Constant evaluation of Contracts

Timur Doumler (papers@timur.audio)

Document #: P2894R0
Date: 2023-08-22
Project: Programming Language C++
Audience: SG21

Abstract

This paper discusses the semantics of contract-checking annotations during constant evaluation. We investigate what needs to be specified, enumerate the possible choices, establish our design goals, and propose a solution that provides a reasonable tradeoff between these design goals. We propose that CCAs should be considered during constant evaluation; if the predicate of a CCA is not a core constant expression, or evaluates to `false`, it should be implementation-defined whether the program is ill-formed or the CCA is ignored during constant evaluation.

1 Introduction

In order to deliver a Contracts facility that can be included in Standard C++ (see [P2695R1]), we need to fully specify the compile-time and runtime behaviour of contract-checking annotations (CCAs) in all cases. The specification for some cases is still missing (see [P2896R0]). One of these cases is the question of how CCAs should behave during constant evaluation. This paper proposes a solution.

The topic of constant evaluation of CCAs has been discussed before in the appendices of [P2834R1]. That paper proposes the following:

> If an expression or conversion is manfiestly constant-evaluated, it is ill-formed if the evaluation of the predicate of a CCA with a runtime-checked semantic disqualifies that expression from being a core constant expression.

However, we since adopted [P2877R0] for the Contracts MVP. This paper removes the notion of build modes and makes the semantics of a CCA implementation-defined and in general unknowable at compile time. The ideas from [P2834R1] therefore no longer apply, as we can no longer make it dependent on the contract semantic (whether it is runtime-checked) whether a piece of C++ code is ill-formed. We need to start over.

Note that SG21 has not yet settled on a syntax for CCAs. In this paper, we use attribute-like syntax [P2487R0] for the code examples, however the proposal is independent of the choice of syntax and would work in the same way with lambda-based syntax [P2461R1] or condition-centric syntax [P2737R0].

2 What do we actually need to specify?

First of all, note that we do not need to specify anything for the "happy case": if a CCA is encountered during constant evaluation, the predicate of that CCA is a core constant expression, and that expression evaluates to `true`, then the semantics are simply that the CCA should have no semantic effect whatsoever apart from ODR-using the entities that appear in the predicate.

The two interesting cases, which we still need to specify for the MVP, and which we discuss in this paper, are:

— **Case 1:** The predicate is not a core constant expression,

— **Case 2:** The predicate is a core constant expression but that expression does not evaluate to `true`.

Let us study both cases further.

2.1 Case 1: Not a core constant expression

First of all, note that since [P2448R2] was adopted for C++23, we do not need to do anything about CCAs on a `constexpr` or `consteval` function if that function is not actually called during constant evaluation:

```
int pred();  // predicate not constexpr

int f() constexpr [[ pre: pred() ]];  // OK; never called during constant evaluation
int g() consteval [[ pre: pred() ]];  // OK: never used

int main() {
  return f();  // not constant evaluation of f
}
```

In these cases, the CCA is only ever checked during runtime, and therefore will have the same semantics as it always does.

The interesting case is what should happen if the compiler actually encounters the predicate above during constant evaluation (we do not have to distinguish between `constexpr` and `consteval` here)?

```
int pred();  // predicate not constexpr

int f() constexpr [[ pre: pred() ]];  // ???

int main() {
  std::array<int, f()> a;  // constant evaluation of f
  // ...
}
```

There are four options to choose from:

— Make the above code ill-formed;

— Make it ill-formed, no diagnostic required (IFNR);

— Ignore the CCA during constant evaluation;

— Ignore the CCA normatively, but recommend that a warning be issued.

In addition, we can choose to specify that it is implementation-defined whether one option or another should apply. Note also that even if we ignore the CCA during constant evaluation, entities inside the predicate will still be ODR-used and therefore can trigger template instantiations and lambda captures.

2.2 Case 2: Does not evaluate to `true`

The other case that needs to be specified is: what should happen if a CCA is encountered during constant evaluation, the predicate *can* be evaluated during compile time (therefore Case 1 does not apply), but that predicate does not evaluate to `true`? For example, how should the following program behave?

```
int f(int i) constexpr [[ pre: i > 0 ]];

int main() {
  std::array<int, f(0)> a;  // out-of-contract call during constant evaluation
  // ...
}
```

At runtime, there are many ways in which a predicate can *not* evaluate to true:

— It can evaluate to `false`,

— It can throw an exception,

— It can `longjmp`,

— It can terminate the program,

— It can be undefined behaviour.

In the Contracts MVP, at runtime, we treat the first two as a contract violation; in all remaining cases, the user "gets what they get" (the behaviour "escapes" the CCA). In addition, at runtime, a CCA can have one of three semantics: *ignore*, *observe*, or *enforce*. With the *ignore* semantic, the predicate is not checked at all. The other two are checked semantics that only differ in what happens after the violation handler is invoked.

During constant evaluation, things look very differently. First of all, as we already established, it is unknowable during constant evaluation whether any given CCA will have *ignore*, *observe*, or *enforce* semantics at compile time; we need to specify the semantics for constant evaluation completely separately. Further, during constant evaluation there is no violation handling, therefore it makes no sense to distinguish between *observe* or *enforce*; instead, the CCA is simply either ignored or not. Further, if it is not ignored, and it does not evaluate to `true`, the only other option is that it evaluates to `false`, so this is the only other case we need to consider here. During constant evaluation, you cannot throw an exception, you cannot `longjmp`, you cannot terminate the program, and there cannot be undefined behaviour; an expression that would do any of these things at runtime is not a core constant expression and therefore its semantics during constant evaluation are defined by Case 1.

Therefore, to specify Case 2, we only need to define what should happen if a contract predicate is a core constant expression and that expression evaluates to `false` (as in the code example at the beginning of this section). Again, there are the same four options to choose from:

— Make the above code ill-formed;

— Make it ill-formed, no diagnostic required (IFNR);

— Ignore the CCA during constant evaluation;

— Ignore the CCA normatively, but recommend that a warning be issued.

In addition, we can again choose to specify that it is implementation-defined whether one option or another should apply.

3 Design goals

Before proposing a solution, we should first agree on the *design principles* that our proposal should satisfy. The following principles seem to be the most relevant for deciding on the correct solution for constant evaluation of Contracts:

1. **Correctness.** Contracts are first and foremost a tool for reasoning about the correctness of a program. Constant evaluation of Contracts should support this primary goal.
2. **Flexibility.** There is a multitude of different use cases (see [P1995R1]) that a Contracts facility should be able to support. Constant evaluation of Contracts should be usable for the "multiverse" and not just a single use case.
3. **Teachability.** Constant evaluation of Contracts should be designed such that the rules are easy to learn and teach, intuitive to the user, and free of unnecessary complexity.
4. **Compile times**. Constant evaluation of Contracts should not significantly increase compile times for real-world code.

No possible design will satisfy all four design principles above. Picking the correct semantics therefore involves picking the right tradeoff between different, potentially conflicting goals. This will involve prioritising the goals relative to each other. This relative priority should reflect the needs of our users.

4 Possible solutions

Given the above design principles, we can move on to discussing concrete solutions.

4.1 Option A: make both cases ill-formed

The first possible solution is to make both Case 1 and Case 2 ill-formed. This solution prioritises *correctness* above the other design goals. With this specification, whenever the user did something wrong, the compiler is required to tell them. For correctness, it is best to not allow predicates that cannot be checked (Case 1), and if a predicate can be checked, it is better to catch such a predicate failing at compile time than at runtime (Case 2), just like in all the other cases in C++ where catching a bug at compile time is better than running into the bug at runtime. This solution is also very good for *teachability* as this set of rules is simple, intuitive, and easy to remember. However, we are trading off *flexibility* as well as *compile times*: the compiler is forced to fully evaluate all CCAs at compile time. According to some participants of SG21, the resulting increases in compile times will be unacceptable for some code bases, which will force these users to wrap CCAs into macros to be able to manually ignore them.

4.2 Option B: make both cases ignored

The opposite solution is to ignore the CCA for both Case 1 and Case 2 (other than for ODR-use). This solution prioritises compile times above the other design goals. The compiler will simply skip all CCAs in the program during constant evaluation, and only consider them at runtime, ensuring that Contracts do not lead to a huge increase in compile times due to constant evaluation. This solution has good teachability as well. However, we are trading off flexibility and correctness: this solution would outright prohibit compilers from diagnosing errors in CCAs during constant evaluation, even if the compiler in question can do it and the user considers it useful.

4.3 Option C: make both cases implementation-defined

The compromise solution is to make it implementation-defined whether the CCA is ill-formed or ignored (other than for ODR-use), for both Case 1 and Case 2. This solution prioritises flexibility and seeks a compromise between correctness and compile times by allowing the compiler to prioritise either, and give the choice to the user. However, we are trading off *teachability* as we are making the rules more complex, and placing another aspect of Contracts outside of the realm of the C++ Standard and into the hands of compiler vendors. With this specification, compilers can offer their own compiler flags or implementation-defined "build modes" to let the user choose whether they prefer to check their CCAs during constant evaluation (correctness) or ignore them (compile time). The compilers could choose to tie this choice to the choice of runtime semantic, which is also implementation-defined, such that in an *ignore* build mode, CCAs would be ignored also during constant evaluation, but in the other (checked) build modes, they would be checked also during constant evaluation. They could also choose to make the choice of compile-time semantic independent of the choice of runtime semantic, so you could for example have a "fast to compile" configuration that nevertheless checks contracts at runtime. A hostile compiler implementation could also choose to only implement Option A or Option B and still be conformant.

4.4 Other options

We believe that Options A, B, and C above are the only ones worth considering. We do not believe that making Cases 1 or 2 IFNDR makes sense, because this would be somewhere in between making them outright ill-formed and making the semantic implementation-defined, but without the requirement that the implementation documents how to determine which semantic is being applied. From a usability perspective, this is strictly worse than Option C without any additional benefit. Similarly, we do not believe that the option of "ignore but recommend a warning" makes sense, because it is essentially the same as making it implementation-defined, but again without the requirement that the implementation documents the behaviour and so strictly worse without any additional benefit. Finally, we also do not believe that choosing one option for Case 1 but another for Case 2 makes sense, as it would be detrimental to the design principle of teachability (by making the rules very complex), without no apparent overall benefit for meeting the other design principles.

5 Proposed solution

Option A is desirable in principle, as correctness is the most important design goal. However, it could hurt the adoption of Contracts if they make compile times unacceptably slow for some users and they start wrapping their Contracts into macros to be able to ignore them at compile time. The worst part about this is that they would then necessarily also be ignored at runtime as well, throwing correctness checks over board, as Option A does not allow any universe in which CCAs are ignored for constant evaluation but still considered at runtime (which is by far more important than considering them during constant evaluation!)

Option B is similarly not viable. Shorter compile times are good, but they are of no use if the resulting program is *incorrect* and has potential undefined behaviour because it does not meet its contracts, or because its contracts are not written correctly. The Standard should at least allow to diagnose these cases.

We therefore propose Option C (implementation-defined). We believe that this option provides the best tradeoffs. Introducing yet another compiler switch is somewhat unfortunate, but is necessary in this case to give all users of Contracts the possibility to use them effectively.

6 Consistency with P2834R1

[P2834R1] was proposing that the compile-time semantic of a CCA follows its runtime semantic: in a build mode where all CCAs are ignored at runtime, they should be ignored at compile time as well; in a build mode where all CCAs are checked at runtime, Cases 1 and 2 should be ill-formed.

[P2877R0] did away with build modes; the runtime semantic of a CCA is no longer known at compile time and therefore the proposed solution in [P2834R1] no longer possible. However, arguably the solution proposed in this paper is consistent with the design principles of both papers. [P2877R0] replaced build modes with the notion that the runtime semantics are implementation-defined; we perform the same transformation with the compile-time semantics, which are now implementation-defined in the same manner.

7 Comparison with `[[assume]]`

When specifying `[[assume]]` (see [P1774R8]), we faced the exact same problem: how should assumptions behave during constant evaluation? The same two cases exist there as they do for CCAs. For `[[assume(expr)]]`, the following solution was chosen:

— Case 1: If `expr` is not a core constant expression, the assumption is ignored;

— Case 2: If `expr` does not evaluate to `true` during constant evaluation, it is implementation-defined whether the assumption is ill-formed.

Note that for Case 2, the semantic is the same as the one proposed here for Contracts, but for Case 1, it is different. We believe that this seeming inconsistency is totally fine, because the design goals for `[[assume]]` are entirely different than the ones for Contracts. As discussed above, Contracts are a feature primarily intended for reasoning about the correctness of a program. On the other hand, `[[assume]]` is an inherently unsafe feature whose sole purpose it is to improve *performance*. Correctness has meaning both during constant evaluation and at runtime, whereas performance only has any significance at runtime. Since assumptions serve no useful purpose at compile time, it makes sense to simply ignore assumptions that cannot be evaluated at compile time (which is not the case for Contracts). At the same time, if an assumption *can* be evaluated at compile time, we should allow (but not force) the compiler to diagnose if this assumption does not hold (same as for Contracts).

8 Proposal summary

We propose that the following be added to the Contracts MVP (wording to be provided after design approval by SG21):

— CCAs are considered during constant evaluation;

— If the predicate of a CCA is not a core constant expression, it is implementation-defined whether the CCA is a core constant expression;

— If the predicate of a CCA is a core constant expression that evaluates to `false`, it is implementation-defined whether the CCA is a core constant expression.

Acknowledgements

Thanks to Oliver Rosten and Gašper Ažman for their very helpful feedback during the drafting of this paper.

References

[P1774R8] Timur Doumler. Portable assumptions. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p1774r8.pdf, 2022-06-14.

[P1995R1] Joshua Berne, Timur Doumler, Andrzej Krzemieński, Ryan McDougall, and Herb Sutter. Contracts – Use Cases. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1995r1.html, 2020-03-02.

[P2448R2] Barry Revzin. Relaxing some `constexpr` restrictions. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2448r2.html, 2022-01-27.

[P2461R1] Gašper Ažman, Caleb Sunstrum, and Bronek Kozicki. Closure-Based Syntax for Contracts. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2461r1.pdf, 2021-11-15.

[P2487R0] Andrzej Krzemieński. Attribute-like syntax for contract annotations. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2487r0.html, 2021-11-12.

[P2695R1] Timur Doumler and John Spicer. A proposed plan for Contracts in C++. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2695r1.pdf, 2023-02-09.

[P2737R0] Andrew Tomazos. Proposal of Condition-centric Contracts Syntax. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2737r0.pdf, 2021-11-15.

[P2834R1] Joshua Berne and John Lakos. Semantic Stability Across Contract-Checking Build Modes. https://wg21.link/p2834r1, 2023-05-15.

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

[P2896R0] Timur Doumler. Outstanding design questions for the Contracts MVP. https://wg21.link/p2896r0, 2023-08-22.