Completing Anonymous Structures and Unions

Document Number: N3926
Author: Fred Veldmeijer, Fontys University of Applied Sciences
Date: 2026–07–15
Submitted To: WG14
Proposal Category: New Feature
Target Audience: Developers, Compiler Implementers

Abstract

C11 introduced anonymous structures and unions, whose members are considered to be members of the enclosing structure or union. This proposal extends this behavior to members of previously defined structure and union types.

Under this proposal, declaring a member of a structure or union type without an identifier creates an unnamed member. The members of this unnamed member are evaluated and accessed exactly as they are for existing anonymous types.

Because the C grammar already permits this construct, the change requires no new syntax; only its semantics are new. This proposal standardizes prior practice established by Microsoft Visual C, GCC, and Clang.

1. The Problem

The conventional way to embed one structure or union inside another is to declare a named member:

```
struct point {
    int x, y;
};

struct widget {
    struct point pt;
} w;
```

Accessing the nested members requires an intermediate identifier (e.g., `w.pt.x`).

C11 introduced anonymous structures and unions [1]. For these types, the members are considered to be members of the enclosing structure or union:

```
struct widget {
    struct {            // C23 anonymous struct
        int x, y;
    };
} w;
```

This permits direct access to the nested members (e.g., `w.x`).

However, this behavior is not permitted for previously defined types:

```
struct point {
    int x, y;
};

struct widget {
    struct point;       // C23 CONSTRAINT VIOLATION
};
```

To achieve direct member access, the type definition must be duplicated inline. Because each inline definition constitutes a distinct type, modifications to the original structure must be manually replicated in every copy. Failure to synchronize these definitions results in mismatched layouts, without a compiler diagnostic.

For example, Xlib defines some thirty event structures (XKeyEvent, XButtonEvent, etc.) that must begin with the same five members (`type`, `serial`, `send_event`, `display`, `window`) in a fixed order [2]. Without the ability to include an unnamed member of a previously defined type, these common members are duplicated manually in every event structure. This relies entirely on convention to maintain ABI compatibility, with no compiler enforcement.

The current language thereby creates a dichotomy: developers must either use named members, which requires intermediate identifiers, or duplicate definitions inline, which introduces maintenance hazards.

2. The Solution

This proposal eliminates this trade-off by extending the semantics of anonymous structures and unions to previously defined types without introducing new syntax. Because the grammar for this construct already exists, this change simply makes the behavior of unnamed members uniform, regardless of where their type is defined. This permits direct member access while reusing a single type definition.

The following declarations achieve the same layout and access semantics:

```
struct widget {
    struct {            // C23 anonymous struct
        int x, y;
    };
};
```

and:

```
struct point {
    int x, y;
};

struct widget {
    struct point;       // Proposed here
};
```

In both cases, `x` and `y` are considered to be members of `widget` according to the existing rules for anonymous structures. Unlike the duplicated-definition workaround described in Section 1, referencing `struct point` establishes a single, authoritative definition. Modifications to `struct point` automatically apply to all enclosing structures, eliminating both the manual effort required to synchronize multiple definitions and the risk of silent divergence.

This single definition also guarantees identical memory layout across translation units, resolving the ABI enforcement issue described in Section 1. For example, the shared prefix required by Xlib event structures can be defined once and safely embedded, relying on the compiler rather than manual convention to ensure layout compatibility.

3. Scope

This proposal is deliberately minimal: it strictly extends existing anonymous member semantics to previously defined types. It introduces no new rules for layout, lookup, or initialization. In summary, this proposal:

- permits previously defined structure and union types to be declared as unnamed members;
- applies the existing evaluation and access rules of anonymous structures and unions to these unnamed members;
- maintains the existing constraint that all member names within the enclosing structure or union must be unique, including those introduced by multiple unnamed members.

4. Rationale

This proposal extends the C23 mechanism for anonymous structures and unions to previously defined types. Five points:

1. **Completeness.** N1406 [3] restricted anonymous members in C11 to inline-defined types. No rationale for excluding previously defined types appears in that proposal or its discussion. Removing the restriction restores orthogonality between the two forms and eliminates the duplication it currently forces.

2. **Expressiveness.** A previously defined type can be embedded as an unnamed member in multiple enclosing structures or unions. With this proposal, C can express shared substructure by naming one authoritative type, rather than re-declaring its fields in each enclosing type.
3. **Precedent.** The Xlib example in Section 1 shows the problem in practice: an ABI invariant maintained by manually duplicating fields across roughly thirty structures. This proposal lets the invariant be stated once and checked by the compiler. Similar patterns occur in other large codebases.
4. **Legibility.** Duplicated field lists, like Xlib's, do not by themselves convey that they must stay in sync. A reader must consult external documentation to learn the invariant. Embedding the shared type puts that intent in the code.
5. **Existing implementations.** MSVC, GCC, and Clang already accept this syntax as an extension. Standardizing it gives programmers a conforming and portable alternative to compiler-specific extensions.

Conformance impact on other implementations is limited to relaxing one constraint; layout, initialization, and ABI rules are unchanged.

5. Existing Practice and Prior Art

Unnamed structure and union members based on previously defined types appeared in Plan 9 C in 1990 [4]. Ken Thompson implemented the feature and used it for composition. Although Thompson advocated its inclusion in C before C99 [3], C11 standardized only the inline-defined subset.

Major implementations already support unnamed members of previously defined types as an extension. MSVC provides it as a legacy extension. GCC and Clang implement and document it for compatibility under `-fms-extensions` [5, 6]. GCC also provides `-fplan9-extensions`, which permits the embedded type name to denote the substructure. This proposal standardizes only the common subset; the additional Plan 9 semantics are excluded (see Section 9).

In 2025, the Linux kernel enabled `-fms-extensions` [7]. Its broader effects led kernel developers to request a narrower flag, `-fms-anonymous-structs`, isolating this extension [8]. The requested scope is the scope standardized here.

6. Compatibility

This proposal permits declarations that are currently constraint violations. No strictly conforming programs are affected. Name lookup and initialization follow the existing rules for anonymous members. Major compilers already support the feature as an extension; this proposal standardizes existing practice.

C/C++ compatibility. C and C++ have diverged on anonymous structures since C11. This proposal generalizes that divergence; it does not create a new one.

7. Examples

Basic Usage with Designated Initializers:

```
struct point {
    int x, y;
};

struct widget {
    struct point;
    int z;
};

struct widget w = { .x = 10, .y = 20, .z = 30 };  // Flat access
```

Nested Composition:

```
struct point {
    int x, y;
};

struct size {
    int width, height;
};

struct rect {
    struct point;
    struct size;
};

struct shape {
    struct rect;
    int color;
};

struct shape s = { .x = 10, .y = 20, .width = 30, .height = 40, .color = 50 };  // Flat access
```

Typedef and Union:

```
typedef struct { int x, y; } Point;

union value {
    int i;
    double d;
};

struct object {
    Point;          // Typedef'ed member
    union value;    // Union member
};
```

Shared Subtype Across Unrelated Types:

```
struct msg_header {
    int type;
    int length;
};

struct ping_msg {
    struct msg_header;  // Shared struct
    int sequence;
};

struct data_msg {
    struct msg_header;  // Shared struct
    char payload[256];
};
```

Passing an Unnamed Member to a Function:

```
void draw_point(struct point *p);

struct point {
    int x, y;
};

struct widget {
    int id;          // Offset 0
    struct point;    // Non-zero offset
    int z;
};

struct widget w = { .id = 0, .x = 10, .y = 20, .z = 30 };

// The address of the first promoted member can be cast to a pointer to its original type:
draw_point((struct point *)&w.x);  // Yields a 'struct point *'
```

As with any structure, if the unnamed member is the initial member, existing layout rules apply: a pointer to the enclosing object (`&w`) can be directly cast to a pointer to the unnamed type.

Collision Through Nested Unnamed Members:

```
struct point { int x, y; };

struct sprite {
    struct point;       // Screen position
    int texture_id;
};

struct hitbox {
    struct point;       // Collision-box position
    int width, height;
};

struct entity {
    struct sprite;
    struct hitbox;      // ERROR: duplicate members 'x' and 'y'
};
```

Incomplete Type Cannot be an Unnamed Member:

```
struct point;       // Valid C: Incomplete type 'struct point'

struct widget {
    struct point;   // ERROR: incomplete type cannot be a member
};
```

This follows from the existing requirement that structure and union members have complete type; this proposal leaves that requirement unchanged.

8. Suggested Changes to the Standard

The following wording is illustrative. The exact wording is left to WG14 editorial review.

The grammar of the standard already permits this construct [9]. The existing production in §6.7.3.2

> *member-declaration*:
>
> *attribute-specifier-sequence*$_{\text{opt}}$ *specifier-qualifier-list member-declarator-list*$_{\text{opt}}$ **;**
>
> *static_assert-declaration*

allows a *member-declaration* to be a *specifier-qualifier-list* without a *member-declarator-list*. The semantics paragraph governing this production currently restricts it to a *struct-or-union-specifier* that does not declare a tag. This proposal removes that restriction by revising ¶16 to read:

> An unnamed member is a member declared by a *member-declaration* without a *member-declarator-list*; if its type is specified by a *struct-or-union-specifier* with no tag, it is called an *anonymous structure* or *anonymous union*. The type specifier shall be one of the following:
>
> 1. a *struct-or-union-specifier* with no tag;
> 2. a *struct-or-union-specifier* with a tag and no *member-declaration-list*, denoting a previously declared complete type;
> 3. a *typedef-name* denoting a complete structure or union type.
>
> The members of the type of an unnamed member are considered to be members of the containing structure or union, preserving the layout they have in that type. This applies recursively through nested unnamed members.

Add the following paragraph after ¶16:

> An unnamed member of structure or union type constitutes an object of that structure or union type, residing at the corresponding offset within the enclosing structure or union. For the purposes of effective type rules, the effective type of the storage occupied by these members is that structure or union type.
>
> **NOTE:** Because an unnamed member constitutes an object of its specified type, the usual pointer conversions between a structure or union object and its members apply. For an unnamed structure with at least one member, a pointer to its first member, suitably converted, points to the unnamed member object. For an unnamed union, a pointer to any of its members, suitably converted, points to the unnamed member object.

8.1 Interactions with Existing Rules

Unnamed members are subject to the same constraints and semantics as structure and union members, except where this proposal states otherwise. In particular, the following interactions do not require any wording changes:

Flexible array members. The existing constraints for structures containing flexible array members apply unchanged. A structure containing a flexible array member cannot be a member of another structure. Consequently, a previously defined structure type containing a flexible array member cannot be used as the type of an unnamed member.

Bit-fields. Bit-field layout, padding, and allocation unit behavior remain unchanged. A previously defined structure or union type containing bit-fields may be declared as an unnamed member; its bit-fields are members of the enclosing structure or union, as for an anonymous structure or union in C23.

Type qualifiers and alignment. The existing rules regarding type qualifiers and alignment specifiers on unnamed members apply unchanged. For example, declaring `const struct point;` causes the nested members to be treated as const-qualified members of the enclosing structure, and `_Alignas(16) struct point;` applies the alignment requirement to the unnamed member object. Consequently, assignment to such a member (e.g., `w.x = 42;`) is a constraint violation, exactly as it would be for a named const-qualified member (`w.pt.x = 42;`).

9. Deliberate Exclusions

The following features are intentionally excluded from this proposal:

Inline Tagged Definitions. An unnamed member may not define a new tagged type inline (e.g., `struct outer { struct inner { int x; }; };`). The tag `inner` introduced by such a definition would have visibility in the enclosing scope under C's existing rules for tagged type definitions, despite the member itself being unnamed. This proposal does not permit such definitions, avoiding this form of tag leakage; see N3925 [10] for a related discussion.

Using the Type Name as a Member Name. The type name of an unnamed member cannot be used as a pseudo-member name for access or designated initialization (e.g., `.Point = {10, 20}`). While supported by Plan 9 C extensions, allowing this would name the unnamed member, blurring the strict separation between type namespaces and member namespaces. Consequently, designated initializers must target the nested members directly (e.g., `.x = 10, .y = 20`), adhering to existing initialization rules for anonymous structures.

10. Conclusion

This proposal completes the C11 anonymous structure and union feature by allowing previously defined types as unnamed members. The change requires no new syntax and standardizes prior practice established by major implementations. By providing a practical mechanism for type composition, it replaces the maintenance hazards of manual duplication with layout guarantees enforced by the compiler.

References

1. ISO/IEC 9899:2011, Committee Draft N1570, 2011.
   Available at https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf
2. X.Org Foundation, *Xlib – C Language X Interface*. (Chapter 10 on events)
   Available at https://www.x.org/releases/X11R7.7/doc/libX11/libX11/libX11.pdf
3. WG14 N1406, *Anonymous Member-Structures and -Unions*, David Keaton, September 2009.
   Available at https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1406.pdf
4. Ken Thompson, *Plan 9 C Compilers*, Bell Labs, 1990. (Section 3.3 on unnamed substructures)
   Available at https://plan9.io/sys/doc/compiler.pdf
5. GCC 14.1 Manual, *Unnamed Structure and Union Fields*.
   Available at https://gcc.gnu.org/onlinedocs/gcc/Unnamed-Fields.html
6. Clang 19 Documentation, *Language Extensions: Microsoft Anonymous Structs and Unions*.
   Available at https://clang.llvm.org/docs/LanguageExtensions.html#microsoft-anonymous-structs-and-unions
7. Michael Larabel, *Linux 6.19 Goes Ahead And Enables Microsoft C Extensions Support*, Phoronix, December 2025.
   Available at https://www.phoronix.com/news/Linux-6.19-Enables-MS-Ext
8. Nathan Chancellor, *Switch from '-fms-extensions' to '-fms-anonymous-structs' when available*, Linux Kernel Mailing List, February 2026. Available at https://lkml.org/lkml/2026/2/23/1644
9. WG14, *Programming Languages – C*, N3854 Working Draft, March 2026.
   Available at https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3854.pdf
10. WG14 N3925, *Deprecating Type Definitions in Member Declarations*, Fred Veldmeijer, July 2026.
    Available at https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3925.pdf