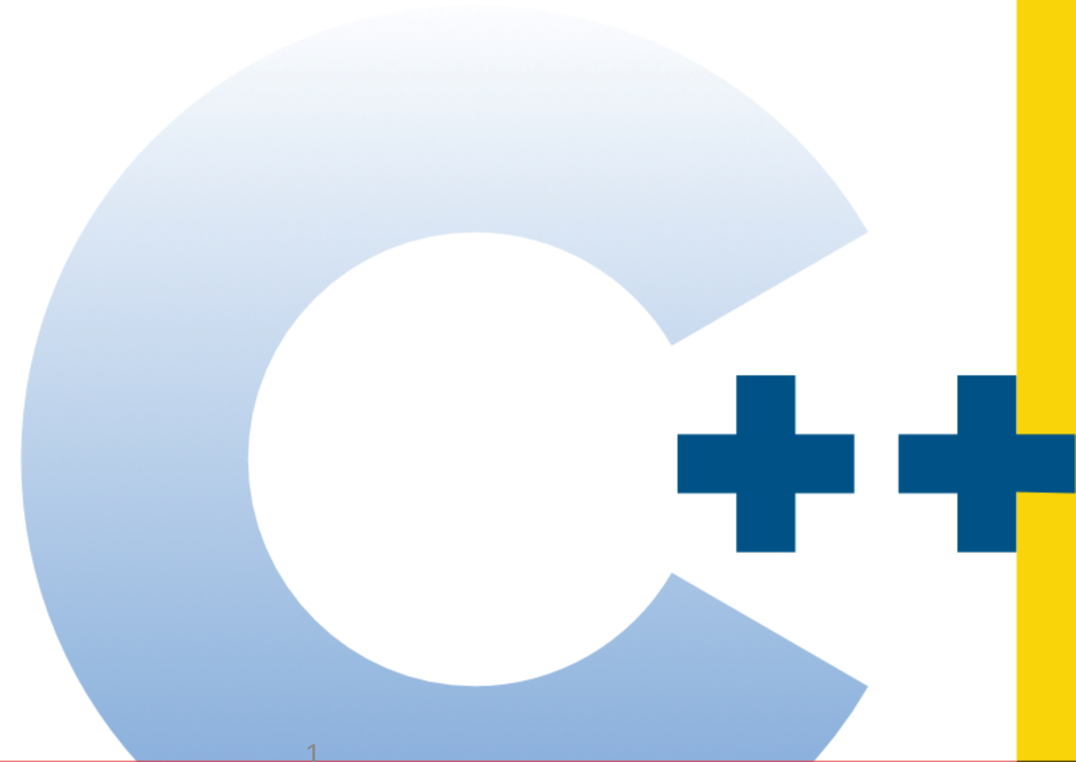
# Tightening the Screws with C++ Profiles



BECPP SYMPOSIUM 2026 | March 30th | Kortrijk

# Part I

The Problem We Actually Have

# The Engineering Reality

- C++ is infrastructure: kernels, browsers, databases, compilers, game engines
- Billions of lines, delivering value every day
- The default guarantees tolerate constructs that, unintentionally, constitute defects
- A dangling pointer is well-formed C++
- An uninitialized local is well-formed C++
- A `reinterpret_cast` from `char*` to `int*` is well-formed C++

# The Constraints

- No pause button — can’t stop the world and rewrite billions of LOC
- No future-only — must help code deployed today
- No vendor lock-in — portability of safety guarantees

> *Without a common framework: “an incompatible mess of checks impossible to unify in a future standard.”*

# The Aliasing Elephant in the Room

Two pointers alias when they refer to the same object.

In C++ this is not a bug — it is a feature.

```
void swap(int& a, int& b) {
    int tmp = a;  a = b;  b = tmp;
}
int x = 1;
swap(x, x);   // a and b alias — perfectly legal

// Self-assignment must be safe
Matrix& Matrix::operator=(const Matrix& rhs) {
    if (this == &rhs) return *this;  // aliasing check
    // ...
}
```

# Why Aliasing Defeats Simple Analysis

```
void dangerous(std::vector<int>& v, int& elem) {
    v.push_back(42);   // may reallocate v's storage
    use(elem);          // is elem still valid?
}

int main() {
    std::vector<int> v = {1, 2, 3};
    dangerous(v, v[0]); // elem aliases into v's storage
}
```

- Function signature doesn’t reveal the aliasing
- Conservative analysis → flood of false positives
- Call-graph analysis → expensive and undecidable
- C++ cannot ban aliasing (self-assignment, this pointer, iterator pairs)

# What Profiles Do About Aliasing

Profiles accept aliasing. They focus on what is diagnosable despite it:

- Lifetime of locals — the local is gone, no aliasing analysis needed
- Invalidation events — `push_back` invalidates; tracked, not alias-resolved
- Known-dangerous patterns — `reinterpret_cast`, uninitialized variables
- Where analysis is insufficient: `[[profiles::suppress]]` with justification

> *Suppression is not hiding the problem — it is documenting the engineering judgment that this aliasing pattern is correct.*

# Part II

C++Profiles, With Only Four Attributes

# [[profiles::enforce(...)]]

Request enforcement — must appear before any non-empty declaration

```
[[profiles::enforce(std::type)]];

#include <vector>

int main() {
    std::vector<int> v = {1, 2, 3};
    int* p = &v[0];
    v.push_back(4);     // reallocation may invalidate p
    int x = *p;         // diagnosed: dangling pointer
}
```

- Dominion = from attribute to end of translation unit
- Multiple profiles compose by juxtaposition
- Repetition of same profile — OK; partial repetition — error

# [[profiles::enforce]] on Module Interfaces

```
module;
#include <unistd.h>

export module Kai.Utils [[profiles::enforce(std::type)]];

export using ::getpid;
//  ...
```

- Dominion extends to all module implementation units
- The library advertises its guarantees in the module declaration itself
- Not in documentation — in code, checked by the compiler

# [[profiles::suppress(...)]]

Local, scoped deviation — mark the corners, don't pretend they don't exist

```
[[profiles::suppress(std::type,
     rule: "type.cast.reinterpret",
     justification: "wire protocol requires exact byte layout")]]
auto* header = reinterpret_cast<PacketHeader*>(raw_bytes.data());
```

- Appertains to a single declaration or statement
- Named-rule suppression: finer granularity than whole-profile
- Three arguments: profile name, rule, justification
- Evolved from a decade of `[[gsl::suppress]]` deployment
- Justification strings — the single most valuable artifact for code review

# [[profiles::require(...)]]

Compile-time contracts between components

```
// Library
export module Kai.Utils
    [[profiles::enforce(std::type, std::lib::hardened)]];

// Consumer 1
import Kai.Utils [[profiles::require(std::type)]];            // OK
import Kai.Utils [[profiles::require(std::lib::hardened)]];  // OK

// Consumer 2
import Kai.Utils [[profiles::require(std::ranges)]];          // error!
```

C++ has never had a way for one translation unit to state a machine-checked precondition on the safety guarantees of another.

# [[profiles::exempt(…)]]

Legacy header interop — the pragmatic bridge

```
[[profiles::enforce(std::type)]];
[[profiles::exempt(std::type, angle_header: "unistd.h")]];
[[profiles::exempt(std::type, quote_header: "legacy-ffi.h")]];

#include <unistd.h>       // exempted from std::type
#include <stdio.h>        // NOT exempted - still under std::type
#include "legacy-ffi.h"   // exempted from std::type
```

- `angle_header`: / `quote_header`: mirrors the two forms of #include
- Covers the nominated header and everything it transitively includes
- Enforce on your code, exempt what you can’t change, track the exemptions

# Part III

The Structure Underneath: Formal Semantics

# Profiles as Sets of Restrictions

- A profile P is a set of additional language restrictions (rules)
- Applied after translation phase 7

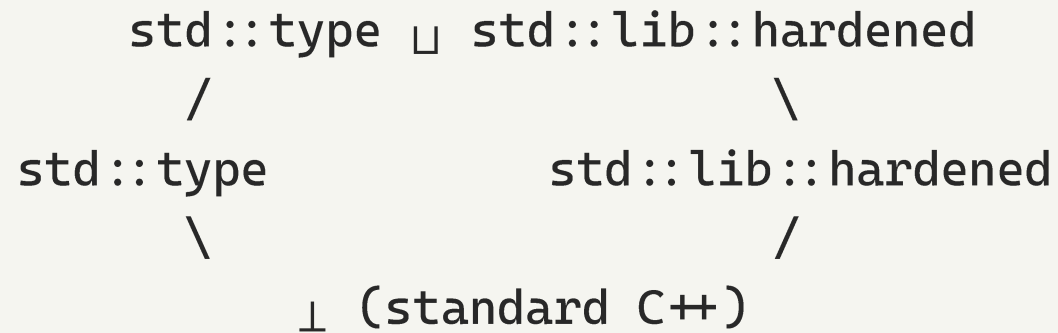
> *A profile does not change the meaning of a well-formed program with no undefined behavior.*

- Profiles restrict the set of accepted programs
- They never change what an accepted program does
- No interference with overload resolution, SFINAE, template instantiation
- “No spooky action at a distance — your metaprogramming works exactly as before”

# The Restriction Partial Order

$P_1 \leq P_2 \quad \Leftrightarrow \quad \text{rules}(P_1) \subseteq \text{rules}(P_2)$

$P_2$ is at least as restrictive as $P_1$.




- ⊥ at the bottom = standard C++ with no profiles
- Two profiles may be incomparable (partial order, not total order)

# Join: Composing Profiles

Enforce two profiles = union of their rule sets = join

$\text{enforce}(P_1, P_2) = P_1 \sqcup P_2 = \text{rules}(P_1) \cup \text{rules}(P_2)$

```
[[profiles::enforce(std::type)]];
[[profiles::enforce(std::lib::hardened)]];

// Active profile: std::type ⊔ std::lib::hardened
```

- Commutative: $P_1 \sqcup P_2 = P_2 \sqcup P_1$
- Associative: $(P_1 \sqcup P_2) \sqcup P_3 = P_1 \sqcup (P_2 \sqcup P_3)$
- Idempotent: $P \sqcup P = P$
- These are the axioms of a join-semilattice

# The Lattice in Action

```
// Geometry – enforces type safety
export module Geometry [[profiles::enforce(std::type)]];

// Network – enforces type safety AND hardened library
export module Network [[profiles::enforce(std::type, std::lib::hardened)]];

// App – the consumer
[[profiles::enforce(std::type, std::lib::hardened)]];
import Geometry [[profiles::require(std::type)]];             // OK
import Network  [[profiles::require(std::type)]];             // OK
import Network  [[profiles::require(std::lib::hardened)]];    // OK
import Geometry [[profiles::require(std::lib::hardened)]];    // error!
```

The consumer requires a profile above what `Geometry` provides.

The lattice makes the reasoning mechanical.

# Suppression: Scoped Weakening

P_eff = rules(P) \ rules(Q) — suppress whole profile Q

P_eff = rules(P) \ {r} — suppress single named rule r

- Scope = dominion of the single declaration or statement
- Outside that dominion: full P is in effect again
- Reversible and auditable
- You can enumerate every suppression site, categorize by profile and rule, and track the count over time as a safety metric

# Compatibility and the Standard Sublattice

> *All standard profiles are compatible with each other.*

- Standard profiles form a closed sublattice
- Any join of standard profiles is well-defined
- Compatibility on redeclarations holds automatically
- Monotonicity: a declaration under profile P requires all redeclarations under a compatible profile
- Cannot silently weaken between header and implementation file

# Module Accords as Pre/Post Conditions

- enforce on a module interface = postcondition:
  "My interface was written under profile P"
- require on an import = precondition:
  "I depend on profile P being satisfied by this module"

The compiler checks: $P_require \le P_enforce$

Hoare-style design, rooted in axiomatic semantics, lifted to the component level and checked by the compiler at the boundary between translation units.

# Dominion Nesting and Effective Profile

```
Translation unit scope
|
├─ [[profiles::enforce(std::type, std::lib::hardened)]];
|   |
|   |  active: std::type ⊔ std::lib::hardened
|   |
|   ├─ [[profiles::suppress(std::type)]]
|   |  char buffer[1024];
|   |                  ← active: std::lib::hardened only
|   |
|   |  active: std::type ⊔ std::lib::hardened  (restored)
|   |
|   ├─ [[profiles::suppress(std::type,
|   |       rule: "type.cast.reinterpret")]]
|   |  auto* p = legacy_c_api(arr, n);
|   |                  ← active: both, minus one rule
|   |
|   |  active: std::type ⊔ std::lib::hardened  (restored)
```

# Part IV

Memory Safety and Lifetime in Practice

# Before and After: string_view of Local

Without profiles

```
// Without profiles
// Compiles silently with
// -Wall -Wextra (may or may not)

std::string_view get_greeting() {
    std::string s = "hello";
    return s;
    // dangles
}
```

With profiles

```
// With profiles
[[profiles::enforce(std::type)]];

std::string_view get_greeting() {
    std::string s = "hello";
    return s;
    // error [std::type]: returning
    // view of local variable 's'
}
```

“The code did not change.  The guarantee did.”

# Dangling Pointers: span of Local vector

```
[[profiles::enforce(std::type)]];

std::span<const int> last_three(int n) {
    std::vector<int> tmp = {n, n+1, n+2};
    return std::span(tmp);
    // diagnosed: returning view of local vector
}
```

- `tmp` owns heap storage; span is a non-owning view
- When `tmp` is destroyed, the span dangles
- Same reasoning as `string_view` — different vocabulary type

# Uninitialized Variables

```
[[profiles::enforce(std::type)]];

int compute(bool flag) {
    int result;
    if (flag)
        result = 42;
    // diagnosed: result may be uninitialized
    return result;
}
```

- Standard C++ permits `int result;` without initialization
- The profile requires definite assignment before use
- Restriction, not language change: smaller set of accepted programs, meaning of accepted programs unchanged

# Runtime Effects: std::lib::hardened

```
[[profiles::enforce(std::lib::hardened)]];

void process(std::vector<int>& v) {
    int x = v[10];
    // if v.size() ≤ 10: runtime trap, not silent UB
}
```

- Dynamic safety net — not static rejection
- Out-of-range access → trap instead of silent memory read
- Cost: bounds check per access (many codebases already pay via .at())
- Static + dynamic compose: `std::type` for compile-time, `std::lib::hardened` for runtime bounds checks
- One attribute, uniform coverage

# Safe Suppression at FFI Boundaries

```
[[profiles::enforce(std::type)]];
[[profiles::exempt(std::type, angle_header: "sys/mman.h")]];
#include <sys/mman.h>

class MappedFile {
    MappedFile(const char* path, size_t len) {
        int fd = open(path, O_RDONLY);
        [[profiles::suppress(std::type,
             rule: "type.cast.reinterpret",
             justification: "mmap returns void*")]]
        data_ = reinterpret_cast<const std::byte*>(
            mmap(nullptr, len, PROT_READ, MAP_PRIVATE, fd, 0));
        close(fd);  length_ = len;
    }
    // ... rest under full enforcement
};
```

Exactly one statement suppressed, for one named rule, with a justification.

# Measuring Suppression Debt

```
Profile suppression report — Kai.Utils
──────────────────────────────────────────────────
std::type                          7 suppressions
  type.cast.reinterpret            4  (FFI boundary)
  type.init.uninitialized          2  (POSIX buffer fill)
  type.lifetime.dangling           1  (arena allocator)
std::lib::hardened                 1 suppression
  lib.hardened.unchecked           1  (hot loop, bounds ok)
──────────────────────────────────────────────────
Total: 8 suppressions in 12,400 LOC (0.065%)
```

- Concrete, trackable safety metric
- The number goes down as you eliminate suppressions
- Justification strings tell you why each one remains
- No new tooling beyond a text search (or AST grep)

# Lifetime Across Module Boundaries

```
// display.ixx
export module Display [[profiles::enforce(std::type)]];
export class Canvas {
public:
    void paint(std::span<const Pixel> pixels);
    void resize(int w, int h);
};

// app.cpp
[[profiles::enforce(std::type)]];
import Display [[profiles::require(std::type)]];
void render(Canvas& c) {
    auto pixels = generate_frame();
    c.paint(pixels);  // safe: verified under std::type
}
```

If Display drops enforcement, every require site becomes a compile error.

Safety guarantees become part of the API contract.

# Part V

Inside the Implementation

# The Three-Stage Pipeline

- Stage 1 — Attribute parsing (translation phases 1–6)
  Parse and record enforce, suppress, exempt, require attributes
-
- Stage 2 — Profile enforcement (after phase 7)
  Walk the abstract semantic graph; check active profile rules at each node
  Suppressions create holes; exemptions exclude nominated headers
-
- Stage 3 — Cross-TU checks
  require on import checked against enforce on module interface
  Simple set-containment check on profile designators

# Tracing an Example End to End

```
[[profiles::enforce(std::type)]];    // (A)

std::string_view get_greeting() {
    std::string s = "hello";
    return std::string_view(s);      // (B)
}
```

- Parse: record `std::type` enforced from (A) to end of TU
- Phase 7: standard C++ — overload resolution, type checking — unchanged
- Enforcement: walk semantic graph, reach return at (B)
  → `string_view` captures pointer into local `s`

```
get_greeting.cpp(4): error [std::type]:
  returning view 'std::string_view'
  that refers to local variable 's'
```

# Abstract Domains for Profile Enforcement

| Domain | Question | Cost | Scope |
|---|---|---|---|
| Definite Assignment | Is this variable initialized? | 1 bit/var, linear | Intra |
| Lifetime State | Is the storage still alive? | 1 state/ptr, linear | Intra + Inter |
| Nullness | Can this pointer be null? | 2 bits/ptr, linear | Intra + summaries |
| Interval | Can this index be out of range? | 2 values/var, moderate | Intra |
| Points-To | What can this pointer refer to? | set/ptr, cubic worst | Intra → Inter |
| Invalidation | Has the container been mutated? | 1 state/pair, linear | Intra + lib model |
| Typestate | Is this object in a valid state? | 1 state/obj, linear | Intra + summaries |

These are the domains compilers already use internally for their warning flags.

Profiles give the results a standardized reporting mechanism.

# Intra-procedural vs. Inter-procedural

| Intra-procedural (local) | Inter-procedural (across calls) |
|---|---|
| Uninitialized variables | Use-after-free (heap) |
| Return address of local | Iterator invalidation across calls |
| Local iterator invalidation | Aliasing across parameters |
| Integer overflow, div-by-zero | Array bounds (param from caller) |
| reinterpret_cast, unsafe casts | Module guarantee matching |
| Use after std::move | Lifetime across call boundaries |

Most high-value checks are intra-procedural.

Inter-procedural gaps bridged by summaries, not whole-program analysis.

This is why profiles are implementable today — not someday.

# Standard Library Invalidation Rules

| Container | Insert | Erase | Key Subtlety |
| --- | --- | --- | --- |
| vector | Realloc → all invalid | At/after erase point | Most dangerous |
| deque | Middle → all; ends → iters only | Symmetric to position | Iterators ≠ references |
| string | Any non-const op → all | Same | Broadest rule (SSO) |
| list, forward_list | Nothing invalidated | Only erased elements | Node-based: safe |
| map, set, multi* | Nothing invalidated | Only erased elements | Node-based: safe |
| unordered_* | Rehash → all iterators | Only erased elements | Refs stay valid |
| span, string_view | No mutations | — | Dangle if underlying modified |

> *A profile doesn't invent these rules —*
> *it enforces what the standard already specifies.*

# The Fragmented Landscape

- Clang: 48 safety-relevant warnings
  (12 spatial, 17 temporal, 13 arithmetic, 6 pointer)
- GCC: 29 safety-relevant warnings
  (7 spatial, 7 temporal, 11 arithmetic, 4 pointer)
- Same problems, different names, different granularity
- No in-source notation
- Suppression: `#pragma` diagnostic ignored
  — no justification, no named rule, not standardized

> *This is exactly the "incompatible mess" the framework document warns about.*

# -fbounds-safety: Right for C, Wrong for C++

Apple's Clang extension: `__counted_by(N)`, `__sized_by(N)`, fat pointers

Deployed at scale on Apple's production C codebases – example of "C++ Profiles", but C-style

- Annotates pointers, not types — C++ has `span`, `string_view`, `vector`
- Fat pointers (2–3× size) break ABI — `span<T>` is always explicit
- Cannot express lifetime — bounds-checked freed memory is still use-after-free
- Single-vendor, C-only

> *Fat pointers are what you build when your language doesn't have span.*
> *C++ has* `span`.

# Part VI

What This Means for You

# If You Maintain a Library

```
export module MyLib [[profiles::enforce(std::type)]];
```

- Pick one module interface
- Add `[[profiles::enforce(std::type)]]`
- Fix or suppress the diagnostics
- Ship it
- Your consumers can now write `[[profiles::require(std::type)]]`
- Your safety guarantee becomes part of your API
- Do one module this quarter. Do the rest next quarter.

# If You Maintain a Large Codebase

- Pilot: pick 10 translation units
- Add `[[profiles::enforce(std::type)]]` at the top of each
- Exempt system headers with `[[profiles::exempt]]`
- Suppress what you can't fix yet — with justification: and rule:
- Count the suppressions → that is your baseline
- Track it quarterly → that is your safety progress curve
-
- The cost is not rewriting code.
- The cost is triaging suppressions — and you only triage once.

# If You Build Tools

- Parse the `[[profiles::enforce( ... )]]` attribute — 5 grammar productions
- Register your existing analysis passes as named profiles
- Emit diagnostics for unrecognized profile names
- Organizations opt into your analysis using the same notation as standard profiles
- No new configuration files, no new command-line flags

```
[[profiles::enforce(msvc::Ruleset("CoreCheck", fortify: 3))]];
```

# If You Write New Code

```
[[profiles::enforce(std::type, std::lib::hardened)]];
```

Start with enforcement from line one.

Zero suppression debt from day one.

Catches dangling pointers, uninitialized variables,
unsafe casts, and out-of-range access
before your code reaches code review.

> *The profiles framework does not ask you to abandon C++.*
> *It asks you to be explicit about the guarantees you intend.*
> *Explicitness is not a burden — it is engineering.*

# Key Takeaways

- Framework, not wish list. Four attributes, precise dominion.
- Does not change C++. Phase-7 enforcement.
- Lattice: join is composition, suppression is scoped set-difference.
- Module contracts: machine-checked pre/post conditions.
- Aliasing accepted. Diagnose what's diagnosable, suppress the rest.
- Infrastructure exists but is fragmented.
  48 Clang + 29 GCC flags, no common notation. Profiles unify.
- `-fbounds-safety` is right for C, wrong for C++.
  Use the types that already carry bounds.
- Suppression debt is measurable. Track it.
- Evolutionary path. Enforce, exempt, suppress, tighten.

# THANK YOU!