# Deprecating Type Definitions in Member Declarations

**Document Number:** N3925
**Author:** Fred Veldmeijer, Fontys University of Applied Sciences
**Date:** 2026–07–07
**Submitted To:** WG14
**Proposal Category:** Deprecation
**Target Audience:** Developers, Compiler Implementers

## Abstract

C permits structures, unions, and enumerations to be defined within the member declaration of an enclosing structure or union. However, because a structure or union does not establish a scope for tags, these nested definitions leak into the enclosing scope. This behavior complicates teaching, code maintenance, tooling, interoperability, and reasoning about code. This proposal classifies such definitions as obsolescent, encouraging implementations to issue a warning, without altering existing underlying language semantics.

## 1. Introduction

The scoping of types defined within member declarations reflects a rule inherited from early C that differs from the lexical scoping used by most other languages. Kernighan and Ritchie described this scoping rule in 1988, in §A11.1 of [1]:

> The scope of a structure, union, or enumeration tag or enumeration constant, begins at its appearance in a type specifier, and persists to the end of the translation unit (for declarations at the file level) or to the end of the block (for declarations within a function).

Although the syntax appears nested, tags and enumerators introduced by these definitions have the scope of their enclosing file or block, not a scope local to the enclosing structure or union. Because they *leak* from the member declaration into the surrounding scope, this proposal refers to the phenomenon as *leakage* (classified here as *obsolescent* in normative contexts and *deprecated* informally). Leakage creates a disconnect between visual structure and actual semantics.

## 2. Problem Statement

This section illustrates four consequences of this leakage: namespace collisions invisible from the source structure, silent bugs from misapplied lexical scoping rules, maintenance hazards from scope that looks local but is not, and implications for generated code and automated assistance arising from the same underlying mismatch. These consequences demonstrate that the construct is misleading rather than merely stylistically undesirable.

### 2.1 Namespace Collisions

The syntactic nesting of the code suggests local scope, aligning with lexically scoped languages such as C++, Java, and C#, making it a common trap for both cross-language developers and automated tools. However, in C, this structural implication is false, though not in a way the compiler will flag here:

```
struct outer {
    struct inner { int x; } a;  // Nested definition of 'struct inner'
    struct inner b;             // Valid in C and C++
};
struct inner c;  // Valid in C, ERROR in C++
```

The tag `inner` leaks into the enclosing scope, making the declaration of `c` valid even though `struct inner` appears to be defined inside `struct outer`. The same leakage that permitted `c` above without complaint becomes an error once the tag is reused in an unrelated structure or union:

```
struct S {
    struct inner { int x; } a;
};
...
union U {
    struct inner { float y; } b;  // ERROR: redefinition of 'inner'
};
```

The behavior extends beyond tags; `enum` definitions are similarly affected:

```
struct download {
    enum { PENDING, COMPLETE } state;
};
...
enum approval_status { PENDING, APPROVED };  // ERROR: redefinition of enumerator 'PENDING'
```

A collision of this kind was documented in 2013, when a Clang bug filed by Mark Adler (co-author of zlib) recorded a nested enumerator colliding with an unrelated one, undetected by both the programmer and the compiler [9].

Note that nesting depth does not stop leakage. No matter how deeply hidden a definition sits, it leaks all the way to the enclosing scope:

```
struct mine {
    struct {
        struct {
            struct {
                enum { DIRT, ROCK, ORE } material;
            } deepest;
        } deeper;
    } deep;
};
...
enum music_genre { CLASSICAL, ROCK, JAZZ };  // ERROR: redefinition of enumerator 'ROCK'
```

In all these cases the dependency is invisible from the source structure: definitions that appear independent are not, reflecting a mismatch between visual structure and scope.

## 2.2 Silent Bugs from Misapplied Lexical Scoping Rules

Unlike the collisions in the previous section, which the compiler rejects outright, assuming lexical scoping rules in C can produce code that compiles cleanly but fails silently at runtime:

```
#include <stdio.h>

struct inner { int x; };

int main(void) {
    struct outer {
        struct inner { double x; } member;
    };
    struct inner v;

    v.x = 42;
    printf("%d\n", v.x);  // Valid in C++, UNDEFINED BEHAVIOR in C
}
```

Because C++ scopes nested types to their enclosing structure, this program, when compiled as C++, treats the nested `struct inner` as local to `struct outer`. The subsequent declaration of `struct inner v` therefore binds to the global definition (the one containing `int`). Indeed, when compiled as C++, this code outputs 42.

In C, however, the nested definition shadows the global tag for the rest of the enclosing block. The variable `v` therefore binds to the local definition (the one containing `double`). When compiled as C, this code is syntactically valid, but passing a `double` to `%d` causes undefined behavior, typically printing a garbage value rather than failing loudly.

### 2.3 Code Maintenance

The construct creates a maintenance hazard not because it introduces complexity, but because it hides it. A type defined inside a structure looks local, so a maintainer restructuring the outer scope, renaming members, or reorganizing a header gets no visual cue that its tag or enumerators are visible beyond its apparent lexical boundary. By placing the definition where its scope is not, the construct requires whole-program knowledge to reason about what appears to be local code. Nothing in the code signals that it will break. It may arrive as a redefinition error or as a silent misbehavior, with no indication of the cause at the point of change.

Declaring the tag or enumeration outside a structure demands the same awareness of shared visibility, but makes that visibility explicit in the source code. The definition then appears where its scope actually is, so a maintainer's local reasoning remains valid.

### 2.4 Automated Code Generation and Refactoring Tools

Large language models are trained on code drawn from many programming languages, most of which (such as C++, Java, and C#) employ lexical scoping for nested type definitions *despite otherwise resembling C in syntax and structure*. As AI-assisted development becomes more common, generated code may increasingly reflect those assumptions, particularly in constructs where C syntax suggests locality but the language semantics do not. This may increase the likelihood of code that is misleading to readers, difficult for tools to reason about, or subtly incorrect, especially when generated code is used without detailed familiarity with the underlying language rules.

## 3. Existing Coding Standards

No explicit prohibition of nested type definitions was found in the coding standards reviewed for this proposal. However, standards that emphasize scope minimization and naming clarity effectively preclude this construct:

- **Kernighan and Ritchie [1].** While not a formal standard, *The C Programming Language* served as the *de facto* standard and shaped C's idiomatic coding style. Throughout the book, whenever an outer type contains an inner type, the inner type is explicitly defined outside the enclosing type, avoiding leakage. No example in the book allows a tag's scope to diverge from where it visually appears.
- **MISRA C:2012 [2].** Rule 5.3 states: "An identifier declared in an inner scope shall not hide an identifier declared in an outer scope." Because these nested definitions leak into the surrounding scope, they can trigger MISRA violations by shadowing globally defined tags or enumerators. The construct also conflicts with MISRA's emphasis on visual clarity: lexical nesting misleadingly suggests local scope.
- **BARR-C:2018 [3].** Rule 5.1.b requires that the keywords `struct`, `union`, and `enum` are used only within `typedef` statements or in anonymous declarations. Because a `typedef` cannot appear inside a structure or union, nested type definitions are precluded entirely.
- **SEI CERT C [4]**. Rule DCL19-C recommends minimizing the scope of variables and functions. Although the rule does not address tags or enumerators directly, the underlying principle applies equally to nested type definitions.

Notably, Jens Gustedt's *Modern C* [5] observes in §6.3.3 *Nested structures* that "all `struct` declarations in a nested declaration have the same scope of visibility" and then restructures the example to place the nested definition outside the enclosing structure, under the heading "a more adequate version." This recommendation aligns with the direction of this proposal and, given that Gustedt is a member of WG14, may reflect a broader sentiment within the committee.

## 4. Real-World Prevalence

To evaluate real-world impact, we surveyed the Linux kernel (v6.18.34, 62,126 source files totalling 35,874,179 lines — nearly 36 million lines of code) as an initial corpus: large enough to be representative, and diverse enough to cover both legacy and modern C. The survey identified 3,300 nested type definitions across 601 (0.9674%) files, of which 407 (67.7%) are in `.h` files and 194 (32.3%) in `.c` files. Broken down by construct:

| Construct | Occurrences | Lines per Occurrence | Prevalence |
|---|---|---|---|
| `struct S {...}` | 2,916 | 12,303 | 0.0081% |
| `union U {...}` | 89 | 403,081 | 0.0002% |
| `enum E {...}` | 24 | 1,494,757 | 0.0001% |
| `enum {...}` | 271 | 132,377 | 0.0008% |
| **Overall total** | **3,300** | **10,870** | **0.0092%** |

The overall prevalence is 1 per 10,870 lines (0.0092%). Roughly one third of all occurrences are concentrated in MIPS register headers (~1,200 occurrences).

A similar survey of the SQLite amalgamation (v3.53.2, 4 source files totalling 321,777 lines) found 28 occurrences (26 structures, 2 unions, no enumerations), yielding an overall prevalence of 1 per 11,492 lines (0.0087%), comparable to the Linux kernel figures above. Across both corpora, the low prevalence suggests the construct serves a narrow use case, making deprecation tractable.

**Methodology.** These figures were obtained via a scan tracking brace nesting without preprocessing or parsing; they are an estimate from source patterns rather than derived from actually compiling the source. Multi-line block comments and character literals containing braces are not stripped before matching, so counts may include (a small number of) false positives from these cases. Prevalence is reported as the percentage of scanned lines containing a match.

# 5. Scope of the Proposal

This proposal targets type definitions nested within member declarations of an enclosing structure or union. Specifically, it classifies the following as obsolescent:

1. Any definition of a structure or union that introduces a tag;
2. Any definition of an enumeration, whether tagged or untagged.

Because the restriction applies to the entire *member-declaration*, it covers both direct member definitions and those nested within any of its sub-constructs, such as function prototypes:

```
struct S {
    struct T { int x; } a;              // Obsolescent: tagged struct definition
    union  U { int y; } b;              // Obsolescent: tagged union definition
    enum   V { X, Y } c;                // Obsolescent: tagged enum definition
    enum     { A, B } d;                // Obsolescent: untagged enum definition (leaks A and B)

    void (*f)(struct P { int z; } *p);  // Obsolescent: nested tagged struct definition
};
```

The same applies when the enclosing type is a union. Unlike untagged structures and unions, which leak no identifiers, untagged enumerations leak their enumerator constants into the enclosing scope and are therefore included in the deprecation.

**Unaffected Constructs.** Member declarations that do not define a new tagged type or enumeration are outside the scope of this proposal, preserving common idioms such as anonymous structures or unions:

```
struct A { int a; int b; };

struct S {
    struct S *next;       // Not affected: self-referential pointer
    struct Z *item;       // Not affected: pointer to incomplete type
    struct A c;           // Not affected: member of previously defined type
    struct { int d; } e;  // Not affected: untagged inline struct definition
    struct { float f; };  // Not affected: anonymous structure
};
struct Z { float g; float h; };
```

# 6. Rationale

While the construct offers a degree of convenience and has historical precedent, its lexical structure does not reflect its scope semantics, making its continued availability increasingly difficult to justify.

## 6.1 The Construct is Misleading

The current syntax communicates locality while the semantics create non-local visibility. As demonstrated in Section 2, this lexical nesting creates the appearance of a scope boundary that does not exist. Because the construct looks safe to refactor when it is not, this disconnect constitutes a hazard for maintainers, educators, and tooling rather than a mere stylistic quirk.

## 6.2 The Construct's Utility Does Not Justify Its Risks

The combined form — defining a tag while declaring a member — offers a degree of convenience: the type definition is co-located with its use, and the result is compact. These are real benefits, but the convenience for the original author comes at the cost of obscuring the scope boundary for every subsequent maintainer and tool that encounters it.

```
struct outer {
    struct inner { int x; } member;
};
```

The tag here may serve a documentation purpose, giving a named type to a sub-region of the structure to signal its conceptual boundary, but this purpose does not require the inline form. The construct can be rewritten as either an **externally defined tagged type**:

```
struct inner {
    int x;
};
struct outer {
    struct inner member;
};
```

making the scope explicit and preserving the documentation value of the name, or, when no tag is needed, as an **untagged inline definition**:

```
struct outer {
    struct { int x; } member;
};
```

## 6.3 The Construct is Asymmetric with `typedef`

The C grammar does not permit `typedef` declarations inside a structure or union; they must be declared outside, in the enclosing block scope or file scope. This creates an asymmetry: while C rejects type declarations via `typedef` in a member declaration list, it permits type definitions via `struct`, `union`, and `enum` specifiers within that same list:

```
struct outer {
    typedef int T;              // ERROR: Forbidden by grammar
    struct inner { int x; } a;  // Valid in C, proposed to become obsolescent
};
```

Classifying this construct as obsolescent harmonizes the language's rules, bringing nested type definitions in line with the established behavior of `typedef`.

## 6.4 C23 Redefinitions Make Leakage Less Visible

C23 relaxed the restrictions on repeated tagged definitions [6], allowing structures, unions, and enumerations to be defined multiple times in the same scope when the definitions are identical. As a side effect, some nested definitions that were previously rejected now compile without warning:

```
struct inner { int x; };
...
struct outer {
    struct inner { int x; } a;  // ERROR in C17, valid in C23
};
```

In C17, this construct was rejected because the nested definition introduced a tag that collided with the existing `struct inner` in the enclosing scope. In C23, it is valid because the definitions are identical, and the diagnostic disappears. By making the leakage less visible to the programmer, this change can reinforce the false impression that the nested `struct inner` is local to `struct outer`.

## 6.5 C23 Enlarges the Surface Area

C11 introduced several constructs that accept nested type-names, including `_Atomic` specifiers, `_Generic` association lists, alignment operators, and static assertions.

C23 expanded this surface area by introducing the `typeof` and `typeof_unqual` operators, which provide new pathways to embed type definitions inside member declarations. Moreover, the surface area continues to grow: the working draft for C2Y, N3854 [8], introduces the `_Countof` operator, which also accepts a parenthesized type-name.

Classifying type definitions as obsolescent when they appear within a member declaration maintains the visual and semantic clarity of structure declarations as the language evolves.

## 6.6 The Construct Impairs C/C++ Compatibility

The differences in scope between C and C++ are illustrated throughout Section 2. Identical source code may be accepted by one language but rejected by the other, or accepted by both while having different meanings. Deprecating this construct eliminates one source of divergence between C and C++ by removing a form whose scoping differs. This reduces the risk of cross-language surprises and improves portability between the languages.

## 6.7 Prior Art Demonstrates the Utility of Deprecation Diagnostics

As part of its multi-language support, GCC already provides a non-standard extension that can track this construct for C/C++ compatibility checking. When these cross-language warnings are enabled (using the `-Wc++-compat` flag), the compiler emits the diagnostic `warning: struct defined in struct or union is not visible in C++` for nested structure definitions, and a corresponding one for unions. This demonstrates both the feasibility and utility of the check, and provides a reference for other compilers to follow.

## 6.8 The Construct is Absent in Ada and the Wirthian Tradition

In domains that overlap with C's own — safety-critical and embedded systems — Ada goes further than lexical scoping: its record types permit no embedded type definitions at all [10]. Ada shares this discipline with the Wirthian family of languages — Pascal, Modula-2, and Oberon — in which every component's type must be declared separately, prior to the record type. This tradition, known for simplicity and clarity of design, shows the construct can be omitted without sacrificing expressiveness.

## 6.9 The Construct Complicates Tooling and AI-Assisted Development

The construct is one of the few places in C where syntactic nesting does not correspond to scope containment. Classifying this construct as obsolescent would simplify educational tooling, parser diagnostics, and IDE scope analysis. It provides tools a formal standard to reference, helping to discourage continued use in both human-written and AI-generated code.

The construct can defeat compiler diagnostics as well as human judgment. The Clang bug in Section 2.1 shows a redefinition check failing to fire because the colliding enumerator was nested inside a member declaration, a case its diagnostic logic did not anticipate. Similarly, the GCC check in Section 6.7 covers the structure and union cases but does not extend to nested enumerations, despite the underlying divergence being identical in kind.

## 6.10 Deprecation Has Low Compatibility and Implementation Risk

This proposal is limited to deprecation and introduces no new language semantics. It does not affect object layout, ABI compatibility, runtime behavior, or existing scoping rules.

Implementation impact is expected to be low. As noted in Section 6.7, GCC already diagnoses this construct for C/C++ interoperability, and the recommended diagnostic is a non-fatal warning consistent with existing behavior. The proposal merely standardises optional diagnostics and classifies the construct as obsolescent.

A warning may be enabled under existing options (for example, `-Wpedantic` or a dedicated warning flag) rather than being issued unconditionally. Existing code therefore continues to compile as before. Projects that wish to eliminate the construct can enable the diagnostic and address occurrences incrementally.

# 7. Alternatives Considered

## 7.1 Lexical Scoping

An alternative is to retain nested type definitions but restrict their visibility to the enclosing structure, as lexically scoped languages do:

```
struct outer {
    struct inner { int x; } member;
};
struct inner obj;  // Would become invalid
```

This proposal rejects lexical scoping in favor of classifying the construct as obsolescent, for the following reasons:

1. **Risk of Silent Changes.** If a nested type's scope were restricted to the enclosing structure, existing programs could silently change behavior. Consider a global `struct inner` and a function that locally defines a `struct outer` containing a nested `struct inner`:

   ```
   #include <stdio.h>

   struct inner {
       float buffer[100];
       int x;
   };

   int main(void) {
       struct outer {
           struct inner { int x; } member;
       };
       struct inner obj;              // SILENT CHANGE: global type under restricted scope

       obj.x = 42;                    // Compiles today and under restricted scope
       printf("%zu\n", sizeof(obj));  // Prints 4 today but 404 under restricted scope (IB)
   }
   ```

   The exact printed values are implementation-defined; the difference in magnitude is not. Under current language rules, the nested `struct inner` leaks into the function's block scope, hiding the global definition within the remainder of the block, and subsequent references to `struct inner` bind to the local definition. Under restricted scoping, the same declaration would instead bind to the global definition: same member name, completely different type, layout, and size — the global definition's 100-element buffer causes a substantially larger footprint. The program would still compile without diagnostic, but the meaning of the program would change despite no source modifications. Silently altering the execution semantics of existing valid programs would introduce a quiet change, posing a hazard to backward compatibility.
2. **Lack of Utility.** The primary purpose of a tag or enumerator is to allow it to be referenced elsewhere. If visibility were confined to the enclosing structure, there would be no existing C syntax to refer to a tag or enumerator outside the definition. For a structure or union, this makes the construct functionally identical to the existing anonymous structure or union, and therefore redundant; for an enumeration, no such fallback exists, since C has no anonymous enumeration equivalent. Adopting lexical scoping in C would therefore leave the construct redundant or without practical use, or require scope-qualification that C does not otherwise have.

Restricting visibility is therefore not a viable path for C.

## 7.2 Constraint Violation

An alternative to deprecation is to classify nested type definitions as a constraint violation, requiring conforming compilers to reject the construct outright. This proposal rejects that approach for three reasons:

1. **Protection of Legacy Code.** Although the construct is rare (as noted in Section 4), it does exist in active codebases, and deprecation imposes no immediate burden: no build breaks, no behavior changes, and no required migration. The warning serves as a signal to maintainers, giving library authors time to update their code without halting user compilation.
2. **No Immediate Risk.** Immediate removal is usually reserved for features that actively and frequently cause undefined behavior, memory corruption, or severe security vulnerabilities. The issue with nested type definitions is primarily one of visibility, teachability, and human factors. While it is a language anomaly that can produce undefined behavior, the construct does not generate unsafe code when used as intended. Therefore, the disruption of an immediate constraint violation is not justified by a corresponding security benefit.

3. **Precedent.** This approach is consistent with the committee's treatment of other legacy constructs. For example, the deprecation of legacy octal literals in C2Y [7] follows the same steps: acknowledging historical support while recognising that the construct warrants deprecation, and giving the ecosystem time to migrate before formal removal. Actual removal (making it a constraint violation) is typically deferred to a subsequent revision of the standard. This proposal aligns with this established practice.

Deprecation retires a form of technical debt when the cost is still manageable, rather than shifting it into a future where the construct will be more entrenched and the programmer population less familiar with its behavior.

# 8. Suggested Changes to the Standard

The following sketches the intended direction for normative wording, based on the current C standard working draft [8]. Exact phrasing is left to WG14 editorial review; this section is intended to convey intent and scope rather than to serve as final normative text.

## 8.1 Future Language Directions

Add a new subclause to §6.11 *Future language directions*:

> **6.11.x Type definitions in a member-declaration**
>
> A *struct-or-union-specifier* containing both an identifier and a *member-declaration-list*, or an *enum-specifier* that contains an *enumerator-list*, is an obsolescent feature when it appears within a *member-declaration*.

The scope of this clause is intentionally broad. By anchoring the rule to *member-declaration*, it catches not only direct type definitions, but also cases where a tag or enumerator is reachable only indirectly:

```
struct {
    static_assert(sizeof(enum { E }) >= sizeof(int));  // Obsolescent: enumerator E in member
    typeof(struct S { int x, y; }) a;                  // Obsolescent: struct S in member
    int b[sizeof(union U { int x; double y; })];       // Obsolescent: union U in member
    void (*f)(struct P { int z; } *p);                 // Obsolescent: struct P in member
};
```

## 8.2 Informative Notes

Add the following note after the description of the *member-declaration-list* in §6.7.3.2 *Structure and union specifiers*:

> **NOTE:** Defining a tagged structure or union directly within a *member-declaration* introduces the tag into the enclosing scope rather than into a scope local to the enclosing structure or union. This is an obsolescent feature (§6.11.x). The alternatives are a tagged type declared in the enclosing scope, an untagged inline definition, or an anonymous structure or union.

Add a similar note to §6.7.3.3 *Enumeration specifiers*:

> **NOTE:** Defining a (tagged or untagged) enumeration directly within a *member-declaration* introduces its enumerators (and its tag, if any) into the enclosing scope. This is an obsolescent feature (§6.11.x). The alternative is an enumeration declared in the enclosing scope.

## 8.3 Recommended Practice

To encourage migration, add the following to the new subclause 6.11.x:

> **Recommended Practice**
>
> Conforming implementations are encouraged to issue a diagnostic message for a *struct-or-union-specifier* or *enum-specifier* that is an obsolescent feature under 6.11.x. A typical diagnostic message is:
> `warning: defining a type as part of a member declaration is an obsolescent feature.`

Programs using this construct remain valid and must be accepted. Object layout, ABI, and all other semantics remain unaffected.

## 9. Future Directions

This proposal classifies the construct as obsolescent as a first step. It is the intention of the author that a future revision of this standard may make the construct a constraint violation, subject to the experience gained from the deprecation period and ecosystem feedback. No such change is proposed at this time, and all existing semantics remain unchanged.

## 10. Conclusion

This proposal targets the construct whereby structures, unions, and enumerations defined within member declarations of structures or unions leak their tags and enumerators into the enclosing scope. Although historically supported, the construct provides limited utility in practice, instead creating a persistent mismatch between lexical nesting and scope. That mismatch affects teaching, code maintenance, tooling, C/C++ interoperability, and the interpretation of both human-written and AI-generated code. This proposal takes a first step toward resolving that mismatch.

## References


1. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1988.
2. MISRA C:2012, *Guidelines for the Use of the C Language in Critical Systems*, 3rd ed., 1st rev., Nuneaton: HORIBA MIRA, 2019. Available at https://www.misra.org.uk/app/uploads/woocommerce_uploads/2021/06/pdf2-3zxsfw-lnbvtq.pdf.
3. Michael Barr, *Embedded C Coding Standard*, Germantown, MD: Barr Group (Integrated Embedded, LLC), 2018. Available at https://barrgroup.com/sites/default/files/barr_c_coding_standard_2018.pdf.
4. SEI CERT C Coding Standard, Carnegie Mellon University Software Engineering Institute. Available at https://wiki.sei.cmu.edu/confluence/display/c/DCL19-C.+Minimize+the+scope+of+variables+and+functions.
5. Jens Gustedt, *Modern C*, 3rd ed. Shelter Island, NY: Manning Publications, 2025.
6. Martin Uecker, *Improved Rules for Tag Compatibility*, WG14 Document N3037, 2022. Available at https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3037.pdf.
7. Aaron Ballman, *Obsolete Implicitly Octal Literals and Add Delimited Escape Sequences*, WG14 Document N3353, 2024. Available at https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3353.htm.
8. WG14, *Programming Languages — C*, N3854 Working Draft, March 2026. Available at https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3854.pdf.
9. Mark Adler, *Redefinition of enumerator in enum not issuing an error*, LLVMbugs Bug 15071, 2013. Available at https://lists.llvm.org/pipermail/llvm-bugs/2013-January/026970.html
10. ISO/IEC 8652, *Ada Reference Manual*, §3.8 Record Types, Ada Resource Association, 2023. Available at http://www.ada-auth.org/standards/22aarm/html/aa-3-8.html.