A Profile for Runtime-Checkable Core-Language Undefined Behavior: std::core_ub

Document Number:	P4317R1
Date:	2026-08-01
Intent:	Inform
Audience:	EWG, SG22

Reply-to:	Vinnie Falco vinnie.falco@gmail.com

Table of Contents

Abstract

P4297R1 [1] asks EWG to sever P3100R8's architecture claim from its case-by-case wording review. This paper demonstrates the severing is feasible.

The same 77 runtime-checkable cases of core-language undefined behavior, enumerated by Doumler and Berne in P3100R8 [2], are brought together under a single profile with zero foundational wording changes and no handler dependency. The form of our solution: named checks, per-build activation, terminating response, is what ships today in hardened production setups.

This paper is design exploration, not a proposal for adoption.

This paper asks for nothing.

Revision History

R1: August 2026

- Restructured around the methodology: how each P3100R8 line item slides into a profile by severing the implicit contract assertion.
- Positioned as the implementation of P4297R1's severance ask.
- Added before/after table for two cases and six EWG telecon quotes (with permission).
- Qualified instrumented-case guarantee: both proposals face the same sanitizer limits.
- Cited D4277R0 (late D-paper) for prototype status.
- Cut SD-10 evaluation tables (former Section 5) and committee-direction section (former Section 7); compressed concerns from seven to three.

- Added Section 5 (Prototype) with Compiler Explorer examples for 7 locally-checkable cases.

R0: July 2026

- Initial version.

1. Introduction

Doumler and Berne enumerate 77 runtime-checkable cases of core-language undefined behavior in P3100R8 [2] (80 total, 3 not runtime-checkable).

P4297R1 [1] identifies the bundling problem with this approach: P3100R8 ties wording for 77 cases to an architecture claim: Profiles as a preset over Contracts machinery. This paper shows the unbundled form.

Our design is driven by a single principle: standardize what already ships.

Quoted with permission. Source: EWG telecon, 10 August 2026.

> *"Can we please separate the part where there's the true consensus - the erroneous value - separate that from the architecture choice, so we can get some value out of our time investment." - Vinnie Falco*

There is consensus for the undefined behavior enumeration, and not the architecture. By separating the two, a profile inherits this consensus without the architecture that rides along with it.

> *"We do make changes that make future changes effectively difficult or impossible, even though they're not officially procedurally impossible. There does have to be recognition that the choices we make now can foreclose future directions, and we have to be cautious of that." - John Spicer, EDG*

Foreclosure is avoidable: the same 77 cases are equally applicable under a different architecture. The companion papers P4297R1 [1], P4306R1 [4], and P4310R1 [5] address adjacent questions (ownership, configuration comparison, response merits).

2. The Guarantee

When `std::core_ub` is enforced, a violated runtime-checkable precondition among the 77 cases avoids undefined behavior.

- 62 terminate on violation; 15 receive a well-defined replacement value (Section 3, Appendix A.4).
- 19 locally checkable (Appendix A.1): the guarantee holds unconditionally.
- 58 instrumented (Appendix A.2 and A.3): the guarantee holds within the instrumented domain.

No current sanitizer catches all 58 reliably; this paper and P3100 face the same instrumentation limits. Sanitizers represent the detection frontier. [6]

The profile is transparent to programs with no undefined behavior. P3589R2 [3] requires this: a profile does not change the meaning of a well-formed program with no UB.

For the 15 replacement cases (12 unconditional, 3 built-in-types-only): the profile defines the meaning directly, fixed for every conforming implementation. P3984R0 [7] grants this authority. Signed overflow is wraparound, out-of-range conversion is erroneous value, and so on per Appendix A.4.

3. Activation and Response

Activation

Framework syntax activates the profile:

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

Dominion runs to end of the translation unit. `[[profiles::suppress(std::core_ub)]]` is the local escape. No annotation appears in ordinary user code.

Three response candidates

All three terminate:

1. **Trap.** Trap instruction. Diagnostics recovered out of process by crash reporter. Smallest codegen. Apple's `-fbounds-safety` and libc++ hardening ship this.
2. **Diagnostic, then abort.** Print failed check plus source location, call `abort()`. libstdc++ ships this.
3. **Non-returning handler.** Replaceable profile-specific function; may log; must not return; if it returns, the program terminates. Shape of Bloomberg's `bsls_assert` [8] where post-violation state is undefined.

All three candidates provide the check identifier and source location to the response mechanism (crash reporter, diagnostic stream, or handler argument), giving deployment tooling enough to locate the violated constraint.

Interop note

A deployment can route through the C++26 contract-violation handler as an interop path, but this reintroduces the Contracts dependency the design avoids.

The noexcept question does not arise

P4308R1 [14] enumerates eight responses to a throwing implicit contract assertion - the question forced when a violation handler may throw through a core-language expression the `noexcept` operator reports as non-throwing. Under the profile, that question does not arise. All three response candidates terminate; none invokes a handler that may throw; no exception escapes a checked expression. The `noexcept` operator keeps both its value and its meaning unchanged.

P4308R1's requirements (1), (2), and (3) form a trilemma: at most two of noexcept-value-kept, unwinding, and noexcept-meaning-kept hold at once. The profile sidesteps the trilemma by not unwinding. A response that never throws has no interaction with the operator to resolve.

P3608R0 precedent

P3608R0 [9] (Dos Reis, Voutilainen, Wakely) proposed this shape for library hardening: "a concrete profile that switches on the standard library hardening, and makes the violations of hardened preconditions just terminate the program, without any additional flexibility for C++26," with vendors "encouraged not to close the door for other violation handling strategies... in the future." The one difference is scope: P3608R0 covers library preconditions; `std::core_ub` covers core-language cases.

Bloomberg boundary

`bsls_review` [8] logs and continues at the library level (post-violation state defined). `bsls_assert` [8] terminates where the state is language-undefined - the class this profile guards. P4310R1 [5] sets out the full case.

4. How to sever each case from the architectural addon

For each of the 77 runtime-checkable cases enumerated by Doumler and Berne in P3100R8 [2], sever the implicit contract assertion and its five evaluation semantics from the checking obligation, and state the check as a profile constraint. The checking is identical; the routing is the difference.

The enumeration, the checking strategies, and the replacement behaviors are the work of Doumler and Berne. Appendix A is their enumeration, reproduced with credit.

Before/after: two cases

Case 1: Division by zero (`{expr.mul.div.by.zero}`, [expr.mul]/4) - locally checkable

Aspect	Under P3100R8	Under std::core_ub
Check	Divisor is nonzero	Divisor is nonzero
Mechanism	Implicit contract assertion	Profile constraint (quality of implementation)
Semantics available	5 (ignore, observe, enforce, quick-enforce, assume)	1 (enforced: terminate or replace)
Handler dependency	Contract-violation handler invoked on violation	None; profile owns the response
Replacement behavior	Erroneous value (under ignore semantic)	Erroneous value (fixed for all conforming implementations)

The prototype (Section 5) demonstrates this case live on Compiler Explorer: https://godbolt.org/z/s5Exo86K6

Case 2: Flow off end of function (`{stmt.return.flow.off}`, [stmt.return]/4) - locally checkable, has replacement behavior

Aspect	Under P3100R8	Under std::core_ub
Check	Execution does not flow off end of value-returning function	Execution does not flow off end of value-returning function
Mechanism	Implicit contract assertion	Profile constraint (quality of implementation)
Semantics available	5 (ignore, observe, enforce, quick-enforce, assume)	1 (enforced: return erroneous value for built-in types, terminate otherwise)
Handler dependency	Contract-violation handler invoked on violation	None; profile owns the response
Replacement behavior	Erroneous value (under ignore semantic)	Erroneous value for built-in return types (fixed); terminate otherwise

Pattern

The check column is identical in both rows. The routing column is where they diverge. Every row in Appendix A follows the same pattern: same check, different owner.

The six foundational clauses

Under the profile, none of the six foundational wording changes P3100R8 requires is needed (P4297R1 [1] Table 2 catalogues them). UB stays as-is in the standard; the profile adds rules on top via P3589R2 [3]. The 15 replacement behaviors are profile-specific semantics under the authority P3984R0 [7] grants and do not require normative changes to the referenced core-language clauses.

D4277R0 [6] presents an alternative wording strategy for P3100R8 that may reduce the six-clause count; the count of six applies to P3100R8's primary wording as presented in R8.

C++26 already standardized erroneous values for scalar initialization without implicit contract assertions and without routing through any violation handler:

> *"The scalar initialization precedent happened without implicit contract assertions and without routing through any violation handler. So my question is: why do we have to bundle the implicit precondition, when it wasn't needed for the example you just gave?" - Vinnie Falco (EWG telecon, 10 August 2026)*

The precedent is live. The profile extends it to the remaining cases. The before/after tables above are what the severed form looks like in practice - same check, different routing. Presented with this form in the telecon:

> *"The proposal suggestion made by Vinnie looks eminently sensible to me; and, I'm saying that not as a philosopher but as someone whose daytime job is to write actual C compilers that are used to compile product that runs the planet. I would like to see evidence of that. Otherwise it is just vehement assertion with no evidence; we cannot build a standard used by engineers and scientists that way. We need evidence." - Gabriel Dos Reis (quoted with permission, EWG telecon, 10 August 2026)*

Closing

The appendix is this methodology applied to all 77 cases.

5. Prototype

The C++ Alliance Clang fork implements `std::core_ub` enforcement for 7 cases across the locally-checkable subset. [15] The prototype is available on Compiler Explorer as "clang (std::core_ub profile - P4317)". Each example below adds one `enforce` line and demonstrates a violation that the profile catches at runtime. Comment out the `enforce` line to see what the code does today; add `-O2` for the optimized answer.

Repository: https://github.com/cppalliance/clang/tree/profiles-core-ub

#	Case	Group	Link
1	`{basic.align.object.alignment}`	A.1	https://godbolt.org/z/bMG14s5cG
2	`{expr.unary.dereference}`	A.2	https://godbolt.org/z/obMj3EYTb
3	`{expr.add.out.of.bounds}`	A.2	https://godbolt.org/z/YbxWbxMP9
4	`{expr.mul.div.by.zero}`	A.1 + A.4	https://godbolt.org/z/s5Exo86K6
5	`{expr.mul.representable.type.result}`	A.1 + A.4	https://godbolt.org/z/3vTjbdnxP
6	`{expr.shift.neg.and.width}`	A.1 + A.4	https://godbolt.org/z/1nrccTq1s
7	`{conv.fpint.float.not.represented}`	A.1 + A.4	https://godbolt.org/z/dbcKb6W8s

Case identifiers are from P3100R8 Appendix A. Cases marked A.4 are those where the paper specifies a defined replacement value rather than termination - the implementation traps on those anyway.

5.1 Misaligned access

`{basic.align.object.alignment}` - A.1 - https://godbolt.org/z/bMG14s5cG

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

#include <cstdint>
#include <cstdio>

std::uint32_t source_addr(const unsigned char *frame) {
  return *reinterpret_cast<const std::uint32_t *>(frame + 14 + 12);
}

int main() {
  alignas(4) unsigned char frame[64] = {};
  frame[26] = 10; frame[29] = 1;
  std::printf("src = %08x\n", source_addr(frame));
}
```

- **Today:** `src = 0100000a` - correct at `-O0` and `-O2` alike
- **Enforced:** `Program returned: 132` - *misaligned pointer access under profile 'std::core_ub'*

An Ethernet header is 14 bytes, so every 32-bit IP header field lands at an odd multiple of 2. Correct on x86 for years; faults on a strict-alignment target.

5.2 Null dereference

`{expr.unary.dereference}` - A.2, null pointer case - https://godbolt.org/z/obMj3EYTb

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

#include <cstdio>

struct Session {
  int id;
  int session_id() const { return this ? id : -1; }
};

Session *lookup(int) { return nullptr; }

int main() {
  Session *s = lookup(42);
  std::printf("id = %d\n", s->session_id());
}
```

- **Today:** `-O0` -> `id = -1` - `-O2` -> **Segmentation fault**
- **Enforced:** `Program returned: 132` - *null pointer dereference under profile 'std::core_ub'*

A null `this` cannot happen, so at `-O2` the compiler deletes the test. The failure arrives with a compiler upgrade, not a code change.

5.3 Out-of-bounds subscript

`{expr.add.out.of.bounds}` - A.2, array bound statically known - https://godbolt.org/z/YbxWbxMP9

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

#include <cstdio>
#include <cstring>

struct Config { char name[16]; int port; };

void set_name(Config &c, const char *src) {
  std::size_t n = std::strlen(src);
  if (n > sizeof(c.name))
    return;
  for (std::size_t i = 0; i <= n; ++i)
    c.name[i] = src[i];
}

int main() {
  Config c{};
  c.port = 8080;
  set_name(c, "sixteen-char-key");
  std::printf("name = %s, port = %d\n", c.name, c.port);
}
```

- **Today:** `name = sixteen-char-key, port = 7936` - was 8080
- **Enforced:** `Program returned: 132` - *array index out of bounds under profile 'std::core_ub'*

The bound test is off by one and the loop copies the terminator too, so the write lands on the next member.

5.4 Division by zero

`{expr.mul.div.by.zero}` - A.1 + A.4 - https://godbolt.org/z/s5Exo86K6

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

#include <cstdio>

long throughput(long bytes, long elapsed_seconds) {
  return bytes / elapsed_seconds;
}

int main() {
  std::printf("%ld bytes/sec\n", throughput(4096, 0));
}
```

- **Today:** `-O0` -> **SIGFPE** (returned 136) - `-O2` -> `140728979890376 bytes/sec`
- **Enforced:** `Program returned: 132` - *integer division or remainder by zero under profile 'std::core_ub'*
- **Per A.4:** the division yields an erroneous value; the program continues

P4317 Section 4's own worked case. A transfer finishing inside one clock tick measures zero elapsed seconds. At `-O2` the optimizer takes the divisor for nonzero and never issues the division at all.

5.5 Signed overflow

`{expr.mul.representable.type.result}` - A.1 + A.4 - https://godbolt.org/z/3vTjbdnxP

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

#include <cstdio>

struct Entry { char name[16]; };

int table_bytes(const unsigned char *hdr) {
  int count = hdr[0] | (hdr[1] << 8) | (hdr[2] << 16) | (hdr[3] << 24);
  return count * (int)sizeof(Entry);
}

int main() {
  const unsigned char hdr[4] = {0, 0, 0, 0x10};
  int bytes = table_bytes(hdr);
  if (bytes > (1 << 20)) {
    std::puts("header rejected");
    return 1;
  }
  std::printf("allocating %d bytes\n", bytes);
}
```

- **Today:** `allocating 0 bytes` - the `> 1 MB` guard passes because `0 <= 1 MB`
- **Enforced:** `Program returned: 132` - *signed integer overflow under profile 'std::core_ub'*
- **Per A.4:** the multiplication yields an erroneous value, which the guard still passes

The shape behind a long line of CVEs. The erroneous-value replacement does not rescue this one: the overflow is upstream of the programmer's own check, so only termination stops the zero-byte allocation.

5.6 Shift out of range

`{expr.shift.neg.and.width}` - A.1 + A.4 - https://godbolt.org/z/1nrccTq1s

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

#include <cstdint>
#include <cstdio>

std::uint32_t extract(std::uint32_t word, unsigned pos, unsigned width) {
  std::uint32_t mask = (1u << width) - 1;
  return (word >> pos) & mask;
}

int main() {
  std::printf("%08x\n", extract(0xdeadbeef, 0, 32));
}
```

- **Today:** `-O0` -> `00000000` - `-O2` -> `67c80238`, `cd9977e8`, `97528678` - a different value on every run
- **Enforced:** `Program returned: 132` - *shift out of bounds under profile 'std::core_ub'*
- **Per A.4:** the shift yields an erroneous value, the same one on every conforming implementation

Every width from 1 to 31 is correct; the full-width field shifts by 32.

5.7 Float-to-integer conversion

`{conv.fpint.float.not.represented}` - A.1 + A.4 - https://godbolt.org/z/dbcKb6W8s

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

#include <cstdio>

int hit_rate(long hits, long total) {
  double pct = 100.0 * (double)hits / (double)total;
  return (int)pct;
}

int main() {
  std::printf("hit rate = %d%%\n", hit_rate(0, 0));
}
```

- **Today:** `-O0` -> `hit rate = -2147483648%` - `-O2` -> `975548632%`, different again next run

- **Enforced:** `Program returned: 132` - *floating-point to integer conversion overflow under profile 'std::core_ub'*
- **Per A.4:** the conversion yields an erroneous value

The division by zero is fine - floating point defines it. The conversion is what is undefined, and the result arrives downstream as an ordinary-looking integer.

Outputs captured from clang 24.0.0git @ `6b9ff00`, the build behind the Compiler Explorer entry, on x86-64 Linux. Undefined values differ between runs by nature; the trap does not.

6. Checking Tiers and Composition

The full guarantee is all 77 cases. "Fully instrumented" is a target, not current capability.

ASan misses stack and global use-after-free. UBSan's vptr check misses non-polymorphic type errors. D4277R0 [6] reports 38% of subcategories with no checks on either prototype compiler. P3100R8 faces identical limits.

19 locally checkable cases carry negligible cost at any optimization level. 58 instrumented cases carry sanitizer-class cost.

The cost story is not uniform across the 19 locally checkable cases. Some require a runtime check; at least one may not require a runtime mechanism at all:

> *"If we forward P3100 with the implicit precondition, then future papers have to amend the existing standard wording rather than starting from a clean slate. That's a hard bar." - Vinnie Falco (EWG telecon, 10 August 2026)*

Making flow-off-end ill-formed requires no runtime check, no handler, and no profile. The compiler rejects the program. The claim that this would be expensive has not been substantiated:

> *"There was a suggestion that this causes a massive performance regression. Has that been measured? I'm asking as a compiler writer who writes this sort of static analysis on a daily basis. Has that difference been measured?" - Ville Voutilainen (quoted with permission, EWG telecon, 10 August 2026), on making flow-off-end-of-function ill-formed instead of UB*

An implementation may ship the locally checkable subset as a cheaper build mode (analogous to libc++ fast/extensive/debug tiers). Such a mode is an adoption aid below the profile. Enforcing the profile means all 77.

Cross-TU: enforcement is per-region. Locally checkable cases hold regardless of how other TUs were compiled. Instrumented cases degrade gracefully under partial instrumentation - partial coverage yields partial diagnosis, never a false guarantee.

The ABI boundary for instrumented cases (shadow state, lifetime records) is inherent to the instrumentation, not to the routing. P3100R8 faces the identical boundary. The cross-TU instrumentation cost (shadow state, lifetime metadata, ABI

surface) is inherent to the sanitizer, not to the routing; the profile imposes no cross-TU overhead beyond what P3100R8 faces for the same cases.

7. Deployed Practice

Table 5: Deployed production hardening

Implementation	Shipped	Category	Response	Measured cost	Scale
libc++ hardening	LLVM 18, 2024	library preconditions	trap	~0.30% (Google)	hundreds of millions of LoC
libstdc++ assertions	GCC 6, 2016	library preconditions	diagnostic, `abort()`	not separately reported	default at `-O0` since GCC 15.1
MSVC STL hardening	VS 2022 17.14, 2025	library preconditions	`__fastfail`	not separately reported	opt-in
WebKit	2024	library preconditions	trap (libc++ extensive)	not separately published	release builds
Firefox	2025	library preconditions	vendor-selected	not separately published	opt macOS default; release pending
Android UBSan	Android 7.0, 2016	core-language: arithmetic, bounds	abort	not public	per-component (media, Bluetooth)
Chrome CFI	production	core-language: control flow	SIGILL	not public	official builds
Apple `-fbounds-safety`	production	core-language: bounds	deterministic trap	not public	millions of LoC of C

Every row terminates on a violation. None constructs a violation object. None routes through a replaceable handler.

Three core-language rows (Android, Chrome, Apple) check subsets of the 77 cases. No deployed system yet checks the full 77-case scope; the profile's complete guarantee is a standardization target, not a report of current practice.

Google's 0.30% figure [10] measures library-precondition hardening, not core-language type-and-lifetime instrumentation. The profile does not claim its full guarantee at 0.30%.

libc++ authors identify their trap as "precisely the quick-enforce evaluation semantic" of C++26 Contracts. [11]

Section 5 demonstrates the profile's own prototype: 7 locally-checkable cases enforced under `std::core_ub`, with live Compiler Explorer links.

8. Potential Concerns

Concern 1: the profile's implementation is partial

A prototype covers 7 locally-checkable cases (Section 5), available on Compiler Explorer. The full 77-case scope - including the 58 instrumented cases - remains a target, not current capability.

The checking is deployed technology (sanitizers, the hardened libraries of Section 7). The checking instrumentation is the same work under either routing. The P3589R2 [3] framework has a public Clang implementation; the `std::core_ub` prototype [15] builds on it. D4277R0 [6] reports prototype checks on GCC and Clang p3850 branches (38% of subcategories uncovered). Those prototype checks are the same instrumentation either routing can use.

Concern 2: deployed systems check library preconditions, not core-language cases

Partly true. Apple `-fbounds-safety`, Android IntSan/BoundSan, and Chrome CFI check core-language subsets in production.

The type-and-lifetime subset (most of the 58 instrumented cases) is not yet a shipped production default. The profile's form, response, and per-build activation are the deployed shape. The enumeration says what an implementation must eventually check.

Concern 3: the profile terminates, so it cannot be adopted into working legacy code

15 defined-replacement cases do not terminate (including signed overflow to wraparound). `[[profiles::suppress(std::core_ub)]]` is the in-source escape; `[[profiles::enforce(std::core_ub)]]` widens enforcement one TU at a time.

Bloomberg's `bsls_review` [8] logs and continues at the library level (post-violation state defined); `bsls_assert` [8] terminates where the state is language-undefined. The profile takes the same line.

The non-returning handler still runs for logging before the program ends. Termination does not cost telemetry. P4310R1 [5] sets out the full case.

9. Questions for the Committee

SD-10 [12] (adopted December 2024) governs evolution design. P3970R0 [13] (Direction Group, January 2026) designates Profiles as the primary strategy for C++29 safety. Thirteen direction polls over four years support continued Profiles work.

> *Question 1. Should proposals for the runtime checking of core-language undefined behavior follow the design principles in SD-10?*

> *Question 2. Should proposals for the runtime checking of core-language undefined behavior be informed by implementation and deployment experience?*

> *Question 3. Is a standard profile `std::core_ub` that guards the runtime-checkable cases of core-language undefined behavior (as enumerated by P3100R8) under the P3589R2 Profiles framework worth further work?*

The three questions locate the profile within the direction the committee has already chosen.

10. Conclusion

`std::core_ub` guards the 77 cases with a single profile under P3589R2 [3]. Zero foundational changes; the alternative routing requires six. The profile standardizes the deployed form: named checks, per-build activation, terminating response. The profile owns its guarantee, enumeration, and response. `noexcept` is untouched.

P4297R1 [1] asks EWG to sever the architecture from the wording. This paper shows the severed form works.

The enumeration belongs to P3100R8 [2]. What remains is the response and replacement behaviors, still to be settled.

11. Disclosure

Vinnie Falco is the founder of the C++ Alliance, which funds a Clang implementation and a GCC implementation of the Profiles framework; the Clang implementation is public, with regularly released experimental builds that implement the framework attributes, an initial slice of the `std::init` profile, and a prototype of `std::core_ub` covering 7 locally-checkable cases (Section 5).

This paper describes a profile specification. It does not propose wording. This is a companion to P4297R1, P4306R1, and P4310R1 in the August 2026 mailing. It works from the published record and uses machine-assisted drafting.

Acknowledgments

- Timur Doumler and Joshua Berne: exhaustive enumeration and classification in P3100R8. Appendix A is their work.
- John Lakos: Bloomberg assertion facilities (`bsls_assert`, `bsls_review`) inform violation-response options.
- Gabriel Dos Reis: Profiles framework of P3589R2.
- Can Cagri: review of R1.

- Bjarne Stroustrup: Profiles concept, D&E principles, P3984R0.
- John Spicer: EWG telecon contributions quoted with permission.

References

[1] P4297R1 - "Severing P3100's Profiles Claim from Its Case-by-Case Review" (Vinnie Falco, Ville Voutilainen, 2026). https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2026/p4297r1.pdf

[2] P3100R8 - "A framework for systematically addressing undefined behaviour in the C++ Standard" (Timur Doumler, Joshua Berne, 2026). https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2026/p3100r8.pdf

[3] P3589R2 - "C++ Profiles: The Framework" (Gabriel Dos Reis, 2025). https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/p3589r2.pdf

[4] P4306R1 - "Configuring Runtime Checking: Profiles and Implicit Contract Assertions" (Vinnie Falco, Ville Voutilainen, 2026). https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2026/p4306r1.pdf

[5] P4310R1 - "Hasta la Vista, Undefined Behavior: Why std::core_ub Should Terminate by Default" (Falco, Voutilainen, 2026). https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2026/p4310r1.pdf

[6] D4277R0 - "Overview and Implementation Report for P3100" (Berne, 2026). https://isocpp.org/files/papers/D4277R0.pdf

[7] P3984R0 - "A type-safety profile" (Bjarne Stroustrup, 2026). https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2026/p3984r0.pdf

[8] bsls_assert and bsls_review component documentation (Bloomberg BDE, retrieved 2026). https://bloomberg.github.io/bde-resources/doxygen/bde_api_prod/group__bsls__assert.html and https://bloomberg.github.io/bde-resources/doxygen/bde_api_prod/group__bsls__review.html

[9] P3608R0 - "Contracts and profiles: what can we reasonably ship in C++26" (Voutilainen, Wakely, Dos Reis, 2025). https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/p3608r0.html

[10] Retrofitting spatial safety to hundreds of millions of lines of C++ (Rebert, Yasuda, Shavrick, Google Security Blog, 2024-11-15). https://security.googleblog.com/2024/11/retrofitting-spatial-safety-to-hundreds.html

[11] Practical Security in Production (Dionne, Rebert, Shavrick, Varlamov, ACM Queue Vol. 23 Iss. 5, 2025). https://queue.acm.org/detail.cfm?id=3773097

[12] SD-10 - "Language Evolution (EWG) Principles" (EWG chairs, 2024-12-02). https://isocpp.org/std/standing-documents/sd-10-language-evolution-principles

[13] P3970R0 - "Profiles and Safety: a call to action" (Vandevoorde, Garland, McKenney, Orr, Stroustrup, Wong, 2026). https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2026/p3970r0.pdf

[14] P4308R1 - "Eight Responses to a Throwing Implicit Contract Assertion" (Vinnie Falco, Ville Voutilainen, 2026). https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2026/p4308r1.pdf

[15] C++ Alliance Clang fork, `profiles-core-ub` branch (Krystian Stasiowski, 2026).
https://github.com/cppalliance/clang/tree/profiles-core-ub

Appendix A: Enumeration of Guarded Operations

The enumeration below is the work of Doumler and Berne, reproduced from P3100R8 Appendix A. Their exhaustive identification of every case of explicit core-language undefined behavior, the classification of each by diagnosability, the checking strategies, and the replacement behaviors are the foundation this profile stands on. The 77 runtime-checkable cases are grouped here by whether a check can be performed locally; the three cases P3100R8 identifies as not runtime-checkable are omitted.

A.1 constitutes the profile - the 19 Enforceable Behavior cases. **A.2, A.3, and A.4 constitute the extension surface**, to be added as instrumentation matures.

A.1 Locally checkable (19 cases)

No cross-program instrumentation is required; these are checkable at any optimization level.

Identifier	Clause	Checking strategy
`{basic.align.object.alignment}`	[basic.align]/1	Insert alignment check
`{expr.mptr.oper.member.func.null}`	[expr.mptr.oper]/6	Insert null pointer check
`{expr.assign.overlap}`	[expr.assign]/7	Check overlap of the two address ranges
`{class.abstract.pure.virtual}`	[class.abstract]/6	Insert `pre(false)` into the pure-virtual stub
`{expr.expr.eval}`	[expr.pre]/4	Check the value is valid
`{conv.double.out.of.range}`	[conv.double]/2	Check the value is valid
`{conv.fpint.float.not.represented}`	[conv.fpint]/1	Check the value is valid
`{conv.fpint.int.not.represented}`	[conv.fpint]/2	Check the value is valid
`{expr.static.cast.enum.outside.range}`	[expr.static.cast]/9	Check the value is valid
`{expr.static.cast.fp.outside.range}`	[expr.static.cast]/10	Check the value is valid
`{expr.mul.div.by.zero}`	[expr.mul]/4	Check the divisor is nonzero
`{expr.mul.representable.type.result}`	[expr.mul]/4	Check the value is valid
`{expr.shift.neg.and.width}`	[expr.shift]/1	Check the right operand is valid
`{intro.execution.unsequenced.modification}`	[conv.rank]/10	Check unsequenced read and write refer to the same address
`{stmt.return.flow.off}`	[stmt.return]/4	`contract_assert(false)` at end of function body (if a separate proposal makes flow-off-end ill-formed, no UB remains and this case drops from the profile automatically)
`{dcl.attr.noreturn.eventually.returns}`	[dcl.attr.noreturn]/2	Insert `post(false)`
`{basic.stc.alloc.dealloc.throw}`	[basic.stc.dynamic.deallocation]/4	Assertion in a catch handler

Identifier	Clause	Checking strategy
`{expr.new.non.allocating.null}`	[expr.new]/22	Insert `post(r: r)`
`{stmt.return.coroutine.flow.off}`	[stmt.return.coroutine]/3	`contract_assert(false)` at end if no `return_void`

A.2 Locally checkable only in special cases (6 cases)

Checkable locally under the stated condition; otherwise they require instrumentation.

Identifier	Clause	Condition	Checking strategy
`{expr.add.out.of.bounds}`	[expr.add]/4	array bound statically known	Track pointer provenance, insert bounds check
`{expr.add.sub.diff.pointers}`	[expr.add]/4	array bound statically known	Track pointer provenance, insert bounds check
`{conv.ptr.virtual.base}`	[conv.ptr]/3	null pointer case	Track lifetime and type, or ctor-dtor state; null check
`{expr.dynamic.cast.pointer.lifetime}`	[expr.dynamic.cast]/7	null pointer case	Track lifetime and type, or ctor-dtor state; null check
`{expr.static.cast.downcast.wrong.derived.type}`	[expr.static.cast]/11	null pointer case	Track lifetime and type, or ctor-dtor state; null check
`{expr.unary.dereference}`	[expr.unary.op]/1	null pointer case	Track lifetime and type, and function address; null check

A.3 Not locally checkable (52 cases)

These require whole-program instrumentation of the kind sanitizers provide. Grouped by category for reference.

Initialization (1)

Identifier	Clause	Checking strategy
`{basic.indet.value}`	[basic.indet]/2	Track whether storage has been initialised

Bounds (3)

Identifier	Clause	Checking strategy
`{basic.stc.alloc.zero.dereference}`	[basic.stc.dynamic.allocation]/2	Track pointer provenance, insert bounds check
`{expr.delete.mismatch}`	[expr.delete]/2	Track pointer provenance, insert bounds check
`{expr.delete.array.mismatch}`	[expr.delete]/2	Track pointer provenance, insert bounds check

Type and Lifetime, object lifetime and type (18)

Identifier	Clause	Checking strategy
{intro.object.implicit.create}	[intro.object]/11	Track whether storage can hold implicit-lifetime objects
{intro.object.implicit.pointer}	[intro.object]/11	Track whether storage can hold implicit-lifetime objects
{lifetime.outside.pointer.delete}	[basic.life]/7	Track lifetime and type of storage
{lifetime.outside.pointer.member}	[basic.life]/7	Track lifetime and type of storage
{lifetime.outside.pointer.virtual}	[basic.life]/7	Track lifetime and type of storage
{lifetime.outside.pointer.dynamic.cast}	[basic.life]/7	Track lifetime and type of storage
{lifetime.outside.glvalue.access}	[basic.life]/8	Track lifetime and type of storage
{lifetime.outside.glvalue.member}	[basic.life]/8	Track lifetime and type of storage
{lifetime.outside.glvalue.virtual}	[basic.life]/8	Track lifetime and type of storage
{lifetime.outside.glvalue.dynamic.cast}	[basic.life]/8	Track lifetime and type of storage
{original.type.implicit.destructor}	[basic.life]/11	Track lifetime and type of storage
{expr.basic.lvalue.strict.aliasing.violation}	[basic.lval]/11.3	Track lifetime and type of storage
{expr.basic.lvalue.union.initialization}	[basic.lval]/11.3	Track lifetime and type of storage
{expr.ref.member.not.similar}	[expr.ref]/9	Track lifetime and type of storage
{expr.dynamic.cast.glvalue.lifetime}	[expr.dynamic.cast]/7	Track lifetime and type, or ctor-dtor state
{expr.static.cast.base.class}	[expr.static.cast]/2	Track lifetime and type of storage
{expr.add.not.similar}	[expr.add]/6	Track whether storage holds an object of the correct type
{class.dtor.no.longer.exists}	[class.dtor]/18	Track lifetime and type of storage

Type and Lifetime, allocation, const, and volatile (6)

Identifier	Clause	Checking strategy
`{creating.within.const.complete.obj}`	[basic.life]/12	Track whether storage holds a const object
`{basic.compound.invalid.pointer}`	[basic.compound]/4	Track whether storage has been allocated and freed
`{expr.type.reference.lifetime}`	[expr.type]/1	Track whether storage has been allocated and freed
`{conv.lval.valid.representation}`	[conv.lval]/3.4	Track lifetime and type of storage
`{dcl.type.cv.modify.const.obj}`	[dcl.type.cv]/4	Track whether storage holds a const object
`{dcl.type.cv.access.volatile}`	[dcl.type.cv]/5	Track whether storage holds a volatile object

Type and Lifetime, function, member-pointer, and reference types (9)

Identifier	Clause	Checking strategy
`{conv.member.missing.member}`	[conv.mem]/2	Track which type the pointer-to-member originated from
`{expr.call.different.type}`	[expr.call]/5	Track function type by address
`{expr.static.cast.does.not.contain.orignal.member}`	[expr.static.cast]/12	Track which type the pointer-to-member originated from
`{expr.delete.dynamic.type.differ}`	[expr.delete]/3	Track dynamic type of non-polymorphic objects
`{expr.delete.dynamic.array.dynamic.type.differ}`	[expr.delete]/3	Track dynamic type of non-polymorphic objects
`{expr.mptr.oper.not.contain.member}`	[expr.mptr.oper]/4	Track pointer-to-member origin and dynamic type
`{dcl.ref.incompatible.function}`	[dcl.ref]/6	Track function types by address
`{dcl.ref.incompatible.type}`	[dcl.ref]/6	Track whether storage holds an object of the correct type
`{dcl.ref.uninitialized.reference}`	[dcl.ref]/6	Track whether references have been initialised

Type and Lifetime, construction and destruction state (9)

Identifier	Clause	Checking strategy
`{class.base.init.mem.fun}`	[class.base.init]/16	Track whether objects are being constructed or destroyed
`{class.cdtor.before.ctor}`	[class.cdtor]/1	Track whether objects are being constructed or destroyed
`{class.cdtor.after.dtor}`	[class.cdtor]/1	Track whether objects are being constructed or destroyed
`{class.cdtor.convert.pointer}`	[class.cdtor]/3	Track whether objects are being constructed or destroyed
`{class.cdtor.form.pointer}`	[class.cdtor]/3	Track whether objects are being constructed or destroyed
`{class.cdtor.virtual.not.x}`	[class.cdtor]/4	Track whether objects are being constructed or destroyed
`{class.cdtor.typeid}`	[class.cdtor]/5	Track whether objects are being constructed or destroyed
`{class.cdtor.dynamic.cast}`	[class.cdtor]/6	Track whether objects are being constructed or destroyed
`{except.handle.handler.ctor.dtor}`	[except.handle]/11	Track whether objects are being constructed or destroyed

Threading (1)

Identifier	Clause	Checking strategy
`{intro.races.data}`	[intro.races]/17	Track inter-thread access and synchronization (TSan-style; a subset only)

Control Flow (3)

Identifier	Clause	Checking strategy
`{basic.start.main.exit.during.destruction}`	[basic.start.main]/4	Track whether static or thread-local objects are being destroyed
`{basic.start.term.use.after.destruction}`	[basic.start.term]/4	Track the lifetime of static objects
`{stmt.dcl.local.static.init.recursive}`	[stmt.dcl]/3	Recursion counter in the static and thread-local init guard

Coroutines (2)

Identifier	Clause	Checking strategy
`{dcl.fct.def.coroutine.resume.not.suspended}`	[dcl.fct.def.coroutine]/9	Track the suspension state of each coroutine handle
`{dcl.fct.def.coroutine.destroy.not.suspended}`	[dcl.fct.def.coroutine]/12	Track the suspension state of each coroutine handle

A.4 Cases with well-defined replacement behavior (15 cases)

The other 62 guarded cases have no replacement: a violation ends the program. For these 15 the profile adopts the defined behavior below in place of termination, fixed for every conforming implementation (12 unconditional, 3 for built-in types only; for those 3 the replacement applies to built-in types and the operation terminates otherwise).

Identifier	Replacement behavior
`{basic.indet.value}`	Erroneous value (built-in types only)
`{conv.lval.valid.representation}`	Coerce invalid representations to erroneous values
`{expr.expr.eval}`	Coerce to erroneous value
`{conv.double.out.of.range}`	Coerce to erroneous value
`{conv.fpint.float.not.represented}`	Coerce to erroneous value
`{conv.fpint.int.not.represented}`	Coerce to erroneous value
`{expr.static.cast.enum.outside.range}`	Coerce to erroneous value
`{expr.static.cast.fp.outside.range}`	Coerce to erroneous value
`{expr.mul.div.by.zero}`	Coerce to erroneous value
`{expr.mul.representable.type.result}`	Coerce to erroneous value
`{expr.shift.neg.and.width}`	Coerce to erroneous value
`{intro.races.data}`	Make primitive memory accesses implicitly atomic
`{intro.execution.unsequenced.modification}`	Sequence the operations in some unspecified order
`{stmt.return.flow.off}`	Return erroneous value (built-in return types only)
`{stmt.return.coroutine.flow.off}`	Return erroneous value (built-in return types only)