Constant evaluation of Contracts

Timur Doumler (papers@timur.audio)

Document #: P2894R1
Date: 2023-12-07
Project: Programming Language C++
Audience: SG21

Abstract

This paper proposes semantics for constant evaluation of contract annotations. We propose that during constant evaluation, contract annotations should be evaluated with an implementation-defined choice of *ignore*, *enforce*, or *observe* semantics, analogous to their runtime counterparts; in a manifestly constant-evaluated context, a contract annotation with a checked semantic and a predicate that is not a core constant expression renders the program ill-formed, while a contract predicate that evaluates to `false` emits a diagnostic if the contract semantic is *observe* or *ignore*, and additionally renders the program ill-formed if the contract semantic is *enforce*. Special rules are required for the case of evaluating contract annotations in the initialiser of a non-`constexpr` variable that may or may not be constant-initialised, such as variables with static or thread storage duration and variables of `const`-qualified integral or enumeration type.

1 Introduction

In order to deliver a Contracts facility for Standard C++ (see [P2695R1]), we need to produce a complete specification for the behaviour of contract annotations both at compile time and at runtime. The specification currently in development (see [P2900R1]) still has some design holes (see [P2896R0]). One of these design holes is the question of how contract annotations should behave during constant evaluation. In this paper, we develop a solution to this question.

This question has first been discussed in the appendices of [P2834R1]. The solution proposed there no longer applies, since it is tied to the notion of checked and unchecked build modes, which was removed with the adoption of [P2877R0]. An updated solution is proposed in [P2932R1] section 3.3, derived from the principles proposed in [P2932R1] section 2. In this paper, we instead derive a solution from practical considerations: how would C++ users expect contract checks to behave when evaluated at compile time, and how can we best meet their needs? As we will see, the specification we arrive at in this paper is actually identical to the one proposed in [P2932R1] section 3.3.

2 Discussion

2.1 To consteval or to not consteval

The first question that arises is whether we should consider contract checking during constant evaluation at all.

On the one hand, for many users the answer will be "yes". While we may think of contract checks primarily as a tool to identify bugs at runtime, similarly to `assert` macros today, contract checking can undoubtedly be useful at compile time, too. It provides immediate value for `constexpr` functions that can be evaluated either at compile time or during runtime: for program correctness it is desirable to identify bugs in such function calls at compile time where possible. Consider the following program:

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

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

Going beyond issuing compiler diagnostics in cases like the above, it is desirable to have a solid framework for the semantics of contract annotations at compile time if we wish to expand usage of Contracts in C++ to use cases like static analysis and formal program verification.

On the other hand, for many users the answer might be "no". Checking contracts at compile time has the potential to significantly increase compile times, which is already a huge problem in many domains where C++ is used. Forcing compile-time contract checking on every user of C++ will likely result in many users wrapping their contract annotations into macros to be able to ignore them at compile time, or even worse, not adopting Contracts in C++ at all.

The answer is therefore that we must give the user both options. Whether contracts are checked during constant evaluation should be implementation-defined, allowing implementations to offer both options, e.g. through compiler flags.

2.2 Contract semantics *ignore* and *enforce* at compile time

In a previous revision of this paper, we suggested that letting the user choose whether they want to check their contracts at compile time could be accomplished by making it implementation-defined whether contract checks in the program are checked during constant evaluation or not, essentially introducing a "global switch" for turning compile-time contract checking on or off. We since realised that this is insufficient, because this setting might be different from one translation unit (TU) to another. Moreover, for an inline function that has contract annotations in a shared header, this setting might be different from one constant evaluation of a contract annotation to another constant evaluation of the same contract annotation; such cases must not be considered ODR violations.

The answer is therefore that we should adopt the same approach for constant evaluation of any given contract annotation as we already do in the Contracts MVP [P2900R1] for *runtime* evaluation of any such contract annotation: that the *contract semantic* of every contract check is implementation-defined and can vary from contract annotation to contract annotation and also from one evaluation of a contract annotation to another evaluation of the same contract annotation.

At runtime, there are three possible contract semantics in the Contracts MVP: the *unchecked* semantic, *ignore*, and the *checked* semantics, *enforce* and *observe*. The *ignore* semantic can be extended to constant evaluation in a straightforward way: do nothing[1]. The same is true for the *enforce* semantic: we can say that an *enforce*d contract check is checked during constant evaluation, and if this check fails, a compiler diagnostic is issued (which is the compile-time analogue of the default contract-violation handler printing a diagnostic message at runtime) and the program is ill-formed (which is the compile-time analogue of terminating the program after the contract-violation

[1] Note that even though the *ignore* semantic means that a contract is not evaluated (neither at compile time nor at runtime), its predicate is still parsed (so it must be a valid expression) and the entities in the predicate are still ODR-used.

handler returns). The only difference is that constant evaluation has no compile-time analogue for installing a user-defined contract-violation handler: as a user-replaceable function that is added at link time, a user-defined contract-violation handler is inherently a runtime-only feature.

2.3 Contract semantic *observe* at compile time

It turns out that the *observe* semantic has a useful compile-time implementation as well: an *observe*d contract check is checked during constant evaluation, and if this check fails, a compiler diagnostic is issued; however, unlike an *enforce*d contract check, this does not render the program ill-formed, which means the compiler can keep going. This is useful for the same reason that the *observe* contract semantic is useful at runtime: it allows to introduce new contract annotations to an existing legacy codebase without that codebase immediately breaking if a contract violation is detected.

We initially had concerns about enabling the *observe* semantic for constant evaluation, because it seems to introduce a novelty: a normative diagnostic that does not render the program ill-formed, a "standard warning". However, after some discussion it turned out that we already have precedent for such a thing in the Standard since we introduced `#warning` [P2437R1] for C++23. The fact that the latter is a preprocessor directive rather than a "proper" language feature does not seem like a substantial difference.

2.4 Compile-time semantic is independent of runtime semantic

The approach proposed here can be described as essentially taking the block of pseudocode in [P2900R1] section 2.4.10 which describes the mechanics of contract-violation handling, making both `__current_semantic()` and `__check_predicate()` compiler intrinsics `constexpr`-enabled, and then adding a `std::is_constant_evaluated() == true` branch to the algorithm.

One important property of this approach is that since the contract semantic of any given contract check can vary from evaluation of a given contract annotation to the next evaluation of the same contract annotation, it is also perfectly confirming to use one contract semantic for all constant evaluations and another contract semantic for all runtime evaluations of a given contract annotation. This can be very useful since there are plausible use cases for such a setup. For example, in a debug build of a large and slow-to-compile application, one might want to disable contract checks at compile time to speed up compilation (minimising turnaround time during development), while at the same time enabling contract checks at runtime. Our approach enables this and many other use cases by making the contract semantic implementation-defined both at compile time and at runtime, following the spirit of [P2877R0].

2.5 Possible outcomes of constant evaluation

When a contract annotation is evaluated during constant evaluation, there are three possible outcomes:

— The predicate is a core constant expression that evaluates to `true`,

— The predicate is not a core constant expression,

— The predicate is a core constant expression that does not evaluate to `true`.

Our first observation is that we do not need to specify anything further for the first case, the "happy path": if a contract annotation is evaluated during constant evaluation, the predicate of that contract annotation is a core constant expression convertible to `bool`, and the converted expression evaluates to `true`, then the obvious semantics of such a contract check are that it simply has no semantic effect whatsoever. In the remainder of this paper, it is sufficient to study only the second and third case.

2.6 Contracts that cannot be checked at compile time

What should happen if a contract check with a checked semantic (*observe* or *enforce*) encounters a predicate that is not a core constant expression, i.e. cannot be evaluated at compile time?

First of all, note that since [P2448R2] was adopted for C++23, we do not need to do anything about this case if the function in question is `constexpr` or `consteval`, but is never actually called during constant evaluation:

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

constexpr int f()
  pre (pred());  // OK; never called during constant evaluation

consteval int g()
  pre (pred());  // OK: never called

int main() {
  return f();  // not a constant evaluation of f
  // g never called
}
```

For `f`, the contract check is only ever checked during runtime, and therefore will have the same semantics as it always does; for `g`, the contract check is never checked at all. This program is therefore well-formed and its behaviour is unambiguous.

The only interesting case is: what should happen if the compiler actually encounters such a predicate during constant evaluation (we do not have to distinguish between `constexpr` and `consteval` here)? For example, consider the following program:

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

constexpr int f()
  pre (pred());  // ???

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

One possibility is to specify that if the predicate is not a core constant expression, the entire contract annotation is not a core constant expression. This would give a compiler error in the case above and many similar cases. However, it turns out that it is possible to SFINAE on whether an expression is a constant expression and therefore follow a different codepath depending on whether a contract is evaluated at compile time; [P2932R1] section 3.3 has an example of such code. We do not believe that it makes any sense to allow such strange control flow. The answer is therefore that we should treat this case as a contract violation at compile time: the compiler must issue a diagnostic if the contract semantic is *observe* or *ignore*, and in addition the program is ill-formed if the contract semantic is *enforce*.

Conceptually, this specification makes sense. Consider the following function:

```
constexpr int do_something(int i)
  pre (i > 0)
  pre (hardware_thingy_available());  // not constexpr
```

Ignoring the precondition annotation when it is not checkable at compile time and we are using a checked contract semantic at compile time would be wrong, because if the precondition is not checkable at compile time it is also not *satisfiable* at compile time: the "hardware thingy" is literally not available while compiling the code, therefore we cannot satisfy this precondition when calling the function at compile time. We can think of this as conceptually similar to how we cannot satisfy

a precondition at runtime if we cannot determine whether the predicate evaluates to `true` or `false` because evaluating it throws an exception — this case, too, is treated as a contract violation.

If the user wishes to say that it is still *correct* to call the function at compile time, even though the "hardware thingy" does not become available until runtime, they need to express that explicitly in the precondition, thus making the program correct:

```
constexpr bool can_do_something() {
  if (!std::is_constant_evaluated())
    return hardware_thingy_available();
  else
    return true;
}

constexpr int do_something(int i)
  pre (i > 0)
  pre (can_do_something());  // OK
```

2.7 Predicate evaluation at compile time

What can happen if a contract annotation is evaluated during constant evaluation, the predicate of that contract annotation is a core constant expression, but that expression does not evaluate to `true`, i.e. we have identified a contract violation at compile time?

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 cases as a contract violation; in all remaining cases, the user "gets what they get" (the behaviour "escapes" the contract check).

During constant evaluation, the situation is actually much simpler. 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, which should be treated as a contract violation as discussed in 2.6.

It follows that the only other way in which we could get a contract violation at compile time, and the only way in which we could get a contract violation at compile time *if the predicate is a core constant expression* is if the predicate evaluates to `false`. As we already discussed above, in this case the compiler must issue a diagnostic if the contract semantic is *observe* or *ignore*, and in addition the program is ill-formed if the contract semantic is *enforce*.

2.8 Trial constant evaluation

`constexpr` variables have to be constant-initialised. Contract annotations encountered while evaluating the initialiser of such a variable will be evaluated during constant evaluation, following the rules described above.

However, there are certain cases where a non-`constexpr` variable may or may not be constant-initialised, depending on whether the initialiser is a core constant expression. One such case are variables with static or thread storage duration; if the initialiser is a core constant expression, such a

variable will be constant-initialised, otherwise the initialisation will be relegated to runtime (dynamic initialisation) and will happen during startup before entering `main` (see [basic.start.static].2):

```
constexpr int f() { return 42; }
int g() { return 43; }  // not constexpr

static int j = f();     // constant initialisation
static int i = g();     // dynamic initialisation
```

Another case are variables of non-`volatile`, `const`-qualified integral or enumeration type (see [expr.const].3); if the initialiser is a core constant expression, then it becomes a manifestly core constant expression and the variable can be used afterwards in constant expressions (for example, as a non-type template argument); otherwise, the variable will be initialised normally at runtime if and when control flow reaches its definition:

```
constexpr int f() { return 42; }
int g() { return 43; }  // not constexpr

int main() {
  const int i = f();   // constant initialisation
  const int j = g();   // runtime initialisation

  std::array<int, i> a;  // OK
  std::array<int, j> b;  // Error: j is not a core constant expression
}
```

In all the cases above, the compiler needs to determine whether the initialiser is a core constant expression, which will determine the semantics of the program. The compiler may perform so-called *trial* constant evaluation to make this determination (see [expr.const] 19.5 and [expr.const] footnote 73). This gives rise to a new case that we need to specify: what should happen if the initialiser would otherwise be a core constant expression, but contains a contract annotation that is not checkable during constant evaluation? Consider the following program:

```
bool whatever();      // not constexpr

constexpr int f()
  pre(whatever())     // contract check not evaluable at compile time
{
  return 42;
}

static int i = f();  // ???
```

If we treat the contract predicate just like any other expression that is part of the definition of `f`, this would mean that the contract check turns the constant initialisation of `i` into dynamic initialisation in the code example above. However, similar to other cases such as a contract check triggering a lambda capture [P2890R1] or a contract check triggering the deduction of a potentially-throwing exception specification [P2969R0], treating the contract check as "just code" leads to violations of the *zero overhead principle* from [P2932R1]: the mere addition of a contract annotation causes the program to take a different branch at compile time, which in turn can cause "heisenbugs" (a program contains a bug, we add a contract annotation to find the bug, but the contract annotation causes the program to take another branch where the bug does not exist) and potentially measurable performance degradations.

Fortunately, in this case there is a workaround that fixes this problem. We can treat such cases as a compile-time contract violation, in exactly the same way as we would treat a non-core-constant-expression contract predicate during regular constant evaluation. Specifying this workaround correctly requires a bit more trickery than for regular constant evaluation. The reason is that the

initialiser might not be a core constant expression anyway, regardless of the presence of the contract check, but the program might still be valid:

```
bool whatever();      // not constexpr

constexpr int f(int i)
  pre(whatever())  // contract check not evaluable at compile time
{
  if (i == 0)
    return runtime_thingy::get_value();  // not constexpr

  return i;
}

static int i = f(0);  // not a core constant expression when passing in 0!
```

In this case, we do not want to try to check the contract at compile time and make the program ill-formed as a result of that, because trial constant evaluation for `f` will fail anyway and the function will never actually be called at compile time.

The above semantics can be achieved by using the following algorithm. First, we conduct a trial constant evaluation *ignoring* any contract annotations that might otherwise be evaluated during that evaluation, even if these have a checked semantic at compile time; if this trial constant evaluation fails, the initialiser is not a core constant expression and will be called at runtime instead (which means the contract checks will be evaluated at runtime as well) and it therefore does not matter whether any of these contract predicates are core constant expressions. If however, the trial constant evaluation succeeds, we re-evaluate the expression but this time we evaluate the contract annotations as well. If any such evaluation fails to be a constant expression or does not evaluate to `true` we treat it as a contract violation. For each contract violation where the semantic is *observe* or *enforce*, the compiler must issue a diagnostic; additionally, if the semantic is *enforce*, the program is ill-formed.

2.9 No eager contract checking during trial evaluation

There is one final variation on the theme of constant evaluation that is worth analysing. Let us consider again a `constexpr` function that either is or is not a core constant expression, depending on which arguments are being passed in:

```
constexpr int divide(int n, int d) {
  return n / d;    // not a core constant expression if d == 0
}
```

Now, we might want to add a contract annotation to prevent running into this case:

```
constexpr int divide(int n, int d)
  pre (d != 0) {  // contract predicate is always a core constant expression!
  return n / d;
}
```

Everything seems fine. If `divide` is called at compile time, we can enable compile-time checking of contracts and get a nice compiler diagnostic that the precondition has been violated (instead of a much more cryptic error saying that `n / d` is not a core constant expression). If `divide` is called at runtime, we can enable runtime checking of contracts, get a runtime diagnostic that the precondition has been violated, and avoid the undefined behaviour that evaluation of `n / d` at runtime would cause.

However, what happens if we call `divide` in a context where a variable may or may not be constant-initialised? Consider the following program:

```
int main() {
  const int i = divide(17,0);    // ???
}
```

Note that the difference to the case described in section 2.8 is that the predicate actually *is* a core constant expression and can be evaluated at compile time. The user might therefore expect that the above program should cause the compiler to issue a diagnostic that the precondition has been violated at compile time. However, that is not what our algorithm does. Trial evaluation to determine whether `divide(17,0)` is a core constant expression happens with the contract check discarded; this trial evaluation determines that the call is not a core constant expression (because of `n / d`) and delegates the call to runtime. As a result, the contract is never checked at compile time, *even if we choose the enforce semantic for compile time*, and we instead run into a contract violation at runtime. This might be surprising to some users, because when evaluated at compile time, the call to `divide` would run into a compile-time contract violation, but this contract violation is not diagnosed.

However, trial evaluation is not *actual* evaluation, so we cannot rely on the contract annotation being evaluated at compile time even when using a checked semantic at compile time). This must be so because relegating the call to runtime might be intentional, and our algorithm cannot distinguish whether this is the case. Consider:

```
int g() {   // not constexpr
  return 42;
}

constexpr int f(int i)
  pre (i > 0 || !std::is_constant_evaluated)
{
  if (i == 0)
    return g();    // f is not a core constant expression when this branch is taken

  return i;
}

int main() {
  const int i = f(0);
  return i;
}
```

In the code above, there is *no* precondition violation. The first part of the predicate, `i > 0`, evaluates to `false`, but the second part of the predicate clearly states that this is not a defect if the function is evaluated at runtime; simultaneously, if `i == 0`, the call to `f` ends up being not a core constant expression and the function `f` ends up being evaluated at runtime. If we were to check contracts during trial constant evaluation, the contract check would fail, rendering the program ill-formed if the contract is *enforce*d even though *no precondition is being violated*. This would be an undesirable outcome. The correct choice in this case is, therefore, to stick to the algorithm described in section 2.8 and to ignore contract checks during trial constant evaluation.

3 Summary

In this paper, we have explored the design space of evaluating contract annotations at compile time, one of the remaining design holes in the Contracts MVP [P2900R1]. After having analysed how C++ users might expect contract annotations to behave when evaluated at compile time, and how can we best meet their needs, we propose the following specification.

In a manifestly constant-evaluated context, contract annotations should be evaluated with one of the three semantics, *ignore*, *observe*, or *enforce*, analogous to their runtime counterparts. If, during

constant evaluation, a contract check has a *checked* semantic (*observe* or *enforce*) and the contract predicate evaluates to `false`, or the predicate cannot be evaluated at compile time because it is not a core constant expression, this is considered a contract violation and the compiler shall issue a diagnostic. Additionally, if the semantic is *enforce*, the program is ill-formed. Just like at runtime, it is implementation-defined (and thus left to the compiler to provide a selection mechanism for) which contract semantic is used for any given evaluation of a contract check.

Special rules are required for the case of evaluating contract annotations in the initialiser of a non-`constexpr` variable that may or may not be constant-initialised, such as variables with static or thread storage duration and variables of `const`-qualified integral or enumeration type. To get the correct behaviour in all cases, we propose the following algorithm. First, we conduct a trial constant evaluation *ignoring* any contract annotations that might otherwise be evaluated during that evaluation, even if these have a checked semantic at compile time; if this trial constant evaluation fails, the initialiser is not a core constant expression and will be called at runtime instead (which means the contracts will be checked at runtime as well). If however, the trial constant evaluation succeeds, we perform the actual constant initialisation, this time *including* constant evaluation of the contract annotations we ignored earlier; if any of these contract predicates are not core constant expressions at this point, or are core constant expressions that evaluate to `false`, the compiler shall treat these as contract violations during constant evaluation.

Acknowledgements

Thanks to Oliver Rosten and Gašper Ažman for their helpful feedback on an earlier version of this paper; thanks to Ville Voutilainen, Joshua Berne, Tom Honermann, Jason Merrill, and Jens Maurer for their helpful comments on the SG21 reflector thread that led to the current revision of this paper.

References

[P2437R1] Aaron Ballman. Support for `#warning`. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2437r1.pdf, 2022-01-13.

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

[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.

[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.

[P2890R1] Timur Doumler. Contracts on lambdas. https://wg21.link/p2890r1, 2023-11-06.

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

[P2900R1] Joshua Berne, Timur Doumler, and Andrzej Krzemieński. Contracts for C++. https://wg21.link/p2900r1, 2023-10-09.

[P2932R1] Joshua Berne. A Principled Approach to Open Design Questions for Contracts. https://wg21.link/p2932r1, 2023-10-04.

[P2969R0] Timur Doumler, Ville Voutilainen, and Tom Honermann. Contract checks are potentially-throwing. https://wg21.link/p2969r0, 2023-11-06.