Eight Responses to a Throwing Implicit Contract Assertion

Document Number:	P4308R0
Date:	2026-07-13
Intent:	Inform
Audience:	EWG

Reply-to: Vinnie Falco vinnie.falco@gmail.com
Ville Voutilainen ville.voutilainen@gmail.com

Table of Contents

Abstract

The response space for a throwing implicit contract assertion contains at least eight options, not the four before EWG.

C++26 Contracts let a violation handler throw, and P3100 extends that mechanism to implicit assertions on core-language undefined behavior while holding the noexcept operator's value fixed, narrowing the response to Options A through D. This paper restores the foreclosed Option 0, adds E, F, and G, and compares all eight against requirements from P3100R8 and the public record; the requirements grid gives Option A genuine wins, while of the eight response shapes, four are deployed and two have prototypes and the two that throw are not deployed. The paper names no winner and requests nothing.

Revision History

R0: July 2026 (pre-Buzios mailing)

- Initial version.

---

1. Disclosure

The authors provide information and serve at the pleasure of the committee.

One author, Vinnie Falco, is the founder of the C++ Alliance and maintains work on the runtime checking of core-language undefined behavior that competes with the implicit-contract-assertion model examined here. Among the response options this paper compares, the authors prefer the family in which no exception escapes an implicit contract assertion, that is, the terminating and aborting options rather than the propagating one. That preference is a stake in the outcome, and the reader should weigh everything that follows accordingly. It bears only on whether an exception escapes an implicit assertion, not on whether the violation handler runs or whether a violation may be logged; the observe, log-and-continue capability is preserved by five of the eight options compared here, A, B, C, D, and 0, as Section 8.3 records.

This paper is `info`. It enumerates the response options for one question, states one finding, and requests nothing. The finding is that the option space for this question is larger than the four options currently before EWG.

The options labeled A through D come from P3100R8 [1]; the authors characterize that published position, do not speak for its authors, and have written the A-D material to a standard its authors could endorse. Option 0 and options E through G are the

authors' own contribution.

One limitation of this paper's method is that its comparison rests on a set of six requirements derived from P3100R8's own prose. A different requirement set could order the options differently. Section 8 discloses the set, names its provenance, and applies it identically to every option.

This paper is one of a series in the same mailing. Its companion is P4306R0 [2], "Configuring Runtime Checking: Profiles and Implicit Contract Assertions" (Vinnie Falco and Ville Voutilainen).

This paper was prepared with the assistance of generative tools. The authors are responsible for its content.

This paper asks for nothing.

---

2. Introduction

A violation handler in C++26 Contracts may exit by throwing, and P3100 brings core-language undefined behavior under the same contract mechanism. Once an implicit assertion can reach a throwing handler, an exception may escape an ordinary core-language expression that the `noexcept` operator reports as non-throwing. The response to that escape is the question the options in this paper answer.

Options A through D originate in P3100R8 [1], whose own proposal is Option A. Option 0 restores Option 1 of P3541R1 [6], the reading of the `noexcept` operator that P3100's premise forecloses. The companion paper P4306R0 [2] carries the configuration, dialect, and Profiles analysis of the same question; the present enumeration scopes to the response-option space and cross-references P4306R0 rather than repeating it.

This paper makes four contributions:

1. It enumerates eight responses to a throwing implicit contract assertion: Option 0, the four P3100 options A through D, and three further options, E, F, and G.
2. It restores Option 0, the option P3100's premise forecloses, and states both its case and the record that set it aside.
3. It proposes Options E, F, and G.
4. It compares all eight against one standard: six requirements drawn from P3100R8's prose together with five dimensions from the public record.

P3100 adopts the premise that an implicit contract assertion must not change the value of the `noexcept` operator (Section 3.3), which seven of the eight options satisfy; this paper grants that premise, restores Option 0 as the alternative it forecloses (Section 6), and takes the C++26 Contracts facility as given. It is scoped to implicit contract assertions on core-language expressions; explicit contract assertions, and the general question of whether C++29 should add non-throwing evaluation semantics, are out of scope except where an option touches them. Adopting P3100's "implicit contract assertion" vocabulary for the enumeration is a convenience of exposition, not a position on configuration ownership: whether a core-language check is owned by the Contracts facility or by the Profiles framework is the question the companion P4297R0 [44] asks EWG to decide and P4306R0 [2] compares, and nothing in this paper's comparison turns on the answer.

---

3. Background and Terms

This section defines the terms the rest of the paper uses and states the premise P3100 adopts. A reader fluent in C++26 Contracts can skip to the premise in Section 3.3; the definitions come first because the options differ only in details these terms make precise. Section 3.1 describes the P3100 model and the evaluation semantics, Section 3.2 describes the `noexcept` operator and the interaction that creates the decision, and Section 3.3 states the shared premise and the two questions the discussion conflates.

3.1. The P3100 model and the evaluation semantics

C++26 added Contracts through P2900R14 [3]: a function carries preconditions and postconditions, a statement carries `contract_assert`, and each such contract assertion is evaluated under one of a fixed set of evaluation semantics. P3100R8 [1] extends that mechanism from assertions the programmer writes to assertions the language inserts. Where a core-language expression can exhibit undefined behavior that an implementation can detect at run time - a signed-integer overflow, a null dereference, an out-of-bounds subscript - P3100R8 places an implicit contract assertion in front of the operation, checked under the same evaluation semantics as an explicit one. This is white-paper work on the C++29 track: it builds on the C++26 Contracts facility rather than changing it.

The evaluation semantic decides what a checked expression does when its predicate is false. C++26 defines four semantics, and P3100 adds a fifth for implicit assertions:

- *ignore*: the assertion is not checked, and the program proceeds as though it were absent.
- *observe*: the predicate is checked; on a violation the contract-violation handler is called, and if the handler returns, the program continues.
- *enforce*: the predicate is checked; on a violation the handler is called, and if it returns normally, the program is contract-terminated.
- *quick-enforce*: the predicate is checked; on a violation the program is terminated as quickly as possible, without calling the handler.
- *assume*: the predicate is not checked, and the compiler may generate code that relies on it holding. P3100R8 [1] permits this semantic for implicit assertions only.

The contract-violation handler named above is a single program-wide function that `observe` and `enforce` call on a violation. C++26 permits a program to replace it, and permits it to exit by throwing an exception. [3] That last permission is the source of the question this paper is about.

3.2. The noexcept operator and the interaction it creates

The `noexcept` operator answers a compile-time question about an expression: `noexcept(e)` yields `false` when the operand is potentially-throwing per [except.spec], and `true` otherwise. Two properties of the operator bear on what follows. First, since C++17 a non-throwing exception specification is part of a function's type (P0012R1 [4]), so the operator's answer participates in overload resolution and in name mangling, and therefore in the application binary interface (ABI). Second, the operator already disregards undefined behavior: CWG2792 [5] settled that its result depends on whether the operand is potentially-throwing per [except.spec], not on whether the operand might throw by way of undefined behavior.

The interaction P3100 must resolve follows from these two properties together. An implicit assertion evaluated with `observe` or `enforce` can call the violation handler, and the handler can throw. If it does, an exception leaves a core-language expression - `x + 1`, `*p`, `a[i]` - that the `noexcept` operator today may report as non-throwing.

3.3. The shared premise and the two questions

The premise that an implicit contract assertion must not change the `noexcept` operator's value has a real basis, and it is worth stating at its strongest. Moving the operator's answer would be a source-level breaking change. P3100R8 observes that "it is easy to construct a program with well-defined behaviour whose effects depend on the value of such a noexcept operator," [1] and code that dispatches on `noexcept(e)` - the standard-library move that selects the throwing-copy path when the move is potentially-throwing is the familiar case - would change behavior if the answer moved. The premise is also backed by a stated design principle of C++26 Contracts. P2900R14 Principle 3, "Concepts Do Not See Contracts," reads: "The mere presence of a contract assertion on a function or in a block of code should not change the satisfiability of a concept, the result of overload resolution and SFINAE, the branch selected by if constexpr, or the value returned by the noexcept operator." [3] And it is backed by the SG21 poll record on P3541R1: the value-reporting option was rejected and the fixed-value answer carried (Table 3, Polls 1 and 4). [6] [7]

One part of the premise is under-evidenced. The magnitude of the breaking change is unmeasured: the public record contains no study of how much code depends on the `noexcept` of an expression P3100 would make checkable, in either direction. The direction is unsettled as well: P3100R8 shows that a well-defined program's effects can depend on such a `noexcept` value, but not that the code so affected would misbehave rather than take a reasonable branch under either answer, leaving the change unestablished in kind as much as in size. So the premise rests on the principle and the poll, which are on the record, and on a breaking-change magnitude and direction that are not. The same standard - name the evidence, mark the gap - governs every deployment claim in Section 8.

Holding the operator's answer fixed forecloses one option that would otherwise complete the space: letting the operator return `false` for an expression that carries a potentially-throwing implicit assertion. That option is Option 1 of P3541R1 [6] and the subject of SG21's Poll 1. Rather than strike it from the enumeration, this paper restores it as Section 6, between the P3100 options and the authors' options, and gives it the same build-up-and-cost treatment as every other option, so the enumeration omits no response the public record contains.

Two questions travel together in this discussion, and the paper separates them. One is whether C++29 Contracts should gain non-throwing evaluation semantics in general, for explicit assertions as well as implicit ones. The other is what response

model P3100's implicit assertions use when a handler throws. This paper is about the second. It takes no position on the first beyond noting, for each option, whether that option would also answer it.

The scope boundary bears on the recovery motivation that several options invoke. Continuation after a violation is coherent in different ways across the two classes: an explicit, author-written precondition can encode a recoverable condition, so continuation there can be meaningful, whereas an implicit assertion fires at a point the language has left undefined. The recovery use case for a throwing response is therefore strongest for the explicit assertions this paper excludes; the present enumeration neither relies on that case nor forecloses it for the implicit assertions it does cover.

The remainder of the paper enumerates the responses to the second question - Option 0, the four P3100 options A through D, and the authors' options E through G - and compares them against six requirements drawn from P3100R8's own prose.

---

4. Options at a Glance

The response space is Option 0 together with Options A through G, and the two tables below preview all eight before Sections 5 through 7 define them in full. Table 1 scores each option against six requirements drawn from P3100R8's prose; Table 2 gives each option's response shape and points to its deployment record.

Table 1 scores each option against six requirements drawn from P3100R8's prose, in the section order A, B, C, D, 0, E, F, G. A cell reads *yes* when the option meets the requirement, *partial* when it meets it only under some configurations, and *no* when it does not. Requirements (1) noexcept value kept, (2) unwinding, and (3) noexcept meaning kept form a trilemma: an option that unwinds the stack from an implicit violation reaches an escaping exception either by holding the operator's value and shifting its meaning (Option A, keeping 1 and 2, losing 3) or by holding its meaning and moving its value (Option 0, keeping 2 and 3, losing 1), so at most two of the three hold at once and no option reads *yes* across all six. The table shows the trade in both directions.

Option	(1) noexcept value kept	(2) unwinding	(3) noexcept meaning kept	(4) no new exception-handling codegen	(5) user choice	(6) no new semantics
A - propagate	yes	yes	no	no	partial	yes
B - terminate on escape	yes	no	yes	yes	no	yes
C - add noexcept semantics	yes	yes	partial	partial	yes	no
D - implicit non-throwing only	yes	no	yes	yes	partial	no
0 - value-reporting noexcept	no	yes	yes	no	no	yes
E - no handler for implicit	yes	no	yes	yes	no	yes
F - handler then abort	yes	no	yes	yes	no	yes
G - contract-terminate	yes	no	yes	yes	no	yes

Table 2 gives each option's response when an implicit assertion's handler would throw, and a pointer to its deployment record, in the same section order. The response shape is definitional and is filled in here. The deployment experience is empirical, is established option by option in Section 8, and is shown there in this same table with the column completed. The deferral is on the merits: an option's response shape follows from its definition and is knowable now, whereas its deployment experience has to be substantiated per option, and P3100R8 Section 5.5 itself notes that Option A requires new codegen no shipping compiler provides today.

Option	Response shape	Deployment experience
A - propagate	throw	*see below*
B - terminate on escape	terminate	*see below*
C - add noexcept semantics	throw or terminate (selectable)	*see below*
D - implicit non-throwing only	terminate	*see below*
0 - value-reporting noexcept	throw	*see below*
E - no handler for implicit	trap, no handler	*see below*
F - handler then abort	abort	*see below*
G - contract-terminate	contract-terminate	*see below*

Deployment experience for each option is given in Section 8.

---

5. The P3100 Options (A, B, C, D)

P3100 identifies four options for what an implicit assertion does when its handler throws, labeled A through D, and this section presents them on their own terms. It opens by granting the premise the four share - that a throwing response to a contract violation is a legitimate thing to want - because the options are answers to that premise, and a reader who does not see why the premise is reasonable cannot weigh them fairly. Section 5.1 states that case; Sections 5.2 through 5.4 present the options, each with the properties it provides and the costs attached to its design. The comparative dimensions - deployment, security, and compatibility direction - are held for Section 8 and applied there identically to every option, so this section presents each option in its best light.

5.1. Why a throwing response is legitimate

A throwing violation handler is a deliberate feature of C++26 Contracts, not an oversight. Bjarne Stroustrup's P2698R0 [8], whose title is its thesis - "Unconditional termination is a serious problem" - argued that a contracts design offering only termination is unusable for programs that must not stop, and concluded that "a mechanism for not terminating after a contract violation is part of any minimally acceptable contract design." P2969R0 [9] records the outcome: Stroustrup's paper "led to the adoption of throwing violation handlers to the Contracts MVP." The committee chose a handler that can throw, on the merits, after argument.

The reasons are concrete, and each names a real deployment shape. A long-running service that answers requests for months without a restart can catch an exception at a request boundary, abandon the single request whose state is suspect, and keep serving the rest, where termination would end every in-flight request at once. A throwing handler routes a contract violation

into the same error channel a program already uses for recoverable failures - including the error channel of `std::execution` - so recovery composes with the surrounding code. A single program-wide handler gives every defect one reporting path, whether the defect is a library precondition or a core-language operation, so a deployment points one hookable function at its logging and telemetry (the central-handler design of P2900R14 [3]). And observe - log a violation and continue - is the semantic that staged hardening uses to bring a legacy codebase under checking without stopping it, which requires a handler that runs and returns. Options A through D preserve a role for the handler for these reasons, and differ only in whether, and how, they let the handler's exception escape. These reasons are the throwing response at its strongest; how far each transfers from an explicit precondition, whose suspect state the program can still reason about, to an implicit core-language violation, whose state the language does not define, is weighed in Section 8.3 rather than here.

5.2. Option A: the exception propagates

Option A is P3100R8's proposal and reads, in its own words: "If the evaluation of a core-language expression leads to a violation of an implicit contract assertion and that assertion is evaluated with the observe or enforce semantic, the contract-violation handler is called; if that handler exits via an exception, that exception propagates out of the enclosing core-language expression. At the same time, the noexcept operator still returns the same value for that core-language expression as before; that value may be true." [1]

Option A provides four properties:

- It provides the full set of C++26 evaluation semantics for implicit assertions, so an implicit assertion behaves exactly as an explicit one does, with no new semantic to select.
- It provides a stack-unwinding path from an implicit violation, which is the recovery use case P3318R0 [10] documents.
- It provides observe - log a violation and continue - for implicit assertions, because the handler runs and may return.
- It preserves the object program, because the checking is opt-in and the operator's answer is unchanged, so no well-defined program's generated code has to change.

SG21's recorded preference is for Option A. P3100R8 states it plainly: Options A and B "have first been proposed in [P3541R1]. This paper was discussed in SG21 and resulted in strong consensus in favour of Option A and against Option B because the latter precludes the stack-unwinding approach to handling implicit contract violations." [1]

Option A's design carries two costs, and the strongest available response to each is stated with it. The first is implementation work with no measurements yet: for an exception to escape a core-language expression correctly, an implementation must treat every checkable core-language expression as a potential throw site and generate the matching exception-handling metadata, which no shipping compiler does for these expressions today. The bounded-engineering response is that the capability already exists - GCC's `-fnon-call-exceptions` [11] generates code in which trapping instructions, "i.e. memory references or floating-point instructions," can throw, which is the shape Option A needs - and the limit is that the same flag is also evidence of the cost, because it constrains the optimizer and no measurement of Option A's specific overhead exists on the record. Option A is feasible on this evidence and unmeasured on this evidence, and both halves hold.

The second cost is a shift in what the `noexcept` operator means. The operator's value does not change, so there is no source break, but its meaning does: P3100R8 states that the operator no longer means "evaluating this expression cannot throw"

but "evaluating this expression cannot throw unless there is a contract violation." [1] The response is that the shift is confined to paths that are undefined behavior today, so the new reading narrows to exactly the cases P3100R8 newly defines. P3100R8 notes that throwing from such expressions "has existing practice today" in sanitizer callbacks, and states in the same place that this "works today, but not in all cases," because a compiler unaware of the now potentially-throwing nature of the expression "may therefore not correctly generate the necessary exception-handling metadata." [1] Section 8.3 records where the counter to the sanitizer analogy is weighed.

5.3. Option B: terminate on escape

Option B forbids the escape: "The evaluation of an implicit contract assertion is not allowed to exit via an exception; if the contract-violation handler is called and exits via an exception, std::terminate is called instead." [1]

Option B provides three properties:

- It provides the C++26 exception model unchanged, with no new codegen and no new exception-handling metadata for core-language expressions, because nothing new can throw through them.
- It provides the `noexcept` operator its exact current meaning, with no conceptual shift, because an implicit assertion can never make an expression throw.
- It provides observe and enforce for implicit assertions up to the handler's return: a handler may still log and continue under observe, and only an escaping exception is converted to termination.

Option B's cost is the one SG21 weighed against it: it precludes unwinding the stack in response to an implicit violation, which P3100R8 records as having "important use cases (see [P3318R0])." [1] [10] P3100R8 also notes that a user can obtain Option B's behavior under Option A by branching in the contract-violation handler "on whether the violated assertion is implicit," [1] which is the reason SG21 preferred the more general Option A. Option B provides the unchanged exception model at the cost of the recovery path, and that trade is the whole of the choice between A and B.

5.4. Options C and D: new non-throwing semantics

Options C and D add non-throwing evaluation semantics rather than fixing a single response. Option C introduces "new contract-evaluation semantics observe-noexcept and enforce-noexcept which have the same runtime effect as observe and enforce, respectively, except that if a contract-violation handler is called under those semantics, it can never exit via an exception and std::terminate is called instead," selectable "via an implementation-defined mechanism, just like the existing ones." [1] Option D is Option C "with the additional limitation that implicit contract assertions can never be evaluated with a semantic that could result in an exception escaping the contract check, thus excluding the regular observe or enforce semantics." [1] For implicit assertions this makes D essentially the same as Option B, with the difference that the new non-throwing semantics stay available for explicit contract assertions.

Option C provides three properties:

- It provides a per-configuration choice between escape and termination, because the throwing and non-throwing semantics coexist and are selected by the same implementation-defined mechanism that selects the existing semantics.

- It provides a written, prototyped form: D4298R0 [12] specifies exactly these two semantics (as `noexcept-observe` and `noexcept-enforce`) and reports an implementation in the P3850 branches of GCC and Clang on Compiler Explorer.
- It provides both a log-and-continue path (observe-noexcept returns and continues) and a no-escape guarantee (an escaping exception calls std::terminate), so a deployment can keep diagnostics without an escape path.

Option C's cost is the number of semantics: adding observe-noexcept and enforce-noexcept alongside observe and enforce doubles the checking semantics a programmer and an implementation reason about, and D4298R0 marks the enumerator values themselves as provisional. C is the only option in the enumeration that adds novel semantics while also keeping the escape path; every other option that preserves unwinding (A and 0) reuses the existing semantics, and every other option that adds a new semantic (D) forecloses it.

Option D provides three properties: it gives implicit assertions the unchanged `noexcept` meaning and exception model of Option B, because they never reach a throwing handler; it keeps the new non-throwing semantics available for explicit assertions, unlike Option B's blanket rule; and it forecloses the escape path for implicit assertions by construction. For implicit assertions D is essentially Option B, with the added ability to use the non-throwing semantics for explicit assertions as well, at the cost of removing the escape path Option A preserves.

Options A through D thus span the response space P3100 mapped: A lets the exception escape and keeps every semantic, B forbids escape and keeps the exception model unchanged, C offers both by adding semantics, and D offers the non-throwing side of C for implicit assertions. SG21's recorded consensus is for A. What each option looks like when the space is read against deployment, security, and compatibility is the subject of Section 8.

---

6. Option 0

Section 3.3 named the option foreclosed by the premise: letting the `noexcept` operator return `false` for an expression that carries a potentially-throwing implicit assertion. This section restores that option - Option 0 - and gives it the treatment every other option gets: the case for it at full strength, then the costs that removed it from the enumeration. Section 6.1 makes the case, Section 6.2 states the costs that end it, and Section 6.3 states the question the analysis leaves for Section 7.

6.1. The case for a value-reporting operator

Option 0 is the reading of `noexcept` that P3541R1 and P2969R0 describe as the natural one. P3541R1 states it as its Option 1: "Any construct that might end up in either undefined behavior or contract violation is potentially-throwing and this is reflected by the noexcept-operator." [6] P2969R0 reaches the same place from the other direction: "since contract checks can throw an exception, we should treat them as potentially-throwing in the language," which "would mean that the noexcept operator ... would return false," and the paper calls this treatment "straightforward and intuitive." [9] For an expression that can throw, an operator returning `false` reports the potentially-throwing value, and an operator returning `true` does not.

The computation is cheap, and the record says why. P2969R0 places the objection to Option 0 not in implementation cost but in the zero-overhead principle: treating checks as potentially-throwing "violates the zero overhead principle: the addition of a contract check to a program can lead to a different branch being taken at compile time." [9] The front end already knows

whether an expression carries an implicit assertion; reporting that fact through the operator reads information the compiler holds.

One consequence of Option A's own design sharpens the point. P3100R8 states that a conforming Option A implementation "has to treat all core-language expressions that can result in implicit contract-assertion violations as potentially-throwing for the purposes of generating the exception-handling metadata." [1] A value-reporting `noexcept` operator would report exactly that property. So the machinery Option A requires computes precisely what Option 0 would publish: Option 0 keeps the operator's meaning and changes its value, and Option A keeps the operator's value and changes its meaning.

6.2. Why the value-reporting operator does not survive

A value-reporting operator can take one of two forms, and each fails for a different reason. The two are numbered because this section answers both.

First, the build-mode-dependent form - the only form that reports whether *this* build can throw - makes the operator's answer depend on the evaluation semantic in force, and that dependency breaks linking. The evaluation semantic may be selected at "compile time, link time, or run time" (P2877R0 [13]), so in a translation unit that evaluates the implicit assertion with `ignore` or `assume`, `x + 1` cannot call the handler and `noexcept(x + 1)` is `true`, while in a translation unit that evaluates it with a throwing `observe` or `enforce`, `x + 1` can throw and `noexcept(x + 1)` is `false`.

This does not stay confined to expressions a programmer writes out by hand; it reaches the standard library's own type traits, which query `noexcept` on the expressions they are handed. `std::is_nothrow_invocable_v<void(*)(int) noexcept, float>` forms a call whose `float`-to-`int` argument conversion can overflow, an operation P3100 makes checkable, so under a build-mode-dependent operator the trait yields one constant-expression value in a translation unit built with `ignore` and another in one built with throwing `enforce`. The same token sequence naming one entity with two different values across translation units is itself a violation of the one-definition rule (ODR), and it seeds further violations wherever that trait feeds a class layout, an overload set, or a template specialization. `std::is_nothrow_invocable_v<void(*)() noexcept>` breaks the same way through a possibly-null function pointer. These are not contrived expressions: the standard library dispatches on exactly these traits - `std::vector` reallocation selects the move path over the copy path on `std::is_nothrow_move_constructible` - so a trait value that shifts with the build mode shifts library behavior with it.

The minimal mechanism is visible in a single declaration. Because `noexcept` has been part of the function type since C++17 (P0012R1 [4]) and is encoded in the mangling - the Itanium C++ ABI mangles a non-throwing function type with the `Do` production [14] - a function whose type depends on that answer mangles two different ways:

```
int x;
void h() noexcept(noexcept(x + 1));   // potentially throwing? depends on the build
```

In the `ignore` or `assume` translation unit, `h` has type `void() noexcept` and one mangled name; in the throwing-`observe` translation unit, `h` has type `void()` and a different mangled name. The two object files do not link, or link with mismatched declarations of one entity - the same ODR violation surfacing at link time. This is not a new failure mode: making `noexcept` part

of the type in C++17 changed mangled names, which is why GCC ships `-Wnoexcept-type` to "warn if the C++17 feature making noexcept part of a function type changes the mangled name of a symbol relative to C++14." [15] The build-mode-dependent operator re-triggers that episode, keyed to the contract build mode instead of the language version. P2834R1 reached the general form of this conclusion, finding that having the semantics of commonly used C++ language constructs depend on the contract-checking build mode is "anathema to any sensible notion of sound software engineering practice." [16]

Second, the unconditional form avoids the link break only by making the operator return `false` for such an expression in every build, whether or not this build can throw. That escapes the ODR problem by paying the source-level price Section 3.3 describes: `noexcept(x + 1)`, `noexcept(*p)`, and `noexcept(a[i])` flip from `true` to `false` for nearly every scalar and pointer expression in the language, the mass source-level change Section 3.3 described. The flip does not stop at expressions written by hand: it hollows out the standard library's noexcept-querying type traits, so that `std::is_nothrow_move_assignable_v<S>` reports `false` even for a trivial `struct S {}`, because the move-assignment expression it forms could touch an object outside its lifetime, and nearly every such trait changes its answer for the same reason. It also runs against the C++26 design principle directly: P2900R14 Principle 3 states that "the mere presence of a contract assertion ... should not change ... the value returned by the noexcept operator," [3] and the unconditional form is that change.

The value-reporting operator was before SG21, which rejected it and kept the operator's answer fixed: Poll 1 declined the unconditional form and Poll 4 carried the fixed value, with the configurable form of Poll 3 also declined (Table 3). [7]

One boundary keeps the targeting exact: the link break is Option 0's, not Option A's. Under Option A the operator's answer does not depend on the build mode - it stays `true` in every mode - so object files built under different contract modes agree on every mangled name and link cleanly. The C++26 principle "Concepts Do Not See Contracts" exists to guarantee that invariance. The ODR argument above reaches Option 0 and does not reach Option A.

6.3. The question the analysis leaves open

The value-reporting way to let an implicit assertion throw fails on both forms: the build-mode-dependent operator breaks linking, the unconditional operator is the mass source-level change Section 3.3 describes and the principle forbids, and SG21 polled the option down. The value-preserving way, Option A, remains available, and reaches an escaping exception by holding the operator's value while shifting its meaning (Section 5.2). With the value-reporting operator removed, the question the enumeration raises is no longer how to report a throwing implicit assertion, but whether an implicit assertion should invoke a throwing handler at all. Options E, F, and G answer that question, and Section 7 presents them.

---

7. Additional Options (E, F, G)

Sections 5 and 6 leave one question standing: whether an implicit assertion should invoke a throwing handler at all. Options E, F, and G answer it three ways - E removes the handler from implicit assertions, F gives them the response the committee already chose for `assert`, and G gives them the working draft's own ending - and this section presents each with the properties it provides and the cost it carries. It opens with those costs, in Section 7.1, because the case for these options is made by stating their limits at full weight and no more. The deployment record that bears on all three is held for Section 8 and applied there to every option alike.

7.1. What E, F, and G give up

Each option gives something up, and naming the loss precisely is the point of this subsection. Option E gives up the standard `observe` semantic for implicit assertions entirely: under E an implicit assertion cannot call the handler, so it cannot log a violation through the handler and continue. This is a real loss, and Section 7.2 states it in full. Sanitizer and tooling observation remains conforming under E - a build can still run UBSan or ASan and log - but that is tooling-based observation, not the contract handler, and the two are not the same facility. Option F gives up the escape path that Option A preserves, in exchange for the response `assert` already uses. Option G gives up a single hardwired ending, in exchange for the ending the working draft already defines. The endings these options give differ in a way that matters to a deployer: `std::terminate`, `abort()`, and contract-termination each leave a different interception point in place, so B, F, and G are distinct responses rather than one response relabeled. Section 7.4 records the interception point each ending exposes.

7.2. Option E: no handler for implicit assertions

Option E keeps the violation handler out of implicit contract assertions: they may be evaluated only with `ignore`, `assume`, or `quick-enforce`, the semantics that cannot call the handler. The exception question then cannot arise, because nothing in the failure path can throw.

Option E provides three properties:

- It provides the `noexcept` operator its exact C++26 meaning for every core-language operation, with no shift and no new semantics, because an implicit assertion can never reach a throwing handler.
- It provides a one-sentence specification. The wording mirrors a restriction P3100R8 already states in the other direction - "explicit contract assertions are never evaluated with the assume semantic" [1] - so E adds the symmetric clause restricting implicit assertions to the non-handler-calling semantics.
- It provides quick-enforce for implicit assertions, the terminating trap semantic the working draft already defines, so a violation still stops the program at the detection site.

Option E's core case is that one-sentence restriction, and its real cost is the observe gap. E does not answer the log-and-continue requirement, and it must not claim to: `ignore` continues silently with no handler and no diagnostic, quick-enforce terminates, and the sanitizer traps terminate too, so none is a substitute for `observe`. Where a deployment needs the handler to log an implicit violation and continue, E does not provide it.

Two counters to E deserve their strongest statement and a direct answer; they are numbered because there are two.

First, the observe-is-a-production-requirement counter: a deployment that must log and continue loses that capability for implicit assertions under E. Two facts frame the answer without disputing the requirement. Observe is satisfiable without any escaping exception - Options B, C, and D all provide the handler, and observe, while forbidding escape - so a requirement for observe does not by itself favor Option A over the no-escape options; it favors any option that keeps the handler, which among these E alone does not. And providing observe for implicit assertions has a structural consequence worth stating plainly: it routes core-language undefined behavior through the single contracts violation handler, which is a fact about where the handler sits in the checking stack, not a conclusion about which feature should own that slot. This paper states the routing fact and draws no ownership conclusion from it.

Second, the forward-compatibility counter: a guarantee that implicit assertions never call the handler is itself a guarantee that code will be written to rely on, and removing it later is a compatibility event. P3318R0 [10] states the general form of this concern for the opposite default - "closing that door closes it permanently, so that it can't be opened again" - and the concern applies to E's door as much as to A's. One asymmetry is genuine: non-escape by default is the safer ship-time posture, because moving from non-escape to escape later requires an explicit opt-in rather than a silent change, while the reverse does not. Section 8 gives the compatibility direction its full treatment and the evidence on both sides.

7.3. Option F: the assert response

Option F gives implicit contract assertions the response `assert` has carried since C: the violation handler is invoked in a non-throwing form, and then the program calls `abort()` - not `std::terminate`, and not contract-termination, but `abort()`, as `assert` does today.

Option F provides three properties:

- It provides the response shape the C standard mandates for the language's oldest assertion facility: a failed `assert` writes to the standard error stream and then calls `abort()`, shipped that way across every major C library (Section 8.3). P3311R0 [45] proposes routing that response through Contracts as an opt-in, extending "the recommended practice for the default contract violation handler to implement the behavior required by the C standard for a failed assertion", which it specifies as "writing select information to the standard error stream and then calling abort()".
- It provides a committee record on the throwing question specifically: SG22 polled whether the `assert` macro should let handler exceptions propagate and reached consensus against, in both its columns (WG14 0/0/0/5/2, WG21 1/0/2/7/2). [17]
- It provides diagnostics without an escape path: the handler still runs and reports, and the exception cannot escape by construction, because `abort()` follows the handler on every path.

Option F's cost is the escape path it forecloses, the same recovery use case P3318R0 documents for Option A. The question F answers is a consistency question: an implicit core-language check and a C `assert` both detect a defect the program did not expect, and F gives them the same response `assert` carries and that SG22 reviewed for C compatibility, rather than a different one.

7.4. Option G: the working draft's own ending

Option G addresses a narrower inconsistency, the endings. If the handler for an implicit assertion throws, Option G contract-terminates, using the ending the working draft already defines rather than a hardwired `std::terminate()` or `abort()`.

Option G provides three properties:

- It provides the ending the standard already specifies. When a program is contract-terminated, [basic.contract.eval] [18] makes it "implementation-defined (depending on context) whether: std::terminate is called, std::abort is called, or execution is terminated," the wording added by P3520R0 [19].

- It provides consistency across the three endings now in play: `abort()` in the `assert` integration, `std::terminate` in the library API and in Option B, and contract-termination in enforce's own normal ending. G resolves the event toward the one the facility already defines.
- It leaves the interception points to the implementation rather than hardwiring them: mandating `std::terminate()` hardwires the replaceable `std::set_terminate` handler, mandating `abort()` hardwires a catchable `SIGABRT`, and contract-termination hardwires neither, permitting even the smallest ending - plain termination with no library call - that quick-enforce already relies on.

Option G's cost is that its specific mapping is new. Among current proposals none maps a throwing implicit handler to contract-termination, and the direction under Option A is to propagate. G's principle - do not hardwire one ending - is the C++26 status quo for enforce; its application to the throwing-implicit case is the novel part.

7.5. Scope, and the poll record

One clarification applies to all three options: E, F, and G constrain the response of P3100's implicit contract assertions and nothing else. They do not speak to how any higher-level feature may define the meaning of operations that are undefined today.

The public poll record maps onto these options, and the mapping cuts both ways. SG21 took seven polls on the `noexcept` interaction when it discussed P3541R1 in January 2025, and Table 3 reproduces them.

Table 3. The SG21 poll record on P3541R1 (January 2025), the seven polls on the `noexcept` interaction. Every poll shares the preface "If P3081 Profiles add implicit contract checks to core language expressions such as pointer dereference or array indexing"; tallies are SF/F/N/A/SA, as recorded on the public tracker. [7]

Poll	Question, abridged	SF/F/N/A/SA	Result
1	the `noexcept` operator returns `false` for such expressions (Option 1)	0/2/0/7/9	Consensus against
2	contract-violation handlers must be `noexcept` (Option 2)	0/1/2/9/7	Consensus against
3	make the throw and the operator's value configurable (Option 3)	0/2/3/8/6	Consensus against
4	the `noexcept` operator returns `true` for such expressions (Option 4)	5/10/1/2/1	Consensus
5	restrict such checks to semantics that cannot call the handler (Option 5)	2/3/4/6/3	No consensus
6	unconditionally disallow a throw from such a check, calling `std::terminate`	1/1/5/7/3	Consensus against
7	optionally disallow a throw from such a check, calling `std::terminate`	3/9/4/0/0	Consensus

Polls 1 and 4 record the premise P3100 adopts (Section 3.3): SG21 rejected the value-reporting operator and kept the operator's answer fixed. Section 5.2's strong consensus for Option A, which P3100R8 reports, is consistent with Poll 4's preference for the fixed value; that record ranks the value-preserving choice against the value-reporting one, and Options E, F, and G were not before SG21, so it does not reach the wider space enumerated here.

On disallowing a throw, the record separates the mandatory form from the optional one. Poll 6, an unconditional bar ending in `std::terminate`, reached consensus against; Poll 7, an optional bar ending in `std::terminate`, reached consensus. Option B is the unconditional bar's exact shape, escape disallowed with `std::terminate` following, so it maps to Poll 6, and Option D takes that same unconditional shape for implicit assertions by construction. Options F and G provide a non-throwing terminating response in the optional opt-out's spirit rather than as a blanket rule, and their endings differ from Poll 7's `std::terminate`: F ends in `abort()` and G in contract-termination, so they realize that opt-out with a different ending rather than being that poll. Option E does not appear here at all: it restricts the available semantics rather than barring a throw from a handler-calling one, which is Poll 5's shape, recorded next.

Two options map to polls that did not carry. Poll 3, a configurable choice covering both whether a throw escapes such a check and what the operator returns for it, reached consensus against; Option C's per-configuration selection between observe and observe-noexcept realizes the throw-configurability half of that poll while holding the operator's value fixed, so it is a narrower shape than the one polled. Poll 5, restricting implicit checks to the semantics that cannot call the handler, reached no consensus; that restriction is Option E's exact shape.

One scope note keeps the mapping exact. Every poll in Table 3 is prefaced "If P3081 Profiles add implicit contract checks to core language expressions," so the polls were framed for the checks Profiles would add, while P3100 applies the same

response question to all core-language undefined behavior, a broader set than the polls named.

---

8. Reading the Comparison

This section reads the two tables from Section 4. It completes Table 2 - the deployment column, withheld in Section 4, is filled in here with every cell cited - then states what the requirements grid rests on and the five dimensions the grid does not show. Each dimension is a finding from the public record, applied to every option by the same standard. The section reports these dimensions and states no ranking.

8.1. The deployment record

The rows of Table 2 are unchanged from Section 4 and remain in section order; only the deployment column is now filled. No option is implemented as a P3100 implicit-assertion feature, because P3100 itself is unimplemented, so the column reports whether each option's response shape - throw, terminate, trap, or abort - has field deployment, through the option where one exists and through an analogue otherwise, and it applies that standard to every row. The vocabulary is: *none* (the response shape ships nowhere), *analogues only* (a related mechanism ships, but not this response for this case), *prototype* (one experimental or opt-in implementation), *universal* (standard-mandated or shipped everywhere), and *massive-fleet* (measured production deployment at scale).

Option	Response shape	Deployment experience
A - propagate	throw	none; analogues only - GCC `-fnon-call-exceptions` [11] lets trapping instructions throw and MSVC `/EHa` [20] catches such asynchronous faults, but no compiler throws from an implicit UB check
B - terminate on escape	terminate	universal as a response shape - terminate-on-escape from a `noexcept` boundary is standard since C++11 [21], and C++26 Contracts ship in GCC 16.1 (experimental) [22]
C - add noexcept semantics	throw or terminate (selectable)	prototype - D4298R0 is implemented in the P3850 experimental forks of GCC and Clang on Compiler Explorer, not in a release of either compiler [12]
D - implicit non-throwing only	terminate	terminate response deployed as in B; the implicit-only restriction itself is unimplemented, precedent only (P3100 restricts `assume` to implicit assertions by the same shape [1])
0 - value-reporting noexcept	throw	none; analogues only - the C++17 `noexcept`-in-type ABI change shipped with `-Wnoexcept-type` [15], never applied to a throwing implicit assertion
E - no handler for implicit	trap, no handler	massive-fleet as a response shape - libc++ hardening runs the trap response across hundreds of millions of lines at Google [23], though the E restriction itself is unimplemented as a P3100 feature; EWG reached no consensus to mandate a non-throwing default handler (P3577R0) [24]
F - handler then abort	abort	universal as a response shape - `assert` is diagnostic then `abort()` in glibc, musl, the MSVC CRT, and bionic [25], and libc++ hardening deploys the trap form at scale [26]; the integration itself is unimplemented as a P3100 feature
G - contract-terminate	contract-terminate	prototype - GCC 16.1 ships experimental Contracts [22]; the implementation-defined ending is specified and EWG-affirmed [18] [19] [27]

The completed column carries the asymmetry the requirements grid cannot show: the terminating, trapping, and aborting response shapes are deployed, while the throwing response of Options A and 0 has no implementation, and only the capability analogues exist for it.

8.2. What the requirements grid rests on

The six requirements in Table 1 are not neutral givens. They are drawn from the desiderata P3100R8's prose expresses, and P3100R8 is the paper that proposes Option A, so the standard reflects the priorities of a party to the question it judges. This paper applies them because they are the criteria the primary proposal puts on the record, and it applies them identically to every option, including its own: Table 1 scores Option E's observe gap as a *no* on user choice, and it scores Option A's real advantages as *yes* on unwinding, on keeping the operator's value, and on adding no new semantics. A different reader might weight the six differently, or add a seventh, and the ordering would move. The grid is a faithful reading of one party's standard, not a measurement of merit.

The grid also shows the standard cannot be fully met. Requirements (1) noexcept value kept, (2) unwinding, and (3) noexcept meaning kept form a trilemma: unwinding from an implicit violation forces the operator to give up either its value (Option 0) or its meaning (Option A), so at most two of the three hold at once. Every option therefore scores *no* on at least one requirement, and the choice among options is a choice of which one to give up.

Five dimensions bear on that choice and do not appear in the grid. Each is stated below as a finding from the record, applied to every option alike. Each of the five is drawn from the published record and cited in Section 8.3 where it is applied: the deployment survey from vendor documentation and release notes; the security dimension from P2784R0, the NDSS unwinding work, and the CERT rules; the compatibility direction from P1093R0, P2861R0, and P3229R1; the diagnostics dimension from the central-handler design of P2900R14; and the implementation dimension from D4298R0 and P3100R8 Section 5.5. As with the six requirements, a different reader might weight these five differently, or add a sixth, and the balance would move.

8.3. Five dimensions the grid does not show

Deployment lineage. On a detected core-language violation, every mechanism that ships terminates or traps, and none throws. The lineage is consistent: C `assert` ends in `abort()` [25], the sanitizers trap [28], Microsoft's `__fastfail` terminates without running handlers [29], Chromium's `CHECK` reaches an immediate crash [30], and C++26's own quick-enforce is a trap [18]. This is a fact about core-language-UB checks specifically, and it is scoped that way: throwing violation handlers were adopted deliberately, on Stroustrup's argument that "unconditional termination is a serious problem" [8] and by the decision P2969R0 records [9], so the lineage says nothing against throwing handlers in general - only that, for a detected core-language violation, the shipping practice is to stop.

Security posture. Running code, including a handler and any unwinding it starts, through state a check has just found corrupt is a documented exploit surface. The NDSS 2023 CHOP work shows a corrupted stack turning the exception unwinder into a control-flow-hijacking primitive that defeats shadow stacks [31], and in Rust, a panic during cleanup that then unwinds has produced a double-free exploitable to code execution [32]. Microsoft's `__fastfail` [29], SDL Heap Fail Fast [33], and glibc's

abort-on-corruption [34] all terminate rather than run code on corrupt state. P2784R0 traces the destructor cascade a throwing violation starts [35], and CERT ERR56-CPP [36], MSC54-CPP [37], and DCL57-CPP [38], with N3103 [39], record the same hazard from other directions. Two limits are stated plainly. First, no public CVE names a C++ contract-violation handler as the vector, because C++26 Contracts are not deployed in the field yet, so this dimension argues from mechanism and precedent, not from an incident. Second, the corrupted-state premise is strongest for the memory-safety violations - a null dereference or an out-of-bounds access leaves memory in a state an attacker can work with - and weaker for an arithmetic violation such as signed overflow, where the detected value is wrong but memory is not corrupt, so the exploit-surface finding applies most directly to the memory-safety subset of core-language undefined behavior. This concession is bounded, and the boundary keeps it from being read as a license to let the arithmetic subset escape: it bears on whether continuing past such a violation is dangerous, not on whether an exception may escape one. The pressure escape puts on the `noexcept` operator - the value-shift Option A accepts (Section 5.2) or the ODR break Option 0 pays (Section 6.2) - comes from the escape itself, not from memory corruption, and its canonical case is the arithmetic expression `x + 1` (Section 3.2). So the weaker security of the arithmetic subset reaches only the continue-into-defined question - whether a specified wrapped result may be observed and continued, which this paper leaves open - and not the escape question, which the `noexcept` account governs for every subset alike. The finding concerns running code through corruption in general, not the throwing-handler option uniquely.

Compatibility direction. The direction of a future change matters. Relaxing a no-throw guarantee is, in WG21's own compatibility calculus, a silent breaking change - P1093R0 classifies relaxing a postcondition that way [40] - and P2861R0 records that both removing and adding a `noexcept` guarantee are backward-compatibility events [41], while P3101R0 shows a throw path silently converting correct-but-not-exception-safe code into incorrect code [42]. C++26 itself redefined the `noexcept` operator specifically to keep the boundary source-compatible (P3229R1) [43]. One asymmetry is genuine, whichever side it favors: a non-escape default can be relaxed to escape later only through an explicit opt-in, while an escape default cannot be tightened without a break, so non-escape is the safer ship-time posture - the same one-way reasoning P3318R0 invokes for the opposite default [10]. The record does not settle the direction, and the risk sits on the escape side.

Diagnostics and observe. The log-and-continue capability - invoke the handler, record the violation, continue - is provided by A, B, C, D, and 0, and not by E, F, or G. It is orthogonal to escape: B, C, and D provide observe with no exception escaping, so a requirement for observe does not favor Option A over the other handler-keeping options. Providing observe for an implicit assertion routes core-language undefined behavior through the single contracts violation handler, which is the central-handler design of P2900R14 [3]; that routing is a structural fact about where the handler sits, stated here without any conclusion about which feature should own that slot. Routing through the handler is not the only path to a diagnostic: tooling observation, such as the sanitizers that report the same core-language violations [28], records them without the contract handler, so logging is a capability the handler can provide rather than one it alone supplies. Which feature owns it is the configuration question the companion P4297R0 [44] asks EWG to decide and P4306R0 [2] compares; this paper's response-shape enumeration holds either way, and under a Profiles-first outcome the non-escape responses B, E, F, and G are a profile's enforced response rather than a Contracts-handler outcome. This paper points at the ownership question and does not answer it.

Implementation experience. One option in the escape family has a written, prototyped form and one does not. D4298R0 implements Option C's non-throwing semantics in the GCC and Clang forks [12], whereas Option A has no implementation of its escape path, and P3100R8 Section 5.5 itself notes the new codegen that path requires [1]. The sanitizer precedent P3100R8

cites for the escape path (Section 5.2) is evidence of that unmet requirement, not against it. A sanitizer callback can throw through a `noexcept` boundary today only because the guarded operation is undefined behavior, which imposes no obligation on the compiler to make that throw propagate correctly: the escaping throw is itself just more undefined behavior. P3100 removes that footing by defining the check, so the escape becomes defined behavior a conforming implementation must make correct. P3100R8 concedes the obligation directly. It notes that without the matching metadata, exceptions can go "flying through `noexcept` boundaries under certain conditions," with "destructors not being called correctly during stack unwinding," and states, "A conforming implementation of P3100 with Option A would have to prevent such effects." [1] The existing sanitizer practice is thus the behavior Option A must correct, not an implementation of it. Among the options that let an exception escape, the one with implementation experience is the one that also supplies a non-throwing choice. Among the non-escape options, E's trap response ships at massive-fleet scale through libc++ hardening and F's abort response is universal through `assert`; C's prototype is in experimental compiler forks that have not reached a release.

8.4. What the comparison shows

The requirements grid, scored by criteria that come from Option A's own paper, gives Option A genuine wins on unwinding and on adding nothing new. The deployment, security, compatibility, and implementation dimensions point toward the options that do not let an exception escape a just-detected core-language violation. No single option is best on every dimension. This paper names no winner: the requirements grid and the four external dimensions point in different directions, and the paper does not resolve which set governs.

---

9. Conclusion

Four of the eight response shapes are deployed in shipping code, two have prototypes, and the two that throw are not. The terminating, trapping, and aborting shapes of Options B, D, E, and F ship through the `noexcept` boundary, libc++ hardening, and `assert`; Options C and G have prototypes in the contracts forks and in GCC 16.1; and the throwing shape of Options A and Option 0 ships nowhere.

The question is what an implicit contract assertion does when its violation handler throws, and the answers this paper maps are Option 0 together with Options A through G. This paper presents all eight against one comparison: six requirements drawn from P3100R8, and five further dimensions from the public record - deployment lineage, security posture, compatibility direction, diagnostics, and implementation experience.

The record shows a split. The requirements grid, whose criteria come from Option A's own paper, gives Option A its wins on unwinding and on adding nothing new. The deployment, security, compatibility, and implementation dimensions point toward the options that do not let an exception escape a just-detected core-language violation. Requirements (1), (2), and (3) cannot all hold, so no option meets all six, and each option gives up something.

The finding is that the space before EWG is larger than four options. The premise used to narrow it, that the `noexcept` operator's value must not change, stands on the record as a design principle and a poll, and stands unquantified as a breaking-change magnitude. The configuration and Profiles dimensions of the same question are treated in the companion, P4306R0 [2].

Of the eight options, the two that let an exception escape a core-language expression, Option A and Option 0, are the two with no implementation of that escape.

Acknowledgements

The careful design statements in P2900R14 [3] and P3100R8 [1] made the questions in this paper precise. Lénárd Szolnoki raised on the EWG reflector the standard-library type-trait examples that Section 6.2 uses to show the one-definition-rule failure of a value-reporting `noexcept` operator. Any errors are the authors' own.

References

[1] P3100R8 - "A framework for systematically addressing undefined behaviour in the C++ Standard" (Timur Doumler, Joshua Berne, 2026).

[2] P4306R0 - "Configuring Runtime Checking: Profiles and Implicit Contract Assertions" (Vinnie Falco, Ville Voutilainen, 2026).

[3] P2900R14 - "Contracts for C++" (Joshua Berne, Timur Doumler, Andrzej Krzemieński, 2025).

[4] P0012R1 - "Make exception specifications be part of the type system" (Jens Maurer, 2015).

[5] CWG2792 - "Clean up specification of noexcept operator" (Jan Schultke, ISO C++ Core Working Group, 2023).

[6] P3541R1 - "Violation handlers vs noexcept" (Andrzej Krzemieński, 2025).

[7] cplusplus/papers #2178 - WG21 public paper tracker issue for P3541, recording the SG21 poll record of 2025.

[8] P2698R0 - "Unconditional termination is a serious problem" (Bjarne Stroustrup, 2022).

[9] P2969R0 - "Contract annotations are potentially-throwing" (Timur Doumler, Ville Voutilainen, Tom Honermann, 2023).

[10] P3318R0 - "Throwing violation handlers, from an application programming perspective" (Ville Voutilainen, 2024).

[11] GCC -fnon-call-exceptions - GCC Manual, "Options for Code Generation Conventions" (GNU Project / Free Software Foundation, retrieved 2026).

[12] D4298R0 - "Nonthrowing Evaluation Semantics" (Joshua Berne, 2026).

[13] P2877R0 - "Contract Build Modes, Semantics, and Implementation Strategies" (Joshua Berne, Tom Honermann, 2023).

[14] Itanium C++ ABI - "Itanium C++ ABI", Mangling section, the `Do` exception-specification production (community-maintained specification, retrieved 2026).

[15] GCC -Wnoexcept-type - GCC Manual, "Options Controlling C++ Dialect" (GNU Project / Free Software Foundation, retrieved 2026).

[16] P2834R1 - "Semantic Stability Across Contract-Checking Build Modes" (Joshua Berne, John Lakos, 2023).

[17] cplusplus/papers #1943 - WG21 public paper tracker issue for P3290, recording the SG22 2026-07-08 poll on assert exception propagation.

[18] basic.contract.eval - "Evaluation of contract assertions", C++ working draft (ISO/IEC JTC1/SC22/WG21, retrieved 2026).

[19] P3520R0 - "Contracts for C++: Wroclaw technical fixes" (Timur Doumler, Joshua Berne, Andrzej Krzemieński, 2024).

[20] MSVC /EH - "/EH (Exception handling model)", `/EHa` asynchronous exceptions (Microsoft, retrieved 2026).

[21] except.terminate - "Handling an exception: std::terminate", C++ working draft (ISO/IEC JTC1/SC22/WG21, retrieved 2026).

[22] GCC 16 changes - "GCC 16 Release Series: Changes, New Features, and Fixes", P2900R14 Contracts implemented as experimental C++26 (GNU Project / Free Software Foundation, 2026).

[23] Practical Security in Production - "Practical Security in Production: Hardening the C++ Standard Library at massive scale" (Louis Dionne, Alex Rebert, Max Shavrick, Konstantin Varlamov, 2025).

[24] P3577R0 - "Requiring a non-throwing system-provided (default) contract-violation handler" (John Lakos, 2025).

[25] glibc: Consistency Checking - "Consistency Checking", The GNU C Library reference manual (Free Software Foundation, retrieved 2026).

[26] libc++ Hardening Modes - "Hardening Modes", libc++ documentation (LLVM Project, retrieved 2026).

[27] cplusplus/papers #1648 - WG21 public paper tracker issue for P2900, recording the St. Louis 2024-06 EWG poll on the contract-termination ending.

[28] UndefinedBehaviorSanitizer - "UndefinedBehaviorSanitizer", Clang documentation, trap mode (LLVM Project, retrieved 2026).

[29] __fastfail - "__fastfail" (Microsoft, retrieved 2026).

[30] Chromium base/immediate_crash.h - "base/immediate_crash.h", the IMMEDIATE_CRASH trap primitive backing CHECK (The Chromium Authors, retrieved 2026).

[31] Let Me Unwind That For You - "Let Me Unwind That For You: Exceptions to Backward-Edge Protection" (Victor Duta, Fabian Freyer, Fabio Pagani, Marius Muench, Cristiano Giuffrida, 2023).

[32] NVD CVE-2026-6654 - "thin-vec: use-after-free and double free when an element Drop panics" (Mozilla thin-vec advisory GHSA-xphw-cqx3-667j / RUSTSEC-2026-0103, 2026).

[33] HeapSetInformation - "HeapSetInformation (HeapEnableTerminationOnCorruption); SDL Heap Manager Fail Fast" (Microsoft, retrieved 2026).

[34] glibc: Heap Consistency Checking - "Heap Consistency Checking", The GNU C Library reference manual (Free Software Foundation, retrieved 2026).

[35] P2784R0 - "Not halting the program after detected contract violation" (Andrzej Krzemieński, 2023).

[36] SEI CERT ERR56-CPP - "ERR56-CPP. Guarantee exception safety", SEI CERT C++ Coding Standard (Software Engineering Institute, Carnegie Mellon University, retrieved 2026).

[37] SEI CERT MSC54-CPP - "MSC54-CPP. A signal handler must be a plain old function", SEI CERT C++ Coding Standard (Software Engineering Institute, Carnegie Mellon University, retrieved 2026).

[38] SEI CERT DCL57-CPP - "DCL57-CPP. Do not let exceptions escape from destructors or deallocation functions", SEI CERT C++ Coding Standard (Software Engineering Institute, Carnegie Mellon University, retrieved 2026).

[39] N3103 - "Security impact of noexcept" (David Kohlbrenner, David Svoboda, Andrew Wesie, 2010).

[40] P1093R0 - "Is undefined behaviour preserved?" (Andrew Bennieston, Jonathan Coe, Daven Gahir, Thomas Russell, 2018).

[41] P2861R0 - "The Lakos Rule: Narrow Contracts and noexcept Are Inherently Incompatible" (John Lakos, 2023).

[42] P3101R0 - "Differentiating potentially throwing and nonthrowing violation handlers" (Ran Regev, Gašper Ažman, 2024).

[43] P3229R1 - "Making erroneous behaviour compatible with Contracts" (Timur Doumler, Joshua Berne, Gašper Ažman, 2025).

[44] P4297R0 - "Severing P3100's Profiles Claim from Its Case-by-Case Review" (Vinnie Falco, Ville Voutilainen, 2026).

[45] P3311R0 - "An opt-in approach for integration of traditional assert facilities in C++ contracts" (Tom Honermann, 2024).