. It is unclear whether WG21 would be open to these forward declaration features. This runs three risks:
  - first, that programmers will need to remove forward declarations when parsing as C++, as well as potentially removing any dependent attributes which require them in order to be expressed, making it more unwieldy to adopt dependent attributes in headers;
  - second, that C++ will need to come up with its own, likely incompatible language solution for dependent attributes, further increasing the burdens on users of adopting these attributes;
  - and third, that programmers might find those burdens too high to be bothered with, causing them to only enable dependent attributes in a subset of language modes (e.g. not in C++), potentially leading to bugs or surprising gaps in the associated safety features when different modes are in use.
- **Pressure to write unnatural code:** All of these burdens will push programmers towards writing new code in an unnatural order just to avoid needing a forward declaration. This can lead to less consistent interfaces (and thus bugs) across an API, especially where long-established patterns would require forward declarations. For example, functions in the C standard library are always passed a pointer's bound as a later argument than the pointer, meaning that `__counted_by` attributes in the standard library headers will always need forward declarations.
- **Readability impacts:** Forward declarations can also add significant readability impacts on existing code. This is particularly true of the GCC parameter forward declaration extension, in which a forward declaration looks exactly like a normal parameter except for being followed by a semicolon instead of a comma. (The exact separator is a point of contention between the different standardization proposals.) Since programmers frequently read function prototypes to understand how to call them, this is a serious problem. It is perhaps convenient, then, that the need to hide the forward declaration in older language modes also has the consequence of making it more visually distinct, although at the cost of adding a lot of noise to the entire prototype.

  The readability problem is less severe for forward declarations of struct and union fields, since they would need something like a specifier to distinguish them from a normal declaration, and since typical code formatting places each declaration on its own line.
- **Verbosity and programmer error:** Requiring forward declarations introduces significant verbosity, increasing the amount of code programmers must write and maintain. This added complexity not only makes code less readable, but also raises the risk of programmer errors—particularly in large codebases or when attributes must be applied across many declarations.
- **Programmer perception/modernity:** Forward declarations are an inelegant language solution not required

by other languages competing with C such as Rust. Adopting this approach risks creating a negative perception of C's memory safety extensions, making them appear cumbersome or outdated compared to solutions in other modern languages.

Forward declarations in attributes

An alternative to putting forward declarations in the ordinary place that peers appear is to put the forward declarations in the attribute instead:

```
struct double_span {
  const double * __counted_by(size_t length; length) data;
  size_t length;
};
```

This does address some of the concerns above. For one, it keeps the forward declaration out of the peer declarations and therefore avoids any readability impact on them. It also avoids the need to introduce separate language features to handle the three cases of parameters, return types, and fields: a forward declaration in each position would have an unambiguous kind of peer that it would be declaring (although this might not always be completely intuitive to programmers reading the code). This in turn side-steps the need for complex, feature-specific preprocessor guards: compilers which accept the attribute would always allow a forward declaration in it, while compilers which don't accept it would need to have the whole thing hidden anyway with something like

```
#define __counted_by(...)
```

Even though moving the forward declaration to the attribute is much more palatable than putting it among its peers, it is still an extra burden on a very common case. In Apple's adoption experience with the bounds safety attributes, about 85% of attributes needed to refer to a peer declaration, and more than half of those would have required forward declarations. (This is attribute-specific, though. Some attributes, like `__ended_by`, are even more biased towards needing forward declarations; conversely, there are reasons to think that lifetime safety attributes would be idiomatically less likely to need them.)

Retroactive scoping (late parsing)

An alternative that completely sidesteps the need for forward declarations is to make the scope of the declaration retroactive to an earlier point in the file. Semantic analysis for affected expressions within the scope can proceed once all declarations within that scope have been completed. This is not a design idea that is used currently for anything in C, but it is used in several situations in C++ and Objective-C to reduce the need for forward declarations.

The typical implementation technique is to delay parsing of the tokens within the attribute by saving parenthesis-balanced tokens. These tokens are then parsed once the enclosing prototype or aggregate is complete. Therefore, this option is usually called "late parsing", although other implementations are possible, such as producing a raw parse tree that allows for the ambiguity.

One of the most common objections to late parsing is that it is not "C-like". The lookup rules in dependent attributes would be different from those used by C for lookups in the surrounding code. However, Apple's adoption experience with this rule includes many experienced C programmers all using these features without complaint. While the majority of adoption experience is in existing code, we also have a fair amount of experience with adopting these features in new code. We believe programmers find the rule to be intuitive when both writing and interpreting code, and we think there are several reasons for this:

- The first is that the lookup rule is only very slightly different in consequence from the usual rule. (Here we are considering only the difference implied by late parsing, not any change to lookup in structs; we will discuss that in a later section.) In a function prototype, the rule differs only if there is a following parameter which would shadow a name in the enclosing scope. Most programmers would find this an ambiguous situation anyway: even though the language rules are unambiguous about ignoring the as-yet undeclared parameter, someone reading the code will reasonably wonder whether the programmer understood that that would be the compiler's interpretation. It would probably be a better rule if ordinary C lookup always found the later parameter and then, when necessary, rejected the resulting code as invalid. So it's quite understandable that C programmers would find a rule intuitive that allows such a forward reference in a situation where it's clearly sensible.

- The second is that names in C are typically unambiguous in context. C as a language does not make extensive use of lexical nesting, other than of local control statements. Almost all struct definitions and function prototypes appear in global scope, and therefore names in them typically only collide with names from the global scope. Such collisions are rare in practice because C programmers widely follow the convention of putting a "namespace" prefix on almost everything in the global scope, especially in headers. Because parameter and field names are generally not prefixed, there's little chance of confusion about what any particular identifier is intended to name.

  Exceptions to this convention are typically internal to implementation files, which means any conflicting declarations would have to appear later in the same file. A naming collision in such a narrow context would inherently be confusing for the programmer even if it never caused a language problem. One could easily imagine, for example, a file in which it makes sense to have a global variable called `task_count`; but if there are also parameters or struct fields in that file called `task_count`, there are clearly multiple senses of "`task count`" in the file now. Most programmers would agree that it's not a good idea to also have a global variable with that name in such a situation: perhaps it's outlived its original purpose and ought to be removed, or perhaps it just needs to be renamed to something more specific, like `active_task_count` or `max_task_count`. Good programmers will naturally make changes like this while maintaining code even if they never run into any shadowing issues.

As a result, we do not believe that late parsing is unnatural for C programmers or likely to cause any confusion in practice.

There is some compiler performance overhead to late parsing, due to the need to buffer tokens. However, the number of tokens in the argument of a dependent attribute is typically very small; in Apple's adoption experience, more than 90% of arguments are either a single identifier or a single integer literal. Additionally, adding a forward declaration to a source file also has a performance overhead, due to the need to process more source code, to look up names in the declaration, and to validate that it is a legal redeclaration. These costs will generally exceed the overhead of buffering a small number of tokens.

A stronger implementation objection is that C compilers do not otherwise need the logic for late parsing. It is easy to take advantage of late parsing in a compiler like Clang that uses a shared frontend between C, C++, and Objective-C, but other compilers may need to implement it from scratch. We believe that this overstates the complexity of implementing late parsing, especially relative to the other costs of implementing major features like the bounds safety extension [1]. Furthermore, once late parsing is implemented for one attribute, it can be easily re-used for all attributes that need it.

Most features that use dependent attributes do need to restrict the contents of the argument expression. For example, `__counted_by` does not permit the expression to contain arbitrary side-effects, which is implemented with a syntactic check preventing assignments, (most) function calls, and common extensions like statement-expressions. If it would make a late-parsing-like solution more acceptable to other implementations, we think it would be reasonable for the committee to impose similar restrictions grammatically, thus reducing the size of the grammar that must be accepted by the "late parser". If this went as far as forbidding casts, it would actually reduce the expression grammar to a context-free subset and allow the actual parsing to be done eagerly, although semantic analysis would still need to be delayed until all the peers were known.

Simple identifiers only

We have said that simple references to peers are the most common case in dependent attributes. If so, it is reasonable to ask if the argument to the attribute should really be thought of as an arbitrary expression at all. If the argument is just an identifier, and that identifier is always required to name a peer, it can just be looked up when the complete set of peers is known. This is still fundamentally a kind of late resolution, coupled with a lookup change that not only prefers but requires the lookup to resolve to a peer. (That is, this option is also a solution to the field-reference problem.) However, it is such a simplified special case of both that it is acceptable to some people who are uncomfortable with those changes if taken more broadly.

Nevertheless, important dependent attributes such as those for bounds safety and thread safety do need to be able to specify more complex expressions. In a recent sample of adopting code, Apple found that about 75% of the uses of `__counted_by` were simple references to parameters or fields. That's a big fraction of the whole, but only allowing such references would still leave 25% of bounds inexpressible. These remaining uses include many constants, but also some arithmetic expressions involving the values of peers, like `__counted_by(2 * numElts)`.

One way of thinking about the design approach we are proposing is that it starts with a design that accepts simple references to peers without any further ceremony, then allows such references to be generalized by embedding them into arbitrary expressions in a natural way. The changes to lookup are simply how this intuition must be fully specified.

Possible solutions to the field-reference problem

Before discussing solutions to the field-reference problem, it is useful to discuss how it came to be.

Historically, expressions within struct and union definitions have always been pragmatically limited in C. Because C does not allow any kind of function or initializer within a struct or union definition, there are only three remaining places that expressions can appear: a bit-field width, the bound of an array type, and the `typeof` operator. `typeof` is rarely used in field declarations for various emergent reasons. The other two contexts must be integer constant expressions. Prior to C23's introduction of `constexpr`, the only kind of declaration that could be used in an integer constant expression (other than within a `sizeof` operator applied to an expression) was an enumerator. As a result, the C standard has never had much reason to pay attention to the expression lookup rules within struct definitions, because for fifty years, there was almost no reason to refer to any declaration there except a type. For most of that time, when C programmers have wanted a symbolic constant that they could use in an array bound or bit-field width, they have generally used a macro.

The lookup rules within struct and union definitions are unique. It is the only context in the language in which completing a declarator does not bring a name into scope, at least not in any way that can be referred to. Instead, struct fields go into an alternate namespace. This decision likely ultimately descends from the unfortunate design decision of early C to put fields into a common namespace, allowing any field to be accessed on any type of pointer. Again, this has always been acceptable because there has never been much reason to refer to any object or function declaration within a structure or union definition, much less a field. The most fundamental reason that dependent attributes change this picture is because they are the first feature in C to have this need.

Simple identifiers only

We have noted already that Apple's review of its current adoption of the bounds safety attributes found that simple peer references accounted for about 75% of all attributes. This is even more skewed for attributes within structs: about 90% of these attributes are simple field references, with a few percent applying arithmetic to fields and the remainder being constant expressions.

As we discussed in the section on forward references, requiring the argument of the attribute to be a simple identifier that names a peer fixes both the forward-reference and field-reference problems. However, it excludes a lot of possibilities for attributes like `__counted_by` and `__guarded_by`, including constant values, member accesses, function calls, and arithmetic expressions over peers such as `__counted_by(num_values + 1)` (a reasonable bound for an array that is also null-terminated). These possibilities are not nearly as commonly used as simple field references, but they are still important to express in a system which aims to add bounds safety to all pointer accesses.

It is worth nothing here that naming a peer field using a simple identifier has generally been accepted by the community, for attributes like `__guarded_by` and `__counted_by`, and others. However, limiting these attributes to only simple identifiers forces the introduction of a separate attribute (e.g., `__counted_by_expr`) and a new syntax to support more complex expressions. This creates an inconsistency within the same class of attributes: field-referencing attributes would require one form for simple cases and an entirely different form for slightly more complex ones, even though the semantic intent is the same.

Finding fields in name lookup

If naming fields is the main case that needs to be expressed, but allowing arithmetic expressions in terms of those fields is also important, the most gentle generalization path is simply to allow field names to be written within expressions. That is, if the most natural way to write that the size of the array pointed to `ptr` is stored in the field `length` is:

```
int * __counted_by(length) ptr
```

then it follows that the most natural way to write that the size of the array is actually twice the value in `length` is:

```
int * __counted_by(length * 2) ptr
```

Implementing this means, essentially, changing the lookup rules in `__counted_by` expressions written within struct and union definitions.

The most straightforward rule is to simply follow the normal lexical scope rules, such that field names within a struct scope are shadowed by names in inner scopes and themselves shadow names from outer scopes, just like any other scope. That is what we are proposing in this paper. We will propose a precise rule in a later section.

We are only proposing to change the lookup rules within the arguments of dependent attributes. While making a general change to lookup within struct bodies would be more consistent, it is also a major change that could break source compatibility. We do not want the resolution of this proposal to depend on a decision like that. As argued elsewhere, we think field shadowing is rare, and so the practical consequences of inconsistency between normal lookup and lookup within dependent attributes will be minor. It will just be a technical curiosity with an interesting historical explanation; we believe the C community can live with things like that. (For those interested, the Appendix provides examples of potential inconsistencies and suggested mitigation strategies, though such cases are rare.)

This also means that we do not need to give language rules for other expressions that have lookups that resolve to field references. Such rules are not difficult to define, however, and C could reasonably just borrow C++'s rules: a field reference is invalid in a potentially-evaluated context (i.e. not the operand of `sizeof`) that lacks an implicit object (i.e. all contexts in C except for dependent attributes in structs), but otherwise it is considered to be an l-value expression of the field's declared type.

`this` and other special member access syntaxes

The other way to solve the field-reference problem is to force the peer reference to be something other than just the name of the field.

Fields are normally accessed with member expressions, so the simplest option would be to invent a syntax that binds a notional value of the enclosing struct or union, which then can be used as the base of a member expression:

```
struct vector {
  const double * __counted_by(this->length) data;
  size_t length;
};
```

`this` is, of course, borrowed from C++; alternate names include `__this`, `__self`, `__builtin_this()`, and so on.

Another option is to introduce some novel syntax that is understood in context to be a member access to a notional value of the enclosing struct or union:

```
struct vector {
  const double * __counted_by(.length) data;
  size_t length;
};
```

A third option, and the last we will consider, is to use the attribute to name a function that will do the computation in terms of a parameter that will be bound to a value of the enclosing struct or union:

```
static size_t get_length(const struct vector *);
struct vector {
  const double * __counted_by(get_length) data;
  size_t length;
};
static inline size_t get_length(const struct vector *v) {
  return v->length;
}
```

These options share two common drawbacks: they add extra ceremony to the predominant case of dependent attributes within structs and unions, and they encourage a hard split with C++ over these attributes.

As we have already discussed, most dependent attributes within structs and unions refer to at least one field. This is true of about 90% of attributes like `__counted_by`, at least in Apple's current adopting code. As a result, any burden on attributes that refer to fields needs to meet a relatively high standard of justification.

The function option clearly imposes an unjustifiable burden. While it is nicely composed using existing features, declaring a completely separate function to wrap a single member expression is a lot of boilerplate, especially because the function also needs to be forward-declared. One could argue that a separate attribute, e.g., `__counted_by_fun` can be provided to support more complex use cases, while keeping the simple name lookup rules for a single member reference in attributes like `__counted_by`. However, that means every single dependent attribute that takes more than an identifier will need to be duplicated. It's also harder for the compiler to analyze and restrict the body of a separate function than it is to examine an inline expression, and programmers trying to read the code and understand the bounds of the pointer (an understated benefit of attributes like `__counted_by` is that they also work as implicit documentation) would have to take several unnecessary extra steps.

The novel syntax option might only be a slight burden, depending on the syntax chosen. Certainly a single `.` is a very slight burden. Of course, to keep the burden down, this option requires the reservation of some novel syntax for this specific purpose, which has consequences. For example, the `.` syntax collides with initializer designators, and this would need to be resolved somehow in the grammar and in parser implementations. Implementing parsing for an alternate expression grammar in which `.identifier` is a valid primary-expression might require cloning most of the expression parser, especially in parser-generator-based implementations. This would also "claim" the syntax and constrain the future evolution of the language; for example, if C wanted to allow an initializer like `{ .x = foo(), .y = .x }` (copying the value computed in a previous initializer), it would have to resolve how that interacted with this syntax. Attribute implementors are understandably nervous about claiming significant pieces of syntax without the blessing of WG14. It has been done by extensions in the past, but that is not a good reason to keep doing it. Later in this paper, we explore the `.` syntax option as a potential alternative.

The member-access option is more of a middle ground, adding a few largely-redundant tokens to almost every attribute. The additional challenge here is simply picking the spelling of the base expression. Many of the C programmers we consulted seemed very uncomfortable with the idea of using the C++ spelling `this`, but any other spelling would only deepen the conflict with C++ over these attributes, as it would just be a differently-spelled `this`. More elaborate spellings, like a builtin call, would also make the attribute longer and more difficult to read, to little purpose.

That is perhaps the key question that a proposal to adopt any of these options would need to answer: what is the purpose of requiring additional syntax beyond naming the desired field? In previous discussions, proponents have focused on the potential for a name collision between a field and a declaration from the enclosing scopes. We do not believe that this is a serious problem. As we have already discussed, the enclosing scope is likely to just be the global scope, and field names are unlikely to conflict with names from the global scope in typical C programs. Programmers reading the struct will see that the name in the attribute expression matches a nearby field and naturally expect that it is meant to refer to that field, agreeing with both the original programmer's intent and the actual interpretation of the code by the compiler.

In the unlikely event that there is a conflict, and the programmer really does want to specify a value from an outer scope, they can simply introduce a new, more specific name for that value, at least when dealing with local or `constexpr` variables:

```
constexpr size_t max_count = 16;

struct widget_list {
  size_t max_count;
  struct widget * __counted_by(max_count) data;
};
```

becomes

```
constexpr size_t max_count = 16;
constexpr size_t max_widget_count = max_count;

struct widget_list {
  size_t max_count;
  struct widget * __counted_by(max_widget_count) data;
};
```

This is a commonplace sort of change that practiced C header maintainers make all the time.

However, renaming is not always possible, especially when the outer variable is a global or comes from external code. In such cases, providing a special syntax (for example, a :: prefix to indicate global scope) to explicitly refer to the outer variable would make sense, while allowing the default name lookup for dependent attributes to prioritize the struct or union member. This strikes a balance: it matches both common usage patterns and programmer expectations, while still allowingg access to globals or outer-scope variables when truly needed.

The second drawback of these options is that they encourage a hard split with C++. Many C headers are expected to be parseable as C++, and doing so should not require dropping any dependent attributes. The expression must therefore be parseable in C++. While C has never before had much reason to allow references to fields within struct or union bodies, C++ has always allowed code there. Therefore, C++ long ago changed the lookup rules for expressions within classes to be able to find fields. (That this has not caused widespread problems parsing C headers as C++ can be considered evidence of our argument that actual naming collisions between fields and the global scope are very rare in practice.) If it is truly an important goal for lookup in these attributes to ignore fields and instead resolve to declarations in enclosing scopes, then it is an important problem that such attributes will be misinterpreted when parsed in C++.

Of course, that problem can be resolved in several ways. We are proposing that C change its lookup rules to be more C++-like in these attributes, but one could just as well argue that C++ should change its lookup rules to be more C-like. There are limits to this, though. WG21 is unlikely to be pleased to consider adopting novel syntax for field references in dependent attributes purely to solve a problem that C++ does not actually naturally have. A WG21 member might also observe that, if shadowing global names is a serious problem in C, WG14 might consider adding a scope resolution operator like :: rather than fitting the rest of the language around its absence.

If the solution that WG14 adopts to the field-reference problem seems unnecessarily out of step with C++, WG21 may well reject it, leaving the designers of dependent attributes in the awkward position of either picking sides in a conflict between the committees or forcing their users to deal with the language portability concerns on their own.

We do not believe that any conflict here is necessary, because we feel that allowing simple field references within dependent attributes is independently justifiable as the best thing for C for the same reasons that the C++ designers reached the same conclusion: because it is a natural and intuitive way to refer to a field within the scope of the struct, and because naming collisions that should be resolved in favor of an outer declaration are very unlikely and easy to resolve through other means.

Proposal

An attribute may define one or more of its expression operands to be a *dependent attribute expression*. We propose special lookup rules for dependent attribute expressions, adopting retroactive scoping for forward referencing and treating struct or union members as if they were in the ordinary identifier namespace. As a result, the way member names of a struct or union member are looked up in dependent attribute expressions is similar to how name lookup works in C++ method bodies (or to a potential future C method body lookup), because-just like method bodies-dependent attribute expressions should be able to refer to members regardless of their declaration order. The formalized rules are as follows:

- In addition to its ordinary scope, an identifier may have a dependent attribute scope. The dependent attribute scope of a parameter is the dependent attribute scope associated with its function declarator; the dependent attribute scope of a member declaration is the dependent attribute scope associated with its struct or union; for all other identifiers, the dependent attribute scope is their ordinary scope.
- In a function definition or function prototype, the dependent attribute scope of a parameter name extends to dependent attribute expressions attached to the function return type and to any part of the function declarator in that definition or prototype.
- For a parameter declaration in a function declarator appearing in other contexts, the dependent attribute scope of the parameter name extends to all dependent attribute expressions attached to any part of the function declartor and its preceding type specifier, including cases where the type specifier is shared among multiple declarators in a single declaration.
- The dependent attribute scope associated with a non-anonymous struct or union begins at the start and terminates at the end of the `struct-or-union-specifier` that defines it.
- The dependent attribute scope associated with an anonymous struct or union begins at the start and terminates at the end of the nearest enclosing non-anonymous `struct-or-union-specifier` if one exists. If not, it begins at the start and terminates at the end of the outermost anonymous `struct-or-union-specifier`.

- The name of a member is not visible in the dependent attribute scope of any enclosing struct or union of the struct or union that defines the member (except as a member of the struct or union itself), unless the member is declared in an anonymous struct or union, in which case it shares the same dependent attribute scope as the enclosing struct or union.
- The name of a member of a struct or union is not visible in the dependent attribute scope associated with a nested struct or union, except in the case of an anonymous struct or union, whose members are injected into the enclosing scope.
- Otherwise, an identifier in a dependent attribute expression is resolved (other than for the purposes of the `.` and `->` operators) as if identifiers had their dependent attribute scopes, and as if members were in the ordinary identifier namespace. (Note that the dependent attribute scopes described above are still properly nested and therefore still imply shadowing as usual.)
- In a special case, dependent attribute scopes with no nesting relationship may partially overlap-specifically, when multiple declarators in a declaration share a type specifier, such as a return type with an attached dependent attribute. In such cases, the overlapping scopes are merged into a single dependent attribute scope.
- An identifier is ill-formed if it could designate multiple entities in the same scope. (This can occur only when scopes are merged as a result of a shared type specifier with a dependent attribute attached, as desribed above.).

Here is an example demonstrating when dependent attribute scopes for a declaration are merged, and when they are not:

```
typedef int * intptr_t;

// well-formed; `f1` and `f2` have separate dependent attribute scopes, because
// `__counted_by(n)` is attached to the declarator *f1(int n).
int *__counted_by(n) f1(int n), f2(int n);

// ill-formed; dependent attribute scopes associated with `f3` and `f4` are
// merged into a single scope, due to `__counted_by(n)` attached to the shared
// type specifier, `intptr_t`.
intptr_t __counted_by(n) f3(int n), f4(int n);
```

Note that the only entities with unusual dependent attribute scopes are function parameters and member declarations. Because the dependent attribute scopes of these entities precedes the completion of their declarator, identifiers in dependent attribute expressions that may overlap with such a scope cannot be resolved until the scope is complete. A sketch of one possible implementation is given.

Implementation sketch

Dependent attribute scopes allowing forward references can only begin at the start of a declaration or type-name. Furthermore, while there can be multiple of these scopes active at once, they must be the innermost scopes, because no other scopes can be nested within a declaration or type-name. This allows a straightforward implementation based on late parsing.

The parser keeps a stack of active forward-reference scopes. When the parser begins to parse a dependent attribute, if the forward-reference scope stack is empty, it can parse the dependent attribute expression as normal. Otherwise, it must save a record of the attribute, including the buffered tokens of the expression and space for "filling in" each entity in the forward-reference scope stack, in the innermost forward-reference scope. When the parser ends a forward-reference scope, and there are any forward-reference attributes collected within it, it first fills in each attribute with some representation of all the forward-referenceable entities in the scope (e.g. by collecting a list of all the parameters in the function declarators whose dependent attribute scope just ended). If this scope was the bottom of the stack, it can now late-parse all of the dependent attribute expressions in the attributes; otherwise, it adds them to the enclosing forward-reference scope so that they can be filled in when it completes.

To late-parse a dependent attribute expression, the parser must be configured to parse out of the saved tokens, and any identifiers in the expression must be looked up in a special mode which checks the filled-in contexts (in order) before looking at the normal scope tree.

Comparison to proposals for forward declaration of parameters

Difference in scope

N3433 [3] is the latest in a series of proposals to allow parameters to be forward declared. One of the stated purpose of these proposals is to allow the array bound of a parameter to declare its size using another parameter that is declared later. This is important considering that every function in the C standard library passes the bound of a pointer after the pointer. Such a proposal would also solve the forward-reference problem for parameters in dependent attributes.

We have argued that we believe that forward declarations are not a good solution for the forward-reference problem in dependent attributes: they are syntactically heavyweight, they are awkward to adopt in portable headers, they tie adoption of the safety features supported by dependent attributes to adopting a newer C standard, and they may permanently conflict with C++. They also do not solve the problems of referring to a parameter in a return type or forward-referencing a struct field. Otherwise, we do not take a stand on N3433.

This section will discuss various interactions between this proposal and N3433, including whether it would serve as a replacement.

Using a dependent attribute as an alternative to N3433

While we are not currently proposing adding any specific dependent attributes to C, a dependent attribute using the lookup rules in this proposal can certainly express the array bounds in N3433 without the use of parameter forward declarations. For example, the bounds safety extension uses `__counted_by(N)` to generate bounds checks, but a compiler could also use the information in the attribute to feed the optimizer, the same way that compilers do with `[static N]`:

```
// N3433
void f1(int size; int arr[size], int size);
void f2(int size; int arr[static size], int size);

// using a dependent attribute to forward reference `size`
void f3(int arr[__counted_by(size)], int size);
```

This is sufficient to allow any remaining K&R-style functions to migrate, such as the following:

```
void f(arr, size)
  int size;
  int arr[size];
{ ... }
```

Applying retroactive scoping for array parameter size

This proposal does not modify the ordinary lookup rules. It uses late parsing only for dependent attributes that are commonly needed in the context of memory safety. That said, late parsing could potentially be applied to array parameter sizes, because array parameter sizes also express relationship between two parameters of the same function. This is known to potentially change the behavior of existing code.

Here is an example of the behavior change. Under C23 rules, the size of the array `p` in the following code is the global `constexpr` variable `n`:

```
constexpr int n = 10;

void f(int p[n], int n);
```

If a future proposal adopted late-parsing rules for ordinary lookup, the size of the array `p` would be the parameter `n`, which is declared in an inner scope and therefore shadows the global variable.

If made soon, the expected impact of this change would be small. `constexpr` is a very recent addition to C, and almost all C code in practice still uses `#define`s to define this kind of global constant. (Enumerations are available as an older way of writing this without `#define`, and some code does use them, but `#define` is much more common.) Moreover, we have argued that this kind of shadowing is very rare in practice for both emergent and practical

reasons. The programmer may even have made an earnest mistake here, one which would have been caught with a compiler error were it not for an unexpected global constant, which may be declared in a little-thought-of header. (The POSIX function index often catches errors of this sort, although thankfully it does not work as a constant expression.)

While we do not suggest changing the behavior for array parameters immediately, at least it might be worth reporting a warning as it is likely to be a mistake, and gathering data about whether code like this exists and how frequently it is a mistake. If it turns out that this pattern is frequently a mistake, adopting retroactive scoping for the bounds of array parameters may become a desirable direction.

Compatibility between N3433 and this proposal

Our proposed approach is largely compatible with N3433 because forward parameter declarations and late parsing can coexist. The idea in N3433 and similar papers is that a parameter used in a dependent attribute is already forward declared, so there is no need for the name lookup to delay parsing. Once a parameter is forward-declared, however, a later parameter cannot conflict with it, so the lookup performed by late parsing will still find the same parameter:

```
void f1(int size; int *__sized_by(size) buf, int size);
```

Forward-declared and ordinary parameters can be freely mixed by late parsing:

```
void f2(int size; int *__sized_by(elems * size) buf, int elems, int size);
```

Mutual dependencies between parameters

N3394 [6] listed mutual dependencies between parameters as one of the reasons not to adopt retroactive scoping, citing [7], which provides the example:

```
void g((*p)[sizeof(*q)], (*q)[sizeof(*p)]) { ... }
```

Such mutual dependencies cannot be expressed using the forward declaration approaches proposed in N3394 or N3433, since both require specifying the type of each forward declaration in sequential order and only allow references to previously declared parameters. As a result, these approaches inherently avoid the possibility of cyclic dependencies. With retroactive scoping, the programmer is able to write code like this, but the lookup rule can still be defined so that such dependencies are diagnosed as ill-formed. Therefore, we do not consider the possibility of mutual dependencies between parameters to be a compelling reason to reject retroactive scoping.

Extending special name lookup to array member sizes

In C, struct members are not in scope within the member list for purposes like array bounds for constant expressions.

Adopting special name lookup rules to allow this for array sizes would break existing code that uses a global `constexpr` for an array size, but where that name conflicts with another member in the structure:

```
constexpr int n = 10; // or enum { n };

struct s {
  int n;
  int array[n]; // stops building because a member declaration cannot be an array size.
};
```

However, we have argued that such name conflicts are likely to be very rare in practice. When they do occur, they may often indicate a programmer error-one that would have been caught by a compiler error if not for the presence of the global constant with the same name.

While we do not suggest changing the behavior for array members immediately, it may be worthwhile to at least issue a warning and collect data about whether this pattern actually appears in real-world code, and how often it is the result of a mistake. If it turns out that such code is both rare and usually a mistake, adopting the proposed name lookup rules for the bounds of array parameters may become a viable direction.

The proposed name lookup rules for fields could unlock additional idioms for array members. For example, it would enable using a `constexpr` member for the array size:

```
struct s {
  constexpr int N = 20;
  int array[N];
};
```

It would also help supporting VM types for struct members in the future:

```
struct s {
  int n;
  int (*array)[n];
};
```

Alternatively, this could be achieved by introducing a new syntax (such as `.` or `__self`) to make field access explicit, rather than changing the default lookup rules. However, a potential downside is that it becomes easy to omit this syntax, causing the program to behave differently from what the programmer intended:

```
constexpr int n = 10;
struct s {
  constexpr int n = 20;
  // The intention was to refer to the member `n`, but it resolves to the global `n`.
  int array[n];
};
```

Moreover, for many programmers, code like the example above might naturally read as if the array member is sized with the member `n`, because that's very common idiom. Thus, more study is needed to assess both the frequency of such patterns and how often they represent programmer intent or error.

Historical context and adoption

While this paper was written with consideration for several other past and potential future features that make use of dependent attributes, it arose primarily in the context of discussions about adding the bounds safety extension in Clang and GCC.

The bounds safety extension for C, originally invented by Apple, guarantees bounds safety through a combination of compile-time and run-time checks. To accomplish this, it introduces a set of attributes that programmers can use to specify bounds for various pointer and array types. For instance, `__counted_by(N)` indicates that a pointer or array has at least `N` elements in memory:

```
void process_matrix(const double * __counted_by(nrows * ncols) data,
                    size_t nrows, size_t ncols);
```

When code within this function reads from `data`, the compiler emits an implicit runtime check that the index is within the `__counted_by` bound. When the function is called, the compiler emits an implicit runtime check that the `data` argument points to an array at least as large as the `nrows * ncols` bound. If the compiler cannot determine the size of the argument array, it rejects the program.

The bounds safety extension provides several other dependent attributes for expressing bounds in slightly different ways, but `__counted_by(N)` is a reasonable representative.

Apple originally implemented the bounds safety extension in a vendor branch of Clang. In this branch, the expression `N` can be a constant literal, a simple declaration reference, a binary arithmetic and logical expression (`+`, `-`, `/`, `*`, `>>`, `<<`, `|`, `&`, etc.), or a call to a function with the const attribute and no side effects. `__counted_by` is allowed in struct members, function parameters, return types, as well as local and global variables.

To allow `N` to refer to declarations that appear later in the function prototype or struct member specification, the expression is parsed using "late parsing", where the parser first collects delimiter-balanced tokens from the lexer, then parses them only when the prototype or member specification is complete. Additionally, name lookup in the expression must use a variant rule that finds struct members as if they were ordinary names in scope. Both of these rules are novel in C. Late parsing has been already available in Clang, as it is required for the C++ and Objective-C frontends, and Clang uses a shared codebase for all languages. Other compilers that do not share implementations with C++, such as GCC, would need to implement this feature separately. Notably, some patches to add delayed parsing support for bounds attributes were recently submitted for feedback in GCC [8].

Mainline Clang and GCC have also started implementing the `__counted_by(N)` attribute used by the bounds safety extension, but for now only in limited contexts. In Clang, the `__counted_by(N)` attribute is supported for struct members including pointers and flexible array members. In GCC, the `__counted_by(N)` attribute is applicable only to flexible array members. It can be used to refine the array bounds sanitizer and the `__builtin_dynamic_object_size` builtin. In both cases, `N` is currently limited to be an identifier that names another field of the same structure, but both projects are looking to extend it to support some limited form of expressions to support more use cases. The discussions around that have substantially informed the drafting of this paper.

Clang's `-Wthread-safety` warning uses very similar dependent attributes, such as `__guarded_by(L)`, `__ptr_guarded_by(L)` [4]. The analysis warns when certain fields are used from a context that is not statically known to be holding a particular lock (identified by `L`). The expression `L` can be a simple field reference, member access, address-of, dereference, a function call that returns a lockable object, or a sentinel value like `!identifier` to indicate that the annotated field is never guarded. The attribute is available to use in both C and C++.

Clang's lifetime safety checks also use a dependent attribute, `lifetime_capture_by(P)`. This attribute can be used on a function parameter or implicit object parameter, indicating that its lifetime is captured by another parameter. The attribute can refer to one or more parameters (separated by commas), regardless of the declaration order (e.g., `lifetime_capture_by(s1, s2)`). While lifetime attributes are primarily used for analysis in C++ today, they could also play an important role in enabling safe interoperation between C/C++ and other memory-safe languages [9].

In theory, many other properties or invariants can be expressed through dependency relationships, making them potential use cases for dependent attributes-supporting static analysis, API contracts, and concise documentation. Potential examples include: a pointer that is an alias of another pointer, a reference borrowed from another object, a variable that serves as a key or index for a container, or a pointer parameter that is non-null if a peer is not zero.

Apple adoption experience with the bounds safety extension

Apple has substantial adoption experience with the bounds safety attributes, enough to feel like we can speak with reasonable authority about how it is generally used in practice. In one recent sample, we found that bounds expressions are overwhelmingly very simple:

- 72% were a simple reference to a peer declaration (a parameter or struct member).
- 8% were a deference of a simple reference to a peer. (the bounds safety extension only allows this on parameters.)
- 17% were integer constant expressions, primarily single literals.
- The remaining cases were arithmetic expressions over peers, with less than half a percent involving multiple peers.

This heavy bias towards simple peer references strongly motivates Apple's desire to make that case easy to write. These are roughly even split between forward and backward references.

Our sample does not include any references to `constexpr` global variables, which have not been adopted significantly in Apple's C code.

Alternative approach to field lookup: Using special member access syntax

As discussed previously, introducing additional syntax to access fields in dependent attribute arguments raises several concerns:

- It increases programmer burden due to extra tokens.
- If designator syntax is chosen (e.g., `.identifier`), a change to the C and C++ standards would be required to allow designator syntax as a primary-expression.
- It risks a hard split with C++, which C++ already has mechanisms for field access; justifying new, parallel syntax in both languages would be challenging.

However, adding a dedicated syntax does offer the benefit of potentially providing a consistent model—particularly for array lengths—without altering the meaning of existing standard C code. Thus, it is worth considering what this direction would entail.

The first concern (programmer burden) may not a hard blocker if a succinct syntax is chosen. If, however, designator syntax is desired, it would require approval from both WG14 and WG21 to make `.identifier` a valid primary-expression. For further analysis, let us assume both committees are amenable to introducing such a syntax for dependent attributes, or that a mutual acceptable new syntax has been identified.

With an added field-access syntax, unqualified names could still be used, defaulting to each language's standard name lookup rules. The challenge is that code like the following would behave differently in C and C++:

```
constexpr int len = 10
struct s {
    int len;
    int *__counted_by(len) buf;
}
```

In C, `__counted_by(len)` would refer to the global `len`, while in C++, it would refer to the member `len`. This silent behavioral divergence is undesirable, especially if headers are to be shared between C and C++.

One way to avoid this divergence would be for C++ to adopt C's name lookup rule for unqualified names in dependent attributes—so that `___counted_by(len)` would always resolve to the global `len`. However, this behavior is counter-intuitive and error-prone, not only in C++ but also in C, since it is common for the length of a buffer to be stored as a peer member in the struct. Consequently, the main use case for dependent attributes is referencing a peer declaration. Thus, this behavior is unlikely to match the intent of the code or how the code would typically be interpreted. This approach would also introduce silent changes to the meaning of existing code using dependent attributes.

To prevent such unintended changes in meaning, a potential workaround is to disallow the use of an unqualified name if it matches any field name in the enclosing struct or union. In this model, a global specifier syntax (e.g., `::`) would be needed instead to refer to a global variable that conflicts with a field name. For example:

```
constexpr int len = 10
struct s1 {
    int len;
    int *__counted_by(len) buf; // error: missing `.` to refer to member `len`
};

struct s2 {
    int *__counted_by(len) buf; // error: missing `.` to refer to member `len`
    int len;
};

struct s3 {
    int *__counted_by(.len) buf; // resolves to member `len`
    int len;
};

struct s4 {
    int len;
    int *__counted_by(::len) buf; // resolves to global `len`
};
```

This solution avoids silent changes in behavior, but it would cause most existing code that uses dependent attributes on struct fields to fail to build, as shown in the examples for `struct s1` and `struct s2`. Additionally, a new syntax for accessing fields would still need to be accepted by both C and C++.

Introducing a new syntax also does not fully resolve the issue of forward referencing. While the new syntax could be used to implicitly create a forward declaration in some contexts-similar to the proposal in N3188 [10], which uses designator syntax for array lengths in VM types-there are important differences. Unlike array lengths, the types of declarations referenced by a dependent attribute are not necessarily known in advance; they may include integers, pointers, or other types. Therefore, even with a new syntax, late parsing would still be required to handle forward references, just as it would be without the new syntax.

Appendix

Inconsistency between normal lookup and lookup within dependent attributes

Here is an example illustrating inconsistency between normal C lookup and lookup within dependent attribute expressions. In this example, the size of the member `int arr[len]` resolves to the global `constexpr len`, whereas the operand of the dependent attribute `int *__counted_by(len)` ptr used in the same struct resolves to the member len (because the global `constexpr` is shadowed by the member):

```
constexpr int len = 10;

struct s1 {
  int len;
  int arr[len]; // resolves to global `len`
  int *__counted_by(len) ptr; // resolves to member `len`;
  // suggested warning: `len` has multiple meanings in this scope
};
```

As explained earlier, we believe field shadowing is extremely rare and so are the practical consequences. This can be further mitigated by issuing a warning where a name used within a struct would have different meanings in a dependent attribute versus in the other contexts.

A second example demonstrates the potential for confusion when a type name and a field name used in a dependent attribute expression conflict:

```
typedef int T;

struct s2 {
  int T;
  int m;
  int *__counted_by((T)+m) ptr; // resolves to member `T`
  // suggestion: limit the dependent attribute expression to not use casts
};
```

In this case, `T` in `__counted_by((T)+m)` always resolves to the field `T`, since the field name shadows the type name within the dependent attribute scope. However, this syntax may lead a programmer or reader to wonder whether `(T)` is intended as a type cast or is simply parenthesizing the field name in a binary expression.

This can also be addressed by either issuing a warning indicating that `(T)` is not a type cast (and suggesting removal of the parentheses to clarify intent), or more robustly by limiting dependent attribute expressions to prohibit type casts entirely. As discussed earlier, dependent attribute expressions can be defined as a context-free subset to prevent such ambiguities.

Limits to late parsing

This proposal does not propose using late parsing in contexts outside of dependent attribute expressions. However, WG14 members may be interested in how broadly the approach could be used in the language. We will try to analyze this point.

We believe it is generally a good goal for C to remain a language that can be parsed in a single pass. The late parsing we are proposing is not really at odds with that because it is carefully limited in extent: the compiler only needs to go back and revisit tokens within a single declaration or struct or union specifier. This is not deeply different in its consequences from the way that a compiler might re-walk the fully-parsed declarator structure to build the type of a declaration, or how it might perform struct layout in a separate pass rather than online as fields are parsed. Both of those techniques are common in compilers today. In contrast, a use of late parsing that required entire function bodies to be late-parsed, or arbitrary amounts of code at the top level to be late-parsed, would be more inherently at odds with this goal of C.

One key to limiting the possible complexity introduced by late parsing is avoiding the need to delay analysis for arbitrary uses of a declaration. Most importantly, this can happen when something that needs to be late-parsed would affect the type of a declaration, because the type of a declaration generally must be known in order to process an expression that refers to it. Determining the type of a declaration with a late-parsing-sensitive type could require

the compiler to first determine the types of other declarations, e.g. if they use `typeof` or `sizeof`; this could even lead to cycles. Resolving this while retaining maximal expressivity would require major architectural changes to the compiler, e.g. to be more demand-driven in resolving the types of parameters while parsing a prototype. This is better off avoided, which it can be in two ways: first, by keeping late-parsing sensitivities out of the type system as much as possible; and second, by preventing uses of declarations with late-parsing-sensitive types before their types can be fully resolved.

The simplest way to keep late-parsing sensitivities out of the type system is to continue to require anything that would affect the type to be eagerly parsed. For example, if parameter scopes were made retroactive to the start of the parameter list, arbitrary identifiers in parameter types (such as `typedef`-names or in `typeof` specifiers) should probably still be forbidden from forward-referencing a parameter so that they can be eagerly resolved. As noted in the implementation sketch, this restriction can be achieved without making the lookup rules fundamentally different for different contexts by ensuring that lookups that would have designated a parameter if late-parsed are still diagnosed as invalid. This can be done by just performing an eager lookup as normal, then diagnosing any collisions with parameter names upon the completion of the parameter list. Any exceptions to this eager-parsing requirement within types should be introduced judiciously because of the potential complexity for implementations.

Some places where late parsing could be used would inherently not affect the type system. A static array bound for a parameter, for example, is not part of the type, and so there are no implementation problems with allowing this expression to be late-parsed.

Late-parsing can be allowed in places that would affect the type system without major architectural changes to implementations by just preventing the use of the declaration before its type can be resolved. In practice, it is rare to use something like `typeof` with a parameter within the parameter list itself, and there are always other options.

For example, array bounds in parameters could be late-parsed, allowing them to forward-reference other parameters, a significant generalization for VLA use in parameters:

```
void foo(double (*array)[size], size_t size);
```

When trying to resolve the type of the parameter eagerly, the implementation would note the presence of an array bound, build a type with the array bound still missing, and mark the declaration as invalid to use. At the completion of the prototype, the implementation would perform late parsing, rewrite the parameter's type to fill in its late-parsed bounds, then make the parameter usable again. This would have to happen in some specified order, e.g. left-to-right in the parameter list, in case late parsing included something like a `sizeof` involving a parameter.

Of course, any changes like this to the core language would have the potential to break source compatibility, as discussed in the previous section. Again, we are not proposing this, just analyzing its viability for the consideration of the committee.

Acknowledgements

We would like to acknowledge David Tarditi and Aaron Ballman for their valuable feedback on this work.

References

[1] "-fbounds-safety: Enforcing bounds safety for C." Available: https://clang.llvm.org/docs/BoundsSafety.html
[2] "N3211: Memory-Safety in C." Available: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3211.pdf
[3] "N3433: Alternative syntax for forward declaration of parameters." Available: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3433.pdf
[4] "Thread Safety Analysis." Available: https://clang.llvm.org/docs/ThreadSafetyAnalysis.html
[5] "Safely Mixing Swift and C/C++." Available: https://www.swift.org/documentation/cxx-interop/safe-interop/#lifetime-annotations-in-detail
[6] "N3394: Forward Declaration of Parameters v4 (updates N3207)." Available: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3394.pdf
[7] "Variable-Size Arrays in C." Available: https://www.nokia.com/bell-labs/about/dennis-m-ritchie/vararray.pdf
[8] "GCC Mailing List." Available: https://gcc.gnu.org/pipermail/gcc-patches/2025-July/690024.html
[9] "[RFC] Lifetime annotations for C++." Available: https://discourse.llvm.org/t/rfc-lifetime-annotations-for-c/61377

[10] "N3188: Identifying array length state." Available: https://www9.open-std.org/JTC1/SC22/WG14/www/docs/n3188.htm