Proposal for C2x
WG14 N3923

Title: Forward Parameter References through Retroactive Scoping, v2 (Updates N3830)
Date: June 13, 2026
Authors: Yeoul Na (Apple) and John McCall (Apple)
Proposal Category: New Features
Target Audience: WG14 Committee, Developers and compiler engineers

Abstract: This paper proposes enabling forward references to function parameters in C through retroactive scoping. The proposal allows declarations like `void process_array(int arr[n], size_t n)` by extending the scope of names introduced in a parameter declaration list — parameter names, tags, and enumeration constants — to include the array bound expressions of array declarators of parameters declared earlier in the same list. This addresses a regression from K&R C, provides a migration path for legacy code, and establishes a standard foundation for memory safety extensions. The scope extension is confined to array bound expressions, avoiding parsing and lookup complications elsewhere in the parameter list. A phased adoption approach with diagnostics ensures backward compatibility and smooth migration.

Forward Parameter References through Retroactive Scoping

Reply-to: Yeoul Na and John McCall ({yeoul_na,rjmccall}@apple.com)
Document No: N3923
Date: June 13, 2026

Revision history

N3923 (June 2026) — Revised in response to reflector feedback ([SC22WG14.35589] and follow-ups):

- Narrowed the retroactive scoping rule to reach only into array bound expressions of earlier parameters, rather than the entire parameter list. This eliminates lookup and parsing complications in contexts such as `typeof`, bit-field widths, `alignas`, and enum initializers.
- Extended the scoping rule to cover parameter names, tags, and enumeration constants introduced anywhere in the parameter list, so that all names in the list are uniformly visible in earlier bound expressions. Note that this extension also broadens the shadowing constraint in 6.7.7.4 ¶7 to cover tag and enumeration-constant shadowing, not only parameter-name shadowing. Programs that currently rely on such shadowing patterns will need to disambiguate under Phase 2.
- Provided a semantic definition of dependency between array bound expressions: A depends on B if determining A's value requires B's value to have been determined. This clarifies which patterns constitute a "circular dependency" and permits legitimate uses such as pointer-arithmetic-based bounds.
- Added an explicit constraint disallowing tag introductions inside array bound expressions (both definitions and implicit forward-declarations), removing ordering ambiguities that would otherwise require multi-phase implementations.
- Restricted the shadowing constraint to array bound expressions and generalized it to cover any name introduced later in the parameter list (not only parameters).
- Added an implementation sketch describing a two-phase strategy compatible with compilers that do not maintain an expression AST.
- Addressed the macro-composability concern raised in the thread by scoping the discussion to function prototypes.

N3830 (February 23, 2026) — Initial version.

Motivation

The C language currently lacks a way to refer to a later parameter in the specification of an earlier parameter. This directly prevents programmers from writing the bound of an array parameter when it would be given by a later parameter:

```
void process_array(int arr[n], size_t n);  // Not currently valid in standard C
```

This is a very common pattern in practice. Programmers who need to write such declarations must instead either reorder the parameters, remove the array bound, or use a language extension. Since it was possible to write these declarations in K&R C:

```
void foo(arr, n)
    size_t n;
    int arr[n];
{ ... }
```

This represents a real regression in the language and complicates some programmers' migration to a standard version of C. This expressivity gap also impacts a wide variety of memory safety extensions, many of which face a similar need to refer to parameters "out of order", and prevents them from adopting a standard solution.

In the absence of a language solution, programmers are forced to choose between unsatisfactory workarounds:

- Being unable to adopt a new version of C prevents programmers from benefiting from many other improvements in the language, such as the type safety in function calls that comes from declaring a function with a proper prototype.
- Reordering parameters is often not possible because of source and binary compatibility constraints. Even when it is possible, it may be undesirable to switch parameters around because of longstanding conventions about

their order. For example, it is very common practice in many APIs to write a size parameter after an array parameter.
- Removing an array bound removes an important piece of information from the declaration. Even when this information is not directly meaningful to the language (as with a top-level, non-static array bound on a parameter), it can be useful as implicit documentation or to enable something like a compiler warning.
- Relying on non-standard language extensions (such as GCC's forward parameter declaration syntax or compiler-specific bounds annotations) creates portability issues and fragments the ecosystem. Without a standard solution, different implementations develop incompatible approaches, making it harder for developers to write portable code and for tooling to provide consistent support across platforms.
- Developers of memory safety extensions often struggle with the lack of language consensus around this problem, making it harder and slower to deliver safety extensions to the community. Existing memory safety extensions (e.g., `void foo(void *__sized_by(n) ptr, int n);`) have had to invent ad-hoc scoping rules to enable forward references within attributes. Standardizing retroactive scoping provides a consistent foundation for such extensions and eliminates the need for each implementation to define its own rules.

This proposal enables forward referencing of parameters through retroactive scoping, allowing more natural and type-safe function declarations while providing a standard foundation for memory safety extensions.

State of the art

Forward parameter declaration

GCC supports forward declaration of parameters, though this feature has seen limited adoption in production environments. Standard proposals have been submitted to formalize this capability [1], [2], but the approach raised concerns about readability and usability. The requirement to duplicate parameter declarations (once in the forward declaration section, then again in the actual parameter list) was seen as error-prone and contrary to modern language design principles where compilers handle such details. Unlike other forward declarations in C that can occur at file scope, these forward parameter declarations must appear in a narrow window immediately before the parameter list, creating an unusual and potentially confusing pattern. This direction [1] lost consensus at the 2025 Brno meeting, and the committee voted against adopting the proposed wording from [2] at the February 2026 committee meeting.

Identifying array length state (`.identifier` syntax)

An alternative direction proposed in [3] introduces a `.identifier` syntax to name an array length directly, without introducing forward references in general-purpose expressions. Under this approach, a bound such as `int arr[.n]` names a length designator `n` that is resolved by an explicit scope-based rule depending on where the array declarator appears: within a parameter declaration, `.n` must name a parameter in the same parameter list; within a struct or union, `.n` must name a member of the same structure; at file scope, `.n` must name a file-scope variable. The proposal also covers structure flexible array members, file-scope arrays, and return types, so its scope is broader than this proposal in some dimensions.

Because the semantics of `.n` is fixed by the context in which it appears, and because `.n` is a new syntactic form that cannot be confused with an expression, the outer-scope shadowing problem that arises with expression-based forward references does not arise here.

The tradeoff is expressivity. Because `.identifier` names an array length designator directly rather than participating in an expression, bounds computed from arithmetic, `sizeof`, ternary expressions, or other operations cannot be written in this form. Legitimate patterns such as

```
char *stpecpy(char dst[dst ? end-dst : 0], char end[0], const char *restrict src);
```

(the Plan 9 [4] `strecpy`-style API, where the bound is a computed expression involving multiple parameters) are not expressible under the `.identifier` approach. Such patterns are common in real-world APIs and constitute an important use case that this proposal preserves.

This proposal and the `.identifier` approach are not mutually exclusive: the `.identifier` syntax could be adopted as a lightweight shorthand for the common case where a bound is simply a named parameter or member, alongside retroactive scoping for the general expression case. Under the current proposal's rules, `int arr[n]` and (hypothetically) `int arr[.n]` in a parameter declaration would resolve to the same parameter; the latter would only add optional syntactic disambiguation.

Retroactive scoping

Several production compilers have recently had successful experience with an alternative approach, retroactive scoping, though limited to the context of dependent attributes.

Dependent attributes: [5] proposed retroactive scoping for parameters and struct fields within dependent attributes to support memory safety extensions. While related, this proposal focuses specifically on applying retroactive scoping to array parameter sizes in the core language. This addresses the more immediate concern facing WG14 of providing a migration path from deprecated K&R style functions.

Apple Clang Implementation: Apple's Clang compiler supports forward referencing of parameters within "dependent attributes" as part of the bounds safety extension [6]:

```
// Supported in Apple Clang
void foo(void *__sized_by(n) ptr, size_t n);
void bar(int arr[] __counted_by(n), size_t n);
void *__sized_by(n) baz(size_t n);
```

In these cases, the parameter n is treated as being in scope retroactively from the beginning of the outermost declarator containing the parameter list, but only within attribute contexts. This scoping rule is limited to attributes and does not affect the core language semantics.

Deployment and Adoption:

- Deployed in production as part of Apple's bounds safety initiative
- Started getting adoption in open-source projects [7]
- No reported issues with the scoping model in practice

Structure Field Forward References: Both Clang and GCC support similar forward referencing for structure fields within dependent attributes [8]:

```
struct buffer {
    int *data __counted_by(count);  // Forward reference to 'count'
    size_t count;
};
```

These demonstrate that retroactive scoping is well-understood and implemented in multiple contexts across major compilers.

Proposal

Allow forward referencing of parameters by extending the scope of names introduced in a parameter declaration list — parameter names, tags, and enumeration constants — to retroactively include the array bound expressions of array declarators of parameters declared earlier in the same list:

```
void foo(int arr[n], size_t n); // `int arr[n]` refers to the parameter `n`
```

The scope extension is confined to array bound expressions of earlier parameters. It does not reach into other parts of a parameter declaration (type specifiers, `typeof`, bit-field widths, `alignas`, enum initializers, or default arguments), which continue to follow existing C lookup rules. This narrow scoping addresses the core motivating case — array bounds that depend on later parameters — while avoiding parsing and lookup complications in contexts where forward references would otherwise require multi-pass semantic analysis or type-directed parsing.

Intuition

A bound expression is evaluated as if it appears at the end of the parameter list, seeing the full environment introduced by the enclosing declaration. Constraints in 6.7.7.4 ¶6 and ¶7 reject the specific configurations where this end-of-list resolution would silently differ from a naive source-order reading, forcing the programmer to disambiguate or restructure the declaration. These constraints are guardrails against silent meaning-changes, not a separate lookup rule.

Backward compatibility consideration

A potential issue arises when an outer-scope variable has the same name as a parameter:

```
int n;
void foo(int arr[n], size_t n);
```

If such code already exists, this proposal would change its behavior—the array size would refer to the parameter n rather than the outer n. **However, this pattern likely represents a mistake where the developer intended to use the parameter anyway.**

The purpose of the constraint violation in 6.7.7.4 ¶7 is to prevent such silent semantic changes: any program in which the outer-scope resolution differs from the parameter-scope resolution is rejected outright, forcing the programmer to disambiguate rather than allowing the meaning to shift depending on the C version. To ensure this transition is gradual and evidence-based, we propose the following adoption path:

1. Phase 1: Encourage implementations to add diagnostics and gather field data on existing code patterns
2. Phase 2: Make this pattern a constraint violation (require disambiguation or produce an error)
3. Phase 3: Adopt the change in the standard

On macro composability. A concern raised on the reflector is that a constraint violation on outer-scope shadowing can interact poorly with macros that generate code using common identifier names — such a macro could compile in most contexts but fail when instantiated where a colliding outer-scope name happens to be visible. The constraint here is scoped narrowly (function parameter lists only, and only when a name introduced in the list is forward-referenced in an earlier bound expression), so its reach is smaller than a general shadowing constraint would be. However, we do not have strong data on how commonly this pattern arises in function-prototype contexts specifically. Phase 1's warning-based deployment is intended in part to gather this evidence before Phase 2 commits to the constraint; if the data indicates significant harm to real code, the design should be revisited — options include downgrading Phase 2 to a persistent warning, providing explicit disambiguation syntax, or scoping the constraint more narrowly.

Implementation experience for Phase 1

At the 2025 Brno meeting, the committee did not object to implementations adding warnings for potentially confusing cases where an array parameter size refers to a global variable or constant but the programmer could have meant another parameter declared later. Based on this direction, Clang is implementing a warning to diagnose cases where an array parameter's size could ambiguously refer to either a global variable or another parameter. Note that while -Wshadow already warns about general shadowing cases including this one, it is not widely used in practice due to being overly noisy. A more targeted warning specific to this context may provide actionable diagnostics without the false positive burden.

Circular dependency

A parameter's array bound expression may depend on another parameter's complete type only when the expression requires that complete type — for example, when applying `sizeof` to a parameter of struct or array type. Mere use of a parameter's identifier as a value (in arithmetic, comparisons, pointer subtraction, etc.) does not require the parameter's complete type; the declarator shape (whether it is a pointer, integer, etc.) suffices.

A cycle in the resulting dependency graph is a constraint violation. For example:

```
void foo(int arr1[sizeof(arr2)], int arr2[sizeof(arr1)]); // constraint violation
```

Here `sizeof(arr2)` requires `arr2`'s complete type, which requires `arr1`'s complete type — creating a cycle.

Self-references that do not require the parameter's own complete type are permitted. For example:

```
char *stpecpy(char dst[dst ? end-dst : 0], char end[0], const char *restrict src);
```

`dst`'s bound uses `dst` and `end` only as pointer values (for null-check and pointer subtraction). Neither use requires `dst`'s complete type, so no cycle exists.

Non-cyclic chains of forward references are also permitted. For example:

```
void f(int a[sizeof(*pb)], int (*pb)[n], int n);
```

`a` depends on `pb`'s type, which depends on `n`, but the dependency graph is acyclic.

Rationale: The types of parameters must be determinable in a well-defined order. Circular dependencies make it impossible to determine the complete type of any parameter involved in the cycle. Distinguishing between references that require complete types and those that only need declarator shapes preserves useful patterns (such as pointer-arithmetic-based bounds) while rejecting truly unresolvable specifications.

Tag introductions in array bound expressions

Existing C permits tag and enumeration definitions inside array bound expressions (for example, `int a[sizeof(struct foo { int x; })]`). Combined with the retroactive scoping proposed here, such patterns create ordering ambiguities between the parameter list and its bound expressions that significantly complicate the implementation and can produce parse-order-dependent semantics.

This proposal disallows introducing new tags inside array bound expressions, either by definition or by implicit forward-declaration. Bound expressions may still reference tags declared in enclosing scopes or introduced by parameter declarators elsewhere in the same parameter list. The affected patterns are exotic in practice, and the restriction allows implementations to treat array bound expressions as opaque token sequences during the first pass over the parameter list.

Implementation sketch

An implementation can adopt a straightforward two-phase strategy:

- **Phase 1 (declarator walk):** Walk the parameter list in source order. For each parameter, process the declaration-specifiers and declarator normally — tags and enumeration constants introduced by declarator type-specifiers go into the prototype scope in source order. Skip the tokens of any array bound expression (brace-match through `[ ... ]` and record the token range). Register the parameter identifier. Continue to the next parameter.
- **Phase 2 (bound expression parsing):** At the end of the parameter list, parse each recorded bound expression's tokens.

Lookup in Phase 2 uses the extended scope from 6.2.1. All parameters, tags, and enumeration constants introduced anywhere in the parameter list are visible for both ordinary-identifier and tag lookup — regardless of source-order position within the list.

Constraint checks are applied as separate post-lookup steps:

- **6.7.7.4 ¶6 check (tag introductions in bounds):** When a tag specifier appears in a bound expression, check whether the tag would have been visible if the specifier were resolved at its source position — i.e., whether the tag was declared in an enclosing scope or by a declarator earlier in source order than the bound's parameter. If not, the specifier would introduce a new tag (by definition or by implicit forward-declaration) and is a constraint violation.
- **6.7.7.4 ¶7 check (shadowing):** When an identifier reference in a bound resolves via the extended scope to a name introduced later in the same parameter list, check whether a same-named identifier is visible in an enclosing scope. If so, it is a constraint violation.

Both checks can be implemented by associating each name introduced in the parameter list with a source-order index (essentially the parameter number of the declarator that introduced it). When resolving a reference in a bound expression, compare the resolved name's index to the bound's parameter index; this is O(1) per reference and requires only a small amount of extra state per name.

Cycles among array bound expressions can be detected during Phase 2 using standard dependency-graph techniques or by marking each bound as "in progress" while it is being resolved and reporting a cycle on re-entry.

All array bound expressions are parsed in Phase 2 uniformly, including those with no forward references; the additional cost of deferring their parsing is bounded (token-range recording during Phase 1, one re-entry into the expression parser during Phase 2).

Proposed wording

The following changes to the C standard are proposed to enable retroactive parameter scoping:

6.2.1 Scopes of identifiers, type names, and compound literals

[. . .]

7 Structure, union, and enumeration tags have scope that begins just after the appearance of the tag in a type specifier that declares the tag. Each enumeration constant has scope that begins just after the appearance of its defining enumerator in an enumerator list. An ordinary identifier that has an underspecified definition has scope that starts when the definition is completed; if the same ordinary identifier declares another entity with a scope that encloses the current block, that declaration is hidden as soon as the inner declarator is completed.

8 As a special case, a type name (which is not a declaration of an identifier) is considered to have a scope that begins just after the place within the type name where the omitted identifier would appear were it not omitted. A compound literal (which is an expression that provides access to an anonymous object) is associated with the scope of the type name used in its definition; that scope is either file scope, function prototype scope, or block scope.

9 As a special case, the scope of a parameter name, tag, or enumeration constant declared in a parameter declaration list, whether in a prototype or in a function definition, extends backward to include the array bound expressions of array declarators of parameters declared earlier in the same list.

[. . .]

6.7.7.4 Function declarators

Constraints [. . .]

4 A parameter declaration shall not specify a void type, except for the special case of a single unnamed parameter of type void with no storage-class specifier, no type qualifier, and no following ellipsis terminator.

5 Within a parameter declaration list, an array bound expression A is said to depend on an array bound expression B if determining the value of A semantically requires the value of B to have been determined (as when A applies sizeof to a type whose determination requires B). Merely using a parameter's identifier as a value does not by itself constitute such a dependency. Such dependencies shall not form a cycle.

6 Within a parameter declaration list, an array bound expression shall not contain a struct-or-union-specifier or enum-specifier that introduces a new tag, either by defining the tag with a body or by implicitly declaring the tag where no declaration of the tag is visible in an enclosing scope or from a declaration appearing earlier in the same parameter list.

7 Within a parameter declaration list, if an identifier used in a parameter's array bound expression refers to a name declared later in the same list, no declaration of an identifier with the same name shall be visible from an enclosing scope.

Future Work

While this proposal focuses on enabling forward parameter references within parameter lists, a related extension could build on this foundation:

Extended Scope for Return Types

Extend function prototype scope to support return types that depend on parameters, enabling both core language features and memory safety annotations:

Declarations of functions returning pointers to variable-length arrays:

```
int (*foo(size_t n))[n];  // Function returning pointer to array of n ints
```

Dependent attributes on return types:

```
void *__sized_by(n) bar(size_t n);      // Return pointer with size dependent on parameter
ptr_type_t __counted_by(n) baz(size_t n);  // Return pointer with size dependent on parameter
```

Currently, function prototype scope terminates at the end of the function declarator, preventing parameters from being referenced in return type specifications. Supporting these use cases would require extending the scope to cover the entire function declaration, allowing parameters to be referenced both in declarator portions that follow the parameter list (such as array bounds) and in attributes and type qualifiers that precede it.

Acknowledgements

We would like to thank the participants of the WG14 reflector discussion of N3830 in March 2026 — Joseph Myers, Martin Uecker, Aaron Peter Bachmann, Christopher Bazley, Alex Celeste, Alejandro Colomar, Thiago Adams, and Jens Gustedt — whose feedback shaped the design choices in this revision, including the narrowing of the retroactive scoping rule to array bound expressions, the semantic definition of dependency, the treatment of tag introductions within bounds, and the implementation strategy for compilers that do not maintain an expression AST.

References

[1] "Alternative syntax for forward declaration of parameters." Available: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3433.pdf

[2] "Alternative syntax for forward declaration of parameters (v2)." Available: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3681.pdf

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

[4] "strcat(3): concatenate, copy, compare, index, tokenize strings." Available: https://9fans.github.io/plan9port/man/man3/strcat.html

[5] "Dependent Attributes, v2 (Updates N3656)." Available: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3796.pdf

[6] "-fbounds-safety: Enforcing bounds safety for C." Available: https://clang.llvm.org/docs/BoundsSafety.html

[7] "libwebp -fbounds-safety adoption commit." Available: https://github.com/webmproject/libwebp/commit/ed054141687596753c8a6fbf145c452f07bf2673

[8] "Compiler Explorer: __counted_by support in GCC and Clang." Available: https://godbolt.org/z/qeEnsKPz3