Contract Evaluation in Constant Expressions

Document #:	P1671R0
Date:	2019-06-16
Project:	Programming Language C++
Audience:	EWG

Reply-to:	Joshua Berne <jberne4@bloomberg.net>
	Alisdair Meredith <ameredith1@bloomberg.net>

Contents

1 Introduction **1**

2 Proposed Changes **3**

3 Formal Wording **3**

4 References **4**

Abstract

The current working C++ draft [N4810] makes violations of checked contracts not core constant expressions, but requires that nothing be done during constant evaluation for any unchecked contracts. This requires that code which might be undefined behavior (UB) at runtime be treated as valid at compile time, a distinct difference from most other treatments of UB in constant evaluation contexts. We propose some wording clarifications and changes to fix this situation.

1 Introduction

The current draft has two locations which define the interaction between contracts and constant expression evaluation. The first is in **[dcl.attr.contract.check]/4**:

> During constant expression evaluation (7.7), only predicates of checked contracts are evaluated.

The second is far away in **[expr.const]/4.11**:

> An expression `e` is a core constant expression unless the evaluation of `e`, following the rules of the abstract machine (6.8.1), would evaluate one of the following expressions:
>
> - ...
> - a checked contract (9.11.4) whose predicate evaluates to `false`;
> - ...

The impact of this behavior needs to be understood in two contexts - those that are *definitely* compile-time evaluated, such as initializing a `constexpr` variable or evaluating the value of a template parameter, and those that are *potentially* compile-time evaluated, which is most other expressions that might potentially be core constant expressions. The current wording makes violations of checked contracts ineligible for use in compile time evaluations. Violations of unchecked contracts, however, while undefined behavior if they are evaluated at runtime, are well-formed if evaluated at compile time.

Consider a function `f` (independently of whether such a function might be a good idea) which cannot be called in contract, and how well formed uses of that function might be in different contexts:

```
constexpr int f() [[ expects : false ]] { return 0; }

void g()
{
  static int x = f();
    // if default contracts are on, invokes the violation handler.
    // if default contracts are off (assumed), this is undefined behavior.

  constexpr int y = f();
    // If default contracts are on, doesn't compile (f is not a core constant
    // expression).
    // If default contracts are off, this is well formed and y == 0.
}
```

This last case is the inconsistent one, and we believe a conforming compiler should be allowed to treat this as ill-formed if it is able to identify that the predicate is `false`. Note importantly that this example predicate, `false`, is clearly reliably evaluable even if the check is disabled, but being disabled in general is an indicator that the developer does not feel the checks are necessarily evaluable at all (or might take too long to evaluate), so we cannot require that the compiler evaluate all predicates and identify all violations at compile time.

An important point to consider is that currently all violations of contracts in standard library functions are treated as being potentially not core constant expressions due to the last part of **[expr.const]/4**:

> If `e` satisfies the constraints of a core constant expression, but evaluation of `e` would evaluate an operation that has undefined behavior as specified in Clause 16 through Clause 32 of this document, or an invocation of the `va_start` macro (17.13.1), it is unspecified whether `e` is a core constant expression.

This treats identifying a violation of library contract at compile time as up to the implementation, and gives implementations the leeway to make such violations into not-core-constant-expressions.

We believe the following items should be addressed:

- The wording in **[dcl.attr.contract.check]** does not make it clear that even when a checked contract predicate is evaluated during constant expression evaluation the violation handler will never be executed in such a situation. While it would be redundant, a normative note

here to replace this would greatly clarify that the definition of the violation handler is not needed to compile a single translation unit.

- Instead of allowing violations of unchecked contracts during constant expression evaluation, we think that they should be treated in the same way as violations of library contracts, leaving it up to the implementation to diagnose a violation if it can.

2 Proposed Changes

The clause in **[dcl.attr.contract.check]/4** regarding constant expression evaluation should be removed or replaced with a note referring to **[expr.const]/4** indicating that expressions with contract violations might not be core constant expressions.

Violations of unchecked contracts should be added to the list of expressions which are not specified to be core constant expressions.

3 Formal Wording

In **[dcl.attr.contract.check]** the following is changed:

~~During constant expression evaluation (7.7), only predicates of checked contracts are evaluated.~~ During constant expression evaluation (7.7), only predicates of checked contracts are evaluated, and any predicate that would evaluate to `false` is not a core constant expression. In other contexts, it is unspecified whether the predicate for a contract that is not checked under the current build level is evaluated; if the predicate of such a contract would evaluate to `false`, the behavior is undefined.

In **[expr.const]/p4** the following is changed (though changing the entire trailing paragraph into a separate bulleted list should be considered): An *expression* `e` is a core constant expression unless the evaluation of `e`, following the rules of the abstract machine (6.8.1), would evaluate one of the following expressions:

- ...
- a checked contract (9.11.4) whose predicate evaluates to `false`;
- ...

If `e` satisfies the constraints of a core constant expression, but evaluation of `e` would evaluate an operation that has undefined behavior as specified in Clause 16 through Clause 32 of this document, an unchecked contract (9.11.4) whose predicate would evaluate to `false`, or an invocation of the `va_start` macro, it is unspecified whether `e` is a core constant expression.

4 References

[N4810] Richard Smith, *Working Draft, Standard for Programming Language C++* http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/n4810.pdf

[P1429R2] Joshua Berne, John Lakos *Contracts That Work*