Tokyo Technical Fixes to Contracts

Document #:	P3119R1
Date:	2024-05-09
Project:	Programming Language C++
Audience:	SG21 (Contracts)
Reply-to:	Joshua Berne <jberne4@bloomberg.net>

Abstract

During EWG discussion of Contracts at the 2024 Tokyo WG21 meeting a number of minor issues were brought up. This paper discusses them and proposes resolutions.

Contents

Revision History

Revision 1

- Updated proposal on unbounded evaluations
- Added results of polls on proposal 1 and Proposal 2.

Revision 0

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

1 Introduction

On march 20, 2024 EWG met to discuss [P2900R6]. A few minor issues were brought up, which are discussed below along with reasoned proposals for their resolution.

2 Array Parameters

In postconditions we require that any function parameter that is ODR-used be marked `const`. This becomes a problem for array parameters due to array to pointer decay, as there is no way to mark such a parameter so that the resulting pointer itself is const, as the following two function declarations are equivalent and you can see that the parameter is a pointer to `const int` and not a `const` pointer:

```
void f(const int a[]);
void f(int * const a)
```

Adding a postcondition to the first declaration above will not do the right thing, as the pointer itself is not `const`:

```
void f(const int a[]) post( a[0] == 5 )
{
  static int x[1];
  a = x;
  a[0] = 5;  // postcondition will be satisfied
}

void g()
{
  int b[5] = {0,1,2,3,4,5};
  f(b);
  contract_assert(b[0] == 5);  // oops, that didn't happen.
}
```

Because there is no way to make the resulting pointer after pointer decay `const`, we should just disallow this usage. Any developer that wants to use an array parameter in a postcondition can change that parameter to be a pointer manually – there is no change in signature, ABI, or anything else significant in requiring this change.

Proposal 1: No Array Parameters In Postconditions

ODR-using an array parameter from the predicate of a postcondition is ill-formed.

This proposal was discussed on April 4, 2024 during an SG21 telecon. The following poll was taken:

2024-04-04 Poll 7

For the Contracts MVP, make it ill-formed to odr-use an array parameter from the predicate of a postcondition, as specified in Proposal 1 of P3119R0.

SF	F	N	A	SA
3	12	0	0	0

Result: Consensus

3 Use of C variadic functions parameters

We must consider how C variadic function parameters can be used in preconditions and postconditions.

In general, the sequence of va_start to va_end must occur within the same function. For contract assertions, it seems like we should have two modifications to these rules:

1. Any use of `va_start` within a contract assertion predicate must be matched by a use of `va_end` in the same predicate. In other words, for the purposes of C variable argument lists each contract predicate is a distinct function.
2. A postcondition assertion cannot make reference to C-style variadic arguments as there is no mechanism to make them `const`.

Unfortunately, the first requirement (as with the C requirement on the matching of `va_start` and `va_end` in a function) cannot be statically checked and thus must be made undefined behavior. Therefore it might be better to outright prevent the use of `va_start` in contract predicates entirely.

For now, this conservative approach is what we propose:

Proposal 2: No C variadics

If a contract predicate encloses a use of `va_start`, the program is ill-formed.

This proposal was discussed on April 4, 2024 during an SG21 telecon. During that discussion, it became clear that some platforms would not be able to diagnose this issue because the macros used for handling C variadic functions transform into expressions that have no identifiable traits once preprocessing is complete. Therefore, the room decided that, while this should still be ill-formed, no diagnostic should be required for this problem.

The following poll was taken:

2024-04-04 Poll 8

For the Contracts MVP, make it ill-formed, no diagnostic required for a contract predicate to enclose a use of `va_start`.

SF	F	N	A	SA
5	10	1	0	0

Result: Consensus

4 Unbounded Evaluations

Two problems with allowing an unbounded number of evaluations to occur for contract assertions within a contract assertion sequence have been brought up:

- A contract assertion that will exhibit UB after a number of repeated assertions could be considered to exhibit UB always — the particular example given was for a contract assertion that incremented an `int` as a side effect, something which will always eventually have undefined behavior if repeated a sufficient number of time. Treating the contract assertion evaluation as UB would, of course, require a particularly hostile compiler — yet it is worth considering something that might mitigate this concern.
- Real time systems which require a hard bound on the runtime complexity of software will be unable to use contracts if the Standard allows an unbounded number of evaluations. Even if, in practice, all platforms might be able to provide a practical limit on the number of repeated evaluations they might emit, this lack in the specification itself might lead some to avoid adopting Contracts in the first place.

There are, however, still reasons to allow repeated evaluations:

- With a mix of caller-side and callee-side evaluations across different translation units it can become impossible for a platform to guarantee that contract assertions are evaluated at least once when requested. Permission to, in some configurations, emit checks on both sides of the function invocation boundary prevents cases where a compiler would have to instead err on the side of not checking at all — a much worse possibility. This argues for allowing at least 2 evaluations of each contract assertion.
- The possibility of repeated evaluations helps make it even more clear to users that side effects are discouraged in contract assertions, as any dependency on the exact number of times side effects will occur is going to be unreliable.
- A particularly thorough mechanism to test for destructive contract assertions is to evaluate them repeatedly during testing and observe if results change. A conforming compiler option to request an arbitrary number of repetitions is an excellent mechanism to verify this – and on a compiler that is instructed to do this, most subsequent evaluations will be elided away completely.

A solution to the above problems that prevents the guaranteed undefined behavior, keeps contract assertion evaluation time bounded, all without preventing the motivating cases for repeated evaluations is to simply have implementations define a limit on the number of evaluations. This prevents the Standard from needing to provide an arbitrary number while allowing implementations to choose between the freedom of setting a particularly high number or anything as low as 1.

A value of 64 is recommended for this implementation limit as it is a number of iterations where `i++` is not going to be guaranteed undefined behavior for any signed type for `i`.

Proposal 3.A: Implementation defined limit on evaluations

The number of times a contract assertion may be repeated in a contract assertion sequence is an implementation limit (added to [implimits]) whose recommended value is 64.

There are, however, issues which this proposal does not address:

- Many users need to reason about exact upper bounds on the number of operations their functions might perform. In particular, multiplying the number of operations a contract assertion could perform by an unknown integral limit is a hindrance to providing cross-platform guarantees. Often, this is not as specific as number of machine operations which might be subject to variance with different optimization levels but instead is measured in terms of higher level functions that might be executed, such as allocations and deallocations that might occur.
- Not having a clear understanding of the number of evaluations that may occur is a common concern with the general specification of contract assertions in [P2900R6]. Even an implementation-defined limit still causes concerns about how high the cost of a contract assertion might be and how many times an observed contract violation might invoke the violation handler.
- Implementation limits as specified in [implimits] are predominantly not of this flavor. The existing limits are all limits that an implementation puts on a program beyond which there will not be support, not limits the implementation puts on what it will do. Limitations on the implementation itself are generally specified by making the corresponding quantities unspecified or implementation defined with a corresponding bound.

Initially,[1] the proposals for repetition were left completely unspecified to maximize implementation freedom and not require that implementations fully document all possible permutations that might lead to multiple evaluations, especially when mixing translation units compiled with different contract flags. In retrospect, this has led to a large amount of concern that compilers will take this as an opportunity to wantonly evaluate the same contract assertion far more times than desired.

Taking a page from [P2877R0], it seems that the solution that clarifies all of our expectations is to put the number of repeat evaluations into the same bucket that we put the semantics with which those contract assertions will be evaluated. Currently, we make that implementation defined with the understanding that this means that implementations will provide all needed build options to select explicitly how those values will be chosen. We can, and seemingly should, do the same for repeated evaluations of a contract assertion.

In addition, to parallel what has been done for specifying the selection of contract semantics, we

[1]Note that this is the author of this paper providing information about unstated motivation for the contents of an earlier paper ([P2751R1]) by the same author.

should provide a recommended practice about what users might be able to specify and what the expected default should be.

Proposal 3.B: Implementation defined number of repetitions

Within a contract-assertion sequence a previously evaluated contract assertion may be evaluated again with the same or a different evaluation semantic, up to an implementation-defined number of times.
Recommended Practice: An implementation should provide an option to perform a specified number of repeated evaluations for contract assertions. By default no additional repetitions should be performed.

Note a few things:

- It would be expected that implementations which support mixing caller-side and callee-side checking would provide corresponding documentation that such mixes would, by default, repeat evaluations of precondition and postcondition assertions one additional time. When asking for multiple evaluations, an implementation might document that it introduces those extra repetitions to caller-side checks, callee-side checks, or both.
- This formulation, hopefully, makes it more clear that repetition is a feature that users must opt in to with open eyes and not a problem that must be endured. With non-destructive contract assertions there will be no semantic problems when repeating evaluations of contract assertions, and by choosing higher repetitions a user is clearly accepting the potential added runtime cost.
- The selection of repetitions, as with the chosen semantic for those evaluations, is explicitly not constrained to being set globally or to even be the same value on different executions of the same function.
- Rich configuration options such as repeating just preconditions assertions, postconditions assertions, assertion statements, or even individual contract assertions are all conforming options as long as the available options have their effects included by the platform in the definition of how many repetitions there might be.

5 Proposed Wording

Wording will be produced when time is available or when SG21 has consensus on these proposals. Wording is relative to [P2900R6].

6 Conclusion

This has hopefully made [P2900R6] an even more robust proposal for inclusion into C++.

Acknowledgements

Thanks to EWG for the productive discussion on the details of [P2900R6].

Bibliography

[P2751R1] Joshua Berne, “Evaluation of *Checked* Contract-Checking Annotations”, 2023
http://wg21.link/P2751R1

[P2877R0] Joshua Berne and Tom Honermann, “Contract Build Modes, Semantics, and Implementation Strategies”, 2023
http://wg21.link/P2877R0

[P2900R6] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, “Contracts for C++”, 2024
http://wg21.link/P2900R6