Revisiting side effects, elision, and duplication of contract predicate evaluations

Timur Doumler (papers@timur.audio)

Document #:	P3228R1
Date:	2024-05-21
Project:	Programming Language C++
Audience:	SG21, EWG

Abstract

The current Contracts MVP [P2900R6] allows side effects in contract predicates. It further specifies that evaluating a predicate when performing a contract check can be elided when the value of the predicate is known without evaluation, and that evaluations of contract assertions can further be duplicated, leading to observable side effects either not occurring, or occurring any number of times. During the design review of [P2900R6], concerns were raised whether this is really the correct design. It has been suggested that the status quo should be tightened to require exactly one evaluation, or at least introduce some upper bound on the allowed number of evaluations. This paper attempts to inform and structure this discussion. We consider what side effects are and which kinds of side effects are allowed in contract predicates in the Contracts MVP programming model. We then perform a thorough exploration of the available design space. We discuss the different conflicting design requirements and their motivation, list the possible solutions for how predicate evaluation could be specified, and analyse which of these solutions satisfy which design requirements.

Contents

1 Introduction . . . 3
  1.1 The Contracts MVP status quo . . . 3
  1.2 Existing practice in C++ . . . 3
  1.3 Existing practice in other programming languages . . . 3
  1.4 Previous Contracts proposals for C++ . . . 4
  1.5 Current discussion . . . 5
2 Side effects . . . 6
  2.1 Side effects in the core language . . . 6
  2.2 Side effects outside of the cone of evaluation . . . 6
  2.3 Why allow side effects in contract predicates? . . . 7
  2.4 Benign and destructive side effects . . . 8
  2.5 Independence of predicate evaluations . . . 9
  2.6 Side effects in assumed assertions . . . 10
3 Design requirements . . . 12
  3.1 Exactly one evaluation . . . 12

3.2 Deterministic upper bound on number of evaluations 14
3.3 Allow duplications 15
3.4 Allow elisions 16
3.5 Allow more than two repetitions 17
4 The solution space 17
5 Discussion 19
Bibliography 20

Revision history

Revision 1 (2024-05-02; Pre-St. Louis Mailing)

— Added new section "Side effects" and restructured other sections accordingly

— Updated material to reflect most recent SG21 discussions

Revision 0 (2024-04-16)

— Original version

1 Introduction

1.1 The Contracts MVP status quo

The current Contracts MVP [P2900R6], as forwarded by SG21 to EWG and LEWG for design review, allows the predicate p of a contract assertion to be a C++ expression that has side effects when evaluated, such as printing a message or modifying an object (with the caveat that modifying a local variable requires a `const_cast` due to [P2900R6]'s implicit `const`-ification rule).

When performing a contract check, i.e. evaluating the contract assertion with a checking semantic (*observe*, *enforce*, or *quick_enforce*), if the implementation can statically determine that evaluation of the predicate p will return a value B (rather than throwing an exception, terminating, `longjmp`-ing, etc.) and it can further statically determine the value B itself (`true` or `false`), [P2900R6] allows the implementation to elide the evaluation of the predicate expression and instead use B directly as the result of the contract check. In this case, any side effects from the evaluation of p will not occur, establishing an exception from the as-if-rule similar to copy elision. Note that such an elision of p can *never* lead to a call to the contract-violation handler being elided or to the program continuing execution past an enforced contract violation.

[P2900R6] further allows the same contract assertion to be evaluated twice, or even more times, with no specified upper bound on the number of evaluations. Inside a contract assertion sequence, i.e., a sequence of consecutive contract assertions (separated only by vacuous operations), any previously evaluated contract assertion may be evaluated again. This has the consequence that the following example program,

```
int i = 0;
void f() pre ((++i, true));

void g() {
  f();
  std::cout << i;
}
```

is allowed to print `0`, `1`, or any other integer value such as `42`.

1.2 Existing practice in C++

The only Contracts facility that exists in C++ today is the `assert` macro, and by extension, similar non-standard assertion macros. Contract checks using the `assert` macro can be either enabled or disabled with the macro `NDEBUG`. When contract checks are enabled, the predicate is evaluated exactly once, and any potential side effects are observed exactly once.

We are not aware of any implementation and deployment experience with elision and duplication of contract predicate evaluations as proposed in [P2900R6].

1.3 Existing practice in other programming languages

A number of programming languages offer a Contracts facility as a core language feature; we considered Ada, D, and Eiffel. All three languages follow the same model as the `assert` macro in C++. Contract checks can be either enabled or disabled, with varying granularity (for example, in Eiffel, this choice can be made per class). When contract checks are enabled, the predicate is evaluated exactly once, and any potential side effects are observed exactly once.

None of these languages attempts to prevent side effects in contract predicates. Notably, the D programming language actually has a facility to reject functions with side effects at compile time using the “pure” annotation, however D chose to not require contract predicates to be “pure”.

While prior art in other programming languages is certainly relevant, it should be considered with a grain of salt. First, [P2900R6] has been designed to enable use cases that none of the above-mentioned languages support, for example a “mixed mode” where the same contract assertion in the same function can have checking semantics in one translation unit and non-checking semantics in another translation unit in the same program. Second, contracts-based programming in these other languages has failed to become a widely established practice. Ada enjoys some success in certain safety-critical applications such as avionics, air traffic control, railways, banking, military and space technology, but is overall nowhere near as popular or as widely used across many different industries as C++ is. D and Eiffel are arguably niche languages that are no longer particularly relevant today.

1.4 Previous Contracts proposals for C++

Early Contracts proposals for C++ either did not consider the question of side effects ([N1613], [N1669]), adopted the model of macro `assert` in which side effects are treated as in any other C++ expression ([N3604], [N4378]), or stated that side effects “should not be allowed” without proposing any concrete mechanism for disallowing them ([N4110]).

Elision of predicate evaluations was first proposed in [N1669] and revisions thereof, to be allowed if the compiler can determine that the predicate is `true`, similar to the current specification in [P2900R6] (with the difference that the latter also allows elision if the compiler can determine that the predicate is `false`). Duplication was first considered in [P0247R0], which stated that “evaluating some checks twice seems tolerable and in general not avoidable”. Most other early Contracts proposals either did not mention elisions and duplication at all, or adopted the `assert` model where neither elisions nor duplications can happen.

In [P0542R5] (“C++2a Contracts”), as adopted into the C++20 Working Draft, the issue of contract check elision and duplication was side-stepped by specifying that evaluating a contract predicate that has observable side effects is undefined behaviour. For contract predicates without side effects, elision and duplication is unobservable under the as-if rule, and therefore does not require special treatment (except that duplication of an *observed* contract assertion may lead to multiple calls to the contract-violation handler — a situation that [P0542R5] did not consider).

[P1670R0] proposed to change the specification in the C++20 Working Draft to make predicates with side effects well-defined but allow elision of the predicate evaluation, and provided extensive motivation for allowing such elision. This paper was never adopted into the C++20 Working Draft because Contracts were removed from it before the paper was considered; however, post-C++20, this specification for side effect elision made its way into an early version of the Contracts MVP [P2388R4]. The latter is also the first proposal that explicitly allowed duplication of the evaluation (but not an arbitrary number of repetitions as [P2900R6] does).

This Contracts MVP specification went through several subsequent iterations. [P2388R0] clarified that it should be allowed to elide or duplicate all (as opposed to “some”) side effects of the evaluated predicate, as long as this does not affect the result of that evaluation. [P2388R3] relaxed this restriction and allowed eliding or duplicating side effects per subexpression of a predicate. [P2521R5] strengthened the rule again to “all or none” of the side effects of a predicate.

As work on the Contracts MVP progressed, SG21 spent an extensive amount of time discussing this topic. [P2751R1] proposed to loosen the [P2521R5] model: instead of just allowing elision or duplication, the number of evaluations of a *checked* contract assertion is deliberately made *unspecified*. Such an evaluation can therefore be elided, evaluated once, twice, or even more times, with no specified upper bound. [P2751R1] provided extensive motivation and use cases for this proposal. A counter-proposal, [P2756R0], instead proposed to strengthen the [P2521R5] model by specifying that the predicate of a *checked* contract assertion should be evaluated exactly once. SG21 ended up adopting [P2751R1] and rejecting [P2756R0] for the Contracts MVP (poll results see [P2751R1], Section 5).

The direction paper [P2680R1] proposed a different design direction whereby contract predicates with side effects outside of the cone of evaluation of the contract assertion would be ill-formed by default. However, even this paper provided an escape hatch in the form of so-called "relaxed" predicates that can exhibit side effects like any other C++ expressions, and therefore did not remove the need to specify the behaviour of predicates with side effects (an issue that the paper itself did not address). The design direction proposed by [P2680R1] ultimately failed to get consensus in SG21.

1.5 Current discussion

At the March 2024 WG21 meeting in Tokyo, [P2900R6] went through a first round of design review in EWG. During this design review, the following concerns about the current approach of allowing elisions as well as an arbitrary number of evaluations were raised:

— A contract assertion that will exhibit undefined behaviour after a number of repeated assertions (say, repeated, accumulating signed integer addition) can be considered to exhibit undefined behaviour always, as there is no specified upper bound on the number of evaluations;

— Low-latency and real-time systems require a deterministic upper bound on the runtime complexity of a contract assertion;

— For some safety-critical systems, a deterministic upper bound is not sufficient, and a guarantee is required that a checked assertion is evaluated a known, deterministic number of times.

In addition, an EWG guidance poll revealed that a significant number of people prefer that contract assertions should not be allowed to be evaluated more than once (see [D3197R0]):

EWG Poll 2024-03-20 (Tokyo)

P2900R6 Contracts should not be able to evaluate preconditions/postconditions/assertions more than once per invocation.

SF	F	N	A	SA
13	8	15	10	8

The paper [P3119R0] was written in response to EWG's review. It attempts to address the issues with undefined behaviour and the lack of a deterministic upper bound by introducing an implementation-defined upper bound, and recommending a value of 64. However, the paper does not attempt to address requests by EWG members that the number of evaluations be specified as exactly once or not more than once.

In light of this new situation, [D3197R0], the response paper to EWG's review, proposed to re-discuss the issue of contract check elision and duplication in SG21. This proposal gained SG21 consensus.

Given that the room seems to be split on this issue, the solution proposed in [P3119R0] may be insufficient and we may have to consider a more deterministic model for predicate evaluation if we wish to gain approval for the Contracts MVP by EWG, CWG, and the WG21 plenary. Alternatively, if SG21 confirms that the current specification (with or without the modification proposed in [P3119R0]) is the intended one, we need to strengthen the motivation for it to gain approval. The goal of the present paper is to provide a solid basis for understanding the tradeoffs and motivations of the different possible solutions in order to help SG21 and EWG make an informed decision.

2 Side effects

Before we can consider the design space for elision and duplication of contract predicate, we need to gain a solid understanding of the concept of *side effects* in contract predicates and how they are treated in the programming model of [P2900R6] and other programming models for contract assertions. In this section, we review the concepts that are essential for understanding the current discussion. For more in-depth discussion of predicates with side effects, see [P2570R2], [P2712R0], [P2751R1], and references therein.

2.1 Side effects in the core language

The contract predicate is a C++ expression contextually converted to `bool`. According to the C++ Standard ([intro.execution]), such an expression has *side effects* when evaluated if it does any of the following operations:

— reading an object designated by a `volatile` glvalue,

— modifying an object,

— calling a library I/O function,

— calling a function that does any of those operations.

Under this definition, most C++ expressions have side effects. C++ expressions without side effects are limited to reading non-`volatile` values and comparing those values, as well as performing value computations on prvalues.

No algorithm exists to statically prove whether an arbitrary C++ expression will have side effects when executed. It is therefore impossible to make contract predicates ill-formed based on whether the predicate expression will have side effects when executed. First, any such analysis is made impossible by the existence of opaque functions whose definitions are in a different translation unit. Second, even if all function definitions were transparent to the compiler, proving that an arbitrary C++ expression is side-effect-free would still be equivalent in complexity to the Halting problem.

It is possible to define a subset of C++ expressions for which the side-effect-free property *can* be proven, and C++ compilers usually have internal logic to make this determination for the purposes of various optimisations. However, we currently lack the specification tools to specify such a subset in the C++ Standard, and in addition this subset is very small and excludes all but the most trivial of allowed forms of expressions.

2.2 Side effects outside of the cone of evaluation

[P2680R1] considered the concept of a predicate that is side-effect free when seen from the outside of its *cone of evaluation*. In addition to side-effect-free predicates, this set includes predicates that during evaluation modify only (non-`volatile`) objects whose lifetime lies entirely within the evaluation of the predicate, thereby making any side effect unobservable after the evaluation of the predicate is complete. Consider:

```
int f(int i) {
  ++i;
  return i;
}

int g(int i)
  pre(f(i) > 0);
```

In the above example, the contract predicate in the declaration of `g` is not side-effect-free (because it calls another function `f` that modifies an object), but is side-effect-free outside of its cone of evaluation (because the modified object does not outlive the predicate evaluation). However, if we change `f` such that it modifies the passed-in object rather than a copy, for example if we make `f` take `i` by non-`const` reference, this property is no longer satisfied.

This concept has existing practice in C++: side effects are not allowed outside of the cone of constant evaluation, thereby preventing "stateful metaprogramming". However, this same approach cannot be applied to contract predicates to identify if they will have side effects. For constant evaluation, the prevention of side effects is applied as an expression is evaluated at compile time with a particular set of input values, and errors are reported only when there are inappropriate side effects for that particular evaluation with those particular inputs. To identify (at compile time) if a contract predicate would have side effects (at run time) would require that this determination be done for all possible inputs, and thus require identifying all possible valid flows of control through a function, a problem that is essentially equivalent to solving the Halting problem for all but the most trivial of allowed forms of expressions.

2.3 Why allow side effects in contract predicates?

As discussed in Sections 2.1 and 2.2, rejecting predicates with side effects outside of their cone of evaluation at compile time would require reducing the set of allowed predicates to a small subset of C++ expressions. For the Contracts MVP, SG21 considered this subset to be too small to be useful for any practical application of Contracts, as this would exclude any opaque function call and any expression not sufficiently simple that a compiler can prove it to be side-effect-free for any possible runtime input (see also [P2700R1]). In addition, we currently lack the specification tools to specify such a subset for the C++ Standard.

At the same time, making predicates with side effects outside of their cone of evaluation undefined behaviour like in [P0542R5] would be a particularly user-hostile choice that would undermine the safety of the proposed Contracts facility and go against the declared design goal of [P2900R6] to not intentionally add any new undefined behaviour to the C++ language (for a more detailed discussion, see [P1670R0]). Making them erroneous behaviour as defined in [P2795R5] would avoid the safety concerns, but would be similarly user-hostile, because many useful contract predicates could lead to unexpected termination of the program when checked, even if these predicates would evaluate to `true` and no contract violation would occur.

There are many use cases for predicates with side effects outside of their cone of evaluation. [P2712R0] provides a taxonomy of different kinds of such predicates, with many examples where side effects outside of their cone of evaluation of the predicate occur and are useful.

For example, while checking a contract assertion, the user might want to allocate memory to perform an algorithm that asserts some non-trivial property of a range, or lock and unlock a mutex to assert the value of a variable that can be accessed concurrently from multiple threads. Further, a contract predicate expression might call an function `f`, perhaps located in a different component of the program such as a third-party library. This library function is unaware that it is being used for a predicate evaluation by the program. It also does not advertise to clients whether it guarantees a lack of side effects when evaluated, and there is no mechanism in the C++ language for such an advertisement. The owner of `f` might add a statement to the implementation of `f` that has a side effect entirely unrelated to the rest of the program, such as logging for debugging purposes; doing so should "just work" and not break the program.

For all these reasons, the Contracts MVP chose to not make predicates with side effects outside of their cone of evaluation ill-formed, nor to make them undefined or erroneous behaviour. The only option left is that evaluating predicates with side effects must be well-formed and well-defined behaviour. Any solution for the problem of elision and duplication must ensure that this is the case.

2.4 Benign and destructive side effects

While we choose to allow predicates with side effects, it is helpful to distinguish between so-called *benign* and *destructive* side effects. The former should be explicitly supported, while the latter should be considered a bug (while still being well-formed and well-defined behaviour).

A side effect is benign if whether or not the side effect occurs (or how many times it occurs) does not have an impact on whether the program is *functionally correct*, i.e. whether the program satisfies its *plain-language contract*. Another way to say this is that a side effect is benign if it does not affect the *essential* behaviour of the program (a term defined in [P2053R1]). A third way to say this is that a side effect is benign if it is *lippincott-indiscernible* (a term defined in [P2461R1]). All three definitions are equivalent. A destructive side effect is a side effect that is not benign.

An interesting property of this definition is that it is unspecifiable in the C++ Standard whether a side effect is benign, as this depends on the plain-language contract of the program, which is not provable or even specifiable in the general case.[1] For example, logging to standard output while evaluating a contract predicate would be considered a benign side effect in many programs, but not if the standard output of the program is part of its contract (consider a command line utility such as `grep`). For another example, allocating a buffer to perform some algorithm on a range would be considered a benign side effect in many programs, but not if the program keeps track of the number of allocations and it is considered a bug in the program if this number exceeds a certain limit.

A corollary is that it is impossible to treat benign and destructive side effects differently within the purview of the C++ abstract machine, for example, by specifying that the latter is ill-formed, or undefined or erroneous behaviour, while the former is not.

While destructive side effects are well-defined behaviour, as a rule they should always be considered a bug. This is an important design principle of the Contracts MVP. Fundamentally, contract assertions should *test* the correctness of a program without *changing* the correctness of that program. If adding a contract assertion to an existing program would alter the behaviour such that the new program with the contract assertion added would become correct where the original program was incorrect, or incorrect where the existing program was correct, such contract assertions fail at their primary purpose of diagnosing bugs, and instead themselves introduce so-called Heisenbugs. Many design choices[2] in the Contracts MVP have been made specifically to satisfy this design principle.

[1]The plain-language contract of a program — i.e., which guarantees the program provides about its behaviour, and under what circumstances it can be considered functionally correct — can be specified through any combination of contract assertions, human-readable specification, or implicitly by convention and the developer's intent. In general, it is possible to specify only a *subset* of the plain-language contract with contract assertions, which means that contract assertions cannot be used to prove that a program is correct, only that it contains a contract violation. For a detailed discussion of the difference between plain-language contracts and contract assertions, see [P2900R6], Section 2.

[2]Examples include: making it ill-formed for a contract assertion to trigger an implicit lambda capture, thereby preventing situations where adding a contract assertion could change the properties of the closure type; defining that a contract assertion is always a core constant expression even if its predicate is not, thereby preventing situations where adding a contract assertion could change which overload is selected due to SFINAE; and making `contract_assert` a statement rather than an expression, such that `noexcept(contract_assert(expr)` is ill-formed, thereby preventing situations where adding a contract assertion could change the result of the `noexcept` operator.

2.5 Independence of predicate evaluations

From the above definition for benign and destructive side effects, we can conclude that if evaluating a contract assertion has a side effect that can change the result of evaluating a different contract assertion, then such a side effect is destructive.

Let us consider a program without contract assertions. If we now add N contract assertions to this program, and the side effects of evaluating one contract assertion could change the result evaluating a different contract assertion, this creates up to 2^N possible program states to consider, some of which may be correct while others are not. On the other hand, if all of the contract assertions have only benign side effects, adding these contract assertions does not add any additional states to the program, or at least not in a way relevant to the correctness and essential behaviour of the program.

In the programming model of the Contracts MVP, whether or not any particular contract assertion is checked (evaluated with a checking semantic) is implementation-defined. [P2900R6] does not mandate any particular mechanism for the selection of evaluation semantic: it may happen at compile time, link time, load time, or run time. The evaluation semantic may vary across translation units, vary across different contract assertions in the same translation unit or even in the same function, or even vary for subsequent evaluations of the *same* contract assertion.

In practice, the choice of semantics will most likely be controlled by a command-line option to the compiler, and [P2900R6] recommends to provide an “all assertions ignored” and “all assertions enforced” option, but other mechanisms of selection are equally conforming. For example, an implementation could provide a way to start a process with assertions ignored, and then later at an arbitrary point in time attach a debugger to this process and enable assertion checking from that point on.

It follows from this programming model that if a contract assertion has a side effect that can change the result of subsequent evaluations of the *same* contract assertion, then such a side effect is destructive.

Such assertions occur in practice. For example, an assertion may count the number of times a recursion or iteration occurs or a certain statement is executed, and report a contract violation if this number exceeds some fixed limit. Consider the following function from Clang:[3]

```
/// Return the MCSchedClassDesc for this instruction. Some SchedClasses require
/// evaluation of predicates that depend on instruction operands or flags.
const MCSchedClassDesc *TargetSchedModel::
resolveSchedClass(const MachineInstr *MI) const {

  // Get the definition's scheduling class descriptor from this machine model.
  unsigned SchedClass = MI->getDesc().getSchedClass();
  const MCSchedClassDesc *SCDesc = SchedModel.getSchedClassDesc(SchedClass);
  if (!SCDesc->isValid())
    return SCDesc;

#ifndef NDEBUG
  unsigned NIter = 0;
#endif
  while (SCDesc->isVariant()) {
    assert(++NIter < 6 && "Variants are nested deeper than the magic number");

    SchedClass = STI->resolveSchedClass(SchedClass, MI, this);
    SCDesc = SchedModel.getSchedClassDesc(SchedClass);
  }
  return SCDesc;
}
```

[3]Full source code: https://shorturl.at/BJOQ2

The contract assertion inside the `while` loop increments the counter `NIter` to keep track of the number of iterations; exceeding a certain number is considered a contract violation. The value of `NIter` is not observed anywhere in the program outside of this particular assertion, however the side effect of incrementing `NIter` affects the result of subsequent evaluations of the same assertion and is therefore a destructive side effect according to our definition.

When using macro `assert`, every check is either always on or always off,[4] depending on whether `NDEBUG` is defined. In other programming languages with language-level Contracts facilities such as Ada, D, and Eiffel, the same applies: every check is either always on or always off. With such a Contracts facility, the above assertion works as intended. In the Contracts MVP programming model however, the evaluation semantics of this assertion can vary freely, even at runtime, and therefore any evaluation of this assertion cannot rely on previous evaluations of the same assertion, and the above assertion is broken.

If the user attempts to port the code as written above to the Contracts MVP by replacing `assert` with `contract_assert`, it will not compile. First, the Contracts MVP does not offer a facility like `NDEBUG` for conditionally declaring a variable only when the associated contract assertion is checked. However, it is plausible that we would add a facility as a post-MVP extension to conditionally enable such code. Second, in the Contracts MVP, local variables such as `NIter` are implicitly `const` and the increment would not compile as it is modifying a local variable.

The user might be tempted to wrap `NIter` into a `const_cast`, or make it `static`, to circumvent the compile error. This would make the above code compile, but it would introduce a bug unless it is somehow statically guaranteed that the assertion is always checked or always unchecked, which is not something that can be expressed the Contracts MVP, *regardless* of whether elisions or duplications of *checked* assertions are allowed. Instead, the only way to make this assertion work in the Contracts MVP is to factor the destructive side effect out of the predicate:

```
while (SCDesc->isVariant()) {
  ++NIter; // increment outside of the assertion
  contract_assert(NIter < 6);
  // ...
}
```

We conclude that `contract_assert` cannot be used as a drop-in replacement for `assert` for any stateful predicate, regardless of whether elisions or duplications of *checked* assertions are allowed or exactly one evaluation of the predicate of a *checked* assertion is guaranteed.

2.6 Side effects in assumed assertions

If we ever want to add the *assume* semantic to C++ Contracts,[5] or should any platform choose to provide one,[6] any predicate with a destructive side effect will immediately be even worse, introducing assumptions possibly unrelated to the correctness of the program. Consider the following example:

[4]Compiling the same macro-based assertion with checks on in one translation unit and with checks off in another translation unit is in general a violation of the ODR rule and not supported by the C++ Standard.

[5]The *assume* semantic has been deliberately left out of [P2900R6] to limit the scope of the MVP, but is a planned post-MVP extension. Like the *ignore* semantic, the *assume* semantic is a non-checking semantic, i.e., the predicate is not evaluated. However, unlike the *ignore* semantic, the *assume* semantic gives the compiler the permission to assume that the predicate would be *true* if evaluated, and optimise the program based on this assumption; if the predicate would not be `true` if evaluated, the behaviour is undefined. In other words, `contract_assert(X)`, when evaluated with the *assume* semantic, is equivalent to `[[assume(X)]]`. For more information about assumptions, and in particular the meaning of side effects in an assumed predicate, see [P1774R8]. For additional motivation why the *assume* semantic should be part of a C++ Contracts facility, see [P3100R0].

[6]This is possible today by providing a build mode where an assertion macro resolves to `[[assume(X)]]`, or for pre-C++23 compilers, to one of the appropriate compiler built-ins that offer the same functionality.

```
struct List {
  int d_data;   // index of node in List, starting with 0 for the head node
  List* d_next;
};

void f(List* l) {
//#ifndef NDEBUG
  int index = 0;
//#endif
  while (l) {
    contract_assert(++index == l->d_data);
    process(l->d_data);
    l = l->d_next;
  }
}
```

The assertion above is another example of a destructive side effect that affects subsequent evaluations of the same contract assertion. In this example, the destructive side effect has particularly unfortunate consequences when the contract assertion is evaluated with the *assume* semantic. In this case, the compiler will know that `index` is never modified, so the line

```
contract_assert(++index == l->d_data);
```

becomes equivalent to

```
[[assume(1 == l->d_data)]];
```

which in turn allows the compiler to transform the function `f` into the following:

```
void f(List* l) {
  while (l) {
    process(1);
    l = l->d_next;
  }
}
```

This transformation silently breaks the program and results in the wrong data being processed, even though the program is correct apart from the destructive side effect in the assertion. Just like before, this bug can be fixed by factoring the increment out of the predicate, which will make the assumption work as intended:

```
void f(List* l) {
//#ifndef NDEBUG
  int index = 0;
//#endif
  while (l) {
  //#ifndef NDEBUG
    ++index;
  //#endif
    contract_assert(index == l->d_data);
    process(l->d_data);
    l = l->d_next;
  }
}
++index;
contract_assert(index == l->d_data);
```

This kind of breakage is another reason why in the programming model of the Contracts MVP, side effects that maintain state within a contract assertion are considered destructive and are not supported, even though we cannot make them ill-formed or undefined or erroneous behaviour; this property holds regardless of whether elisions or duplications of *checked* assertions are allowed or exactly one evaluation of the predicate of a *checked* assertion is guaranteed.

3 Design requirements

Now that we have established a framework to reason about side effects in contract assertions, we can return to the main subject of this paper: reconsidering whether such side effects should be allowed to be elided, duplicated, or occur an arbitrary number of times when a contract assertion is evaluated, as currently proposed in [P2900R6], or whether we should introduce some upper bound to the number of evaluations or perhaps specify that the side effects should occur at most once or exactly once when a contract assertion is evaluated. In this section, we summarise the different, partially conflicting design requirements that have motivated different proposals in this space, and discuss the known motivating use cases for all of these requirements.

For the following discussion, we do not consider what happens when a contract assertion is evaluated with the *ignore* semantic, as such an evaluation never evaluates the predicate, but only what should happen when a contract assertion is evaluated with a *checking* semantic. This is an important distinction: on the one hand, a predicate might not get evaluated because the contract assertion is evaluated with the *ignore* semantic, and on the other hand, a predicate might not get evaluated even if the contract assertion is evaluated with a *checking* semantic in case elisions are allowed (see Section 3.4) and such an elision has been performed. When we speak about elisions in this paper, we always mean the latter case and never the former.

Further, we consider only contract predicates that have side effects (as defined in Section 2.1) when evaluated. For "pure" contract predicates with no side effects, according to the as-if rule, it is unobservable whether they are evaluated zero, one, or multiple times, as long as the compiler has correctly determined whether the result of such an evaluation would be `true` or `false`, and therefore no specification for elision or duplication of of "pure" contract predicates is necessary or even possible within the C++ abstract machine.

Finally, we do not consider contract assertions whose evaluation may end up being elided because of undefined behaviour occurring either during evaluation of the predicate itself or elsewhere in the program (see [P2900R6], Section 3.6.4). For the following discussion, we consider only programs that would have well-defined behaviour if a given predicate was evaluated once.

3.1 Exactly one evaluation

Guaranteeing a deterministic number of predicate evaluations is a desirable property for a number of use cases. This requirement effectively translates to guaranteeing exactly one evaluation, as it does not seem useful to specify, for example, that each contract assertion is always evaluated twice.

From a language specification perspective, a deterministic number of evaluations is required if we wish to minimise the amount of implementation-defined and unspecified behaviour added to the C++ language by the Contracts MVP. This has been brought up as a significant concern in EWG.

Further, deterministic behaviour is a requirement in some safety-critical systems, which may be unable to use Contracts at all unless this requirement is satisfied. To our knowledge, general coding guidelines for safety-critical systems such as MISRA typically do not place deterministic requirements on aspects of an algorithm such as the number of evaluations, but only requirements on a deterministic upper bound or "worst-case behaviour", which is addressed in Section 3.2. However, more stringent requirements on deterministic behaviour are required for certain use cases.

For example, in systems that perform work in between regularly timed callbacks on a real-time thread, need to maximise the amount of work done in these intervals, and need to provide a guarantee that the work is completed before the next callback occurs, it is necessary to be able to reason about the exact operations that a C++ statement may perform when executed. Note that requiring a deterministic observable behaviour with regard to side effects does not imply a deterministic execution time, or a deterministic sequence of CPU instructions. These aspects of a program's behaviour, while observable and sometimes important, are outside of the purview of the C++ abstract machine and are not side effects in the core language sense (see Section 2.1). Requiring a deterministic observable behaviour is therefore a necessary, but not a sufficient requirement for such real-time use cases. Note that deterministic observable behaviour could also be achieved for these cases by banning contract predicates with side effects in the coding guidelines.

Independently of any safety or real-time requirements, it seems desirable to be able to reason about which exact statements are actually being executed when a contract predicate is checked. Benign side effects on contract predicates include operations such as locking and unlocking a mutex, allocating memory, or utilising some other resource in order to perform the contract check. It seems desirable to be able to reason about how many times such a contract check will attempt to acquire a lock, or allocate memory, rather than having these operations occur an unspecified, non-deterministic amount of times when contract checks are enabled, even if this is not part of the program's plain-language contract (see Section 2.4). In particular, when debugging misbehaving code with contract assertions enabled, diagnosing the bug (either in the program or in the contract predicate itself) can be more difficult if the developer cannot reason about which operations are being performed by the program.

In order to reason about the operations being performed, it would arguably be helpful if the act of performing a contract check could be mapped to equivalent C++ code. [P2900R6] Section 3.5.8 proposes the following mapping:

```
evaluation_semantic _semantic = __current_semantic(); // semantic may be determined at
if (evaluation_semantic::ignore == _semantic) {       // compile time, link time, run time...
  // non-checking semantic - do nothing
}
else if (evaluation_semantic::observe       == _semantic
      || evaluation_semantic::enforce       == _semantic
      || evaluation_semantic::quick_enforce == _semantic) {
  // checking semantic - determine the value of the predicate
  bool _violation;
  try {
    _violation = __check_predicate(X);    // no guarantee whether/how many times X is evaluated
  }
  // handle violation
}
```

If we instead guarantee that a *checked* contract assertion evaluates its predicate exactly once, the mapping becomes:

```
  // ...
  // checking semantic - determine the value of the predicate
  bool _violation;
  try {
    _violation = !X;    // evaluates X exactly once
  }
  // handle violation
}
```

[P2756R0] argues that such a mapping that evaluates the predicate exactly once is the most simple, intuitive, and easy to reason about solution. Guaranteeing one evaluation is also the only solution that follows existing practice of the `assert` macro, other C++ assertion macros, and other programming languages with a Contracts facility (Ada, D, Eiffel, etc.), all of which evaluate the predicate of a *checked* assertion exactly once, and guarantee that any potential side effects are observed exactly once.[7]

[7]Note that even if we guarantee exactly one evaluation, there are other aspects of how a *checked* contract assertion behaves in the Contracts MVP that differs from the behaviour of these existing facilities, such as how an exception thrown from the predicate evaluation is handled.

While implementations can certainly provide a conforming "exactly once" mode with any of the solutions discussed in this paper, only the solution guaranteeing exactly one evaluation allows the user to *portably* rely on the side effects of a predicate evaluation, and in particular *benign* side effects that do not affect the correctness of the predicate itself or the surrounding program.[8] This is the behaviour that users are familiar with. Most users will likely intuitively expect this behaviour when they start using the new C++ Contracts facility, leading to surprise and frustration when the actual behaviour is subtly different.

As discussed in section 2.5, stateful predicates with a destructive side effect affecting subsequent evaluations of the same assertion, which work with macro `assert`, would stop working if `assert` is changed to `contract_assert`, or to `pre` or `post` and moved outside of the function body, without further changes. Most such cases would not compile if `assert` is changed to `contract_assert` due to implicit constification and the lack of a facility similar to `NDEBUG`. However, some such cases might exist in production code, and work as intended, that *would* compile if `assert` is changed to `contract_assert`, in particular if the variable modified inside the predicate is not a local variable, and its declaration is not wrapped in an `#ifdef NDEBUG` block.

In such cases, changing `assert` to `contract_assert` is a transformation that can silently break the program and potentially introduce undefined behaviour without any diagnostic message. The possibility of such breakage due to turning deterministic into non-deterministic evaluation is arguably a particularly user-hostile way to break users' assumptions about how assertions work; the mere existence of this possibility, however theoretical, could significantly hamper the adoption of C++ Contracts.

On the other hand, as discussed in section 2.5, guaranteeing exactly one evaluation when checked would not actually be sufficient to make the transformation from `assert` to `contract_assert` work for any such stateful assertions. In addition, we would have to roll back the adoption of [P2877R0] and revert the flexible evaluation semantics model to static build modes where either all assertions are on or all assertions are off, with unspecified semantics for mixed mode, as in [P0542R5] ("C++2a Contracts") and [P2388R4] ("Minimum Contract Support: either *No_eval* or *Eval_and_abort*").

3.2 Deterministic upper bound on number of evaluations

Many low-latency and real-time systems such as video games and audio processing software do not necessarily require full deterministic behaviour, but do require at least a deterministic upper bound on the runtime complexity of a contract assertion because such systems need to ensure that a deadline for computing a result such as a video frame or an audio buffer is always met. Similar "worst case" requirements are also common in safety-critical systems and mandated by coding guidelines such as MISRA. Such systems may be unable to use Contracts at all unless this requirement is satisfied.

In addition, without a deterministic upper bound on the number of evaluations, a contract assertion that will exhibit undefined behaviour after a number of repeated assertions (for example, accumulating signed integer addition) can be considered to exhibit undefined behaviour always. It is therefore conforming for a particularly hostile compiler to treat such contract assertions as unreachable code. Both of these issues are discussed in more detail in [P3119R0].

There are two ways to specify such an upper bound: either normatively specify a concrete number in the Standard (for example, "at most two evaluations"), or merely specify that an implementation has to define *some* deterministic upper bound but leave the actual number unspecified. The latter is proposed by [P3119R0].

[8] Of course, one could argue that if the predicate evaluation has a side effect that the user wishes to rely upon, such a side effect then effectively becomes a part of the plain-language contract of the program (or in other words, a part of the *essential* behaviour of the program), and therefore is a *destructive* side effect that the Contracts MVP does not support (see Section 2.4).

For consumer-facing, cross-platform applications, which often need to support different compilers, a normatively specified upper bound seems preferable to an implementation-defined one, because the latter could change across compilers or even across different versions of the same compiler, making it harder to reason about the code and the guarantees it provides.

Regardless of whether such an upper bound on the number of evaluations is normatively specified or implementation-defined, there are use cases where neither is a sufficiently strong guarantee, and the stricter "deterministic number of evaluations" guarantee is required instead (see Section 3.1).

3.3 Allow duplications

The main motivation for allowing duplication of predicate evaluations is to allow the implementation to perform caller-side checking[9] while preserving ABI compatibility. To our best knowledge, caller-side checking cannot be implemented without either allowing duplication or requiring ABI changes. When considering such changes, preserving ABI compatibility is crucial for allowing user applications to link against shared libraries such as system libraries: regardless of the shape of the application's dependency graph, and which components in that graph are compiled with or without Contracts support, the program must still link and execute correctly.

Consider a shared library that has function contract assertions on its function declarations, and an application that uses this library by compiling against a header and then dynamically linking against a shared library. The provider of the shared library may ship the compiled library binary with callee-side checks disabled or enabled; the developer of the application may not have any control over this.

The developer of the application should have the choice of compiling the application with caller-side checks either enabled or disabled (the latter resulting in the caller-side-checkable subset of the library checks being checked, which can be useful to diagnose problems). Either version of the application should be compatible with either version of the shared library, without having to recompile and re-link[10] when switching between a shared library that has callee-side checking enabled and one that has them disabled. When linking an application having caller-side checks enabled and a library having callee-side checks enabled, running the program will result in some function contract assertions in that library being checked twice.

We can consider implementation strategies that would allow this use case while also guaranteeing that contract checks be evaluated exactly once (or not more than once). However, this requires that the implementation provides a way for function calls compiled with caller-side checking to skip callee-side checking and thereby avoid a duplicated check. Such strategies come with tradeoffs. One possible strategy would be to compile the library binary such that each function with precondition or postcondition assertions has two entry points, one that performs the callee-side check and one

[9]Note that only a subset of contract checks can be implemented caller-side. Some contract checks can be implemented only callee-side. This is the case for indirect function calls, for example, through a function pointer or a facility like `std::function`. In such scenarios, a callee-side check can be generated only if the compiler front-end sees the function contract assertion related to the function call, i.e. it knows which function will be called and can see the declaration of that function. It is also the case for checking postconditions in the Microsoft ABI, as this ABI performs argument destruction callee-side and postcondition checks are guaranteed to happen before argument destruction. Note further that, if we adopt the proposed design in [P3097R0] and [P3165R0] for supporting function contract assertions on virtual functions, the reverse will also become true: only a subset of contract checks can be implemented callee-side. In particular, checks for function contract assertions of the statically called function in a virtual function call can be generated only caller-side as the statically called function is unknown to the callee.

[10]Without this requirement, guaranteeing that contract checks be evaluated exactly once would be relatively easy: if in the library, checks are disabled, in the application we call a thunk wrapper, which checks precondition assertions and then calls the function; otherwise, we just call the function. The library's build never affects whether the preconditions are checked, only the application build does. However, in this implementation model, we cannot flip precondition checks on and off by just building the library differently and then running an unmodified application with it. Instead, we would need to recompile the application, which might be prohibitively slow or impossible.

that does not, thereby allowing function calls compiled with caller-side checking to choose the latter entry point. However, this can significantly increase the amount of symbols in the binary, and requires a change to the ABI. Another possibility is to compile the library such that the choice whether to perform callee-side checks can be made dynamically at runtime, but this would incur additional runtime overhead and again require a change to the ABI.

For some users, the tradeoffs of either strategy will be unacceptable: if the ABI change is not backwards-compatible, this makes the strategy undeployable, and in a world where some C++ applications require tens of gigabytes of memory to link due to the sheer amount of symbols, there is a strong incentive to avoid adding more symbols. Further, there is currently no implementation experience for either strategy[11]. Avoiding these tradeoffs however requires us to either explicitly allow duplication of predicate evaluations, or to place this use case outside of the scope of the C++ Standard and treat support for it as a non-conforming vendor extension, as we do for example for `-fno-exceptions`.

3.4 Allow elisions

Allowing elisions of predicate evaluations is discussed in more detail [P1670R0] and [P2751R1]. One reason to allow elisions is that this would make it clear that side effects in contract predicates cannot be relied upon, thereby dissuading developers from writing contract assertions with destructive side effects (see Section 2.5). This comes along with an increased likelihood of benefitting from assumptions (see Section 2.6).

Another reason is that allowing elisions enables reasoning about contract assertions at a higher level rather than merely in terms of their immediate runtime effects. Consider the following example:

```
int f(int i)
  pre (i > 0);   // opaque function

int g(int i)
  pre (i > 0) {
  return f(i) - 1;
}

int main() {
  int i;
  std::cin >> i;
  return g(i);
}
```

In the example above, the preconditions of `g` and `f` constitute a sequence of contract assertions, creating a situation where checks can be elided according to the specification in [P2900R6]. At compile time, the implementation can perform symbolic evaluation to prove that, if the precondition check in `g` succeeds, the precondition check in `f` must succeed as well, and elide the second check, even though the value of `i` is not known at compile time. When the program runs, only the precondition of `g()` is checked; nevertheless, we can rely on the precondition of `f()` always being `true`.

In a large code base that is rigorously annotated with contract assertions, being able to elide checks from sequences in this fashion — even in the presence of opaque function calls — has the potential to greatly reduce the amount of runtime overhead that these checks require, while still having a program at is rigorously checked, because many of the contract assertions can be statically proven to hold. However, such symbolic evaluation is possible only if it can rely on repeated evaluations

[11]Caller-side checking itself does not have implementation experience either, but it is arguably somewhat less theoretical, because compilers know how to parse the function contract specifiers of a function declaration and rewrite a function call `f()` into a caller-side checked function call such as `(pre_check() ? f() : abort())`.

of the same predicate (or an equivalent predicate) anywhere in the sequence to result in the same value. A predicate satisfies this criterion when it is free of destructive side effects.

Contract predicates that have destructive side effects, such as in the `NIter` example in Section 3.1, make such symbolic evaluations and correctness proofs impossible. At the same time, it is in general impossible to prove that a C++ expression does not have side effects; it is further impossible for a C++ compiler to distinguish between benign and destructive side effects. Therefore, enabling symbolic evaluation and correctness proofs as described above requires treating predicates *as if* they had no side effects, and perform elisions under this assumption. This approach invariably leads to the possibility that any observable side effects (including benign ones) may be elided by the compiler.

Note that treating predicates as if they had no side effects does not imply that a contract predicate that has side effects leads to undefined behaviour, the way it did in [P0542R5] ("C++2a Contracts"). In [P2900R6], non-complying predicates can lead to elisions of observable side effects, but not to any other nondeterministic consequences such as unbounded undefined behaviour and time-travel optimisations.

3.5 Allow more than two repetitions

Supporting caller-side checking while preserving ABI compatibility may require allowing evaluation to occur twice, but we are not aware of any scenario where a particular configuration would require allowing evaluation to occur three times or more. It seems therefore that it would be enough to allow evaluation to occur up to twice.

However, [P3119R0] describes a use case for allowing an unspecified number of evaluations larger than 2. We discussed in Section 2.5 and 2.6 how the Contracts MVP programming model requires contract predicates to be free of destructive side effects. Failure to do so is a bug, but not ill-formed or undefined or erroneous behaviour. Allowing an unspecified number of evaluations, similarly to allowing elisions, would make it clear that side effects in contract predicates cannot be relied upon, thereby dissuading developers from writing contract assertions with destructive side effects (see Section 2.5). Beyond this, allowing an unspecified number of evaluations enables a strategy to identify such buggy assertions, described in [P3119R0]). This strategy consists of running a test where each assertion is evaluated an arbitrary number of times N, where N can be passed via a compiler flag. For sufficiently high N, breakage due to destructive side effects will be more easily observable in the form of failing tests and/or altered program behaviour.

If the Contracts MVP allows more than two repetitions, such a compiler flag can be implemented in a conforming way. It is desirable for this flag to be conforming, because otherwise it becomes more difficult to argue that an assertion with destructive side effects that breaks in such a test mode is actually a bug, and incentivise a library author to fix such a bug.

4 The solution space

Now that we have discussed different design requirements for allowing or disallowing elisions, duplications, or arbitrary repetitions of contract predicate evaluations, we can explore the available solution space.

This paper does not propose any concrete changes to the Contracts MVP. Instead, we list all plausible specification strategies for how many times the predicate of a checked contract assertion may be evaluated, and provide an analysis of which solutions satisfy which design requirements. The intent of the paper is to add some structure to the discussion and to highlight the engineering tradeoffs that each solution involves, to help reach consensus on the best solution in SG21 and EWG.

Whether elisions are allowed is orthogonal to the other concerns, so we can split the solution space into solutions that allow elisions and solutions that do not. The four known solutions that allow elisions are as follows (all four have been proposed at some point):

A_0. At most once, i.e. evaluation may be elided but not duplicated ([P1670R0], [D3197R0]).

B_0. At most twice ([P2521R5]), i.e. evaluation may be both elided and duplicated.

C_0. An unspecified number of times with an implementation-defined upper bound N ([P3119R0]).

D_0. An unspecified number of times with no upper bound (status quo, [P2900R6]).

Further, we can construct four solutions that are analogous to the above but do not allow elisions (only one of those, A_1, has been formally proposed):

A_1. Exactly once ([P2756R0]).

B_1. Once or twice, i.e. evaluation may be duplicated but not elided.

C_1. An unspecified number of times but at least once with implementation-defined upper bound N.

D_1. An unspecified number of times but at least once with no upper bound.

Remember that we consider only evaluations of contract assertions with *checked* evaluation semantics (`observe`, `enforce`, or `quick_enforce`), as evaluations with *unchecked* evaluation semantics (`ignore`) always evaluate the predicate zero times and we do not consider removing `ignore` from the MVP or changing how it is specified. Remember further that we consider only evaluation of predicates with observable side effects. For predicates with no observable side effects, neither elision nor duplication are observable under the as-if rule, and therefore all of the above solutions are equivalent.

A few more solutions than the ones listed above are theoretically possible, such as requiring a deterministic number of evaluations that is not once (e.g. "every contract assertion must always be evaluated twice"), or requiring a normatively specified (as opposed to implementation-defined) upper bound larger than two. We do not consider these solutions here because we are not aware of any benefits these might have over the ones listed above.

Further, there has been a suggestion that we could allow duplications and simultaneously ensure a deterministic amount of evaluations with a solution that says that the predicate must be evaluated exactly N times, where N is implementation-defined. An implementation may then say that, for example, N is 1 if the function called is in a statically linked library, but 2 if the library is linked dynamically. However, a conforming implementation may satisfy such a specification by simply saying that N is any number between 0 and 64 (for example), therefore such a solution is equivalent to solution C_0.

Now that we enumerated all the requirements and possible solutions, we can create a decision matrix that visualises which possible solutions satisfy which design requirements (Table 1). The order of the design requirements in this matrix does not imply a ranking by importance; we are not attempting to perform such a ranking in this paper.

Design requirement Nr. of evaluations:	A_0 $0-1$	A_1 1	B_0 $0-2$	B_1 $1-2$	C_0 $0-N$	C_1 $1-N$	D_0 $0-\infty$	D_1 $1-\infty$
Exactly one evaluation	❌	✅	❌	❌	❌	❌	❌	❌
Normative upper bound	✅	✅	✅	✅	❌	❌	❌	❌
Some deterministic upper bound	✅	✅	✅	✅	✅	✅	❌	❌
Allow duplications	❌	❌	✅	✅	✅	✅	✅	✅
Allow elisions	✅	❌	✅	❌	✅	❌	✅	❌
Allow more than two repetitions	❌	❌	❌	❌	✅	✅	✅	✅

Table 1: Decision matrix for the number of contract predicate evaluations.

5 Discussion

The decision matrix in Table 1 reveals that it is impossible to satisfy all design requirements. Of the six design requirements presented, all of which can be well-motivated as we have seen above, at most four design requirements can be satisfied simultaneously by any solution considered in this paper. Choosing a solution will therefore require making a tradeoff between contradicting requirements.

For example, SG21 and EWG will need to decide whether it is more important to have a normatively specified deterministic number of evaluations (which effectively translates to "exactly one evaluation when checked" — see Section 3.1), or whether it is more important to allow duplications of evaluations in order to support caller-side checking without an ABI break (Section 3.3), as satisfying both of these requirements is impossible.

In case the latter is deemed to be the better tradeoff, four more decisions need to be made: whether to allow elisions (Section 3.4), whether to allow more than two repetitions (Section 3.5), whether to require a deterministic upper bound on those repetitions (Section 3.2), and whether such a deterministic upper bound should be normative or implementation-defined.

If SG21 and EWG can get consensus on the above questions, it becomes unambiguous which concrete solution to choose for the specification.

Notably, it is possible to choose one solution for `pre` and `post` and a different solution for `contract_assert`. One such proposal is [P3257R0]. It proposes to retain the status quo (solution D_0) for `pre` and `post`, but to adopt solution A_1 for `contract_assert`. The latter has different tradeoffs than the former. In particular, the need to support caller-side checking does not exist for `contract_assert`, because assertion statements can appear only inside the function body and are therefore always checked callee-side. Instead, there is a new design requirement: it might be desirable to pick the same solution for `contract_assert` as we did for `pre` and `post` so that the three assertion kinds provided in the Contracts MVP behave in a consistent fashion.

That said, the consistency requirement should be weighed against the other requirements. For example, SG21 and EWG can decide that having a deterministic number of evaluations is particularly important for `contract_assert`, and less so for `pre` and `post`, and that further, having a deterministic number of evaluations for `contract_assert` is more important than consistency between the three assertion kinds.

However, breaking the consistency between the three assertion kinds would have implications for teachability, complexity of the language, etc. It would also widen the gap[12] between the use of

[12] `pre` and `post` on the one hand, and `contract_assert` on the other hand, are already not fully consistent. One difference is that `contract_assert` does not allow referring to the return object directly. If the return object is an rvalue, referring to it requires taking an extra copy, which might not be possible if the return type is non-copyable. Another difference is that `pre` and `post` are evaluated outside of a function-try block, while `contract_assert` is

`pre` and the use of `contract_assert` at the start of a function, or the use of `post` and the use of `contract_assert` before returning. Lowering `pre` and `post` into the function body in this fashion is the only mechanism offered by [P2900R6] to insulate precondition and postcondition checks from client translation units when the developer considers them an implementation detail. Further, using `contract_assert` to specify preconditions and postconditions is an effective technique to check the precondition of a function that does not itself declare those preconditions, and establish the postcondition of a function that does not itself declare those postconditions, i.e. to add "missing" precondition and postcondition assertions caller-side.

Another aspect of choosing the right solution is the question of forward-compatibility. Choosing a more loosely specified solution now (with D_0 being the most loosely specified) does not preclude strengthening the specification to a more strictly specified solution later (with A_1 being the most strictly specified), while an evolution in the other direction would not be possible without breaking changes.

It is our hope that the analysis provided in this paper will be helpful for SG21 and EWG to choose the best tradeoffs for the number of evaluations problem, and to arrive at a consensus solution for all three assertion kinds in the Contracts MVP that will be part of the Contracts facility eventually incorporated into the C++ standard.

Acknowledgements

Thanks to John Spicer, Joshua Berne, Ville Voutilainen, Gašper Ažman, Andreas Weis, and Anthony Williams for the illuminating discussions that led to this paper. Thanks to Joshua Berne for reviewing this paper and providing the `List` code example. Thanks to Jonas Persson, Andrew Tomazos, Jens Maurer, Andrzej Krzemieński, John Lakos, Ran Regev, and Greg Marr for their helpful comments. Thanks to everyone who participated in the discussion on the SG21 reflector which this paper attempts to capture.

Bibliography

[D3197R0] Timur Doumler and John Spicer. A response to the Tokyo EWG polls on the Contracts MVP (P2900R6). https://wg21.link/d3197r0, 2024-04-04.

[N1613] Thorsten Ottosen. Proposal to add Design by Contract to C++. https://wg21.link/n1613, 2004-03-29.

[N1669] Thorsten Ottosen. Proposal to add Contract Programming to C++ (revision 1). https://wg21.link/n1669, 2004-09-10.

[N3604] John Lakos and Alexei Zakharov. Centralized Defensive-Programming Support for Narrow Contracts. https://wg21.link/n3604, 2013-03-08.

[N4110] J. Daniel Garcia. Exploring the design space of contract specifications for C++. https://wg21.link/n4110, 2014-07-06.

[N4378] John Lakos, Nathan Myers, Alexei Zakharov, and Alexander Beels. Language Support for Contract Assertions (Revision 10). https://wg21.link/n4378, 2015-02-08.

evaluated inside, so the behaviour of a throwing violation handler will differ. A third difference is that in a constructor, `pre` is evaluated before the member initialiser list, while `contract_assert` as the first statement in the function body is evaluated after, which changes the meaning of the predicate. Despite these differences, in the most common case — moving a `pre` on a free function or regular member function into the function body with a `contract_assert` as the first statement — both will behave consistently with the current specification in [P2900R6].

[P0247R0] Nathan Myers. Criteria for Contract Support. https://wg21.link/p0247, 2016-02-12.

[P0542R5] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, and B. Stroustrup. Support for contract based programming in C++. https://wg21.link/p0542r5, 2018-06-08.

[P1670R0] Alisdair Meredith and Joshua Berne. Side Effects of Checked Contracts and Predicate Elision. https://wg21.link/p1670r0, 2019-06-06.

[P1774R8] Timur Doumler. Portable assumptions. https://wg21.link/p1774r8, 2022-06-14.

[P2053R1] Rostislav Khlebnikov and John Lakos. Defensive Checking Versus Input Validation. https://wg21.link/p2053r1, 2020-08-14.

[P2388R0] Andrzej Krzemieński and Gašper Ažman. Abort-only contract support. https://wg21.link/p2388r0, 2021-06-15.

[P2388R3] Andrzej Krzemieński and Gašper Ažman. Minimum Contract Support: either *No_eval* or *Eval_and_abort* contracts. https://wg21.link/p2388r3, 2021-10-13.

[P2388R4] Andrzej Krzemieński and Gašper Ažman. Minimum Contract Support: either *No_eval* or *Eval_and_abort* contracts. https://wg21.link/p2388r3, 2021-11-15.

[P2461R1] Gašper Ažman, Caleb Sunstrum, and Bronek Kozicki. Closure-Based Syntax for Contracts. https://wg21.link//p2461r1, 2021-11-15.

[P2521R5] Andrzej Krzemieński, Gašper Ažman, Joshua Berne, Bronek Kozicki, Ryan McDougall, and Caleb Sunstrum. Contract support – Record of SG21 consensus. https://wg21.link/p2521r5, 2023-08-15.

[P2570R2] Andrzej Krzemieński. Contract predicates that are not predicates. https://wg21.link/p2570r2, 2023-01-14.

[P2680R1] Gabriel Dos Reis. Contracts for C++: Prioritizing Safety. https://wg21.link/p2680r1, 2022-12-15.

[P2700R1] Timur Doumler, Andrzej Krzemieński, John Lakos, Joshua Berne, Brian Bi, Peter Brett, Oliver Rosten, and Herb Sutter. CQuestions on P2680 "Contracts for C++: Prioritizing Safety". https://wg21.link/p2700r1, 2022-12-17.

[P2712R0] Joshua Berne. Classification of Contract-Checking Predicates. https://wg21.link/p2712r0, 2022-11-13.

[P2751R1] Joshua Berne. Evaluation of *Checked* Contract-Checking Annotations. https://wg21.link/p2751r1, 2023-02-14.

[P2756R0] Andrew Tomazos. Proposal of Simple Contract Side Effect Semantics. https://wg21.link/p2756r0, 2022-12-31.

[P2795R5] Thomas Köppe. Erroneous behaviour for uninitialized reads. https://wg21.link/p2795r5, 2024-03-22.

[P2877R0] Joshua Berne and Tom Honermann. Contract Build Modes, Semantics, and Implementation Strategies. https://wg21.link/p2877r0, 2023-06-09.

[P2900R6] Joshua Berne, Timur Doumler, and Andrzej Krzemieński. Contracts for C++. https://wg21.link/p2900r6, 2024-02-29.

[P3097R0] Timur Doumler, Joshua Berne, and Gašper Ažman. Contracts for C++: Support for Virtual Functions. https://wg21.link/p3097r0, 2024-04-15.

[P3100R0] Timur Doumler. Contracts, undefined behaviour, and unspecified behaviour. https://wg21.link/p3100r0, 2024-04-15.

[P3119R0] Joshua Berne. Tokyo Technical Fixes to Contracts. https://wg21.link/p3119r0, 2024-04-03.

[P3165R0] Ville Voutilainen. Contracts on virtual functions for the Contracts MVP . https://wg21.link/p3165r0, 2024-02-16.

[P3257R0] Jens Maurer. Make the predicate of `contract_assert` more regular. https://wg21.link/p3257r0, 2024-04-26.