Transient Benefit, Perpetual Cost: Implicit Core-Language Assertions

Document Number:	P4318R1
Date:	2026-08-11
Intent:	Inform
Audience:	EWG

Reply-to:	Vinnie Falco vinnie.falco@gmail.com Mungo Gill mungo.gill@me.com

Table of Contents

Abstract

Standardizing a continuing (log-and-continue) response for implicit contract assertions on core-language undefined behavior, as a portable guarantee every implementation carries, returns less than it costs.

P3100R8 proposes to guard the runtime-checkable cases of core-language undefined behavior with implicit contract assertions: checks the compiler inserts and evaluates under the C++26 Contracts semantics. One of those semantics, observe, continues past the violation once the handler has run.

This paper prices a single slice of that proposal: the observe response for the checks whose continuation runs on a state the language leaves undefined, standardized so that every implementation must provide it. To price it, the paper builds a cost model that adapts ordinary opportunity-cost reasoning to a language feature.

The model rejects the slice on two independent grounds. First, standardizing the slice adds almost nothing over the build options libc++ and Bloomberg's BDE already ship, which deliver the same behavior, so its marginal value is near zero; the marginal-value test then ends the analysis, because no other term can rescue it. Second, even if the benefit were positive, it is a benefit that fades: those facilities' own documentation confines it to a codebase's adoption period, while the cost falls on every conforming implementation for as long as the standard exists. Standardization is also the hardest delivery mechanism to reverse, so it locks in the most permanent cost to serve the most temporary need.

That perpetual cost is not runtime overhead, which the ignore default sets to zero. It sits in implementer maintenance, in coupling to other parts of the standard, and in the concept every programmer must now carry.

The paper prices only this continue-into-undefined slice. P3100R8's enumeration of undefined behavior and its terminating responses carry durable value and are not priced here.

In Plain Terms

This section states the argument for readers who skip the model in Sections 3 through 7.

P3100R8 asks the compiler to insert a contract check at each core-language operation that can have undefined behavior. When a check fails, the chosen semantic decides what happens next. The observe semantic calls the violation handler and then lets the program continue past the failure. For a bad pointer dereference, an out-of-bounds access, or a use after an object's lifetime ends, "continue" means the program keeps running on state the language does not define.

This paper prices one decision only: whether to standardize that continue-past-undefined response as a portable guarantee. A portable guarantee means every conforming compiler must provide it, and a program can rely on it behaving the same on GCC, Clang, and MSVC. The paper does not argue against the rest of P3100R8. It supports the list of undefined-behavior cases. It supports the responses that stop the program, enforce and quick-enforce. It supports calling the handler to log a violation. It contests only the portable guarantee to continue past undefined state.

The first argument is that vendors already ship this. libc++ provides the observe semantic as a build option. Bloomberg's BDE ships the same capability as bsls_review. Both are opt-in flags. The vendor documentation for each says to use it only during

an adoption period. A team can already continue past a detected violation today, without the standard requiring it. Standardizing the response would add one thing on top: portability across vendors. A team turns this on per codebase, through a build setting it already controls, on a compiler it has already chosen. Cross-vendor portability buys that team almost nothing.

The second argument is that the benefit fades but the cost does not. A team turns on continuation while it works through the latent violations in a codebase it is bringing under checking. Once the code is clean, the team promotes the checks to a response that stops the program. The team then stops using continuation. The benefit lasts for that adoption window and then falls away. The cost runs the other way. A semantic written into the standard stays in every compiler for as long as the standard exists.

The cost is not a runtime cost. The default semantic is ignore, so a program that does not select continuation pays nothing when it runs. The cost sits in three other places.

- Implementer maintenance: to let an exception leave a checked expression, a compiler must generate exception-handling code around every such expression, and the libc++ team states in P3191R0 that it will not do this.
- Definitional coupling: a handler that may throw changes what the noexcept operator means, so noexcept no longer says an expression cannot throw, only that it cannot throw unless a contract is violated.
- Cognitive load: every C++ programmer now has to know this mode exists, whether or not they ever use it.

The conclusion is that this one slice returns less than it costs. A vendor extension already carries the capability for the teams and the time that need it, and the vendor can retire it when the need passes. The standardized portable guarantee would carry it for every implementation and every programmer, permanently.

This slice sets a decaying benefit against a perpetual cost.

Revision History

R1: August 2026

- Updated companion paper cross-references from July mailing (R0) to August mailing (R1) throughout
- Named libc++ and Bloomberg's BDE explicitly in abstract and conclusion where R0 said "the vendor build option"
- Replaced "the model's threshold" with "the marginal-value test" in abstract and conclusion for terminological consistency
- Reformatted the three cost items in the "In Plain Terms" section as a bulleted list
- Prose tightening throughout; no substantive changes to analysis or model

R0: July 2026

- Initial version.

---

1. Introduction

C++26 Contracts (P2900R14 [1]) define a fixed set of evaluation semantics and a single, replaceable violation handler. P3100R8 [2] extends that machinery from assertions the programmer writes to assertions the language inserts at each runtime-checkable case of core-language undefined behavior. Among the semantics that machinery carries is observe: on a detected violation the handler is called, and if it returns, execution continues past the violation. Two companion papers examine the response question on its merits (P4310R1 [3]) and the full response-option space (P4308R1 [4]); a third compares the configuration-ownership models (P4306R1 [5]). None of them prices the standardization decision itself, which is the subject here.

Related work supplies the model's inputs. P2000R5 [6], the Direction Group's direction paper, states the change strategy that the model turns into its benefit term. P3608R0 [7] applied a deployment-experience standard in this exact domain. P3191R0 [8], from the libc++ team, records the implementation cost that the model's tax term draws on. The hardening facilities that libc++ [9] and Bloomberg's BDE [10] already ship document the adoption-period scope that bounds the model's benefit stream.

This paper makes three contributions:

1. It adapts a cost model - marginal benefit, reach, an interaction tax, a discount rate, and a return-on-complexity threshold - from library-standardization reasoning to a language feature, and states each term's provenance (Section 3).
2. It applies the model to one scoped slice of P3100R8 and reports a negative result on two independent grounds: the marginal value over the deployed vendor option is near zero (Section 4), and a transient, bounded benefit is set against a perpetual, irreversible cost (Section 5).
3. It locates that perpetual cost in implementer maintenance, definitional coupling, and cognitive load rather than in runtime overhead (Section 6), and shows why the usual payoff from standardization reverses for a narrow, decaying constituency (Section 7).

The analysis rests on three assumptions, each stated where it is used and gathered here for the reader who reads only the surface:

- No compiler yet implements implicit contract assertions, so the comparison reasons from deployed analogues rather than from a conforming implementation.
- The strong result covers the class of core-language checks whose continuation runs on a state the language does not define. The class whose continuation is into a defined replacement value (for example a signed overflow specified to wrap) is treated separately and left open, as in P4310R1 [3] Section 6.
- The model's numbers are rough. They rank the options against each other; they do not measure them.

The scope is one slice, and naming it precisely is the whole of the setup. The priced object is the continuing response: log the violation through the handler, then proceed past it, for the class of checks whose continuation is undefined, offered as a portable guarantee that every conforming implementation must carry. Three larger things are deliberately not priced: P3100R8 as a whole; its enumeration of undefined behavior, which is portable to any mechanism and useful to every safety effort; and its terminating responses. Section 2 fixes that object before the model is applied to it.

2. The Priced Object Is One Slice, Not the Framework

A cost model returns a wrong answer when it is pointed at the wrong object, so this section fixes the object before Section 3 supplies the model. The distinction that matters is between the parts of P3100R8 with durable, universal value and the one slice this paper prices.

P3100R8 [2] contributes work that stands on its own. Its Appendix A enumerates the cases of core-language undefined behavior and classifies each by how it can be diagnosed; that enumeration is data, portable to a Contracts routing or a Profiles routing alike, and every safety effort consumes it. Its terminating semantics - enforce, which calls the handler and then contract-terminates, and quick-enforce, which terminates without the handler - describe the response that production hardening already ships (Section 4). None of these is priced here, because none is the contested slice.

The contested slice is narrower. C++26 defines observe as the semantic under which the handler is called and, on a normal return, execution continues past the point of evaluation (P2900R14 [1]). P3100R8 makes observe available for implicit assertions on core-language operations. For the class of those operations whose continuation is into a state the language does not define - a dereference of an invalid pointer, an out-of-bounds access, a use after the object's lifetime has ended - continuing means executing on undefined state. The object this paper prices is the decision to standardize that continuing response as a portable guarantee: a semantic every conforming implementation must provide, so that a program written against it behaves the same across vendors.

Two adjacent things are deliberately excluded. The class of checks with a defined replacement value is excluded, because there continuation is into a specified result and the objection does not apply (P4310R1 [3] Section 6). The handler invocation itself is excluded, because it is preserved by every response this paper would credit, terminating responses included: enforce calls the handler before it terminates, so logging and telemetry survive a terminating response unchanged. What is priced is continuation past undefined state, offered portably, and nothing else.

The scope discipline is the finding of this section: the model that follows is applied only to the continue-into-undefined slice offered as a portable guarantee, and the enumeration, the terminating semantics, and the handler-as-telemetry-hook are outside it.

---

3. A Cost Model for a Language Feature

The model adapts the reasoning a committee already applies to a library addition - does the benefit, summed over its reach and net of its ongoing cost, beat the best alternative use of the same finite capacity - to a language feature. It names where each term comes from, so the reader can weigh the inputs independently. Section 3.1 fixes the terms, Section 3.2 lists the variables, and Section 3.3 states the formulas and each term's provenance. The model is the authors' own construction; its purpose is to force the standardization decision to state estimates that can be argued about rather than left implicit.

3.1 Terms

The recurring terms carry one meaning throughout.

Term	Meaning
implicit contract assertion	A check the compiler inserts at a core-language operation that can have undefined behavior, evaluated like a C++26 contract assertion on the operation's precondition.
evaluation semantic	The mode that decides what happens when a check's predicate is false: ignore, observe, enforce, quick-enforce, or assume.
observe	Call the violation handler; on a normal return, continue past the violation.
the continuing response	The observe outcome for the priced slice: proceed past a state the language does not define.
vendor opt-in	A non-portable build option, macro, or flag by which one implementation offers a behavior without the standard requiring it.
interaction tax	The ongoing cost a standardized feature imposes on the rest of the standard and its implementations for as long as it exists.
Return on Complexity	Net present value delivered per unit of specification and maintenance capacity spent.

3.2 Variables

The variables below make the model's terms measurable. Each is an order-of-magnitude instrument, estimated from the sources named in Section 3.3.

Symbol	Meaning
B	Marginal benefit per beneficiary per year of the standardized guarantee, measured against what a vendor opt-in already delivers
$N(t)$	Beneficiaries at time t
W	Length in years of the adoption window over which $N(t)$ is non-negligible
δ	Fraction of value forfeited to ossification once the design is frozen, $0 \le \delta \le 1$
k	Specification and interaction complexity the slice spends
τ	Annual cost per unit of complexity imposed on everything standardized around it
r	Time discount rate
λ	The return, per unit of specification and maintenance capacity, of the best alternative use of that capacity; $\lambda \ge 0$, since capacity can always be left unspent

3.3 Formulas and their provenance

The first condition is the marginal-value test. A language feature earns standardization only when it delivers something a program cannot already obtain without it:

$$B = B_{\text{in-standard}} - B_{\text{vendor-opt-in}} > 0$$

If everything the slice offers is already delivered by a vendor opt-in, then $B \approx 0$ and no other term matters. Provenance: this is the language-feature form of the direction stated in P2000R5 [6] Section 5, "We change the language and standard library by gradually building on previous work or by providing a better alternative to an existing feature." P2000R5 is an advisory Direction Group paper without a poll behind it, and one of its authors is a party to the safety-design discussion; a reader who rejects the criterion on that ground can read Section 4 as resting on the vendor-deployment record directly.

The annual net flow of an admitted feature is its benefit over its reach, net of ossification, minus the tax it levies every year:

$$d(t) = (1 - \delta)\, B\, N(t) \; - \; \tau k$$

The value of the feature is the present value of that flow:

$$D = \sum_{t=1}^{\infty} \frac{d(t)}{(1 + r)^t}$$

and the admission rule is comparative, because specification and maintenance capacity are finite:

$$\text{ROC} = \frac{D}{k}, \qquad \text{admit iff ROC} \ge \lambda$$

Positive value is not the bar; λ, the return of the best alternative use of the same capacity, is the bar. Because capacity can always be left unspent, $\lambda \geq 0$, so a feature with $D < 0$ fails the rule outright. Provenance: the deployment-experience input to B and N is the standard applied in P3608R0 [7], "the standard library hardening is existing practice, and comes with very positive field experience reports," a paper co-authored by an author on the Profiles side of the wider discussion; the tax input τ is measured against P3191R0 [8] and P2900R14 [1] Section 3.6.6 (Section 6).

The model is ordinal. Its quantities are estimated to an order of magnitude, and the conclusions rest on the direction and rough size of the gaps, not on precise values. Benefit and cost are compared on a common present-value scale, as in ordinary net-present-value reasoning; because the model is ordinal, only the sign and order-of-magnitude ratio of the two present values enter, not a currency figure, so the comparison requires benefit and cost to be rankable, not fungible. For the priced slice k is a fixed scalar; cost scales with it by the definition of τ, while the slice's benefit is estimated directly rather than as a function of k. Modeling δ as a constant fraction is likewise a simplification: ossification loss plausibly grows with the horizon, which would only deepen the discount on far-future benefit, so the constant form is again conservative. The finding of this section is the model itself: a standardization decision for the slice is admitted only if its Return on Complexity clears the return of the best competing use of the same capacity, and the sections that follow estimate the terms.

---

4. The Marginal Value Over a Vendor Option Is Near Zero

The first term to estimate is B, and it is the one that can end the analysis by itself: if the standardized guarantee delivers nothing beyond a vendor opt-in that already ships, then $B \approx 0$ and the admission rule fails regardless of the other terms.

The continuing response already ships as a vendor opt-in. libc++'s hardening documentation describes its observe semantic in exactly these terms [9]:

> *Continuing execution after a hardening check fails results in undefined behavior; the observe semantic is meant to make adopting hardening easier but should not be used outside of the adoption period.*

Bloomberg's BDE library ships the same capability as a separate facility, bsls_review, whose own documentation frames review mode as "an interim step towards lowering the assertion level threshold for an existing application" [10]. Both are opt-in, both are non-portable, and both are documented as bounded to an adoption period. The capability a program obtains by continuing past a detected violation is therefore available today without the standard requiring it.

What standardization would add on top of the vendor opt-in is portability: a guarantee that the continuing response behaves identically across GCC, Clang, and MSVC. For most language features portability is a real benefit, and Section 7 states when it is. For a transitional adoption aid it is close to worthless. A team continues past violations while it drains a backlog of latent findings from a codebase it is bringing under checking. That activity is per-codebase and per-build, expressed by a build setting the team already controls. It does not require the setting to mean the same thing on a compiler the team is not using. The value of cross-vendor portability for a facility whose documented purpose is a temporary, local rollout is near zero.

Setting $B \approx 0$ makes the rest of the arithmetic moot: with the benefit near zero the present value D falls to about $-\tau k/r$, a negative number that cannot clear the threshold λ, whatever the reach. This is the marginal-value test returning the default answer. The finding of this section is that the standardized guarantee delivers, over the deployed vendor opt-in, only a portability whose value for a transitional facility is negligible, so $B \approx 0$ and the admission rule fails on the first term.

---

5. A Transient Benefit Against a Perpetual, Irreversible Cost

Section 4 is sufficient on its own. This section sets that result aside, grants a positive benefit, and shows the model still fails on independent grounds. The argument is the shape of the two streams: even with $B > 0$, the benefit is a decaying finite stream while the cost is a perpetuity, and the delivery mechanism compounds the mismatch.

The benefit stream decays. The beneficiaries $N(t)$ are codebases in an active adoption window - on a toolchain that implements the feature, carrying latent core-language undefined behavior they have not yet fixed, and continuing past it while they fix it. The deployed facilities describe this population's use of continuation as bounded: libc++ states observe "should not be used outside of the adoption period" [9], and BDE frames review mode as "an interim step" [10]. The lifecycle those documents describe is to add a check, continue past its findings while they are triaged, and then promote to a terminating response once the code is clean. A codebase that completes that cycle leaves $N(t)$. The benefit is a spike during adoption that decays toward zero, and its present value is a finite sum:

$$\mathrm{PV}(\text{benefit}) = \sum_{t=1}^{\infty} \frac{(1-\delta)\, B\, N(t)}{(1+r)^t}, \qquad N(t) \to 0$$

The cost stream does not decay. A semantic written into the standard is carried by every conforming implementation for as long as the standard exists. Its present value is a perpetuity:

$$\mathrm{PV}(\text{cost}) = \sum_{t=1}^{\infty} \frac{\tau k}{(1+r)^t} = \frac{\tau k}{r}$$

The closed form holds for $r > 0$, and the level shape is the right one: the costs located in Section 6 are recurring - per-release implementer maintenance, per-later-paper definitional coupling, per-programmer cognitive load - so a one-time specification cost would add to the perpetuity, not replace it. Holding τk constant is moreover conservative: Section 6's coupling argument implies the tax grows as more of the standard is written around the semantic, which for a growth rate $g < r$ would replace $\tau k/r$ with the larger $\tau k/(r-g)$. The constant-cost perpetuity is a floor on the cost, not a ceiling.

A decaying finite stream and a perpetuity are both finite present values, so the comparison is one of magnitudes; the question is what magnitude the benefit would need. Let P be the peak annual benefit, so $(1-\delta)\, B\, N(t) \le P$, with $N(t) = 0$ outside the adoption window of length W the deployed facilities document. Then the benefit's present value is bounded:

$$\mathrm{PV}(\text{benefit}) \le P \sum_{t=1}^{W} \frac{1}{(1+r)^t} = \frac{P}{r}\left(1 - (1+r)^{-W}\right),$$

so matching the perpetuity $\mathrm{PV}(\text{cost}) = \tau k/r$ requires

$$P \geq \frac{\tau k}{1-(1+r)^{-W}}.$$

For a three-year window at $r = 0.05$ that factor is about seven: the slice's peak annual benefit would have to exceed the entire annual cross-implementation tax sevenfold to break even, and the factor only grows as r falls toward the rate appropriate to a standard's indefinite horizon. A transitional adoption aid, selected through a build setting the team already controls, does not deliver several times the whole standard-wide tax in a single year. Discounting does not rescue the benefit, because discounting shrinks the far-future cost too; the perpetuity is already the discounted value of the endless stream. Section 6 establishes that τk is real and non-trivial, which is all the comparison now needs.

The delivery mechanism compounds the mismatch, because the standard is the least reversible way to ship a response. A vendor opt-in can be retired when the adoption-period rationale for it lapses; a build flag can be deprecated and removed. A standardized evaluation semantic, encoded in the meaning of core-language operations, is effectively permanent: the installed base and the stability expectations of the standard hold it in place. A reversible mechanism caps the cost sum at its retirement date T, paying only $\sum_{t=1}^{T} \tau k/(1+r)^t = \frac{\tau k}{r}\left(1-(1+r)^{-T}\right) < \frac{\tau k}{r}$; the standardized guarantee forgoes that cap and pays the full perpetuity. Its irreversibility is exactly that gap. Using the same τk for the vendor mechanism understates the gap: by Section 6 a vendor opt-in avoids the standard-wide coupling and cognitive-load components of the tax. Matching the most permanent, least reversible delivery mechanism to the most transient, adoption-bounded need is the inefficiency stated in one line.

The finding of this section is that the slice fails the model a second time, independently of Section 4. A benefit stream that the deployed facilities themselves scope to an adoption period is set against a perpetual cost carried by every implementation, through the least reversible mechanism available, so $\mathrm{ROC} < \lambda$ even when B is granted to be positive.

---

6. Where the Perpetual Cost Sits

The perpetuity in Section 5 is only decisive if τk is real and non-trivial, and the natural objection is that it is neither, because P3100R8's default is ignore and imposes no runtime overhead. This section locates the cost, and it is not in runtime.

The runtime cost on a program that does not select the semantic is indeed near zero, by construction. P3100R8 [2] states that a conforming implementation may give every case the ignore semantic, so existing programs are unaffected and pay nothing at run time. Placing τ in runtime overhead would locate it where it does not exist, and the objection would be correct. The cost sits in three other places.

The first is implementer maintenance. For an exception to leave a checked core-language expression correctly, an implementation must treat every such expression as a potential throw site and generate the matching exception-handling metadata; P2900R14 [1] Section 3.6.6 records that the compiler "has to generate the correct instructions for exception handling around every contract assertion." The reference implementers decline this. P3191R0 [8], from the libc++ team, sets the production requirement that a contract violation "should generate no code at all beyond the equivalent of a branch and a `__builtin_trap()`," with "no exception-handling code being generated around contract predicates." The committee's own response to this cost was to add the quick-enforce semantic, which skips the handler entirely (P3198R0 [11]). The machinery

the continuing response requires is machinery the implementers who ship hardening have stated they will not carry, and standardizing the portable guarantee obligates every implementation to carry it regardless.

The second is definitional coupling. A continuing handler that may throw shifts what the noexcept operator asserts. P3100R8 [2] Section 5.5 concludes that "the addition of implicit contract assertions must not affect the result of the noexcept operator," and its Option A preserves the operator's value while changing its meaning, so that a `true` result no longer states that evaluating the expression cannot throw but that it cannot throw "unless there is a contract violation." Since C++17 a function's exception specification has been part of its type (P0012R1 [12]), and the noexcept operator is the standard way to compute one, as in `void f() noexcept(noexcept(g()));`. What the operator reports therefore feeds the type system, and C++26 already spent wording to keep that boundary source-compatible (P3229R1 [13]). A change to what a core-language operation may do, and to what noexcept means over it, is coupling that every later paper writes against.

The third is cognitive load. A semantic in the standard is a concept every C++ programmer carries, whether or not they use it. The continuing response for implicit core-language assertions adds a mode that a shrinking, adoption-bounded population ever selects, to the conceptual surface of a language used by everyone.

The finding of this section is that τk is real and sits in implementer maintenance, definitional coupling, and cognitive load. It does not sit in runtime, where the ignore default sets it to zero. Its maintenance component is a cost the reference implementers have stated on the record they decline to carry.

---

7. Why the Usual Payoff From Standardization Reverses Here

Standardization usually pays off precisely by conscripting every implementation, so this section states why that mechanism runs the other way for this slice. The answer is in the reach term $N(t)$ and its scaling.

For a durable feature the whole installed base uses, conscription is the benefit. When the standard requires a facility, every vendor ships and maintains it, and a program can rely on it everywhere; that is the reasoning by which platform-coupled features that no single author could portably provide entered the library. The benefit scales with reach: a facility used across the whole installed base, indefinitely, recovers its perpetual cost because the benefit is also perpetual and universal.

For the priced slice the same mechanism runs with the opposite sign. Conscription still requires every implementation to carry the semantic, but the reach that would justify it is absent: the constituency is narrow (codebases with unfixed latent core-language undefined behavior, on an implementing toolchain) and the benefit is bounded (the adoption period the deployed facilities document). A perpetual, universal cost is incurred to serve a reach that is neither perpetual nor universal. The property that makes conscription pay for a durable feature used across the installed base is exactly the property this slice lacks.

The scaling contrast also explains why the parts excluded in Section 2 are not caught by this argument. The enumeration of undefined behavior and a terminating response have broad, durable reach - every hardened program benefits, indefinitely - so their benefit stream is the kind that recovers a standardization cost. The continuing response for the undefined class does not share that shape, which is why the model separates it out.

The finding of this section is that standardization's usual payoff, conscripting every implementation on behalf of a universal and durable need, reverses for a slice whose need is narrow and adoption-bounded: the same conscription becomes a perpetual universal cost against a transient minority benefit.

---

8. Possible Concerns

This section answers five possible objections to the analysis.

"The default is ignore, so the feature costs nothing"

The runtime cost on a non-selecting program is near zero, and Section 6 grants it. The cost the model prices is not runtime. It is the implementer maintenance of the exception-handling machinery the continuing response requires, which P3191R0 [8] records the reference implementers declining; the definitional coupling of the noexcept meaning-shift that P3100R8 [2] Section 5.5 documents; and the cognitive load of a standard-wide concept. A zero runtime cost on non-users does not zero the perpetuity in those three places.

"The need recurs, so the benefit is not transient"

A large organization adds checks to legacy code continuously, so the adoption-period need recurs rather than ending once. The recurrence is real and does not change the result, for two reasons already on the record. The recurring need is served equally by the non-portable vendor opt-in of Section 4, which is what libc++ and BDE ship for exactly this purpose [9] [10], so it does not require the portable standard guarantee. And the deployed facilities apply continuation at the library level, where the post-violation state is still language-defined, rather than to the core-language-undefined class this paper prices; BDE's own boundary terminates on the harder class through bsls_assert while bsls_review continues on the defined class [10]. Recurrence keeps the vendor opt-in useful; it does not make the portable core-language guarantee earn its perpetual cost.

"The cost model is the author's own, and its numbers are unmeasured"

The model is the author's construction, disclosed as such in Section 11, and its quantities are order-of-magnitude instruments. Its value is that it makes the estimates behind the standardization decision explicit and arguable, and its conclusions rest on the direction and rough size of the gaps rather than on precise values. The two gaps it turns on are large and their direction is not in dispute: a benefit the deployed facilities themselves scope to an adoption period, against a cost every conforming implementation carries without end. A reader who assigns different values to B, τ, or λ can re-run the comparison; the ordering survives any assignment that keeps the benefit bounded and the cost perpetual.

"Portability is the whole point of a standard"

Portability is a genuine benefit for a durable feature used across the installed base, and Section 7 states the case for it. It is near-worthless for a facility whose documented purpose is a temporary, per-codebase rollout aid, because such a facility is selected through a build setting the team already controls and does not need to mean the same thing on a toolchain the team is not using. The portability the standardized guarantee would add over the vendor opt-in is portability of a transitional mode, which is the near-zero B of Section 4.

"This prices P3100, which does far more than one semantic"

It does not. Section 2 fixes the priced object as one slice: the continuing response for the class whose continuation is undefined, offered as a portable guarantee. The enumeration of undefined behavior, the terminating semantics, and the handler as a telemetry hook are excluded and are credited as carrying durable value. The model is applied to the slice and to nothing else.

---

9. Questions Worth Considering

The analysis above raises questions readers might wish to consider. They request no poll and no committee action.

- Why does the continuing response need to be a portable standard guarantee rather than a vendor extension? libc++ [9] and Bloomberg's BDE [10] already ship it as an opt-in, and both document it as bounded to an adoption period.
- What does cross-vendor portability add for a mode selected per-codebase through a build setting the team already controls, on a toolchain the team has chosen to use?
- The deployed facilities scope continuation to an adoption window. What use case requires it to persist as a permanent guarantee that every conforming implementation carries?
- Deployed practice terminates on the core-language-undefined class and continues only where the post-violation state is language-defined: BDE's bsls_review continues on the defined class while bsls_assert terminates on the harder one [10]. What justifies standardizing continuation on the class that deployed practice terminates on?
- The reference implementers state they will not generate exception-handling code around contract predicates (P3191R0 [8]). Who will carry the machinery the continuing response requires once it is a portable obligation on every implementation?

---

10. Conclusion

The runtime checking of core-language undefined behavior is worth standardizing, and P3100R8's enumeration and terminating responses are the parts of that work with the reach to earn it. The continuing response for the class whose continuation is into undefined state, offered as a portable guarantee, is a different object, and priced on its own it does not earn standardization.

The model reaches that result twice over. First, standardizing the guarantee adds almost nothing over the build options libc++ and Bloomberg's BDE already ship, so its marginal value is near zero; the marginal-value test then ends the analysis, because no other term matters. Second, even granting a positive benefit, that benefit fades: the deployed facilities' own documentation scopes it to an adoption period, while the cost falls on every conforming implementation, through the delivery mechanism that is hardest to reverse. That cost is real. It sits in the implementer maintenance the reference implementers have declined to carry, in the coupling created when the noexcept operator changes meaning, and in the cognitive load of a concept every C++ programmer must now hold. It does not sit in runtime, where the ignore default sets it to zero. Standardization normally pays off by requiring every implementation to carry a feature; for a narrow, adoption-bounded slice that same requirement becomes pure cost.

The deployment record shows an asymmetry. A transitional capability ships today as a vendor opt-in, and the documentation for those options bounds it to a rollout period. Standardizing it as a portable guarantee sets that temporary need against a permanent obligation on every implementation and every reader of the language. A vendor extension carries the capability for the codebases and the interval that need it, and lapses when they do not. The standardized portable guarantee carries it for everyone, forever. Whoever takes up the response question next builds on the deployment record and the model placed here.

---

11. Disclosure

The author provides information and serves at the pleasure of the committee.

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.

The intent of this paper is `info`. It argues a position - that a portable, standardized continuing response for one class of implicit core-language assertions returns less than it costs - and it proposes no wording and requests no poll.

Among the response options for a detected core-language violation, the author prefers the family in which execution does not continue past a state the language leaves undefined. That is a stake in the outcome, and the reader should weigh what follows accordingly.

One limitation is disclosed up front: the cost model applied here is the author's own construction, and its quantities are order-of-magnitude instruments meant to rank options, not precise measurements. No compiler yet implements implicit contract assertions with any semantic, so the model reasons from deployed analogues rather than from a conforming implementation.

This paper is one of a set in the August 2026 mailing on the runtime checking of core-language undefined behavior. Its companions are P4297R1 (the ownership question and its polls), P4306R1 (the configuration-ownership comparison), P4310R1 (the response question on the merits), P4308R1 (the response-option space), and P4317R1 (a profile carrying the same enumeration). This paper prices one slice of the same subject and cross-references those rather than repeating them. It works only from the published record and primary vendor documentation; committee-internal materials may contain answers the record does not.

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

This paper asks for nothing.

Acknowledgments

Timur Doumler and Joshua Berne performed the enumeration and classification of core-language undefined behavior in P3100R8 that this paper's scope section relies on to separate the priced slice from the durable parts of that work.

Louis Dionne, Yeoul Na, and Konstantin Varlamov stated the implementation cost in P3191R0 that Section 6 uses to locate the interaction tax.

References

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

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

[3] P4310R1 - "Hasta la Vista, Undefined Behavior: Why std::core_ub Should Terminate by Default" (Vinnie Falco, Ville Voutilainen, 2026).

[4] P4308R1 - "Eight Responses to a Throwing Implicit Contract Assertion" (Vinnie Falco, Ville Voutilainen, 2026).

[5] P4306R1 - "Configuring Runtime Checking: Profiles and Implicit Contract Assertions" (Vinnie Falco, Ville Voutilainen, 2026).

[6] P2000R5 - "Direction for ISO C++" (Jeff Garland, Paul E. McKenney, Roger Orr, Bjarne Stroustrup, David Vandevoorde, Michael Wong, 2026).

[7] P3608R0 - "Contracts and profiles: what can we reasonably ship in C++26" (Ville Voutilainen, Jonathan Wakely, Gabriel Dos Reis, 2025).

[8] P3191R0 - "Feedback on the scalability of contract violation handlers in P2900" (Louis Dionne, Yeoul Na, Konstantin Varlamov, 2024).

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

[10] bsls_review and bsls_assert - component documentation (Bloomberg BDE, retrieved 2026).

[11] P3198R0 - "A takeaway from the Tokyo LEWG meeting on Contracts MVP" (Andrzej Krzemieński, 2024).

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

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