Proposal for C2x
WG14 N3830

Title: Forward Parameter References through Retroactive Scoping
Date: January 26, 2026
Authors: Yeoul Na (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 parameter scope to retroactively start at the beginning of the parameter 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. A phased adoption approach with diagnostics ensures backward compatibility and smooth migration.

Forward Parameter References through Retroactive Scoping

Reply-to: Yeoul Na (yeoul_na@apple.com)
Document No: N3830
Date: February 23, 2026

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.

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: [3] 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 [4]:

```
// 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 [5]
- 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 [6]:

```
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 parameters to retroactively start at the beginning of the parameter list:

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

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.** To address this concern, we propose a gradual 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

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 issue

Circular dependencies between parameter declarations are inherently invalid and should be diagnosed by implementations. For example:

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

Such patterns are a constraint violation as they create unresolvable type dependencies. Implementations should detect and report these as errors.

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.

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.19 As a special case, if the identifier appears within the list of parameter declarations either in a prototype or a function definition, the scope of the identifier retroactively starts at the beginning of the parameter declaration 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, the types of parameters shall not have circular dependencies. Determining the complete type of a parameter shall not require knowledge of its own type (directly or transitively through other parameters).

6 Within a parameter declaration list, if an identifier used in a parameter's type refers to a parameter declared later in the same list, and an identifier with the same name is visible in an enclosing scope, it is a constraint violation.

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

I would like to acknowledge John McCall for his valuable feedback on this work.

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

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

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

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