Repetition, Elision, and `const`-ification With Regard to `contract_assert`: A Principled Analysis

Document #:	P3270R0
Date:	2024-05-22
Project:	Programming Language C++
Audience:	SG21 (Contracts)

Reply-to:	Joshua Berne <jberne4@bloomberg.net>
	John Lakos <jlakos@bloomberg.net>

Abstract

The MVP for the burgeoning C++26 Contracts facility designed in SG21 is a cohesive well-formed solution to the problem as posed. After sharing with a wider audience, concerns were raised, especially with respect to two novel features related to predicates: (1) default *`const`-ification* of variables in predicates and (2) *repetition* and *elision* of runtime predicate evaluation. Using principled design [P3004R1], we began to look into solutions that *removed* those two features from [P2900R7]. Further discussion revealed that the more pervasive concern was the lack of a modern C++ replacement for the standard C `assert` macro. We used the information obtained from our principled analysis to synthesize a new, composite solution ([P3290R0]) that *adds* functionality — i.e., support for a modern replacement for the standard C `assert` macro — and satisfies the needs of the multiverse, in particular those advocating for bespoke, legacy assertion facilities. Our analysis concludes that [P2900R7] with the addition of [P3290R0] is the optimal solution, and we hope this solution will allow the Contracts MVP to gain consensus.

Contents

Revision History

Revision 0, May 2024 Mailing

- Original version of the paper for discussion during an SG21 telecon

1 Introduction

NB: This paper is under development and will include a thorough analysis of how principles that stem from the goals described below relate to potential changes we might or might not make to the MVP, specifically with regard to elision, repetition, and `const`-ification.

What the underlying goals and requirements for proposals for standardization should be is often a topic that arises during discussions of such proposals. In the context of the Contracts facility, this consideration is doubly importantly due to the many potential use cases for Contracts (see [P1995R1]).

We see ten clear goals to consider, each with various points to keep in mind.

1. Improve public opinion.

    - We must demonstrate our commitment to enhancing the safety of modern C++ and C++26 (following C++23). Adding the Contracts facility to C++26 represents our last, best chance to achieve this goal.

    - The Contracts facility is by far the most effective way to reduce undefined behavior (UB) and improve safety by reducing security vulnerabilities and improving correctness *incrementally* in both old and new code — something that isn't true of any other safety-related effort.

    - An IS based on [P2900R7] will provide C++26 developers with a near-term compelling advantage with regard to security and correctness over C++23.

2. Assess teachability of a novel paradigm.

    - We need to learn whether this new use is easy to teach to novice and experienced programmers.

        - Just type `pre(exp)` and try to avoid (and definitely do not depend on) any side effects.

    - Don't teach people that the new C++ Contracts facility is at all intended to be a plug-in replacement for old C-style `assert` macros (unless they happen to be used as a proper approximation to modern C++ contract preconditions, postconditions, or local contract assertion checks, i.e., no destructive side effects).

    - Learning about the other features — `const`-ification, elision, and repetition — reinforces to programmers that they shouldn't have side effects that matter. Importantly, adherence to this limited use of side effects means that their programs will still be correct when all the other methods of fine-grained enabling and disabling individual contract assertions within a TU are finally deployed (post-MVP).

3. Motivate C `assert` users to change.

    - Because P2900 will allow for selective activation of contract assertions in various builds, modern C++ contracts are entirely *unlike* C-style assertion macros, which have a single flag that checks or unchecks all contract assertions, whereas future versions of the Contracts facility will have the needed syntax to control the power set of contract assertions active in any given build.
    - Note that [P2900R7] already allows such flexibility, but we don't yet have the needed labels to make it practicably usable.
    - Note that this ability is fundamentally incompatible with destructive side effects in contract assertion predicates.
        - Definition: A destructive side effect in a contract assertion predicate is one that either potentially (1) interferes with the essential behavior of a program or (2) affects the result of any other contract assertion (or a subsequent evaluation of the same assertion).
    - For `pre` and `post`, we expect that there will be valid build configurations where checks might occur twice, fifty times, or not at all. New reusable code with predicates written to that looser specification will work.

4. Implement reusable libraries.

    - Nothing in [P2900R7] prohibits a build mode that limits the number of predicate evaluations for a given runtime flow-of-control encounter with a contract assertion to be exactly 1, but code that depends on that particular build mode is not guaranteed to work in other builds where the exactly-1 evaluation doesn't hold, much the same way that, in general, code built to require that `NDEBUG` not be set won't work.
    - Reusable library code would need to follow the no-destructive-side-effect rules, which would likely include no print statements and might even go so far as to avoid all stateful changes that are not represented in the VM, such as paired memory allocation and deallocation within the predicate.
    - In other words, software, to be widely reusable, would need to follow this relaxed rule allowing elision and duplication.
        - **Weak Rule:** Given any defect-free program, if we remove any one contract assertion from that program, the program must remain correct.
        - **Strong Rule:** Given any defect-free program, if we fail to check any one contract assertion exactly once at any point in execution of that program, the program must remain correct.

    Failing either of these rules, large applications comprising multiple libraries would be unable to include any such libraries unless they somehow remembered to build those sub-libraries in a platform-defined special build mode, which, although fully Standard compliant, is error prone, requires expert individual knowledge of each library, and simply does not scale to larger development efforts.

5. Create build-independent source code.

    - Standard-Library implementations would necessarily be *unable* to make assumptions that the predicate will be evaluated exactly once (at run time) each time the flow of control reaches it.
    - We could require a mode where contract assertions are either all checked (and each predicate is evaluated exactly once) or all unchecked (and no contract assertion predicates are evaluated).
    - Having such a single build switch would satisfy those who want to employ destructive side effects in C++ contract asserts, yet that ability invites the question of whether such specialized guaranteed semantics (though valid) serve the greater good.
    - Though not incorrect, software that relies on being built with an all-or-nothing switch means that the software would not generally be portable.
    - Relying on exactly one predicate evaluation per contract assertion encounter would be similarly incompatible with wide reuse. Moreover, relying on specialized build modes would play poorly with effective use of planned future syntactic features (e.g., labels) that would give fine-grained control over individual contract assertions.
    - As an analogy, Standard-Library implementations should avoid dependencies on any one specialized build mode just as they should avoid dependencies on libraries that are not exception safe, are not `const` correct, are not minimally thread safe,[1] or otherwise fall short of normal and customary levels of QoI.

6. Maximize implementation options.

    - Forcing the predicate of every runtime-encountered contract assertion to be evaluated exactly once (and especially never twice) eliminates, for example, the implementation strategy of allowing a client TU and a supplier TU to be independently compiled with caller- or callee-side checking. Hence, if both are compiled with different choices for which side of the function invocation is responsible for checking contract assertions, some predicates will inevitably be evaluated twice. Such double evaluation will break any software that relies on the guaranteed single evaluation of every runtime-encountered contract assertion.
    - This problem, which seems bad enough, is much worse in practice. Imagine we are building an application that makes use of two precompiled libraries, A and B. Library A is built with caller-side checks, and Library B is built with callee-side checks. Consider also that both libraries might depend on a third library, Library C — which may even be the Standard Library. As the owner of the application, we might want to turn contract checks on for Library C, but that inevitably means it will have a different option with respect to caller- or callee-side checking than either Library B or Library A. Which option do we pick if either one ends up potentially causing double evaluations of contract checks that are not permitted? What happens when, instead of two libraries, we have 5, 200, or 10,000 libraries, each of which is built with a different configuration of build flags. How

[1]A type is minimally thread safe if two or more independent objects of that type can be safely accessed concurrently, each from its own, distinct thread.

are we supposed to build our application in a way that guarantees that each call into every library API is checked at least once?

  - What's more important?
    - Making client code that is checked at least once?
    - Making sure that we guarantee that destructive side effects work in the MVP?
  - Might destructive side effects wait until we have the needed additional syntax to support both in a future release?

7. Avoid rigid final decisions.
   - SG21 has consensus on at least two controversial decisions, but some colleagues have voiced concerns.
     (a) SG21 supports `const`-ification.
     (b) SG21 supports elision and repetition.
   - Each of these decisions is chosen for the same two reasons.
     (a) SG21 believes that this choice is correct for the long term.
     (b) SG21's choices preserve the option to revert to the other option in a guaranteed backward-compatible way.

   Importantly, primarily by dint of this ability to revert, with full backward compatibility, to the other choice can we make these specific choices for the default in the MVP; otherwise, we would remove the option to change our minds in the future in the unlikely event that the other option proves — through actual experience — to be the better default after all.
   - This mitigation principle bears repeating.
     - **Rule:** Without strong consensus, choose a solution that allows for backward-compatible migration to all other viable solutions.
   - The alternatives (not supporting `const`-ification and not supporting elision and repetition) would make the MVP's default effectively permanent in violation of the aforementioned principle.

8. Address use cases.
   - We cannot address all use cases.
   - To achieve deterministic single evaluation in the C++ MVP, we would necessarily have to eliminate the ability for
     (a) finer-grained control over contract assertions (e.g., keeping all `pre` contract assertions checked and leaving all `post` contract assertions unchecked) in both the MVP and, by default, all future unlabeled contract assertions

    (b) implementations to allow for asynchronously changing semantics at run time (and again for all default contract assertions moving forward)

    (c) a client to ensure that every precompiled library API it uses is checked at least once (without causing an ABI break or forcing a global recompile) for the MVP and by default (without labels) post-MVP

    (d) the default behavior (no labels on the contract assertion) to ever change back to that of [P2900R7]

   - The MVP — by definition — cannot address all the solutions identified in SG21's initial list of desired solutions [P1995R1], and not all solutions are needed for the initial release of the C++ Contracts feature to be viable.
   - In particular, we can significantly improve safety by eliminating UB in narrow contracts and thereby substantially reduce security vulnerabilities while also detecting other logical programmatic defects at run time.
   - Moreover, just having contract assertions — with or without potential side effects — available in the interface is essential to performing static analysis across TUs.
   - By keeping the MVP as is, we enable extensibility of the MVP to solve essentially all the remaining problems post-MVP, i.e., once we have the needed infrastructure and additional syntax in place.

9. Add essential syntax in C++29.
   - The MVP cannot simultaneously satisfy the needs of determinism *and* extensibility because there is no way to identify a contract assertion that happens to have destructive side effects (or, conversely, one that doesn't).
   - For the past several years, Bloomberg, under the direction of Joshua Berne, has been working with outside experts, such as Andrew Sutton and Lisa Lippincott, to develop a complete roadmap (P2755R1) detailing what would be required to realize essentially all the use cases delineated in [P1995R1], except those that are inherently inconsistent with higher-priority use cases.
   - To satisfy simultaneously both flexibility and determinism, we will need to provide additional syntax (and infrastructure) in the form of labels.
   - Once we have labels, we will be able to effectively annotate individual contract assertions to allow them to be paired and otherwise be treated specially from all other contract assertions.
   - This ability to provide a potentially unique identity will
     - allow individual contract assertions to be controlled externally
     - prevent specific contract assertions from being controlled externally
     - indicate that specific subsets of contract assertions be treated as a unit

    - indicate that specific subsets of checked contract assertions not be afforded flexibility to evaluate their respective predicates other than exactly once each time one of those contract assertions is encountered at run time

  Hence these multiple usage paradigms will be able to work together synergistically at scale.

- Alternatively, we plan to provide yet another way in which side effects can be isolated outside of predicates: a new construct, a `contract_support` block, will be a safer and much more modular way to contain side effects in a post-MVP world.

10. Solve all use cases via post-MVP Contracts.

    - We believe that the eventual C++ Contracts facility must support the multiverse.
    - We fully acknowledge those people who do and will continue to want to use this new facility as a drop-in replacement for C-style `assert` because (a) they are used to that, (b) they can reason about it today, and (3) a lot of code out there already uses the standard C++ `assert` macro, and they would like a painless way to replace it without having to look at each and every predicate. *We get that.*
    - We are not saying that having coupled side effects in a predicate is wrong (even though we are not happy that destructive side effects will be propagated rather than fixed once there is a better way to do it).
    - What we *are* saying is that there can be only one default kind of contract assertion (one without labels) in the MVP, and we have to choose something for that default.
    - If we choose [P2900R7], then we (a) will have closed no doors to changing, post-MVP, the default should empirical data demonstrate that what worked well in theory didn't in practice and (b) we can evolve the features synergistically.
    - If we chose anything else, we give up substantial important use cases for large-scale software developers.
        - In particular, we risk major players, e.g., the financial services industry, not adopting the feature.

In conclusion, we want to maximize our ability to achieve the above goals in the Contracts MVP and move forward with targeting it for inclusion in the C++ Standard. We believe our best plan of action at this point is to

1. stabilize [P2900R7] and hold it constant
2. implement [P2900R7] on GCC, Clang, and perhaps on one other platform (e.g., EDG) before the end of 2024 at the very latest (which Bloomberg intends to fund itself)
3. apply the result to large codebases, such as Bloomberg's, to see if repetition, elision, and `const`-ification prove problematic in an environment in which the users are educated *not* to have side effects in their assertion predicates

We simply do not see any other viable approach to getting an uncompromised, fully extensible Contracts MVP into the IS. More importantly, we believe our three-step solution can still be achieved soon enough to be incorporated into the C++26 IS (continuing with the plan laid out by [P2695R1]) and to expediently deliver real improvements to C++ users.

2 SG21 Polls on Elision, Repetition, and `const`-ification

On Thursday, May 16, 2024, SG21 discussed in its telecon several matters related to changing contract assertions in various ways.

Poll 1. For the Contracts MVP, remove the rule that in the predicate of an assertion statement (`contract_assert`), variables with automatic storage duration are implicitly `const`, as proposed in [P3257R0] Proposal 1.

SF	F	N	A	SA
8	1	2	5	4

Result: No consensus

Poll 2. For the Contracts MVP, remove the rule that in the predicate of a contract assertion of any kind (`pre`, `post`, and `contract_assert`), variables with automatic storage duration and the result object of a function are implicitly `const`, as proposed in [P3281R0].

SF	F	N	A	SA
5	2	0	6	8

Result: No consensus

Poll 3. For the Contracts MVP, specify that the predicate of a checked assertion statement (contract_assert) is evaluated exactly once, as proposed in [P3257R0] Proposal 2.

SF	F	N	A	SA
8	4	0	4	3

Result: No consensus

Poll 4. For the Contracts MVP, specify that the predicate of a checked contract assertion of any kind (`pre`, `post`, and `contract_assert`) is evaluated exactly once, as proposed in [P3281R0].

SF	F	N	A	SA
4	2	1	4	10

Result: Consensus against

None of these polls had consensus to change, but clearly members expressed more of an appetite to change `contract_assert` than `pre` or `post`. Unlike `contract_assert`, which is permitted to appear only in the body of a function, `pre` and `post` have no analog with the standard C `assert` macro and can appear on the interface; therefore, changing them would affect many more important use cases that involve multiple evaluations to support caller- and callee-side checking — e.g., across application and library boundaries as well as for virtual functions, function pointers, smart pointers, and so on. Hence, the remainder of this paper focuses exclusively on whether and to what extent any change to the preconditions and essential behavior of just the `contract_assert` *contract assertion* is warranted.

3 Solutions

In this section, we discuss proposed solutions for how to implement contract assertions in general under the assumption that all assertions will be treated the same. We can consider, as a second round, whether we want to treat `contract_assert` differently from `pre` and `post`. (We argue that we do not.) We are also assuming that we want to keep the `const`-ification of predicates in place. Note that having `const`-ification reinforces the desire not to support destructive side effects in the MVP and vice versa.

We employ the following abbreviations in our discussion of solutions.

- **EL:** the elision clause; can ignore at compile time when predicate is always false
- **RE:** the ability to repeat evaluation of a predicate; enables requesting arbitrary repetitions and duplicates for `pre` and `post`
- **IM:** implementation latitude to map distinct semantics within a single TU; semantics of each invocation of every contract assertion is controlled independently
    - **"– IM":** all contract assertions within a TU are fixed at either all checked or all unchecked; similar to C `assert` where exactly 1 flag controls check of all contract assertions

3.1 Solution A: Don't Do Anything for C++26 (Wait for C++29)

- Motivating Principles
    - Leave all options open.
        - Every possible choice is entirely preserved for C++29.

3.2 Solution B: Keep P2900 As Is (i.e., P2900R7)

- Motivating Principles
    - Maintain `contract_assert` consistency with current `pre` and `post` as defined in P2900R7.
        - This relaxed specification for `contract_assert` would be easier to teach alongside `pre` and `post` if all three of these contract assertions behave consistently.
    - Allow for independent library- and application-side checking.
        - This checking is needed for multiple mixed-mode precompiled libraries.
    - Make teaching the feature easier.
        - There is less to teach; we need not teach all of it, just `pre(exp)`. No side effects occur in `exp`.
    - Be consistent with `const`-ification.
    - Facilitate unit and beta testing for destructive side effects.
        - Destructive side effects might lead to correctness, safety, and/or security-vulnerability defects.

  - Discourage, by default, use of destructive side effects.
  - Leave open the widest possible range of implementation choices.
    * Every invocation of every contract assertion can be any semantic.
  - Allow backward-compatible migration of the default behavior to C, D, E, or F.

3.3 Solution C: P2900 Without Elision Clause (P2900 − EL)

- Motivating Principles
  - Enable testing for destructive side effects by repeating many times.
  - Enable reliance on idempotent side effects not statically provable to be 0.
    * An example would be something that sets a flag to indicate that the predicate was encountered during the flow of control and evaluated to false (at least once).
    * Another would be that the runtime-only-evaluable predicate was encountered at least once.
  - Allow backward-compatible migration of the default behavior to E or F.

3.4 Solution D: P2900 Without Repetition (P2900 − RE)

- Motivating Principles
  - The cost of the assertion predicate evaluation will never be multiplied.
  - Facilitate teaching via familiarity to existing developers.
    * Most developers are not familiar with code that might execute more than once.
    * Developers are used to the idea that copies are sometimes optionally elided.
  - Allow backward-compatible migration of the default behavior to E or F.

3.5 Solution E: P2900 Without Elision and Repetition (P2900 − EL − RE)

- Motivating Principles
  - Enable testing for destructive side effects by repeating many times.
  - Enable reliance on idempotent side effects not statically provable to be 0.
  - The runtime cost of the assertion predicate evaluation will never be multiplied.
  - Facilitate teaching via familiarity to existing developers.
  - Allow backward-compatible migration of the default behavior to F.

3.6 Solution F: P2900 Without Elision and Repetition and With Only One Kind of Semantic (i.e., Checked or Unchecked) per TU (P2900 − EL − RE − IM)

- Motivating Principles
  - Enable testing for destructive side effects by repeating many times.
  - Enable reliance on idempotent side effects not statically provable to be 0.
  - The cost of the assertion predicate evaluation will never be multiplied.
  - Facilitate teaching via familiarity to existing developers.
  - Behave like C `assert`; per TU, contract assertions are either all checked or unchecked.
  - Be completely defined (no backward-compatible migration path).

4 Backward-Compatibility Summary

- **EL:** the elision clause
- **RE:** the ability to repeat evaluation of a predicate
- **IM:** implementation latitude to map distinct semantics within a single TU

```
                          [ A ]      // A = nothing in C++26
                            |
                            v
                          [ B ]      // B = P2900
                         /     \
                        v       v
// D = P2900 - EL     [ C ]   [ D ]  // D = P2900 - RE
                         \     /
                          v   v
                          [ E ]      // E = P2900 - EL - RE
                            |
                            v
                          [ F ]      // F = P2900 - EL - RE - IM
```

5 Counteroffers to P3281R0

Rather than break the consistency of the current Contracts MVP, which was developed to provide an extensible framework that could grow to address arbitrary uses cases, we thought about how we might possibly develop solutions that would address the needs of those who view contracts as a modern, more flexible replacement for legacy use of the standard C `assert` macro. We came up with five counteroffers, each of which preserves all the use cases of P2900R7.

1. **Raw Access:** Provide raw access to call the handler, thereby allowing clients to build their own local custom version of C `assert` as a macro or otherwise.

2. **Retrofit C `assert`:** Provide a way to opt in to giving the standard C `assert` macro in C++ the ability to opt in to calling the contract-violation handler.

3. **Two Contract Asserts:** Introduce, in addition to `contract_assert` (which would remain unchanged), another form of contract assertion having a different spelling, say, `contract_cassert` or `contract_c_assert`, that would always have the semantics of Solution F above.

4. **Build Mode:** Require the ability to configure all contract assertions within a TU as either *all* checked or *all* unchecked.

5. **Wait for Post-MVP Contracts:** We plan to have syntax that will allow users to cleanly and modularly specify — either via labels or via `contract_support` — literal semantics in the code itself, which would include the behavior desired by Solution F.

6 A Proposal (P3290) That Can Satisfy the Multiverse

Until now, our opening bid, Solution B (P2900R7) seemed to have an edge on any of its alternatives because it immediately provides substantial value and can grow to meet all needs. Unfortunately, we don't have sufficient syntax to make it work in time for the C++26. To build consensus in time to get an enhanced MVP into the stable state in time for C++26, we used the information obtained from our principled-design approach ([P3004R1]) to synthesize a new solution. Instead of removing capabilities from of the current MVP, we instead provide three independent but mutually compatible proposals that collective provide overlapping support for uses cases involved the standard C++ `assert` macro and other, similar legacy assertion frameworks.

The gist of the three synergistic yet independent proposals are derived from the first three counteroffers in the previous section.

1. Provide direct access (via an additional API) to invoke the MVP's contract-violation handler, thereby allowing legacy assertion systems to benefit from the new, centralized facility with minimal code churn.

2. Modify the C++ definition of C `assert` to conditionally support calling the handler instead of outputting the currently specified diagnostics to `cerror`, thereby allowing existing code to take advantage of the MVP's contract-violation hander without having to modify (or possibly even recompile) the source code.

3. Provide additional syntax in the source, specifically, a new keyword that mirrors `contract_assert` that, by default has behavior that is plug-compatible with that of the standard C++ `assert` macro, thereby allowing those who wish to incrementally upgrade their legacy systems to avoid macros can do so without having to understand how to hand-check whether the predicate each C `assert` is to be converted to is valid for the new more build-time flexible `contract_assert`.

The details that these three proposals comprise can be found in [P3290R0]. Incorporating this new proposal into P2900 leads to our final solution.

6.1 Solution G: P2900 Plus All Three Proposals in P3290 (P2900R7 + P3290R0)

- Motivating Principles
  - All of the above!

7 Discussion of Motivating Principles

For succinctness, we gathered all the principles, refined them, added general principles, determined their importance and objectivity, and ranked them.

7.1 General Principles

- Maximize perception that the Contracts facility is aimed at improving safety.
- Maximize the ability to reduce UB in new and legacy code.
- Maximize the ergonomics of replacing existing `assert` using Contracts.
- Maximize teachability for those using C `assert` for contract checking.
- Maximize teachability for a new paradigm in which side effects are to be avoided.
- Maximize the ability to mix and match prebuilt libraries and applications.
- Maximize the ability to extend the MVP to solve all use cases in P1995R1.
- Maximize the ability to change the defaults if necessary once we have more knowledge.
- Maximize applicability to reusable libraries at scale (e.g., the C++ Standard Library).
- Maximize applicability to self-contained applications.
- Maximize the creation of build-independent source code.
- Maximize options for implementers of the Standard (e.g., to improve safety).
- Maximize the ability to have simultaneous caller- and callee-side checking.
- Maximize the ability to have simultaneous client- and library-side checking.
  - What's more important for the MVP?
    - Keeping our implementation options open with 2900?
    - Making client code that is checked at least once?
    - Making client code that is checked no more than once?
    - Both: i.e., exactly once?

        * Guarantee that destructive side effects work in the MVP?
        * **Rule:** Without strong consensus, choose a solution that allows for backward-compatible migration to all other viable solutions.

    The MVP — by definition — cannot address all the solutions identified in SG21's initial list of desired solutions (P1995R1) unless every single one of them is needed for the initial release of the C++ Contracts feature to be viable.

- Maximize the ability of the MVP to be a drop-in replacement for all C `assert`.
    - To achieve deterministic single evaluation in the C++ MVP, we would necessarily have to eliminate the ability
        1. to have finer-grained control over contract assertions (e.g., keeping all `pre` contract assertions checked and leaving all `post` contract assertions unchecked) in both the MVP and, by default, all future unlabeled contract assertions
        2. for implementations to allow for asynchronously changing semantics at run time (and again for all default contract assertions moving forward)
        3. for a client to ensure that every precompiled library API it uses is checked at least once (without causing an ABI break or forcing a global recompile) with the MVP and by default (without labels) post-MVP
        4. for the default behavior (i.e., no labels on the contract assertion) to ever change back to that of P2900
        5. for independent precompiled client- and library-side use without API breaks
        6. for independent caller- and callee-side checking, as in virtual functions, function pointers, smart pointers, to migrate to Contracts
- Essential syntax is envisioned for a future release.
    - Additional syntax gives us the ability to provide potentially unique identifiers that will allow
        1. individual contract assertions to be controlled externally
        2. prevention of specific contract assertions being controlled externally
        3. indication that specific subsets of contract assertions be treated as a unit
        4. indication that specific subsets of checked contract assertions not be afforded flexibility to evaluate their respective predicates other than exactly once each time one of those contract assertions is encountered at run time

    and hence these multiple usage paradigms will be able to work together synergistically at scale.

8 Prioritizing the Principles

In this section, we follow the process described in P3004R1 of bringing down the motivating and general principles from above, removing duplicates, curating them to apply to all solutions equally. We then assign each a short mnemonic principle ID, and characterize each with respect to objectivity and importance (note that an `@` means provably objective and a `-` means subjective). Finally, we rank them in some rough order of priority (though, this case, the order will turn out not to affect our conclusion).

Table 1: Scoring Importance and Objectivity of Principles

Rk	i	o	Principle ID	Principle Statement
5	9	@	IsConsPre	Consistent with current pre/post.
8	9	-	MaxTeachCA	Easy to teach `contract_assert`.
16	5	5	MaxTestSId	Facilitate unit/beta testing for destructive side effects.
17	5	5	MinSideEff	Discourages, by default, use of destructive side effects.
6	9	5	MaxImplFex	Allows widest possible range of implementation choices.
14	5	@	MaxBackCom	Allows backward-compatible migration of default behavior.
20	1	@	MaxIdempSi	Can rely on most kinds of *idempotent* side effects.
13	5	@	IsFixedCst	Number predicate evals is limited, in source code, to 1.
15	5	@	ISCassert	Supports *all* use cases for the standard C assert macro.
4	9	@	MaxCassert	Supports *most* use cases for the standard C assert macro.
1	9	-	MaxPerSafe	Maximizes perception MVP aims to improving safety
2	9	5	MaxLoseUB	Maximizes ability to reduce UB in new and legacy code
11	5	5	MaxErgCass	Maximizes ergonomics of replacing legacy asserts with MVP.
18	5	-	TeachCass	Maximizes teachability for those familiar with C `assert`
9	9	-	TeachNoSid	Maximizes teachability for those not needing side effects.
3	9	5	MaxExtent	Maximizes ability to extend MVP to solve *all* use cases.
10	9	-	MaxLibrary	Maximizes applicability to reusable libraries at scale.
19	5	-	MaxApp	Maximizes applicability to self-contained applications.
7	9	5	IsBldIndSc	Will eventually allow for build-independent source code.
12	5	@	IsSafeHois	Is safe when hoisting `contract_assert` to `pre`.

Now we reorder the rows.

Table 2: Putting the Principles in Ranked Order

Rk	i	o	Principle ID	Principle Statement
1	9	-	MaxPerSafe	Maximizes perception MVP aims to improving safety
2	9	5	MaxLoseUB	Maximizes ability to reduce UB in new and legacy code
3	9	5	MaxExtent	Maximizes ability to extend MVP to solve *all* use cases.
4	9	@	MaxCassert	Supports *most* use cases for the standard C assert macro.
5	9	@	IsConsPre	Consistent with current pre/post.
6	9	5	MaxImplFex	Allows widest possible range of implementation choices.
7	9	5	IsBldIndSc	Will eventually allow for build-independent source code.
8	9	-	MaxTeachCA	Easy to teach `contract_assert`.
9	9	-	TeachNoSid	Maximizes teachability for those not needing side effects.
10	9	-	MaxLibrary	Maximizes applicability to reusable libraries at scale.
11	5	5	MaxErgCass	Maximizes ergonomics of replacing legacy asserts with MVP.
12	5	@	IsSafeHois	Is safe when hoisting `contract_assert` to `pre`.
13	5	@	IsFixedCst	Number predicate evals is limited, in source code, to 1.
14	5	@	MaxBackCom	Allows backward-compatible migration of default behavior.
15	5	@	ISCassert	Supports *all* use cases for the standard C assert macro.
16	5	5	MaxTestSId	Facilitate unit/beta testing for destructive side effects.
17	5	5	MinSideEff	Discourages, by default, use of destructive side effects.
18	5	-	TeachCass	Maximizes teachability for those familiar with C `assert`
19	5	-	MaxApp	Maximizes applicability to self-contained applications.
20	1	@	MaxIdempSi	Can rely on most kinds of *idempotent* side effects.

9 Evaluating the Solutions

In the compliance table below, we apply our curated principles from the the previous section in the same rank order and score them as explained in P3004.

Solution Legend

A. No MVP for C++26

B. MVP = P2900R7

C. MVP = P2900R7 − EL

D. MVP = P2900R7 − RE

E. MVP = P2900R7 − EL − RE

F. MVP = P2900R7 − EL − RE and − IM

G. MVP = P2900R7 + P3290R0

Table 3: Compliance Table

Rk	i	o	Principle ID	A	B	C	D	E	F	G
1	9	-	MaxPerSafe	-	@	-	-	-	-	@
2	9	5	MaxLoseUB	-	@	-	-	-	-	@
3	9	5	MaxExtent	-	@	-	-	-	-	@
4	9	@	MaxCassert	-	@	@	@	@	@	@
5	9	@	IsConsPre	-	@	-	-	-	-	@
6	9	5	MaxImplFex	-	@	-	-	-	-	@
7	9	5	IsBldIndSc	-	@	-	-	-	-	@
8	9	-	MaxTeachCA	-	@	-	-	-	@	@
9	9	-	TeachNoSid	-	@	-	-	-	-	@
10	9	-	MaxLibrary	-	@	-	-	-	-	@
11	5	5	MaxErgCass	-	-	-	-	-	@	@
12	5	@	IsSafeHois	-	@	-	-	-	-	@
12	5	@	IsFixedCst	-	@	-	-	-	-	@
14	5	@	MaxBackCom	-	-	-	@	@	@	@
15	5	@	ISCassert	-	@	-	-	-	-	@
16	5	5	MaxTestSId	-	-	-	-	-	@	@
17	5	5	MinSideEff	-	@	@	@	@	-	@
18	5	-	TeachCass	-	-	-	-	-	@	@
19	5	-	MaxApp	-	-	-	-	-	-	@
20	1	@	MaxIdempSi	-	-	@	-	@	@	@

The conclusion is that the synthesized solution, G, is the clear winner!

10 Conclusion

[P2900R7] provides an extensible MVP that can grow to meet the needs of all anticipated future use cases. After sharing it with EWG, we learned of an urgent need to address a wider set of use cases, particularly those currently addressed by the standard C `assert` macro. After analyzing the motivating principles for each of the proposed solutions, we used the information obtained from the principled analysis to synthesize a new, comprehensive solution ([P3290R0]) that address essentially all the use cases supported by C `assert` as well as most other custom legacy `assert` facilities. We, therefore, propose that the proposals in P3290R0 be incorporated into P2900R7 and that this new version, P2900R8, become the stable definition of the MVP with no further alterations (especially breaking changes) other than those that substantially increase consensus.

Acknowledgements

Thanks to Timur Doumler for providing feedback on this paper, to Mungo Gill for his thoughtful review comments, to Harold Bott for his help in both writing and coordinating our efforts, and to Lori Hughes for forming this paper into something that is recognizably in the English language.

Bibliography

[P1995R1] Joshua Berne, Andrzej Krzemieński, Ryan McDougall, Timur Doumler, and Herb Sutter, "Contracts — Use Cases", 2020
`http://wg21.link/P1995R1`

[P2695R1] Timur Doumler and John Spicer, "A proposed plan for contracts in C++", 2023
`http://wg21.link/P2695R1`

[P2900R7] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, "Contracts for C++", 2024
`http://wg21.link/P2900R7`

[P3004R1] John Lakos, Harold Bott, Bill Chapman, Mungo Gill, Mike Giroux, Lori Hughes, Alisdair Meredith, and Oleg Subbotin, "Principled Design for WG21", 2024
`http://wg21.link/P3004R1`

[P3257R0] Jens Maurer, "Make the predicate of `contract_assert` more regular", 2024
`https://isocpp.org/files/papers/P3257R0.html`

[P3281R0] John Spicer, "Contract Checks Should Be Regular C++", 2024
`https://isocpp.org/files/papers/P3281R0.pdf`

[P3290R0] Joshua Berne, Timur Doumler, and John Lakos, "Integrating Existing Assertions With Contracts", 2024
`http://wg21.link/P3290R0`