Enhance Clang Diagnostics

Making the Compiler Work Harder for You

Presenter

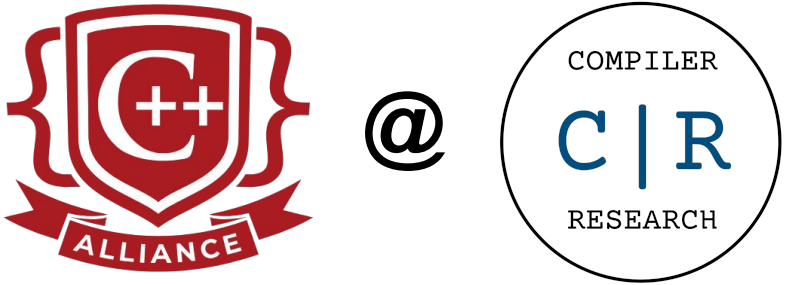
Aditya Medhane

CppAlliance Fellow 2026

Mentors

Vassil Vassilev

Aaron Jomy



Why Diagnostics?

Compilers communicate with us. When something goes wrong in our code, they tell us what happened and where, through errors, warnings, notes, and remarks.

> **But there are a surprising number of cases where Clang just stays silent.**
>
> **And that silence causes real bugs that are hard to track down.**

These silent failures are exactly what this project aims to fix.

The Five Areas

This project targets five categories of diagnostic gaps in Clang.

1. Unimplemented and partially implemented C2y papers

Focusing on Standards conformance.

2. Silent semantic change detection

Catching code that compiles cleanly but silently means something different from what the developer intended.

3. Clang-tidy check upstreaming into Clang proper

Promoting valuable opt-in checks to default warnings, so every Clang user benefits.

4. Diagnostic rewording and fix-it hints

Fixing confusing or factually wrong messages, and adding actionable suggestions.

5. Missing diagnostics where Clang stays silent but other compilers don't

Closing parity gaps with GCC and others.

Category 1 - Unimplemented and Partially Implemented C2y Papers

- C2y is the upcoming C standard, succeeding C23.
- It introduces dozens of new papers and clarifications.
- Clang tracks each one in its public C status page as **Yes**, **No**, or **Partial**.

What we'll work on:

- **N3418** - "Slay Some Earthly Demons XIV" → **No**
- **N3244** - "Slay Some Earthly Demons I" → **Partial**
- This project aims to move both to **Yes**.

Prior Work:

I've worked on a related C2y paper **N3410** which makes it ill-formed to declare an identifier with both internal and external linkage.

```c
static int x;                   // file scope - internal linkage
void test_basic_shadow(void) {
    int x;                      // local variable - no linkage
    { extern int x; }           // requests external linkage
                                // -> err_internal_extern_mismatch fires:
                                //    "declared with both internal and external linkage"
}
```

N3418 -- Universal Character Name Formation via ## is now a Constraint Violation

- Adopted by WG14 in Feb 2025 (straw poll 20/1/3)
- Goal: Turn silent Undefined Behavior into a clear constraint-violation diagnostic

Quick Explainer

A UCN lets you write Unicode directly in source using hex codes: \uXXXX (4 digits) or \UXXXXXXXX (8 digits).

```
// \u00E9 -> é | \u4E2D -> 中
int \u00E9 = 0; // valid identifier
const char *s = "caf\u00E9";
// valid string "café"
```

The Problem: UCN formed by token pasting

```
#define CONCAT(a, b) a##b
int CONCAT(\u00, E9) = 0;
// ^^^^^^^^^^^^^^^^ forms "\u00E9"

// - Pre-C2y: Undefined Behavior. GCC diagnoses. Clang stays silent.
// - C2y: Constraint violation — MUST BE DIAGNOSED.
```

N3244 -- `extern inline` Without a TU Definition (Item 67)

- N3244 is currently marked **Partial** on Clang's C status page.
- Contains two sub-items related to diagnostics still needing work -- we'll focus on the more illustrative one here.

Quick Explainer

- The `inline` keyword tells the compiler a function may be inlined.
- `extern inline` means: "this declaration must be paired with a definition in the same translation unit".
- C2y requires a diagnostic if the definition is missing; Clang stays silent today.

```c
// Declared extern inline, but never defined
extern inline void never_defined(void);

int main(void) {
never_defined(); // Linker error later
return 0;
}

// C2y: Compiler must diagnose at end-of-TU
```

Implementation Strategy

The diagnostic cannot fire at the declaration—the definition could appear anywhere later.

- **Deferred Check:** Maintain a tracking set in `Sema` of every `extern inline` declaration.
- **Cleanup:** Remove entries from the set as soon as their definition is encountered.
- **End-of-TU:** Emit diagnostics in `ActOnEndOfTranslationUnit()` for any remaining items.

Category 2 -- Silent Semantic Change Detection

FLAGSHIP DELIVERABLE

- The most novel compiler check in this project
- Same code. Same lambda. Completely different behavior.

A library update changes a container's element type from **`std::string`** to **`const char*`**. Existing generic-lambda code keeps compiling — but silently flips its meaning.

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

// Two containers - same contents, different element types
std::vector<std::string>  v1 = {"alice", "bob", "carol"};
std::vector<const char*>  v2 = {"alice", "bob", "carol"};

// One generic lambda - reused in BOTH calls below
auto match = [&](const auto& s) { return s == needle; };

std::find_if(v1.begin(), v1.end(), match);
//   s deduces to:  std::string
//   s == needle ->  calls std::string::operator==
//               ->  VALUE comparison   ->  finds "bob"   ✓

std::find_if(v2.begin(), v2.end(), match);
//   s deduces to:  const char*
//   s == needle ->  raw pointer == on two const char*
//               ->  ADDRESS comparison ->  never finds "bob"   ✗
```

Container	What `s == needle` actually does
`vector<std::string>`	Calls **`operator==`** content comparison (**Correct**)
`vector<const char*>`	Plain `==` raw address comparison (**Always fails**)

Why both addresses are always different:

- `needle` points into b's heap buffer
- `v2[1]` points into the read-only string-literal segment
- Same characters, different pointers comparison is always **false**

Category 3 -- Pulling Useful Clang-Tidy Checks into Clang Proper

- Clang-tidy ships hundreds of useful checks but most developers never see them, because they require opt-in.
- Two checks are clean candidates to live in Clang itself, where they'd run on every compilation.

Check 1 : -Wmultistatement-macros

Catches macros that expand to multiple statements used as the body of control flow without braces.

- Only first statement is conditional.
- GCC already ships this warning.

```
#define INCREMENT(x, y) (x)++; (y)++

if (cond) INCREMENT(a, b);

// Expands to: (a)++; (b)++;
// -> (a)++ is conditional on 'cond'
// -> (b)++ ALWAYS runs - silent bug
```

Check 2 : Extending -Wclass-memaccess

Flags raw C memory calls (memset, realloc, etc.) on non-trivial C++ types.

- Bypasses constructors/destructors.
- Fixes gaps for realloc and type traits.

```
std::string s;

memset(&s, 0, sizeof(s));

// std::string is non-trivial
// -> raw memset bypasses construction
// -> silently corrupts internal state
```

Both checks already exist as proven patterns, upstreaming gives every Clang user the benefit by default.

Category 4 -- Better Wording, Helpful Fix-Its

- Open issues where Clang's diagnostic is factually wrong, misleading, or missing a fix-it suggestion.
- Each is self-contained: locate the diagnostic string, rewrite it or attach a FixItHint, update tests.

Example 1: Missing #include fix-it

```c
int main(void) {
printf("hello\n"); // forgot #include <stdio.h>
return 0;
}
```

- **Current:** "implicit declaration of function 'printf'" (error stops).
- **With fix-it:** "add #include <stdio.h>?" → suggestion + auto-insert.
- Internal logic exists; just missing the UI hook.

Example 2: Inverted Switch Location

```c
switch (x) {
default: ... // Error points here (VALID one)
case 1: ...
default: ... // Note points here (DUPLICATE)
```

- Primary error and "previous definition" note are swapped.
- Valid label gets the error; duplicate gets the note.
- Fixing arity mismatches and misleading 'this' constexpr notes.

Category 5 -- Missing Diagnostics

- Cases where Clang stays silent on code that GCC (and others) warn about.

Example 1

Operator precedence inside macros (filed by Richard Smith, former Clang lead)

```
#define FOO 2 + 3

int x = FOO * 4;

// Expands to: 2 + 3 * 4 -> evaluates as 14
// User likely meant: (2 + 3) * 4 -> would be 20
```

- The + lives inside the macro, but * 4 is supplied from outside - operator precedence silently changes the meaning.
- A warning here is feasible with Clang's existing source-location info; the issue is labeled good first issue.

Example 2

-Wunused-value skipped inside macros

```
true; // -Wunused-value: WARNS

#define MACRO() (long)f()
MACRO(); // -Wunused-value: SILENT
```

- Same expression, different behavior depending on whether it came from a macro.
- Root cause: a single check on isMacroBodyExpansion() - fixing it closes two open issues at once.

Thank You!

Any Questions?