

**Proposal for C2y**  
**WG14 N3990**

**Title:** Forward Parameter References through Retroactive Scoping, v3 (Updates N3923)

**Date:** October 12, 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 the parameters in that list, including a parameter’s own. 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: N3990

Date: October 12, 2026

## Revision history

**N3990 (October 2026)** — Revised in response to Joseph Myers’ comments on N3923 and committee feedback from the Ottawa meeting:

- Replaced the semantic definition of dependency between array bound expressions (6.7.7.4 ¶5) with a syntactic definition and corrected the circular-dependency example.
- Specified exactly which array bound expressions the scope extension reaches (6.2.1 ¶9).
- Extended the scope rule to include a parameter’s own array bound expressions (6.2.1 ¶9).
- Broadened the constraint on declarations in array bound expressions (6.7.7.4 ¶6) to forbid declaring any identifier or type, which also covers enumeration constants.
- Added a constraint (6.7.7.4 ¶8) so that a parameter used outside the array bound expressions, as in `typeof` or a bit-field width, has a type that can be determined at that point, and described parsing bound expressions on demand.
- Split the proposed wording into Part 1, which adds only constraints and can be adopted without changing the meaning of any program, and Part 2, which adds the scope extension. Part 1’s constraints are stated in terms of array size expressions, so they do not depend on the scope extension.
- Excluded declarations within nested parameter declaration lists from the shadowing constraint (6.7.7.4 ¶7), since such names are never visible to the enclosing list’s bounds.
- Added alternative A, which replaces Part 2 and applies the scope extension only to bounds that consist of a single identifier, and described its implementation.
- Added straw polls for the parts.

**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. It was possible to write these declarations in K&R C:

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

Losing this ability 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 that of 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[], 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 Part 2 of this proposal preserves; alternative A, described under “Alternative: forward references restricted to identifiers”, does not, until it is widened to Part 2.

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 the parameters in that list, including a parameter's own:

```
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. It does not reach into other parts of a parameter declaration (type specifiers, `typeof`, bit-field widths, `alignas`, or enum initializers), 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.

### Which bound expressions are covered

The scope extension applies to the *array bound expressions* of a parameter (6.2.1 ¶9): the size expressions of the array declarators in the parameter's declarator, excluding those within nested parameter declaration lists and those in the return types of function declarators. Array sizes in the declaration specifiers, such as in structure members or `typeof` operands, are not included either.

```
void f1(int arr[n], size_t n);           // covered
void f2(int (*p)[n][m], size_t n, size_t m); // covered: both bounds
void f3(int ((*g)(void))[n], size_t n);   // not covered: bound in g's return type
void f4(void (*cb)(int a[n]), size_t n);  // not covered: nested parameter list
void f5(struct s { int a[N]; } x, enum { N = 4 } y); // not covered: structure member
```

In `f3`, the bound belongs to the return type of the function `g` points to. Such a bound would most naturally refer to `g`'s own parameters, which is the return-type extension described in Future Work, so this proposal leaves it unchanged; that extension would need to address declarations such as `void f(size_t n, int ((*g)(size_t n))[n]);`, whose `[n]` currently refers to `f`'s `n`.

In `f4`, the bound belongs to `a`, a parameter of the nested parameter declaration list. The rule applies to that list on its own, so `a`'s bound may refer to a later parameter of the nested list, but not to a parameter of the enclosing list. Each parameter declaration list thus resolves its own bound expressions at its own end, which keeps the second pass flat: a nested list's bounds are resolved at the end of the nested list, so they never have to be deferred to the end of the enclosing list, which would require reconstructing the nested scope. In `f5`, the bound belongs to a structure member declared in the declaration specifiers, so it is looked up by the existing rules and cannot refer to the enumeration constant `N` declared later.

The extension also reaches a parameter's own bound expressions, not only those of parameters declared earlier. Under the existing rules a parameter is not in scope in its own declarator, because its scope begins only after the declarator is complete, but the computed-bound pattern this proposal aims to preserve needs it:

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

Here `dst`'s bound refers to `dst` itself. Whether such a self-reference forms a cycle is decided by 6.7.7.4 ¶5, as described in “Circular dependency” below.

The shadowing constraint, 6.7.7.4 ¶7, is stated in terms of source position rather than the scope extension, so it applies to every array size expression in the list, including those in nested parameter declaration lists, in function return types, and in the declaration specifiers. A bound therefore never silently refers to an outer declaration while the same name is declared later in the list or by the bound's own declarator: either it refers to that declaration, or the program is rejected.

```
int n;
void f1(int a[n], size_t n);           // constraint violation
void f4(void (*cb)(int a[n]), size_t n); // constraint violation
void f6(typeof(int[n]) *p, size_t n);   // constraint violation
void f7(size_t n, void (*cb)(int a[n])); // OK: refers to the earlier parameter n
void f8(char n[static n ? 2 : 1]);      // constraint violation: own name n, with the outer n visible
void f9(int a[n], void (*cb)(int n));  // OK: cb's n is local to the nested list
```

References to earlier parameters, as in `f7`, are unaffected, and so are names declared only within a nested parameter declaration list, as in `f9`, since they can never be visible to the enclosing list’s bounds. The rejected declarations are valid today and are subject to the phased adoption described below.

## Intuition

A bound expression is evaluated as if it appeared 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

The proposed wording is split to match: Part 1 is Phase 2, making these patterns constraint violations without changing any program’s meaning, and Part 2 is Phase 3, adding the scope extension.

**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 an array size expression in the list uses a name that is declared later in the list or by the declarator that contains it), 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

Because a bound expression may refer to parameters declared later, it is possible to write a parameter list whose types cannot be determined in any order. 6.7.7.4 ¶5 rules these out with a dependency relation between the array bound expressions and the types of the parameters:

- A bound expression depends on the type of each parameter it names. This holds for every bound expression, including the outermost bound of an array parameter.
- The type of a parameter depends on each bound expression that remains part of that type after array parameters are adjusted to pointers.

Dependency is transitive, and a bound expression that depends on itself is a constraint violation. For example:

```
void foo(int (*p1)[sizeof(*p2)], int (*p2)[sizeof(*p1)]); // constraint violation
```

p1's bound names p2, so it depends on p2's type. That type is "pointer to array of sizeof(\*p1) int", so it depends on p2's bound, which names p1 and so depends on p1's type, which in turn depends on p1's bound. p1's bound therefore depends on itself.

A bound can also depend on itself through its own parameter:

```
void f(char (*p)[sizeof(p)]); // constraint violation
```

p's bound names p, so it depends on p's type, which is "pointer to array of sizeof(p) char" and so depends on that same bound.

The outermost bound of an array parameter is not part of that parameter's type, so a bound that names such a parameter depends on its type but on nothing further. For example, the following declaration has no cycle, even though dst's bound names dst itself:

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

dst's bound names dst and end, so it depends on their types. Both are adjusted to char \*, which contains no array bound, so the chain ends there.

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

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

a's bound depends on pb's type, which depends on the bound n, which depends on the type of the parameter n, which depends on nothing.

## Declarations 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 declaring anything inside an array size expression in a parameter's declarator, including those in nested parameter declaration lists and function return types: no identifier, including a tag or an enumeration constant, and no structure, union, or enumeration type. This includes the implicit declaration of a tag. Bound expressions may still reference tags and enumeration constants declared in enclosing scopes or 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.

## Dependencies from outside bound expressions

Parts of a parameter list other than array bound expressions can use a parameter's type, and so depend on its bound expressions. Both of the following are valid today and remain valid:

```
void g(int (*a)[2], typeof(a) b);
void h(int (*a)[2], struct s { int c : sizeof(*a); } x);
```

The types of b and x need a's bound where they appear, not at the end of the list. 6.7.7.4 ¶8 therefore requires that, at such a use, every name in the bound expressions the parameter's type depends on is already declared:

```
void f1(int (*a)[n], int n, typeof(a) b); // OK
void f2(int (*a)[n], typeof(a) b, int n); // constraint violation
void f3(int a[n], typeof(a) b, int n);    // OK: a's type is int * and depends on no bound
```

With ¶8, an implementation can parse a bound expression on demand, at the first use of its parameter outside the bound expressions, and get the same result as parsing it at the end of the list. Because a bound expression declares nothing (¶6), parsing it early does not affect the rest of the list.

### Alternative: forward references restricted to identifiers

A narrower alternative, suggested at the Ottawa meeting, applies the scope extension only to bounds of the form [ *identifier* ]. Its purpose is to land a simpler form of the feature in C2y, with the intent of widening it to Part 2 in a future revision. All other bounds keep the existing lookup rules, so they can use full expressions but cannot refer forward. This is proposed as alternative A, which replaces Part 2.

```
void f1(int a[n], size_t n);           // OK
void f2(int a[n * 2], size_t n);      // error: n is not in scope in a's bound
void f3(size_t n, int a[n * 2]);      // OK: no forward reference
```

Although the scope then depends on the form of the bound, this cannot change the meaning of any program. A bound that is not a single identifier and names something declared later in the list either fails lookup, or finds an outer declaration with the same name, which ¶7 rejects. So in every program alternative A accepts, a bound that is not a single identifier refers only to names declared earlier or in an enclosing scope, and it would refer to the same declarations under Part 2.

The alternative adds no constraint of its own, so the existing code it rejects is the same as for Part 2: the code rejected by the constraints of Part 1 (¶6 and ¶7). It also simplifies the design. A bound that is a single identifier has integer type, which contains no bound, so no cycle can form and ¶5 can never be violated, and only such bounds need to be deferred. A parameter's own name can never be such a bound, since it has pointer or array type. The constraints on shadowing (¶7) and on dependencies from outside bound expressions (¶8) are still needed, for example for `int n; void f(int a[n], size_t n);` and `void f(int (*a)[n], sizeof(a) b, int n);`.

The cost is expressiveness: `stpecpy` above cannot be written, which is the same limitation noted for the `.identifier` approach. What distinguishes alternative A is that it needs no new syntax, and that it can be widened to Part 2 later, as intended, without changing the meaning of any program it accepts, since Part 2 only extends the scope further.

### Implementation sketch

An implementation can adopt a straightforward two-pass strategy:

- **Pass 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 declaration specifiers go into the prototype scope in source order. Skip the tokens of any array bound expression (bracket-match through [ ... ] and record the token range). Register the parameter identifier. Continue to the next parameter.
- **Pass 2 (bound expression parsing):** At the end of the parameter list, parse the tokens of each recorded bound expression that has not already been parsed. If Pass 1 reaches a use of a parameter outside the bound expressions, such as in `typeof` or a bit-field width, the bound expressions that parameter's type depends on are parsed at that point instead.

Lookup in Pass 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. For a bound expression parsed on demand, lookup sees only the names declared so far, which ¶8 guarantees is enough.

Constraint checks are applied as separate post-lookup steps:

- **6.7.7.4 ¶6 check (declarations in bounds):** While a bound expression is parsed in Pass 2, a structure, union, or enumeration definition, such as `struct s { int x; }` or `enum { A }`, is a constraint violation. A reference to a tag without a definition, such as `struct s` in `sizeof(struct s *)`, refers to an existing tag if a declaration of it is visible, and otherwise implicitly declares a new tag, which is a constraint violation.
- **6.7.7.4 ¶7 check (shadowing):** When an identifier in any array size expression in the list matches a name declared in the same parameter list (outside any nested parameter list), in the same name space, that is

declared after that point or by a declarator containing the expression, check whether a declaration of that name is visible at the beginning of the list. If so, it is a constraint violation. Array size expressions that are not resolved at the end of the list — those in nested parameter declaration lists, which are resolved at the end of the nested list, and those in function return types and declaration specifiers, which are resolved where they appear — are resolved before a later declaration in the list is seen. For these, the identifiers that resolved outside the list are recorded, and the check is applied to them at the end of the list.

The ¶7 check can be implemented by associating each name introduced in the parameter list, outside any nested parameter list, with a source-order index (essentially the parameter number of the declarator that introduced it). A name triggers the check if its index is greater than the bound’s parameter index, or if it is that parameter’s own identifier. 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 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; this works the same for bounds parsed on demand.

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

**Alternative A.** Under alternative A, only bounds of the form [ **identifier** ] are deferred, which needs only a short lookahead in Pass 1. On reaching the [ of an array bound expression, the implementation looks past any **static** and type qualifiers: if the next two tokens are an identifier and ], it records the identifier without looking it up; otherwise it parses the bound immediately under the existing lookup rules. At the end of the parameter list, each recorded identifier is looked up in the extended scope, which is a single lookup rather than a re-entry into the expression parser, so no token ranges need to be recorded. A use of a parameter outside the bound expressions, such as in **typeof**, looks up the recorded identifiers its type depends on at that point, which ¶8 guarantees are already declared. Since a recorded identifier has integer type, no cycle can form, and no cycle detection is needed.

The constraint checks are the same as for Part 2. A bound that is not a single identifier is resolved where it appears, before any later declaration in the list is seen, so the ¶7 check applies to it as it does to the array size expressions in nested parameter declaration lists: the identifiers that resolved outside the list are recorded, and the check is applied to them at the end of the list.

## Proposed wording

The wording is divided into two parts. Part 1 adds only constraints: it rejects every declaration whose meaning Part 2 would change, and it changes the meaning of no program it accepts, so it can be adopted on its own. Part 2 adds the retroactive scoping on top of Part 1; it only relaxes constraints and gives meaning to declarations that Part 1 rejects. Paragraph numbers are those after both parts are applied.

### Part 1: Constraints

#### 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.

6 Within a parameter declaration list, an array size expression in the declarator or abstract declarator of a parameter shall not declare, explicitly or implicitly, an identifier (including a tag or an enumeration constant) or a structure, union, or enumeration type.

7 Within a parameter declaration list, an identifier in an array size expression in the list shall not be declared in the same name space after that expression, or by a declarator that contains that expression, by a declaration in the list other than one within a nested parameter declaration list.

## Part 2: Retroactive 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.<sup>19</sup> Any other identifier has scope that begins just after the completion of its declarator.

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 to include the size expressions of the array declarators in the declarator or abstract declarator of each parameter of the same list, including the parameter being declared, other than those within the parameter type list of a function declarator in that declarator and those whose array declarator contains a function declarator (the *array bound expressions* of the list).

[...]

#### 6.7.7.4 Function declarators

##### Constraints

[...]

5 Within a parameter declaration list, an array bound expression *depends on* the type of each parameter of the list that is denoted by an identifier in the expression, and the type of a parameter, after the adjustment of a parameter declared as “array of type” to “qualified pointer to type”, *depends on* each array bound expression of the list that is the size expression of an array type from which that type is derived. Dependency is transitive. No array bound expression shall depend on itself.

Replace paragraph 7 as added by Part 1 with:

7 Within a parameter declaration list, if an identifier in an array size expression in the list is declared in the same name space after that expression, or by a declarator that contains that expression, by a declaration in the list other than one within a nested parameter declaration list, no declaration of that identifier in the same name space shall be visible at the beginning of the list.

8 Within a parameter declaration list, if an identifier that does not occur in an array bound expression of the list denotes a parameter of the list, each identifier in each array bound expression on which the type of that parameter depends shall be declared before that identifier.

## Alternative A to Part 2: retroactive scoping for identifier bounds only

As Part 2, except that 6.2.1 ¶9 is replaced by:

### 6.2.1 Scopes of identifiers, type names, and compound literals

[...]

9 As a special case, the scope of a parameter name or enumeration constant declared in a parameter declaration list, whether in a prototype or in a function definition, extends to include each array bound expression of the same list that consists of a single identifier. The size expressions of the array declarators in the declarator or abstract declarator of each parameter of the same list, other than those within the parameter type list of a function declarator in that declarator and those whose array declarator contains a function declarator, are the *array bound expressions* of the list.

## Straw polls

1. Does WG14 want to adopt Part 1 into C2y?
2. Does WG14 want to adopt Part 2, on top of Part 1, into C2y?
3. Does WG14 want to adopt alternative A instead of Part 2, on top of Part 1, into C2y?

## Future Work

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

### Widening alternative A to Part 2

If alternative A is adopted, a future revision can replace its 6.2.1 ¶9 with that of Part 2, extending the scope to all array bound expressions. In every program alternative A accepts, a bound that is not a single identifier names nothing declared later in the list: such a name would either fail lookup or, if an outer declaration with the same name were visible, be rejected by ¶7. Widening the scope therefore changes no accepted program, and only gives meaning to declarations that alternative A rejects, such as `void f(int a[n * 2], size_t n);`.

### 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.

We further thank Joseph Myers for his review of N3923, and the committee for its feedback from the Ottawa meeting.

## 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>