Evaluation of *Checked* Contract-Checking Annotations

Document #:	P2751R1
Date:	2023-2-14
Project:	Programming Language C++
Audience:	SG21 (Contracts)

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

Abstract

SG21 is ready to discuss and decide how *checked* contract-checking annotations will be evaluated in the Contracts MVP. We present here a thorough analysis and direction for *handling* all possible results of the normal evaluation of a contract-checking annotation's predicate. We then provide details and motivation for deliberately making *unspecified* the number of evaluations of the predicate of a checked contract; i.e., each such predicate *may* be evaluated zero, one, or more times.

Contents

1 Revision History

Revision 0

- Original version of the paper for discussion during an SG21 telecon in January 2023

Revision 1

- Significant clarifications on Proposals 2.3, 2.4, and 3.4
- Summarized results of SG21 discussions at the February 2023 WG21 meeting in Issaquah

2 Introduction

The Contracts MVP being developed by SG21 (see [P2521R2]), as with most other existing runtime contract-checking facilities, allows for contract checks to be expressed in code as contract-checking annotations and then, in certain situations, to be *checked*.

When a contract-checking annotation is *checked* and a contract violation is detected, the associated behavior, such as invoking `std::abort` in the `Eval_and_abort` mode proposed for the MVP as described in [P2521R2], must occur. The *specifics* of how the predicates of *checked* contract-checking annotations are evaluated to detect a violation are the subject of the current technical discussion and are needed for the MVP to continue on the path laid out by [P2695R0] to produce a Contracts facility for C++26.

We propose a series of rules to govern the evaluation of predicates and thus to determine when a contract violation has occurred. These rules include clarifying how a predicate's evaluation detects a violation as well as what freedoms an implementation has in choosing whether to evaluate that predicate (at run time) for the purpose of detecting a violation.

Taken together, we assert that these rules simultaneously maximize safety, implementation freedom, and benefits to the end user.

3 Proposals

We now present a series of discrete proposals along with motivations for each individual decision. Note that much of this content overlaps heavily with the discussions in the MVP and [P2570R0] and should be considered as additional reasoning and motivation for specific options outlined in those papers.

Understand that we are concerned here with what happens when evaluating a *checked* contract-checking annotation, i.e., one in which, if a contract violation occurs, then an associated observable behavior also occurs, such as invoking `std::abort`. When building in the `Eval_and_abort` mode in the MVP, all contract-checking annotations are *checked*, and hence the runtime boolean value of the predicate must be determined. When building in the `No_Eval` mode, contract-checking annotations are *not* checked, and their predicates are never evaluated.

3.1 Expressions

Proposal 1: Predicates are normal C++ expressions.

A contract-checking annotation's predicate is a C++ expression that contextually converts to `bool` and, when evaluated (including the conversion to `bool`), follows the normal C++ rules for expression evaluation.

Expressing preconditions, postconditions, and assertions as evaluatable C++ expressions has been central to all existing Contracts proposals that have been considered by WG21. The MVP currently does not deviate from this familiar paradigm.

Normal evaluation brings with it two important points.

1. When the evaluation of a predicate occurs, that evaluation, behaving just as with other complete expressions, will not overlap with the evaluation of any other expressions.
2. For any given subexpression of a contract check's predicate, its defined behavior will match that which it would have if evaluated within the body of a function.

Normal evaluation does *not* extend to other guarantees that might inadvertently be assumed.

- The number of times the predicate for a contract-check is evaluated, if at all, is not guaranteed.
- A predicate whose evaluation has undefined behavior results in a contract-checking annotation whose evaluation has undefined behavior and thus is never guaranteed by the Standard.
- Behaviors that would be associated with the specific *context* in which a predicate is evaluated, as opposed to those associated with the evaluation of the predicate itself, are not guaranteed to be the same as in other contexts in which expressions are evaluated and converted to `bool`. For example, as we indicate in Proposal 2.3 below, an exception escaping the evaluation of a predicate might be handled as a contract violation rather than propagating into the enclosing scope.

Requiring that evaluations of contract-check predicates follow the same rules as those experienced C++ developers must already strive to know and understand for all other contexts has several important collateral benefits:

- The use of boolean expressions for the detection of *contract violations* maximizes teachability and understandability. Leveraging the existing semantics of C++ expressions fits naturally alongside the rest of the language and avoids the need to learn a new language or understand alternate semantics for the already complex C++ language.
- When designing APIs employing functions that have narrow contracts, we commonly observe parallel sets of functions having wide contracts that return an error when used outside of the narrow contract's domain. The `operator[]` and `at` accessors of `std::vector` are an example of parallel APIs such as this, where `operator[]` has a narrow contract and `at` has a wide one.

  Lifting such preconditions from a contract check into the narrow function to a conditional in a wide one must not itself, for safety's sake, be a source of bugs. Cases in which the predicate

of the narrow function's contract check evaluates having different semantics than when used within an `if` are needlessly dangerous and are best avoided.

- Ideally, we want a contract-checking annotation's predicate to be able to freely use functions defined in external libraries, possibly those available only as binary distributions. Absent a massive global engineering effort, safely invoking such functions where they potentially have alternate semantics when evaluated from within the predicate of a contract check would not be achievable.

3.2 Interpretation of Expression Results

Proposal 2: When a *checked* predicate fails to evaluate to `true`, something anomalous has happened.
When control would return from the evaluation of the predicate of a contract-checking annotation through the evaluation of that annotation and the result is not `true`, a contract violation has occurred.

This overarching view of the meaning of the evaluation of a contract-check's predicate guides our proposals for handling other potential results of such predicate evaluation.

We deliberately describe interpreting these behaviors in terms of what *would* happen were the predicate to be evaluated, as opposed to what happens when the predicate *is* evaluated. As per Proposal 3.2, when a compiler can emit an alternate, side-effect-free method to identify the same contract violations, the contract checking annotation can be evaluated without having to evaluate the original predicate itself.

Note that the MVP already specifies precisely *when* a contract-checking annotation is evaluated; e.g., a precondition is evaluated immediately after function parameters are initialized. We do not propose any changes to those rules.

When evaluating the predicate of a contract-checking annotation, if the flow of control never returns to the evaluation of the contract-checking annotation (e.g., the program is terminated or `longjmp` is invoked), the anomalous control flow must not be subverted, lest we would violate Proposal 1. On the other hand, control flow that does pass back through the point of evaluation of the annotation yet does not produce a value (`true` or `false`) — e.g., by unwinding the stack due to an uncaught exception — is always a coding defect and thus treated as a *contract violation*.

Proposal 2 is further subdivided across the many ways an expression might produce or fail to produce a boolean value, and each method has distinct motivations for how it should be handled.

Proposal 2.1: A predicate that evaluates to `true` does *not* indicate that a contract violation has occurred.
When a predicate of a *checked* contract-checking annotation would evaluate to `true`, no contract violation is detected and no further action is taken.

Proposal 2.2: A predicate that evaluates to `false` indicates that a contract violation has occurred.

When the predicate of a *checked* contract-checking annotation would evaluate to `false`, a contract violation occurs.

Proposals 2.1 and 2.2 govern how the evaluation of a contract-check's predicate that produces a well-defined value should be handled.

When the predicate would evaluate to `true`, the contract check succeeds, and program execution continues as normal. When the predicate evaluates to `false` or when it can determined that the predicate would evaluate to `false`, a contract violation has been detected, and the associated action (such as invoking `std::abort` and ideally issuing a diagnostic) is taken.

Proposal 2.3: A predicate that throws indicates that a contract violation has occurred.

When the predicate of a *checked* contract-checking annotation throws (or would throw), a *contract violation* occurs.

We could consider three possibilities for handling a contract-check's predicate that throws.

1. Allow the exception to propagate.
2. Treat the predicate as implicitly `noexcept`, invoking `std::terminate`.
3. Catch the exception, consider the contract check to have failed, and proceed just as if any other contract violation had occurred.

Option 1, allowing the exception to propagate, can result in control-flow decisions based on the contract check being enabled, which, in turn, would allow *essential behavior* to differ across distinct build modes.[1]

Allowing the exception thrown by the evaluation of the predicate to propagate would also result in the need to determine what value the `noexcept` operator would return for a function annotated as `noexcept` that had precondition annotations attached to it. We could choose to always consider the precondition check that is internal to the function (even when implemented as a call-side check) and thus subject to the `noexcept` barrier around the function, which, in that case, would naturally have the `noexcept` operator. If we choose to allow such exceptions to propagate, we would be left with a number of decisions related to the `noexcept` operator.

- Return `false` if the build mode would evaluate the contract predicate, `true` otherwise. This causes potentially (vastly) different control flows to be followed depending on the build mode.
- Return `true` always if the function is annotated `noexcept` but allow exceptions to propagate anyway,[2] potentially preventing control flow necessary to preserving exception-safety guarantees from executing.

[1] Note that predicates that throw are distinct from having user-defined violation handlers that might throw, where throwing might enable recovery from detected violations as opposed to recovery from defects in the predicate itself.

[2] This is also known as lying.

- Return `false` in all build modes, preserving consistency yet removing the use of potentially improved algorithms that would rely on the exception-generating properties of the function, which presumably would have been the reason `noexcept` was applied to the function in the first place.

The possible answers come with various ramifications.

- Invoke `std::terminate()` as if the check had been done within the body of the function, and return `true` from the `noexcept` operator.
- Propagate the exception, and always return `false`

Options 2 and 3, then, are the only potentially viable alternatives available to us. Now consider that invoking `std::terminate` reduces the level of information delivered to the client when the predicate clearly failed to validate the contract being checked. Since an uncaught exception can be locally identified (by catching it) without any need to change the semantics of the subexpressions of the predicate, we propose that uncaught exceptions be treated as violations of the contract-checking annotation being evaluated. In such cases, the evaluation of the predicate neither succeeded nor detected a contract violation and as such is likely a defective contract check; hence, identifying such situations as a contract violation is apt.

Proposal 2.4: Predicates having UB indicate a defect.

When the predicate of a *checked* contract-checking annotation would have undefined behavior (UB), a defect occurs; platforms are encouraged to treat any such UB as a *contract violation*.

We do *not* intend to redefine the abstract machine, so a predicate with UB still has UB.

On the other hand, we can recommend how such predicates are interpreted and suggest that, when practicable, a predicate that has UB should be treated as a contract violation. Without the need for normatively specifying this concept, compiler vendors already have ample leeway to make this interpretation themselves since they can provide *any* meaning they desire to UB.

This non-normative *recommendation* would encourage compiler vendors to inject null pointer detection, integer overflow detection, and any number of other checks for core-language preconditions into the evaluation of a contract-check's predicates. Avoiding making this recommendation into a *requirement*, however, gives compilers leeway to act on this recommendation. UB within the predicate expression itself might be easily identified or intractable, and how readily a compiler can detect each form of UB might vary from one implementation to the next; therefore how aggressively these defects are identified as contract violations will inevitably be a matter of quality of implementation (QoI). When invoking functions from other translation units, altering the semantics of even a subset of possible UB would require a bifurcation of how such invocations can take place, so we deliberately leave compilers the leeway to invoke such functions normally.

Note that this recommendation will ideally have a strong impact on how UB in a contract-checking predicate will "time travel" through code prior to the predicate. In general, "time-travel" UB happens as a result of compilers inspecting a control flow graph and identifying which paths in that graph can be discarded. Given the compilers' active nature in leveraging UB in such a way, we are hopeful that, in seeking good QoI, control flow paths into a contract check will be not discarded but

replaced by violation handler invocations. This form of UB is, generally, not the result of hidden implicit assumptions a compiler makes about the behavior of C++ constructs.

Proposal 2.5: Other predicates that do not evaluate normally behave normally.

Other than the cases listed above, a contract-check's predicate that fails to evaluate to a boolean value follows the normal C++ rules for expression evaluation.

Producing a value or throwing an exception are not the only possible results of evaluating an expression; they are simply the only ones by which we can readily control the compiler's response by making changes that are local to the evaluation of the predicate.

Invoking a function that does not return (e.g., one marked `[[noreturn]]`), invoking `std::longjmp`, or entering an infinite loop are all possible ways in which an expression might have behavior we have not covered so far, but other behaviors might occur as well. In the spirit of avoiding alteration of the basic semantics of evaluation of a C++ expression, we propose that all such atypical or pathological behaviors act as they would normally.

The case in which evaluation results in program termination, in particular, would be a behavior that is especially unsafe to subvert within the evaluation of a contract-check's predicate. For example, having an uncaught exception within a `noexcept` function currently always results in the invocation of `std::terminate`, and changing that guarantee should be done only with great care and consideration.

We certainly do not wish to encourage the writing of contract-checking annotations that employ such constructs; they are actively destructive to the control flow of the program itself. Altering the semantics of these operations within the evaluation of a contract-check's predicate would add complexity and reduce safety without yielding any clear overall benefit.

3.3 Number of Evaluations

Proposal 3: A predicate may be evaluated an unspecified number of times.

When determining whether a violation of a *checked* contract-checking annotation has occurred, its predicate may be evaluated zero or more times; any detected contract violation may be handled one or more times.

Proposal 3 is further subdivided into four parts, each with its own nuance and motivations.

Proposal 3.1: Evaluating a contract-check's predicate once is allowed.

When evaluating a *checked* contract-checking annotation, its predicate may be evaluated once.

The only viable route to determine if any arbitrary predicate would detect a violation is simply to evaluate it and see if the result is not `true`. This behavior is familiar from `<cassert>` and, as far as we know, has never proven controversial.

Whether zero or multiple evaluations are allowed, evaluating once at the point where the contract-checking annotation takes effect must always remain a viable implementation strategy.

Proposal 3.2: Evaluating a contract-check's predicate zero times is allowed.

When evaluating a *checked* contract-checking annotation, its predicate may be evaluated zero times.

For a contract-check's predicate that has no *observable* side effects (i.e., one that has no side effects outside of its cone of evaluation), permission to skip evaluating the predicate is already granted to the compiler. If the compiler can identify an equivalent expression that detects the same set of violations, it may replace an unobservable predicate by that alternate evaluation, which is a result of the basic *as-if* rule of the Standard.

Allowing zero evaluations for all predicates extends this rewriting ability to even those predicates having observable side effects. With these rewriting rules, when the compiler can identify an equivalent unobservable evaluation that detects precisely the same set of violations as the observable predicate, the replacement can be made, thereby eliding the side effects without allowing any violations to become undetected.

When a predicate's result can be completely determined at compile time, this rewritten expression might be as simple as `true` or `false`. In other cases, a rewritten expression might be a proper subset of the original expression with *all* observable side effects removed, provided, of course, that the compiler has determined that none of those side effects could alter whether the predicate indicates that a contract violation has occurred.

An important property of this rewriting is that developers and the compiler can still reason locally about the evaluation of a contract-check's predicate, i.e., any given subexpression of a contract-check's predicate behaves the same way it would outside of a contract-checking predicate if it does get evaluated. For each evaluation of a contract-checking annotation, either all side effects of the predicate will be observed or none of them will be.

Allowing zero evaluations even for predicates that would otherwise be *observable* has beneficial properties.

- Allowing essential behavior to depend on the side effects of evaluating a contract-check's predicate is destructive. Yet, as [P2712R0] explains, disallowing all side effects in such predicates is unduly limiting because so many algorithms that would be essential to implementing a contract check might involve allocation, modifying mutable members to cache computational results, or logging for debugging purposes. Hence, anything that discourages affecting essential behavior in a contract-check predicate is a worthy design goal.

  For example, some poorly designed libraries with destructive contract checks or with contract checks whose evaluation is needed for essential behavior might work for a particular client only with contracts always *on* or always *off*. When building a complete program in which one library works only with contracts *on* and one library works only with contracts *off*, a developer is left with an impossible choice, and the Contracts facility is blamed. This unfortunate situation can result in the inability to adopt or upgrade certain libraries, greatly impeding the adoption of the use of the Contracts facility itself.

To actively discourage dependency of any essential behavior on the evaluation of any contract-check predicates, we propose granting permission for the compiler to optionally elide all side effects of any predicate in much the same way as supposedly superfluous copies can sometimes be elided in the name of runtime performance. The result of the predicate must still be determined by what does get evaluated, but either all side effects happen or none happen.

- Encouraging any dependence on side effects of a contract-check's predicate occurring discourages future evolutionary paths that might allow mixed builds in which one contract-checking annotation is checked and a consecutive one might not be. Many of the use cases gathered by SG21 in [P1995R1] require this ability, such as disabling the checking of postconditions or labeling expensive contract checks with a label such as `audit`.

- This rule enables at least one form of symbolic evaluation, in which consecutive contract-checking annotations having functionally identical predicates can be checked once, even when they invoke opaque functions:

```
bool f();  // defined elsewhere
void g()
{
  [[ assert : f() ]];  // #1
  [[ assert : f() ]];  // #2
}
```

  Here, a single evaluation of `f()` can be used to infer one of two possibilities.

  1. The evaluation returning `true` proves that both `#1` and `#2` have been validated.
  2. The evaluation returning `false` proves that `#1` has been violated.

  Although identical consecutive assertions are rare, this same flow happens more frequently in two scenarios:

  1. When a function forwards to another function having the same preconditions
  2. When the postconditions of one function are the preconditions of another function invoked immediately after

  When we can obtain additional performance, we increase the likelihood of a checked build being usable in a production environment, though performance for *checked* contract-checking annotations is not the goal of the design of any contract-checking facility. When *checked* builds are used in production, we see increased safety delivered directly to the end users of the software we build.

  This elision of consecutive contract-checking annotations having similar (e.g., identical) predicates is limited in significant ways because we have little leeway to expand the elision when anything that cannot be reordered happens between the contract-checking annotations. Such expansion to enable more symbolic evaluations could be the subject of future proposals but brings with it the risk of missing a case and allowing a contract violation to go undetected. We assert that having a hundred superfluous contract checks go unelided is better than having one valid one go unchecked.

Proposal 3.3: Evaluating a contract-check's predicate more than once is allowed.

When evaluating a *checked* contract-checking annotation, its predicate may be be evaluated multiple times.

When evaluating a contract-check's predicate multiple times, any one of those evaluations failing to successfully evaluate to `true` indicates that a contract violation has occurred. When multiple invocations of the same predicate fail to agree on whether a contract violation has occurred, the safest interpretation is to always consider the contract violation to have occurred.

After any evaluation that does detect a contract violation is performed, any violation handling routine should immediately follow. In the context of the MVP, with no checked build mode that allows continuation, no more than one failed evaluation of a contract-check's predicate will occur. Should violation handling modes that support continuation be added, detecting the same violation multiple times would then be possible.

Allowing multiple evaluations also brings a few beneficial properties.

- As with evaluating zero times, evaluating a contract-check's predicate having no observable effects multiple times is completely within the allowed behavior of the compiler under the *as-if* rule. In general, compilers do not take advantage of the leeway; doing so is considered poor QoI, but this proposal extends that same leeway to predicates with observable side effects.
- For contract-checking annotations with predicates having observable side effects, all of the concerns related to discouraging any dependency on side effects that we raised in the discussion of Proposal 3.2, with respect to zero evaluations, apply as much (if not more) to allowing multiple evaluations. Side effects that are needed for the essential behavior of a function will typically fare poorly if evaluated multiple times; we argue that such discouragement is not a bug but a welcome feature of this proposal.
- In general, having a contract-checking facility apply blame to the source of a defect as accurately as possible is hugely beneficial. Any group that has supported a library that makes heavy use of macro-based assertions to check preconditions knows that, in the vast majority of cases, problems get assigned first to the owner of the library, even when the problem stems from a client invoking that library's functions with invalid input.

  Enabling compilers to place precondition-checking code on the caller side of a function invocation assists greatly toward allowing simple implementation strategies (e.g., that don't require passing the source location of the caller to the function) to nonetheless produce the caller's identity (file and line number) on precondition violations. A Contracts facility that enabled just that and little else would be a welcome improvement over many macro-based facilities.

  On the other hand, such checking on the caller side becomes difficult to implement when a function is invoked indirectly, such as through a function pointer. Allowing any *checked* contract-checking annotation to go unchecked would violate the most essential requirement we have for a contract-checking facility; hence precondition checks might often be best placed inside the invoked function, not at the call site.

Implementation strategies can alleviate some of these conflicts, and a subset of these strategies might do so by sometimes checking a contract both on the caller side (for improved diagnostics) and the callee side (to guarantee that checks are never skipped). The viability of these implementation strategies might hinge largely on the ability and desire to change calling conventions and ABIs to facilitate contract checking, and we expect the desire to make substantial ABI changes to be minimal to nonexistent with an initial MVP.

Allowing multiple evaluations enables platforms to experiment with different implementation strategies to maximize diagnostic quality while still always guaranteeing that violations of *checked* contracts are detected.

- Though the MVP does not currently provide great flexibility in specifying the preconditions and postconditions of virtual functions, that is one case in which it will potentially be possible in the future for callers to be aware of a distinct contract from the callee, and both of those lists of contract checks will need to be evaluated. When a caller invokes a virtual function through a base-class pointer or reference, the contract the caller is aware of and upon whose postconditions it depends is the contract on the base class's virtual function. When a virtual function is invoked, the particular function depends on its own preconditions, i.e., those on the function declared in the dynamic type of the object.

  Today, when these contract checks are the same, the distinction does not matter. When evolving the MVP into one that meets more of the known use cases for contracts, however, callers and callees can have multiple possibly distinct (but also possibly identical) contract-checking annotations to evaluate on any given virtual function invocation. Leaving the number of evaluations unspecified allows a compiler to check on both sides when the runtime type is unknown but check only once when a virtual dispatch is devirtualized.

- Mixed-build modes is another area for which the MVP does not currently allow but which is nonetheless required for many of the known use cases. To guarantee desired checks get evaluated when a caller might be built in an unknown mode, the compiler of, for example, a library function might wish to always check preconditions — even when that check might be redundant. Once again, allowing multiple evaluations enables the generation of redundant checks to guarantee that no build inadvertently ends up not checking a *checked* contract-checking annotation at all.

- Since the MVP is intended to lay the foundation of a feature that can evolve to satisfy many more of the use cases the community has for a Contracts facility, we would be wise not to overconstrain the design at this early stage in its development. Once it is released, we will begin to gain implementation experience and build toward consensus on a more complete feature with the MVP as its core. If we adopt the most relaxed specification allowing zero or more evaluations, i.e., Proposals 3.2 and 3.3, and then later we find that no implementation issues or user-scenarios require such flexibility, we can always consider tightening the specification. Obviously, going the other way — from more restrictive to less — is not even a remote possibility; hence, our strong preference is to at least start with zero or more evaluations and see where that takes us.

Proposal 3.4: Extra evaluations of consecutive predicates may be reordered.

When a sequence of contract-checking annotations are evaluated consecutively, additional checks of each annotation (allowed by previous proposals) may be inserted anywhere after the first check of that annotation within the sequence. Note: These additional checks *may* but are not *required* to involve a non-zero number of evaluations.

With Proposal 3.3, any single contract-checking annotation might involve multiple evaluations of that annotation's predicate and violation handling. (Note again that the MVP's violation handling always invokes `std::abort` and thus necessarily occurs only once.) Consider, though, a function with multiple precondition checks:

```
void f1()
  [[ pre : p1() ]]
  [[ pre : p2() ]];
```

To check preconditions at both the call site and at the start of the body of `f1`, simply allowing multiple evaluations is insufficient. To have this implementation freedom, compilers need to have the ability to invoke `p1()` and then `p2()` at the call site and then invoke them *again* in the implementation of `f1`. To facilitate that implementation flexibility, we must allow reordering of the repeated evaluation of the predicates.

To understand why we would constrain this reordering, consider the simple case of preconditions where undefined behavior in one precondition is guarded by an earlier precondition:

```
void f(int *p)
  [[ pre : p != nullptr ]]  // #1
  [[ pre : *p != 5 ]];      // #2
```

Reordering `#1` to occur after `#2` would prevent catching the violation when `f(nullptr)` is invoked. On the other hand, repeating the entire sequence of checking `#1` and `#2` would be fine as would partially repeating the sequence after the first invocation. By requiring that checks be performed only after lexically (and possibly logically) prior enabled checks have been performed (at least once), we match the natural expectations a developer might reasonably assume are guaranteed.

Rather than limit this reordering to just single batches of preconditions or postconditions, we propose extending this flexibility to any sequence of consecutive contract checks. Formally, this proposal would consist of two parts:

- A vacuous operation is one that should not, a priori, be able to alter the state of a program which a contract could observe, and thus could not induce a contract violation. Two contract-checking annotations shall be considered consecutive when they are separated only by vacuous operations Examples of such vacuous operations include
  - doing nothing, such as an empty statement
  - performing trivial initialization, including trivial constructors and value-initializing scalar objects
  - performing trivial destruction, including destruction of scalars and invoking trivial destructors

  - initializing reference variables
  - invoking functions as long as none of the function parameters require a nonvacuous operation to initialize

  Consecutive evaluations will occur in a number of common places, which include but are not limited to

  - all precondition checks on a single function when invoking that function
  - all postcondition checks on a single function when that function returns normally
  - the preconditions of a function (`f1`) and the preconditions of the first function invoked by `f1` (`f2`), when preparing the arguments to the invoked function (`f2`) involves no non-trivial operations
  - the postconditions of a function (`f1`) and the preconditions of the next function invoked immediately after `f1` returns (`f2`), when the destruction of the arguments of the first function (`f1`) and the preparation of the arguments of the second function (`f2`) involve no non-trivial operations

- A natural ordering occurs among all contract-checking annotations based on where they appear lexically within a function declaration and the normal ordering of function invocations themselves. When one contract-checking annotation is sequenced before another, at least one evaluation of the first annotation must happen before a check of the second may begin.

Allowing reordering of evaluations of contract-checking annotations having predicates that would otherwise be *observable* also has beneficial properties.

- As with allowing zero or multiple evaluations of a predicate, the compiler can already, under the *as-if* rule, perform this same reordering when the predicates of a contract check have no observable effects. This form of reordering is a frequent target of aggressive optimizations.
- Consider a simple function with two preconditions:

  ```
  void f()
    [[ pre : f() ]]   // #1
    [[ pre : g() ]]; // #2
  ```

  To allow for the possibility of evaluating the contract-checking annotations in both the caller and the callee, we want to allow these checks to be run in the sequence `#1`, `#2`, `#1`, and then finally `#2`. Proposals 3.1, 3.2, and 3.3 would allow us to expand this only to sequences such as `#1`, `#1`, `#2`, and then `#2`.

  Allowing the redundant checks enabled by Proposal 3.3 to be placed anywhere later within the full sequence of evaluations enables us to insert a second check of `#1` at the end of the sequence, followed by a second check of `#2`, resulting in the repetition of the full set of checks that can be done in the caller of `f()` and within `f()` itself.

- The ability to insert duplicates of a check that has been confirmed allows for the elision of different but equivalent checks. Consider a nested set of functions with some related and some unrelated preconditions:

```
bool f();
bool g();
bool h();

void inner()
  [[ pre : f() ]]   // #1
  [[ pre : h() ]];  // #2

void outer()
  [[ pre : f() ]]   // #3
  [[ pre : g() ]]   // #4
{
  inner();
}
```

With no repetition, we would expect the checks when calling `outer()` to evaluate `#1`, `#2`, `#3`, and then `#4`. Ideally, we want to allow for the freedom to remove the check `#3`, which is a redundant check of `f()`. Doing so, however, would not be valid without a reordering guarantee that avoids the need to confirm that the call to `g()` made by checking `#2` does not invalidate the truth of `f()` — something that may well be impossible if `g()` is defined in a different translation unit.

Repetition within the sequence enables us to insert a second check of `#1` immediately prior to the check of `#3`. We can then use that `#1` has already been evaluated by that time and determined to be `true` to evaluate `f()` zero times to validate the second check of `#1`. The result of this check then confirms that we need not evaluate `f()` a second time to check the immediately following check of `f()` within `#3`.

The sequence of annotation checks is then `#1`, `#2`, `#1`, `#3`, `#4`. The sequence of annotation predicates evaluated, however, is just `#1`, `#2`, and then `#4`.

Larger systems with well-connected chains of postconditions and preconditions can then leverage this rule to remove verifiably redundant checks in a wide variety of scenarios.

- Enabling arbitrary reordering of an unspecified number of evaluations further encourages users to treat each individual contract-checking annotation as independent, particularly with respect to any side effects contract predicates might have.

  The freedom to apply Proposal 3.2 depends on not having an expectation that the side effects of blocks of contract-checking predicates may be treated as a single unit. Consider, for example, a function that is written making this bad assumption:

  ```
  void f()
     [[ pre : d_mutex.lock(), true ]]     // #1
     [[ pre : check_state() ]]
     [[ pre : d_mutex.unlock(), true ]];  // #2
  ```

  A compiler may, for various reasons, be capable of concluding that `#2` always returns `true` but not come to the same conclusion for `#1`, resulting in failure to unlock a mutex that has been locked. This situation would be likely if, for example, the definition of `unlock` were inlined and visible while the implementation of `lock` were hidden within a different translation unit.

Enabling users to depend on paired side effects of distinct contract-checking annotations would also prevent future evolutionary paths that allow for fine-grained control of individual contract-checking annotations.

- Independence of distinct contract-checking annotations is also crucial because different translations of the same function might be able to optimize more or less effectively when looking at a set of contract checks. Consider, for example, the situation where both a caller and callee compiler emit evaluations of a sequence of preconditions, the two compilers might have different levels of success analyzing the predicates being compiled and replacing evaluations with side effects with ones that are not observable. One could expect to encounter this disparity with relative frequency since compilation of a function's definition will often have more visibility into other function definitions within the same library than compilations that see only the library's declarations.

We could structure this reordering in other equivalent ways, each of which allows for different understanding of the evaluations we will allow, leading to the reordering rule we have proposed.

- The original wording for this proposal was "When multiple contract-checking annotations are evaluated consecutively, any *extra* evaluations of each predicate may be reordered with respect to each other." The reasoning behind this rewording was that all the proposed transformations described above would be enabled by repeating the checks of each annotation the desired number of times and following that up with shuffling, which continued to meet the constraint on the ordering of initial checks. Our current wording does not fundamentally deviate from that original wording.
- Another view would be to extend the duplication to allow repetition of entire sequences of checks, followed by arbitrary elision of redundant checks or the tail of one of the sequences. With a sufficient number of repetitions and elisions, an arbitrary sequence of checks could be emitted.

  To see this, consider 5 checks named `A`, `B`, `C`, `D`, and `E`. To produce the sequence `ABABCDEAD`, we perform the following steps:

```
ABCDE
ABCDE ABCDE ABCDE ABCDE
AB    AB    ABCDE ABCDE  <-- drop tails
AB    AB      CDE A  D   <-- elide redundant checks
                ABABCDEAD  <-- final sequence of checks
```

- A different formulation for the reordering restriction is that the sequence of *first* checks of each annotation is the complete original sequence, in the original order:

```
ABABCDEAD
AB  CDE     <-- first checks
```

4 Wording Changes

The current MVP (i.e., [P2521R2]) does not contain suggested wording, somewhat by design. A previous paper, [P2388R4], contains standard wording for an earlier iteration of the MVP, and the final wording for the MVP can be expected to evolve from that.

4.1 Consensus Changes

As of this publication, some proposals in this paper have received consensus approval (implicitly or explicitly) from SG21. Proposals 1, 2.1, and 2.2 reflect the status quo within [P2521R2] and [P2388R4].

Proposal 2.5 is not directly addressed by [P2521R2]. In [P2388R4], we need to remove one new form of undefined behavior addded in `[dc.correct.test]`:

- Remove "If the evaluation exits via a call to `longjmp` (17.13.3) the behavior is undefined".

Proposal 3.1 reflects the status quo in both papers. Proposals 3.2, 3.3, and 3.4 clarify an explicit choice and semantics for the controversial topic covered by [P2521R2] in section `{con.eff}`. The wording currently in [P2388R4] allows for zero evaluations or two evaluations of batches of predicates. Proposal 3, including its subproposals, would allow a strict superset of the implementations that [P2388R4] enables. In particular, the number of evaluations may be greater than two and consecutive allocations may be reordered somewhat more arbitrarily. A wording implementation of our proposals would likely remove the concept of a *test sequence* and instead define *consecutive correctness tests* along with a reordering rule to match the one described in Proposal 3.4. Complete wording for our proposed approach can be assembled at a later date and will likely involve some restructuring alongside a more thorough core wording review.

4.2 Deferred Changes

The rest of the proposals in this paper have not (yet) had consensus to adopt.

Proposal 2.3 is a change from the current MVP. In [P2521R2], sections `{pro.ord}` and `{pro.end}` would require updating to include our proposal's recommendation on handling a predicate that throws. In [P2388R4], modifications would be needed to `[dcl.correct.test]p4`:

- Remove "If the evaluation exits via an exception, `std::terminate()` is called."
- Change "where the evaluation of `P` returns `false`" to "where the evaluation of `P` exits via an exception or returns `false`."

Proposal 2.4 is not mentioned in either paper. In [P2388R4], add a recommended practice to `[dcl.correct.test]`:

- "Recommended practice: Implementations should consider treating undefined behavior in the evaluation of a contract predicate as a contract violation."

5 SG21 Discussion History

5.1 February 2023, Issaquah, WA

This paper was reviewed at the February 2023 WG21 meeting in Issaquah, Washington. A number of polls related to this paper were taken.

Two polls related to Proposal 3.

- Poll: Do we want to allow that a contract-checking predicate is evaluated zero or more times in eval_and_abort mode?

SF	F	N	A	SA
14	10	0	1	2

[3]

Result: Consensus.

- Poll: We want to adopt the rules on reordering the evaluation of contract-checking predicates as described in P2751R0 proposal 3.4, contingent on a clarification that this reordering can be equivalently formulated in terms of elision of evaluations inside repetitions of the given sequence of predicates.

SF	F	N	A	SA
12	5	3	1	0

Result: Consensus.

We interpret these results together as direction to move forward with Proposal 3 as is, with the clarifications to Proposal 3.4 present in this revision of this paper.

Three polls related to Proposal 2.3.

- Poll: In case the evaluation of a contract-checking predicate throws an exception, we want `std::terminate` to be called.

SF	F	N	A	SA
0	2	5	14	8

Result: Consensus against.

- Poll: In case the evaluation of a contract-checking predicate throws an exception, we want this to be treated as a contract violation.

SF	F	N	A	SA
7	7	3	3	7

Result: No consensus.

- Poll: In case the evaluation of a contract-checking predicate throws an exception, we want to propagate that exception out of the contract check.

SF	F	N	A	SA
5	5	4	3	10

Result: No consensus.

We interpret these results as leaving on the table a choice between Proposal 2.3 and propagating exceptions instead. Additional discussion on this point has been added to this revision of this paper, and we envision revisiting this result when discussing the forthcoming [P2811R0].

One poll related to Proposal 2.5.

[3] WG21 subgroup polls are taken of all participants in the room with potential responses of strongly favor (SF), weakly favor (F), neutral (N), weakly against (A), and strongly against (SA). See [SD4].

- Poll: When the evaluation of a contract-checking predicate exits the control flow other than by returning or throwing (for example, `abort` or `longjmp`), the behaviour is as if it were not in a contract-checking predicate.

SF	F	N	A	SA
19	11	0	1	0

Result: Consensus.

We take this result to indicate strong acceptance of Proposal 2.5.

One poll related to Proposal 2.4.

- Poll: We would like to non-normatively state the recommended practice that implementations should consider treating undefined behavior in the evaluation of a contract predicate as a contract violation.

SF	F	N	A	SA
5	11	5	2	8

Result: No consensus.

We interpret this result as no desire to move ahead with guidance on handling UB in contract predicates. We hope to in the future offer reasonable proposals to improve behavior related to this topic.

6 Conclusion

SG21 is seeking to produce an MVP that can evolve to meet the many use cases gathered for a C++ contract-checking facility. To achieve an MVP for C++26, the decisions we make today must maximize the flexibility for evolution while pinning down behaviors that are not points of conflicting demands from those use cases.

We have made three primary proposals with those goals in mind.

1 Predicates are C++ expressions and follow the normal C++ rules for expression evaluation.

2 A *checked* predicate fails to evaluate to `true`, something anomalous has happened; if control comes back to the point of evaluation of such a check, a *contract violation* occurs.

3 When determining whether a contract-checking annotation has been violated, its predicate may be evaluated zero or more times.

Defining clearly the precise semantics of evaluating the predicate for each individual contract-checking annotation builds a strong foundation for future evolution. Other interpretations and specifications seem to invite confusion.

Allowing elision and multiple evaluations of the predicate of each individual contract-checking annotation enables a robust range of implementation strategies for the MVP, including those that explore conforming extensions that satisfy many of the not-yet-met SG21 use cases. To pin down too early the precise number of evaluations allowed would block many of those strategies and evolutionary paths, with no clear benefit. Note that, given more implementation experience, we can

choose to remove the permission for zero or multiple evaluations without breaking existing software, yet enabling those evaluations later would run the risk of breaking (already questionable) software.

We hope that the discussion in this paper, along with the closely related but broader-focused [P2570R1], will facilitate consensus and thus help move the specification of our Contracts MVP toward a successful C++26 delivery.

Bibliography

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

[P2388R4] Andrzej Krzemieński and Gašper Ažman, "Minimum Contract Support: either No_eval or Eval_and_abort", 2021
http://wg21.link/P2388R4

[P2521R2] Andrzej Krzemieński, Gašper Ažman, Joshua Berne, Bronek Kozicki, Ryan McDougall, and Caleb Sunstrum, "Contract Support — Working Paper", 2022
http://wg21.link/P2521R2

[P2570R0] Andrzej Krzemieński, "On side effects in contract annotations", 2022
http://wg21.link/P2570R0

[P2570R1] Andrzej Krzemieński, "Contract predicates that are not predicates", 2022
http://wg21.link/P2570R1

[P2695R0] Timur Doumler and John Spicer, "A proposed plan for contracts in C++", 2022
http://wg21.link/P2695R0

[P2712R0] Joshua Berne, "Classification of Contract-Checking Predicates", 2022
http://wg21.link/P2712R0

[P2811R0] Joshua Berne, "Contract Violation Handlers", 2023
http://wg21.link/P2811R0

[SD4] "Practices and Procedures: The "How We Work" Cheat Sheet"
https://wg21.link/sd4