P4008R0: Clean Modular Mode: Legacy Opt-out for C++

Author: Zhiyi Lin (林志逸)

Target: C++29 / C++31

Date: 2026-02-08

Audience: Evolution Working Group (EWG)

Paper Number: P4008R0

Reply-To: lin20101009xv@qq.com

Dear WG21 Convener and Members,

Please find my proposal for a new clean modular mode for C++. This design enables gradual legacy cleanup without breaking backward compatibility, while preserving full low-level system programming capabilities. Thank you for your time and consideration.

==========

1.One-Sentence Vision

Keep full backward compatibility by default, but allow users to opt out of historical C-compatible baggage via modular exclusion—preserving all low-level system programming capabilities in a non-excludable core.

==========

2. Problem We Solve

C++ must remain compatible with billions of lines of legacy code. However, historical C-compatible behaviors (array decay, implicit narrowing, C-style casts, unsafe C stdlib functions) are a persistent source of bugs, complexity, and poor developer onboarding.

Existing "safe subset" efforts either:

- Remove essential low-level system programming power, or
- Are unenforceable guidelines (not compiler-backed).

There is no structured, per-translation-unit/per-module mechanism to **cleanly disable these legacy pitfalls** while retaining full control over low-level features.

==========

3.Core Axioms (Key Insight)

Low-level capability≠ historical garbage

Low-level capability ≠inherently unsafe

Low-level capability ≠ to be removed

Historical baggage ≠ low-level capability

Historical baggage = opt-outable

Default compatibility ≠ mandatory pollution

Clean subset ≠ neutered language

==========

4. Proposal (3 Bullet Points)

1. **Split the standard into 3 fixed, non-overlapping modules**:

-`std.core`: Non-excludable low-level primitives (pointers, type layout, atomics, bit ops, zero-cost abstractions).

-`std.modern`: Safe modern library components (vector, span, smart pointers, ranges, `std::expected`).

-`std.legacy`: C compatibility & legacy pitfalls (array decay, C-style casts, implicit narrowing, C stdlib).

```
//Default behavior (100% backward-compatible):
import std.core;
import std.modern;
import std.legacy;
// Clean Mode (user opt-in)
exclude std.legacy; // Removes only legacy pitfalls, retains full low-level
power
```

5. Safety For Standardization

No ABI break: Type layouts, name mangling, calling conventions are unchanged.

No default behavior change: Existing code compiles exactly as before.

No dialect split: Clean mode is a subset of standard C++ (not a new language).

No forced migration: Legacy code can remain indefinitely; adoption is per-file/per-module.

Builds on C++20 Modules: Uses mature infrastructure (no compiler redesign required).

---

6. Value Proposition

Dual-line evolution: Legacy line (unchanged, default) + Clean line (modern, safe).

Incremental cleanup: No full project rewrite—migrate files/modules one at a time.

Safety without sacrifice: Retains low-level control/performance.

Teachable again: New developers start in Clean Mode, avoiding historical pitfalls.

---

7. What We Are NOT Proposing

Not a new language (this is C++, not "C++ 2.0").

Not an ABI break (no runtime/linkage changes).

Not removing low-level features (std.core is non-excludable).

Not mandatory safety (Clean Mode is opt-in).Not dialect fragmentation (all code is interoperable across modes).

==========

8. Code Examples

8.1 Legacy Mode (Default, Unsafe)

```
// Compiles in default mode (contains legacy pitfalls)
void legacy_code() {
int arr[10];
int* p = arr; // Array-to-pointer decay (unsafe)
double d = 3.14;
int i = (int)d; // C-style cast + implicit narrowing (unsafe)
}
```

8.2 Clean Mode (Compile Errors for Pitfalls)

```
exclude std.legacy;

 // Opt into Clean Mode

void clean_code() {

int arr[10];

int* p = arr; // ERROR: Array decay is disabled

double d = 3.14;

 int i = (int)d; // ERROR: C-style cast is disabled }
```

8.3 Modern Safe Equivalent (Clean Mode)

```
exclude std.legacy;

 import std.modern;

void safe_code() {

std::array<int, 10> arr;

 std::span<int> p = arr; // Safe view (no decay)

double d = 3.14;

 int i = std::narrow_cast<int>(d); // Explicit, checked conversion

}
```

==========

9. Gradual Migration Pathway

Default Mode: All existing code works unchanged.

Hybrid Mode: New files/modules use exclude std.legacy; old files stay in Legacy Mode.

Clean Mode: All files exclude std.legacy (full modern project).

==========

10. Implementation Roadmap

Phase 1 (C++29): Add exclude std.legacy syntax; bind array decay/C-style casts to the flag.

Phase 2 (C++31): Expand to more legacy features (implicit narrowing, C stdlib).

Phase 3: Incrementally restructure the standard library into std.core/std.modern/std.legacy.

==========

11. Conclusion

This proposal is the **minimal, safest, most industry-viable path** for C++ to evolve:

Preserves C++'s system programming soul.

Removes technical debt *structurally* (not just via guidelines).

Supports gradual, risk-free adoption.

Lets C++ remain compatible *and* modern for the next 30 years.