Enhance Clang Diagnostics

Author: Aditya Medhane
Email: <adityamedhane.dev@gmail.com>
GitHub: flash1729 (Aditya Medhane)

Compiler and Technical Experience

I started learning compilers initially by pure interest of what really happens to the code that we write. And going forward I also learnt the basics in my university course and a winter school organised by ACM India which significantly boosted my interest. I was pretty impressed with Lambda Calculus as a subject so I went on to build an app to teach the same for Swift Student Challenge organised by Apple in 2025 where it was selected among one of top 350 globally. Inside this app I wrote a Lambda Calculus Interpreter in Swift to help people directly experiment on expressions inside the learning environment itself.(GitHub).

LLVM Contributions

I have recently started contributing to LLVM. My first contribution was an NFC change to fix minor typos in the APINotes subsystem (PR#183811), which helped me get familiarized with the contribution workflow. My second PR focuses on upstreaming the BoundsSafety format part (PR#186960, related issue #183340), which provided a solid understanding of the APINotes pipeline.

Related to Clang::Diagnostics

My most recent work on Clang::Diagnostics is PR#192116, which introduces a new diagnostic in *Sema::MergeVarDecl* to detect undefined behavior when an identifier is declared with both internal and external linkage. This is a violation of C11 6.2.2p7 that's formally ill-formed in C2y (N3410). The initial patch landed and was then reverted due to documentation issues around some FIXMEs. I'm currently working on the reland (PR#193567), which is in the final steps of review. The implementation involved defining *err_internal_extern_mismatch* in *DiagnosticSemaKinds.td*, integrating the logic into *SemaDecl.cpp*, and writing test coverage for both C2y errors and legacy UB warnings. Going through the land-revert-reland cycle gave me real experience with the Sema declaration merging pipeline, the CI expectations, and how to respond to reviewer feedback.

Programming Language Expertise

- **C++:** Intermediate
- **C:** Intermediate
- **Swift:** Intermediate

Proposed Project: Enhance Clang Diagnostics

Synopsis

Clang is an open-source compiler front-end for C, C++, Objective-C, and related languages built on the LLVM backend. It offers fast compiles, low memory usage, and detailed diagnostics. This proposal addresses the "Enhance Clang Diagnostics" project officially listed on the Compiler Research Open Projects page.
The project targets five task areas:

1. Unimplemented and partially implemented C2y papers.
2. Silent semantic change detection.
3. Clang-tidy check upstreaming into Clang proper.
4. Diagnostic rewording and fix-it hints.
5. Missing diagnostics where Clang stays silent but other compilers don't.

Category 1: Unimplemented and Partially Implemented C2y Papers

Two C2y papers, N3418 and N3244, are marked as **"No"** and **"Partial"** on Clang's c_status.html. Together they contain three bounded, standard-mandated diagnostic gaps. Completing all three moves N3418 from "No" to "Yes" and N3244 from "Partial" closer to "Yes." Each one follows the same workflow I used for N3410: identify the paper, locate the check in Sema or the Lexer, define the diagnostic, write the tests, update c_status.html.

1a. N3418: UCN Formation via ## Token Pasting
N3418 ("Slay Some Earthly Demons XIV") makes the creation of universal character names via ## token pasting a **constraint violation** in C2y. Previously, this was undefined behavior. The WG14 February 2025 meeting adopted N3418 with a straw poll of **20/1/3** (N3583). The gap is also documented in GitHub issue #97741 ("-Winvalid-token-paste fails to catch UCNs which are invalid preprocessor tokens"). The reporter demonstrates that Clang preprocesses \u0000 via ## pasting without any diagnostic, while GCC correctly diagnoses the invalid UCN. The issue is open and unassigned. The fix belongs in **TokenLexer::pasteTokens()** in lib/Lex/TokenLexer.cpp. This function already re-lexes the pasted result token and checks for invalid paste results. A post-paste check for UCN-shaped results is the natural insertion point for the diagnostic. **Important:** C++26, via P2621R3, went the **opposite direction** and made UCN formation via ## well-defined. So the C2y diagnostic must be gated to C language mode only. Estimated effort: 1-2 weeks.
1b. N3244 Item 67: extern inline Function with No TU Definition
N3244 ("Slay Some Earthly Demons I") includes Annex J Item 67, which makes it a **constraint violation** to declare a function extern inline without ever providing a definition in the same translation unit. The c_status.html entry for N3244 is marked **"Partial"** with the detail text: "Clang does not diagnose an extern inline function with no definition in the TU." The test file clang/test/C/C2y/n3244.c already contains explicit scaffolding for this case:

```C/C++
// FIXME: this function should be diagnosed as it is never defined in the TU.
extern inline void never_defined_extern_inline(void);

// While this declaration is fine because the function is defined within the
TU.
extern inline void is_defined_extern_inline(void);
extern inline void is_defined_extern_inline(void) {}
```

Both the failing case and the passing case are already written. The gap is purely the missing diagnostic and the bookkeeping to track it. The diagnostic must be deferred until the end of the translation unit, since the definition could appear anywhere in the TU. The hook is Sema::ActOnEndOfTranslationUnit() in lib/Sema/Sema.cpp, which is the same pattern Clang uses for unused-static-function warnings. A tracking set in Sema (populated when extern inline declarations are encountered, cleared when a definition is found) drives the deferred diagnostic. Estimated effort: 1-2 weeks. The test scaffolding is already in place; the implementation is a tracking set plus an end-of-TU scan.

1c. N3244 Item 69: _Alignas Mismatch on Redeclarations

N3244 Annex J Item 69 makes mismatched _Alignas specifiers on redeclarations a **constraint violation**. The c_status.html detail text reads: "Clang accepts and rejects redeclarations with/without an alignment specifier, depending on the order of the declarations." Clang is currently wrong in both directions:

- **Under-diagnoses (false negative):** Adding an alignment specifier on a redeclaration where the original had none should be an error. Clang silently accepts it.
- **Over-diagnoses (false positive):** Redeclaring without an alignment specifier when the first declaration had one is valid under C2y. Clang incorrectly rejects it.

The test file n3244.c contains **5 FIXMEs** for Item 69 (plus 1 for Item 67), covering distinct scenarios:

Scenario	Current Clang Behavior	Correct C2y Behavior
No alignment on original, _Alignas(8) on redecl	**Silently accepts**	Constraint violation
_Alignas(8) on original, no alignment on redecl	**Incorrectly rejects**	Valid
_Alignas(alignof(T)) on original, no alignment on redecl	**Incorrectly rejects**	Valid
Definition with _Alignas(8), subsequent decl without	**Incorrectly rejects**	Valid

Scenario	Current Clang Behavior	Correct C2y Behavior
Definition without alignment, redecl with compatible _Alignas	**Incorrectly rejects**	Valid

The fix belongs in MergeVarDecl in lib/Sema/SemaDecl.cpp, where declaration merging occurs. The alignment comparison logic needs to distinguish between "no specifier" (inherits natural alignment) and "specifier present" (explicit alignment), rather than treating the absence of a specifier as a mismatch. Estimated effort: 2-3 weeks. Fixing both false negatives and false positives makes this more involved than a point-of-use diagnostic.

Summary

Item	Standard Paper	c_status.html Status	Expected Fix Location	Effort
N3418 (UCN via ##)	N3418	**"No"**	TokenLexer::pasteTokens()	1-2 weeks
N3244 Item 67 (extern inline)	N3244	**"Partial"**	ActOnEndOfTranslationUnit()	1-2 weeks
N3244 Item 69 (_Alignas mismatch)	N3244	**"Partial"**	MergeVarDecl in SemaDecl.cpp	2-3 weeks

Category 2: Silent Semantic Change Detection

This one's the core deliverable from the CppAlliance project listing. The problem is reproducible in this Godbolt link. When a type changes from *std::string* to *const char** during a library update, code using *auto* and == silently flips from value comparison to pointer comparison. No compiler catches this today.

The bug in detail:

```C/C++
std::string b = "bob";
const char* needle = b.c_str(); // points into b's heap buffer

std::vector<std::string> v1 = {"alice", "bob", "carol"};
std::vector<const char*> v2 = {"alice", "bob", "carol"};
```

```
auto it1 = std::find_if(v1.begin(), v1.end(), [&](const auto& s) { return s == needle; }); //value comparison, correct
auto it2 = std::find_if(v2.begin(), v2.end(), [&](const auto& s) { return s == needle; }); //pointer comparison, always wrong here
```

The generic lambda *[&](const auto& s) { return s == needle; }* is identical in both calls. Yet *it1* finds *"bob"* and *it2* does not.

The AST makes this very clear:
The generic lambda is a *FunctionTemplateDecl*. It gets instantiated separately for each *find_if* call, and the two instantiations resolve *==* to completely different operations.
For *v1* (*vector<std::string>*), the instantiated *operator()* produces:

```
None
CXXOperatorCallExpr 'bool' '==' adl
├─ DeclRefExpr ... Function 'operator=='
| 'bool (const basic_string<char,...>&, const char*)'
├─ DeclRefExpr 's' 'const std::basic_string<char>&'
└─ ImplicitCastExpr <LValueToRValue> 'const char*'
```

This is a *CXXOperatorCallExpr*. ADL found *std::string*'s overloaded *operator==*, which performs lexicographic character-by-character comparison. Correct.
For *v2* (*vector<const char*>*), the same lambda produces:

```
None
BinaryOperator 'bool' '=='
├─ ImplicitCastExpr <LValueToRValue> 'const char*'
└─ ImplicitCastExpr <LValueToRValue> 'const char*'
```

This is a plain *BinaryOperator*. No ADL, no overload resolution, no function call. Both sides are raw *LValueToRValue* loads and the CPU just compares two memory addresses. Since *needle* points into *b*'s heap buffer while *v2[1]* is a string literal in the read-only data segment, the addresses are always different even though the content is *"bob"* in both cases.

Why this is hard to diagnose:
In the uninstantiated lambda template, the AST reads:

```
None
BinaryOperator '<dependent type>' '=='
├─ DeclRefExpr 'const auto' ... s
└─ DeclRefExpr 'const char*' ... needle
```

The type is *<dependent type>*. Clang has deferred all semantic analysis until instantiation. No diagnostic can fire at the template definition site because *==* has no concrete semantics yet. The warning has to fire at instantiation time, after overload resolution has (or has not) found an *operator==*.

Exploring the diagnostic approach:
CheckCompareOperands() in *SemaExpr.cpp* is where *diagnoseTautologicalComparison()* and *warn_stringcompare* already fire. It has access to both operand types and the full *Expr** nodes, so it's the natural place to explore adding a check for this pattern. The existing *warn_stringcompare* handles the case where one operand is a *StringLiteral*. What's missing is the case where *neither* operand is a literal but both are *const char**. After reading through the AST dumps, the trigger condition I'm thinking about is: a *BinaryOperator ==* (not *CXXOperatorCallExpr*) where both operands resolve to *const char** and neither is a null-pointer constant (already checkable via *Expr::isNullPointerConstant()*, which *CheckCompareOperands()* uses today) nor derived from a *StringLiteral* decay.
Two-phase approach:

- **Phase 1:** Implement *warn_char_pointer_compare* with null-pointer and string-literal suppressions. Verify the template instantiation path fires correctly. Land tests in *clang/test/Sema/*.
- **Phase 2:** Run against LLVM's own codebase, document false positive patterns (especially in *llvm/include/llvm/ADT/StringRef.h* and table-driven parsers where pointer identity is deliberate), write RFC for warning group placement.

No other compiler catches this pattern today, which makes it a pretty unique deliverable. Effort: 3-4 weeks.

Category 3: Clang-Tidy Upstreaming

Check 1: *bugprone-multiple-statement-macro → -Wmultistatement-macros*
This check detects macros that expand to multiple statements used as the body of an *if*, *else*, *for*, *while*, or *do* without braces. Only the first statement is conditionally executed; the rest run unconditionally:

```C/C++
#define INCREMENT(x, y) (x)++; (y)++
if (cond) INCREMENT(a, b); // (b)++ always runs, silent bug
```

I traced through the clang-tidy implementation (*MultipleStatementMacroCheck.cpp*, ~103 lines). It checks whether the *Inner* statement and the *Next* statement after a control-flow body share the same macro expansion range using *SourceLocation* comparison. If both expand from the same macro token, only one is guarded by the condition.
Clang already has the infrastructure for this. *SemaStmt.cpp* uses *SourceManager::isMacroBodyExpansion()* (line 182) to suppress warnings inside macro bodies, and *getExpansionLoc()* to compare macro origins. The *-Wparentheses* warning in the same file follows an analogous pattern. The natural hooks are *Sema::ActOnIfStmt* (line 950), *ActOnWhileStmt* (line 1790), and *ActOnForStmt* (line 2277), where Clang finishes building the statement body.
GCC already ships *-Wmultistatement-macros*, so there's direct competitive parity value and low reviewer resistance. The new warning group would mirror GCC's name exactly. Since this is purely structural with no interaction with templates, type analysis, or lifetime tracking, the false positive risk is essentially zero.

Check 2: Extending *-Wclass-memaccess* (from *bugprone-raw-memory-call-on-non-trivial-type*)

This one flags calls to C raw memory functions (*memset*, *memcpy*, *memmove*, *realloc*) when the destination or source type is non-trivial in C++ terms, meaning it has a user-provided constructor, destructor, or non-trivial copy. Using these on C++ objects bypasses constructors/destructors and silently causes UB:

```C/C++
std::string s;
memset(&s, 0, sizeof(s)); // corrupts internal string state silently
```

The clang-tidy implementation (*RawMemoryCallOnNonTrivialTypeCheck.cpp*, ~125 lines) checks the pointee type of the destination pointer argument. If the type isn't trivially copyable or trivially destructible, it warns.
Clang already has most of this. The existing *-Wclass-memaccess* warning lives in *SemaChecking.cpp* inside *Sema::CheckMemaccessArguments* (line 10790). It already dispatches on *Builtin::BImemset*, *Builtin::BImemcpy*, and *Builtin::BImemmove*. At line 10905, the condition checks *!PointeeTy.isTriviallyCopyableType(Context)*. After digging through both implementations, I found two gaps:

1. **Missing builtins:** *realloc* isn't in the dispatch table. Adding *Builtin::BIrealloc* to the existing switch is mechanical.
2. **Narrow type trait check:** The current check uses *isTriviallyCopyableType()* only. The clang-tidy version also flags *!isTriviallyCopyConstructible()* and *!isTriviallyDestructible()*. These are already available as *RecordDecl* methods, so extending the condition is incremental.

Estimated effort: 3-4 weeks (2 upstream patches).

Category 4: Diagnostic Rewording and Fix-It Hints

These are open issues where Clang's diagnostic text is factually wrong, misleading, or missing a fix-it hint. Each one is self-contained: find the diagnostic string in *DiagnosticSemaKinds.td*, rewrite it or add a *FixItHint*, update the lit tests, and make sure nothing regresses.

All issues below were verified as open and unassigned at the time of writing.

- **[#171074](https://github.com/llvm/llvm-project/issues/171074): Confusing diagnostic about implicit use of *this* involving immediate-escalating function.** The *note_constexpr_this* diagnostic says implicit *this* usage is "only allowed within the evaluation of a call to a 'constexpr' member function." That wording is factually wrong. *constexpr* is not a requirement for *this* usage. Filed by [@Sirraide] (LLVM member). Targeted string fix, fast turnaround.
- **[#92810](https://github.com/llvm/llvm-project/issues/92810): Confusing diagnostic for "address of consteval function" on arity mismatch.** Calling a *consteval* function with the wrong number of arguments produces a secondary error about "cannot take address of consteval function" when no address is being taken. Filed by [@MitalAshok]. Clean reproducer, zero comments since filing.
- **[#180429](https://github.com/llvm/llvm-project/issues/180429): Diagnostic location inverted for duplicate *default* labels in a switch.** When two *default:* labels appear in a single switch statement, Clang points the primary error at the *first* (valid) occurrence and the "previous definition" note at the *second* (duplicate) one. That's the exact opposite of what's correct. Labeled *confirmed*. A PR (#180447) was opened in February 2026 but has had no review activity since March 2026. Clean pick-up opportunity.
- **[#5757](https://github.com/llvm/llvm-project/issues/5757): Fix-it for missing *#include* directives.** When Clang diagnoses an implicit function declaration (e.g., calling *printf* without *<stdio.h>*), it already knows which header is needed but doesn't offer a fix-it to insert the *#include*. Open since 2009, labeled *confirmed* and *quality-of-implementation*. The infrastructure to suggest header paths already exists in *Preprocessor::getHeaderSearchInfo().suggestPathToFileForDiagnostics()*. What's missing is the *FixItHint::CreateInsertion* at the right source location. Probably the highest user-facing value in this category.

Estimated effort: 3-4 weeks.

Category 5: Missing Diagnostics

These are cases where Clang stays silent on patterns that other compilers already catch. Each cluster shares a root cause, so one fix closes multiple issues.

All issues below were verified as open and unassigned at the time of writing.

5a. Macro operator precedence warning (filed by @zygoloid)

- **[#184672](https://github.com/llvm/llvm-project/issues/184672): Warn on operators in macro expansion with external operands.** Filed by Richard Smith ([@zygoloid], former Clang lead), labeled *good first issue* and *clang:diagnostics*. If *+* is inside a macro like *#define FOO 2+3* but *4* is outside in *FOO*4*, Clang should warn because the expansion produces *2+(3*4)* which is almost never intended. @zygoloid notes "this seems like something we could quickly and easily check for with our existing source location representation." A linked PR exists but has been stalled with no follow-up. Clean pick-up opportunity.

5b. Self-initialization cluster

These two issues describe the same root cause: Clang silently accepts reads from uninitialized variables.

- **[#19252](https://github.com/llvm/llvm-project/issues/19252): No warning for self-initialization (*int x = x;*).** A 12-year-old confirmed bug. GCC warns. Clang is silent. Canonical uninitialized-read example, directly connects to C++26 erroneous-behavior semantics (P2795). CC'd to [@zygoloid].
- **[#191497](https://github.com/llvm/llvm-project/issues/191497): Clang accepts self-variable value assignment without any warnings.** Filed April 2026. Reports *int self = self;* compiles silently even with *-Wall -Wuninitialized -Wextra*. Duplicate of the pattern in #19252, confirms the issue persists on Clang 18. One fix closes both.

5c. *-Wunused-value* silent for macro-expanded expressions

These two issues share the same root cause: *DiagnoseUnused()* calls *SourceMgr.isMacroBodyExpansion()* and skips the warning for macro-expanded expressions.

- **[#106867](https://github.com/llvm/llvm-project/issues/106867): *-Wunused-value* does not warn about *literal;* if the literal is expanded from a macro.** Labeled *clang:diagnostics*, *enhancement*, and *false-negative*. Writing *true;* produces no warning because *true* expands from *<stdbool.h>*. The reporter verified this is a regression from Clang 3.0.0.
- **[#142614](https://github.com/llvm/llvm-project/issues/142614): Different *-Wunused-value* output in preprocessed source code.** Filed by @iii-i (LLVM member). *(long)f();* warns directly but not when wrapped in *#define MACRO() (long)f()*. Root cause: *DiagnoseUnused()* checking *isMacroBodyExpansion()*. One fix addresses both issues.

Estimated effort: 4 weeks.

Timeline

Standards patches come first because they're the most bounded and I've already done this workflow for N3410. The semantic change diagnostic comes early to maximize review iterations on the hardest piece. Clang-tidy upstreaming fills the middle. Rewording and missing diagnostics provide steady landed patches throughout.

Month	Focus	Deliverables
Bonding	Project planning + ramp-up	Finalize all proposed items, deepen knowledge of the Sema pipeline and diagnostic infrastructure through reading and experimentation, get comfortable with Compiler Explorer for AST inspection and reproducer validation
1	C2y standards patches (Cat 1)	N3418 (UCN via paste), N3244 Item 67 (extern inline)
2	C2y completion + semantic change (Cat 1 + 2)	N3244 Item 69 (_Alignas mismatch), *warn_char_pointer_compare* Phase 1
3	Semantic change FP analysis + clang-tidy (Cat 2 + 3)	*warn_char_pointer_compare* Phase 2 (FP analysis + RFC), *-Wmultistatement-macros* upstream
4	Clang-tidy + rewording (Cat 3 + 4)	*-Wclass-memaccess* extension, #171074 and #92810 rewords
5	Missing diagnostics + remaining (Cat 4 + 5)	#180429 (default label), #5757 (include fix-it), #19252/#191497 (self-init), #184672 (macro precedence), #106867/#142614 (-Wunused-value)

Testing Strategy

Every diagnostic change gets:

- **Targeted *lit* regression tests** that verify the diagnostic fires on the intended pattern with correct wording and source locations.
- **Negative tests** making sure no false positives show up on similar-looking but correct code.
- **Edge case coverage** for macro interactions, template instantiations, and system header contexts. These are the places where diagnostics historically break.
- **Compile-time measurement** on representative codebases for anything that runs during normal compilation, especially the upstreamed clang-tidy checks. If a patch introduces measurable regression, it gets reworked before landing.

Stretch Goals

If the core work runs ahead of schedule, or if review latency on primary items creates idle time, these are natural extensions:

Diagnostic rewording (from Category 4 overflow):

- **[#36285], [#74212], [#40382]**: complex rewording issues involving overload resolution, deleted conversions, and lambda capture semantics. These touch deeper compiler paths and carry slower review tracks.
- **[#75248] + [#75663]**: missing *typename* note in parameter pack expansion. Deep in dependent-type diagnostic coverage.

Missing diagnostics (from Category 5 overflow):

- **[#130586], [#149802], [#48943]**: bitshift UB causing silent codegen surprises. Involves optimizer interaction and has an unresolved meta-discussion.
- **[#141908]** and sub-issues (**[#141791], [#141103]**): system header warning suppression. Open architectural problem with competing solution approaches.

Clang-tidy upstreaming (remaining checks):

- *bugprone-use-after-move*
- *bugprone-multiple-new-in-one-expression*

Documentation

As the final deliverable, I'll write a report documenting:

- Which diagnostics were improved and why.
- Which standard papers were closed.
- Which clang-tidy checks were migrated, including any behavioral differences from the original.
- Which open issues were resolved.
- Any remaining work or follow-up items for future contributors.

Why Me

I've already been through the full lifecycle of a Clang diagnostic patch, including a land, revert, and reland. I am familiar with how the diagnostic infrastructure works, from *.td* file definitions to Sema integration to *lit* test validation.

But beyond the technical side, I genuinely want to learn from experienced mentors and pick up good engineering practices while contributing something meaningful. I have a three-month summer vacation coming up where this project will be my primary focus, and going forward I will be consistent for sure. Compilers is an area I want to go deeper into, whether that's through higher studies or building something of my own in the future. Right now I want to work on real problems in a real codebase, and this fellowship is the right opportunity for that.