Contract assertions on coroutines

Timur Doumler (papers@timur.audio)
Joshua Berne (jberne4@bloomberg.net)
Iain Sandoe (iain@sandoe.co.uk)
Peter Bindels (dascandy@gmail.com)

Document #:	P3387R0
Date:	2024-10-09
Project:	Programming Language C++
Audience:	SG21, EWG

Abstract

In this paper, we explore the design space for an extension to the Contracts MVP proposal [P2900R8] that allows placing function contract assertions — `pre` and `post` — on coroutines. We discuss the questions that such a proposal must answer, such as the point of evaluation of `pre` and `post` on a coroutine and the treatment of coroutine parameters in the predicate of `post`. We analyse the available solution space, based on the design proposed in [P2957R1] and exploring some additional options, formulate our design goals, and conclude with the solution that best satisfies those design goals, taking into account the fundamental design principles of both coroutines and Contracts.

Contents

1 Introduction

The C++2a Contracts proposal [P0542R5], and the Contracts proposals before it, did not take coroutines into account because they were not yet part of the C++ Standard at the time. Post-C++20, SG21 initially decided to not support `pre` and `post` on coroutines in the Contracts MVP proposal, "until we have a more complete picture of what we intend to provide" (see [P2932R3]). This remains the case in the current revision [P2900R8], which makes it ill-formed for a function `f` to have function contract assertions — `pre` and `post` — if `f` is a coroutine; it merely allows `contract_assert` inside the body of `f`.

However, there are good reasons to remove this limitation from the Contracts MVP. The benefits of adding `pre` and `post` to a function declaration also extend to coroutines, especially if we consider the fundamental design principle behind coroutines that the coroutine-ness of a function is an implementation detail. We should not give users more reasons to avoid coroutines by hampering the ability to apply `pre` and `post` to them. As pointed out in [P3173R0], coroutines are a foundational facility of modern C++ and thus a Contracts proposal should adequately address uses of contract assertions in combination with coroutines. In that paper, a major compiler vendor names support for `pre` and `post` on coroutines as one of the criteria for the viability of a Contracts MVP proposal.

Significant progress towards a solution was made in [P2957R0], the first proposal to add support for `pre` and `post` on coroutines. In this paper, the authors argue why the only viable semantics for `pre` and `post` on a coroutine declaration are that they apply to the function interface that this declaration exposes to the caller, and not to the internal coroutine aspects of the function; and that they express the preconditions and postconditions on the so-called *ramp function*, and not on the suspend and resume points of the coroutine produced by it. This paper was seen by SG21 in Kona (November 2023). The proposed solution did not gain consensus at the time due to some remaining uncertainty on the proposed design direction.

A revision of the paper, [P2957R1], contained a change that made `post` on a coroutine ill-formed (while `pre` on a coroutine would still work as proposed before), to address concerns in SG21 over how `post` as proposed relates to the coroutine body (not in any straightforward way), and how parameters would behave in `post` (a parameter may be moved-from by the ramp function even if the parameter has been declared `const` by the user). A later paper, [P3251R0], proposed the same solution, and additionally considered how contract assertions could be applied to the return channel of a coroutine by adding `pre` and `post` on the promise type without any changes to [P2900R8].

The present paper does not add any fundamental new ideas to these previous proposals; we support the design direction developed in [P2957R0] and [P2957R1]. What this paper contributes is a more exhaustive exploration of the available design space, including some options and design goals not explicitly discussed in other papers, as well as a specification strategy and formal wording for the concrete solution that best satisfies all relevant design goals and the fundamental design principles of both coroutines and Contracts.

2 Discussion

2.1 Coroutine-ness is an implementation detail

The fundamental design principle of coroutines is that whether `f` is a coroutine is an implementation detail that is not known at the declaration of `f`, or to the caller of `f`. This design principle can and should be applied to Contracts as well. If a Contracts proposal introduces new ways in which coroutines are different from other functions — such as disallowing the usage of `pre` and `post` on them — then such a design is in contradiction with how coroutines are designed and specified in

C++ today. Our design direction should therefore be that, as much as possible, coroutines should work exactly the same as other functions with respect to Contracts.

To illustrate the implementation detail nature of coroutine-ness, let us assume that `generator<T>` is a conforming coroutine return type ([dcl.fct.def.coroutine]). Now, consider the following function declaration:

```
generator<int> iota(int n);
```

This function could be implemented as a coroutine, for example as follows:

```
generator<int> iota(int n) {
  while (true)
    co_yield n++;
}
```

However, another option is to implement `iota` as a non-coroutine function that manually initialises an object of type `generator<int>` and returns it, without using any of the C++ coroutine machinery. It is possible to do this without any observable change in behaviour between the two implementations.

A third possible implementation is to implement `iota` as a non-coroutine function that wraps a coroutine `iota_coro_impl` as follows:

```
generator<int> iota(int n) {
  return iota_coro_impl(n);
}
```

In general, it is impossible to distinguish between these three implementations from the declaration or from the call site of the function. This should remain the case in the presence of `pre` and `post`. To achieve this, it is necessary to remove the current restriction in [P2900R8] that `pre` and `post` cannot be applied to a coroutine.

Consider a coroutine `f` that we wish to augment with `pre` and `post`. Note that, with the current specification in [P2900R8], any call to `f` an be indirected through a single non-coroutine function `g` with the same function signature, and any `pre` and `post` the user wishes to apply to `f` can instead be applied to `g`:

```
auto g()   // not a coroutine
  pre (/*...*/)
  post (r: /*...*/) {
  return f();   // a coroutine
}
```

Given the existence of this workaround, it seems straightforward to specify that `pre` and `post` when placed on `f` directly should behave exactly the same as they would when placed on the wrapper function `g`. However, as with any other attempt to wrap a function in C++, once additional parameters are being passed there are subtle yet important differences in behaviour that must be considered, in particular regarding odr-use of a non-reference parameter in the predicate of `post`, which we discuss in more detail in Section 2.4. Before we get there, let us consider the conceptual meaning of `pre` and `post` on a coroutine declaration in more detail.

2.2 The interface of a coroutine

The full interface of a coroutine consists of two parts. First, there is the *function interface*, that is, the function declaration that the caller sees. This interface covers the initial function call. From the caller's perspective, this function call behaves like a factory function that initiates the coroutine body and creates and returns the coroutine return object. This factory function is informally called the *ramp function*. The implementation of the ramp function is generated entirely by the compiler according to the rules specified in [dcl.fct.def.coroutine]/5 and is composed of the user-provided coroutine body, together with start-up and termination functionality, part of which is specified by customisation methods in the promise type.

From some perspectives, the coroutine has a second, extended, *coroutine interface*. This interface covers the actions of the coroutine body which might, for example, repeatedly yield values via `co_yield`, suspend activity pending some condition via `co_await`, or return a final value via `co_return`. The behaviour of this extended interface is governed both by the user-provided body and the promise and awaiter types that implement the C++ customisation points required. The extended interface is unknown and opaque to the caller, which might no longer exist when parts of the coroutine body execute.

From the design principle that coroutine-ness is an implementation detail it follows directly that the function contract assertions — `pre` and `post` — on a coroutine declaration must specify the contract of the function interface, i.e. the ramp function, as this is the only interface known at the site of the function call.

One of the design principles of the Contracts MVP ([P2900R8], Section 3.1, Design Principle 11) is that `pre` and `post` serve both caller and callee. In this case, crucially, the callee is *not* the user-provided coroutine body, but the compiler-generated ramp function: `pre` on a coroutine is an assertion on the parameters passed into the ramp function when it is called, *before* the coroutine state is constructed, and the state of the program at that point; `post` on a coroutine is an assertion on the object that the ramp function returns to the caller. On the other hand, the coroutine body does not have direct access to the return value of the ramp function;[1] it may resume execution long after `post` is checked; and it may run on another thread.

This lack of a direct connection between function declaration and user-provided function body may seem strange and unintuitive to users not familiar with coroutines; but it is a fundamental part of their design. Adding `pre` and `post` to the function declaration does not change this design and does not make things any more complex than they already are. This key insight allows us to move past “postpone until we have a more complete picture” and propose a coherent design for `pre` and `post` applied to coroutines.

In addition to contract assertions on the function interface, we could consider a novel language feature to express assertions on the extended coroutine interface, i.e., preconditions and postconditions on a coroutine’s suspension and resumption. However, such assertions could not be specified on the function declaration as the coroutine-ness of a function is not known there. We believe that most, if not all, use cases for such assertions can already be accomplished with the existing functionality in [P2900R8] by using `contract_assert` inside the coroutine body or, for contracts that hold for *any* entity that uses a given promise type, by applying `pre` and/or `post` to the various customisation functions in the promise and awaiter types. This latter approach is described in more detail in [P3251R0]. A novel language feature to support this functionality is therefore not required for a viable Contracts MVP. Note also that for any coroutine, it is possible to construct a non-coroutine function with the same observable behaviour (although that might require multithreading in some cases). For the remainder of this paper, we therefore do not discuss contract assertions on the suspend and resume points of a coroutine any further, and focus solely on contract assertions on its function interface.

[1]It is possible to provide access to the return value to the body of the coroutine, but it is tricky to do and not very common. For example, implementations of a `std::optional`-returning coroutine can make use of a special constructor in the `std::optional` type that takes a pointer to the `promise_type` and that registers itself with the promise object when the return value is returned from `get_return_object()`. Then, if the coroutine does not suspend at `initial_suspend()` and continues executing the body, the body can have access to the return-value.

2.3 Point of evaluation of `pre` and `post`

As pointed out in [P2957R1], correctly specifying the point of evaluation of `pre` and `post` needs some slight clarifications for coroutines, however, according to the mental model described in the previous section, this does not conceptually change the point of evaluation itself or compromise the equivalence between coroutines and non-coroutine functions.

[P2900R8] specifies that the precondition assertions are evaluated "immediately after function parameters are initialised and before entering the function body". For a coroutine, by "function body", we do *not* mean the coroutine body that the user wrote, but the body of the ramp function that the compiler generated. Therefore, the precondition assertions of a coroutine are evaluated *before* any of the coroutine-specific events that happen in the ramp function, such as creating a copy of the parameters, allocating the coroutine state, initialising the promise object, etc.

[P2900R8] further specifies that the postcondition assertions are evaluated "after the return value has been initialised and local automatic variables have been destroyed but prior to the destruction of function parameters". Again, for a coroutine, this specification should be applied to the body of the ramp function and not the coroutine body. This means that the "return value" is that of the ramp function, *not* (for example) values yielded by the coroutine; its type is the declared return type of the coroutine; and there are no "local variables" since the local variables of the coroutine body are not in the scope of the ramp function and are not destroyed when the latter returns.

2.4 odr-using parameters in `post`

In [P2900R8], in order to odr-use a non-reference parameter in the predicate of `post`, it has to be declared `const` on every declaration of the function. This allows us to reason about the parameter value not having been modified between the function call and the evaluation of that function's postcondition assertions, which is a prerequisite for writing meaningful postcondition assertions on such a parameter (see [P2900R8], Section 3.4.4). As it turns out, there is one implementation detail of coroutines that is unobservable caller-side today but leads to an interaction with the above rule if we wish to allow `post` on coroutines.

As part of the standard-mandated activity of the ramp, a copy of each parameter may be created in the coroutine state, and the value of each parameter moved into that copy ([dcl.fct.def.coroutine]/13), potentially leaving the original parameter object in a moved-from state. This is necessary to make the parameters accessible inside the coroutine body, whose lifetime can extend far beyond the ramp function returning. Notably, a parameter may be moved from in this fashion even if it is a non-reference parameter declared `const` by the user. The ramp function effectively removes the top-level `const` from the parameter, and then modifies that parameter by using it to move-construct the copy. Put another way, in a coroutine, a parameter object is never actually `const`, even if declared as such in the function declaration.

This specification may seem strange, but does not cause issues in C++ today because the moved-from parameter values are not exposed to the user in any way. Such a ramp function implementation is in fact fully consistent with the design principle that coroutine-ness is an implementation detail. Consider the following declaration of a function `f`:

```
awaitable<int> f(const Widget w);
```

Assuming that `awaitable<int>` is a valid coroutine return type, this function could be implemented either as a coroutine or as a non-coroutine. Note that according to [dcl.fct]/4, the following declaration declares the exact same function as the previous one:

```
awaitable<int> f(Widget w);
```

Note further that while the function `f` has the same type in both declarations, the parameter `w` does not: it is `const` in the first declaration, but non-`const` in the second.

Now, as an implementation detail, the user could choose to provide a non-coroutine definition of `f` that modifies `w` in its body:

```
// f.h
awaitable<int> f(const Widget w);

// f.cpp
awaitable<int> f(Widget w) {
  Widget w_copy = std::move(w); // OK
  // some other code; not a coroutine
}
```

Note that the user *must* remove `const` from the parameter declaration on the definition of `f`, because otherwise, any attempt to move from `w` or otherwise modify `w` in the body of `f` cannot work because it would either not compile (if no `const_cast` is used) or be undefined behaviour as per [dcl.type.cv]/4. This is fine; it is possible to declare `w` is non-`const` on the definition even if an earlier declaration of `f` declares it as `const`.

With the above implementation of `f`, according to the rules specified by [P2900R8] one would not be able to odr-use `w` in the predicate of a `post` that applies to `f`, because for that to work, `w` needs to be declared `const` on *all* declarations of `f`.

Now, if the user instead chooses to provide a definition of `f` that makes `f` a coroutine, then the implementation of `f` will not actually be the function body that the user wrote, but instead the compiler-generated ramp function. Since this compiler-generated ramp function modifies the parameter object, it will effectively have a non-`const Widget` parameter, just like the last implementation of `f` above. In other words, the ramp function behaves as if its defining declaration was rewritten to have any top-level *cv*-qualifiers stripped from parameter declarations (and as we will see in Section 6, our proposed wording includes a clarification of this reality that we intend to see adopted).

Considering the above, the question that any viable Contracts proposal needs to answer — in a consistent and user-friendly way — is how to handle the case of a non-reference parameter being used in the predicate of `post` on a coroutine. On the one hand, if that non-reference parameter is declared `const` on *all* declarations of `f`, including the defining declaration, it would be eligible for odr-use in the predicate of post under the current rules in [P2900R8], which assume that `f` is not a coroutine. On the other hand, as we saw above, if `f` is a coroutine, the parameter is never actually `const`, and therefore the reasoning behind the current rules in [P2900R8] does not apply.

Note that reference parameters are not affected by any of the above. For reference parameters, there is no assumption that the underlying object will not be modified between the function call and the evaluation of that function's postcondition assertions,[2] and therefore the current rule that reference parameters can be freely used in the predicate of `post` can be applied without modifications to coroutines.

3 The solution space

We are aware of the following potential solutions for how `pre` and `post` should behave on a coroutine, all of which have been formally or informally proposed by a WG21 member at some point:

1. Do not allow `pre` or `post` on coroutines at all (status quo in [P2900R8]);
2. Allow `pre` on coroutines, but not `post` (first proposed in [P2957R1]);

[2] Note that this is true even for `const` reference parameters: the implementation of a function could `const_cast` away the `const` and modify the object through the resulting non-`const` reference, which is fine as long as the underlying original object has not been declared `const`.

3. Allow `post` on coroutines, but do not allow odr-using non-reference parameters in its predicate (first mentioned in [P2957R1] as an alternative);
4. Allow odr-using non-reference parameters in the predicate of `post`; an *id-expression* naming a non-reference parameter refers to the copy made for the coroutine state;
5. Allow odr-using non-reference parameters in the predicate of `post`; an *id-expression* naming a non-reference parameter refers to the original object, and:
   a. the parameter copy made in the ramp function is copy-constructed instead of move-constructed;
   b. is ill-formed if the parameter type has a non-trivial move constructor (i.e., the parameter can have a moved-from value);
   c. no further provision is added, i.e. the *id-expression* may refer to a moved-from value, even if the parameter is declared `const` by the user (first proposed in [P2957R0]).

All of the above solutions handle the question of odr-using non-reference parameters in `post` in different ways. In order to choose the correct solution, we need to formulate our design goals and determine which of these solutions best satisfies these design goals. This analysis is performed in the following section.

Some of the above solutions have minor variations we will not discuss explicitly (for example, 5b could be modified to make `post` ill-formed only for parameters of non-trivial type). We believe that the above selection provides an adequate sampling of the entire solution space for the purposes of our analysis.

4 Design goals

4.1 Allow `pre` and `post` on coroutines

The simplest and most practical design goal of this proposal is to allow applying `pre` and `post` to the declaration of a coroutine. This would enable us to use contract assertions to specify preconditions on the parameters passed into the coroutine, postconditions on the object that this function call returns to the caller, as well as preconditions and postconditions on other reachable program state at the point when the coroutine is called and when it returns.

Such assertions can be useful for the same reasons that assertions on any other function can be useful: they allow enhancing the program with configurable checks of its correctness, thereby helping to diagnose and fix program defects.

Solution 1 does not allow either `pre` or `post` on coroutines, while Solution 2 does not allow `post`; all other solutions satisfy this design goal in some form.

4.2 Treat coroutine-ness as an implementation detail

Any solution should be consistent with the fundamental design principle of coroutines that whether a function `f` is a coroutine is an implementation detail that is not known at the declaration of `f`, or at a call to `f`.

Solutions 4 and 5c violate this principle because they make it so that the user of a function will evaluate a `post` differently depending on whether the function is a coroutine. They do so by exposing the parameter copies internal to the coroutine state (Solution 4) or the moved-from values of the original parameter objects (Solution 5c).

The other solutions do not directly violate this principle. Solutions 1, 2, 3, and 5b does not expose coroutine-ness in the declaration, but make the definition ill-formed under certain circumstances if that definition happens to make the function a coroutine. But that does not let the callsite of a function detect whether or not the function is a coroutine, i.e. from the caller's perspective it is still just a normal function; the compiler error occurs in the function definition.

One way to think about this is that solutions 1, 2, 3, and 5b add a few more cases to the language in which a function with a given declaration cannot be implemented as a coroutine, but can only be implemented as a non-coroutine. Note that there are already many such cases in the language today, for example a function that has a non-coroutine-compatible return type, or a function that has a parameter of non-copyable non-movable type. We do not consider any of these cases to be exposing the coroutine-ness of a function.

One could perhaps argue that solutions 1, 2, and 5b *indirectly* fail to fully satisfy the implementation-detail principle: without looking at the definition of a function, one could reason that it must be a coroutine if adding certain kinds of function contract assertions makes the program ill-formed due to an error that is *specific* to coroutines — i.e., nothing other than making the function a coroutine would make the program fail in this particular way, even if the failure is not apparent at the callsite. However, with solution 3, no such reasoning is possible, as the coroutine-ness of a function is not the only possible reason why odr-using a non-reference parameter in `post` would make the program ill-formed.[3]

4.3 Do not expose moved-from parameters in `post`

On the one hand, exposing the moved-from parameter value in `post` is the "honest" choice, as it simply reflects what is going on under the hood. On the other hand, it exposes an implementation detail of C++ coroutines to the caller that is currently not being exposed. Apart from violating the fundamental design principle of coroutines (see previous section), observing such moved-from values when odr-using `const` parameters is likely to be surprising to the user, and can lead to unexpected behaviour and unintended bugs that will be very difficult to diagnose and fix. It should therefore be a design goal to avoid this, guided by an underlying design principle that Contracts and coroutines should not be unnecessarily user-hostile.

Furthermore, it does not seem useful to be able to write a postcondition on a non-reference parameter that can be moved-from, as we cannot reason about the meaning of such a postcondition, in the same way in which we cannot reason about the meaning of a postcondition on a non-reference parameter that is not declared `const` by the user (which is the reason why [P2900R8] makes such postcondition assertions ill-formed). Overall, it is therefore a reasonable design goal to avoid this scenario.

Solution 5c directly violates this design goal, as it makes parameter names in `post` refer to moved-from objects. Whether solution 5b violates this goal as well is a matter of interpretation. On the one hand, it *morally* does, as the parameter names in `post` refer to moved-from objects as well; on the other hand, such a `post` will only compile for types for which the move operation is equivalent to a copy and no actual moved-from values exist.

[3]As discussed in Section 2.4, any non-coroutine function that has a non-reference parameter declared `const` on its first declaration could be implemented such that the same parameter is not declared `const` on a subsequent declaration (which may be a definition). With the rules in [P2900R8] today, this would render render the program ill-formed at the place of that subsequent declaration if that parameter is used in `post`, even if that `post` syntactically appears only on the first declaration.

4.4 Satisfy the Contracts Prime Directive

The most fundamendal design principle of the Contracts MVP, the so-called Contracts Prime Directive ([P2900R8], Section 3.1, Principle 1) states that adding `pre` or `post` to an existing program should not alter the correctness of that program, as this would undermine the purpose of contract assertions — instead of checking the correctness of the program the user wrote, they would check the correctness of some other program, potentially leading to so-called "Heisenbugs" as well as other problems.

A corollary of this fundamental principle is that adding `pre` or `post` to an existing program should not change the compile-time or run-time semantics of that program (see [P2900R8], Section 3.1, Principles 2 and 3).

Solution 4 violates this principle. Note that the coroutine state, along with its copies of the function parameters, can be destroyed on initial suspend, which happens before control is returned to the caller, and thus before `post` is checked. Therefore, making parameters in `post` refer to the parameter copy would require changes to the semantics of the coroutine state to extend its lifetime accordingly, thereby violating the Contracts Prime Directive. Furthermore, such an approach would require heavy lifting in both wording and implementation changes, as we would have to change the nesting of lifetimes in significant ways; but without such changes, there would be no point at which the parameter copies and the return value necessarily exist at the same time, and therefore no point at which `post` could be checked.

Solution 5a violates the Contracts Prime Directive in a different way. With this solution, adding `post` to a coroutine would incur an additional copy of each non-reference parameter odr-used in `post`, even if the *ignore* semantic is used and predicate is never checked. [P2900R8] has been carefully designed to avoid such scenarios. This is the reason why, for example, the program is ill-formed if a contract assertion would trigger an implicit lambda capture. To avoid incurring an additional copy by merely adding a (potentially unchecked) contract assertion, we would have to perform an additional copy of *each* parameter, on *any* coroutine, regardless whether contract assertions are being used. Such an approach would violate the "don't pay for what you don't use" principle of C++, and would introduce performance regressions to existing code.

4.5 Do not introduce additional inconsistencies between `pre` and `post`

A well-designed Contracts feature will naturally compose with other C++ language features, including coroutines. We should avoid adding more complexity to make Contracts and coroutines work together. In particular, we should avoid introducing new inconsistencies between `pre` and `post` that do not exist in the current design, as that could be surprising to the user and hinder effective usage and wider adoption of Contracts.

Arguably, Solution 2 violates this design goal, because being able to apply `pre`, but not `post` to a function would be a new inconsistency and arguably surprising.

Solution 4 also violates this design goal. Currently, if a function has a `pre` and a `post`, and their predicates odr-use the same parameter, then the corresponding *id-expression* will refer to the same parameter object with the same address. Solution 4 breaks this symmetry, which will be unexpected to most users.

Finally, Solution 5c violates this design goal as well. Currently, if a function has a `pre` and a `post`, and their predicates odr-use the same non-reference parameter, that parameter must be `const` and therefore `pre` and `post` are guaranteed to see the same value for that parameter (at least, if the parameter type has been implemented with `const`-correctness). Solution 4 however would lead to potentially different values being observed in `pre` and `post`, respectively.

Note that Solution 3 does not violate this design goal because it does not introduce a new inconsistency. As already discussed above, one of the necessary limitations of `post`, compared to `pre`, is that it is ill-formed to odr-use non-`const` non-reference parameters in the predicate of `post`. However, due to the nature of how coroutines are implemented by the compiler, the parameters of a coroutine are never `const`, even if declared `const` everywhere by the user. Therefore, implementing a function as a coroutine is just another way of making a non-reference parameter non-`const`.

4.6 Do not facilitate remote code breakage

Our design for how Contracts and coroutines work together should not facilitate situations that can lead to unintended, remote code breakage. Solution 5b violates this design goal because it would create a new dependency between the trivial movability of a type and the ability to use it as a function parameter.

For example, if the user adds a move constructor to a type that previously only had a copy constructor (something that we explicitly encourage people to do to modernise their code!), this will break clients who happen to use this type as a coroutine parameter. Likewise, if there is a struct with an `int` and a `float` data member, and the user adds a `std::string` data member, this will also break clients who happen to use this type as a coroutine parameter. Such breakage would happen for highly non-obvious reasons, does not have a good workaround — at least not until we get postcondition captures ([P2461R1], [P3098R0]) as a post-MVP extensions — and therefore seems user-hostile.

4.7 Support caller-side checking of `pre` and `post`

The Contracts MVP has been designed from the start to accommodate a wide range of implementation strategies and usage scenarios. In particular, it allows implementations to check precondition and postcondition assertions caller-side and/or callee-side, and enables the two translation units involved to make the decision on contract evaluation semantics independently, which has important use cases ([P2751R1], [P3228R1], [P3119R1], [P3267R1], [P3321R0]).

In particular, if one has deployed a pre-built library with all contracts ignored, and the user of that pre-built library wishes to verify that it is working correctly when used from particular other translation units, enabling caller-side checking of that library's precondition and postcondition assertions can be very useful, for example to validate a new version of such a library before integrating it into the shipping product. In general, any time we cross the boundary between translation units, it is very helpful (and currently intentionally supported by [P2900R8]) to have both sides of that boundary be able to enable contract checks.

However, in order to be able to enable caller-side checking of `pre` and `post`, the predicate cannot depend on anything that is not accessible caller-side and only known callee-side. For coroutines in particular, it means that `post` cannot refer to the copies of parameter objects that belong to the coroutine state. Solution 4 is therefore not compatible with caller-side checking of `post`, even though it does allow caller-side checking of `pre`.

Admittedly, we are aware of fewer use cases for caller-side checking of `post` than for `pre`. The former seems useful but overall less important to support than the latter. It might also be unimplementable on some platforms.[4] Whether supporting caller-side checking of `post` should be considered a design goal therefore depends on the intended use cases; if we strive to enable the widest range of known use cases for Contracts ("design for the multiverse"), then it arguably should be.

[4]Notably, the Microsoft ABI performs argument destruction callee-side, not caller-side. Since postcondition assertions are specified in [P2900R8] to be evaluated before argument destruction happens, they cannot be checked caller-side on this platform without an ABI break.

5 Proposed solution

Having formulated our design goals, we can now construct a decision matrix that visualises which possible solutions in the available design space satisfy which design goals (question marks represent cases where one could argue one way or the other):

	1	2	3	4	5a	5b	5c
Allow `pre` on coroutines	❌	✅	✅	✅	✅	✅	✅
Allow `post` on coroutines	❌	❌	✅	✅	✅	✅	✅
Treat coroutine-ness as implementation detail	❓	❓	✅	❌	✅	❓	❌
Do not expose moved-from parameters	✅	✅	✅	✅	✅	❓	❌
Satisfy Contracts Prime Directive	✅	✅	✅	❌	❌	✅	✅
Do not make `pre` and `post` more inconsistent	✅	❌	✅	❌	✅	✅	❌
Do not facilitate remote code breakage	✅	✅	✅	✅	✅	❌	✅
Support caller-side checking of `pre` and `post`	✅	✅	✅	❌	✅	✅	✅

The above decision matrix visualises that the only solution that satisfies all our design goals, and furthermore the only possible solution that is compatible with the fundamental design principles of both coroutines (Section 4.2) and Contracts (Section 4.4), is Solution 3: allow `pre` and `post` on coroutines, but make it ill-formed to odr-use a parameter of a coroutine in the predicate of `post` (even if that parameter is declared `const` by the user).

Therefore, we propose to adopt Solution 3 for the Contracts MVP. It is the correct solution for contract assertions on coroutines and provides the most value for the C++ language and its users. Furthermore, as we will see in Section 6, Solution 3 can be specified on top of the existing Contracts and coroutines wording in an elegant way: it reduces to essentially just clarifications in the existing coroutines wording plus the actual removal of the prohibition to place `pre` and `post` on a coroutine. The desired behaviour regarding point of evaluation, non-reference parameters, etc. just automatically falls out from those clarifications. This is another strong hint that it is indeed the most natural and consistent composition of the Contracts and coroutines features.

Note that if we adopt Solution 3, but for whatever reason the user really must odr-use non-reference parameters in the predicate of `post`, a workaround exists: one can wrap the coroutine into a non-coroutine wrapper as shown at the end of Section 2.1. With such a wrapper, there is no question as to whether `post` applies to the original or copy of the parameter, or whether that parameter might have been moved from, as only the original parameter object is visible to the wrapper, and that parameter object is not modifiable. It is then up to the user how to pass that parameter on to the wrapped coroutine (e.g., by copy).

The situation will become even simpler once we get postcondition captures ([P2461R1], [P3098R0]) as a post-MVP extension. Postcondition captures will allow the user to explicitly capture parameters when a function is called, by copy or by reference, with a syntax analogous to lambda captures, and use these captured parameters later in the predicate of `post` when the function returns. This will work even if the parameter in question is non-`const`. With this post-MVP extension, the need for a wrapper will go away completely.

6 Proposed wording

Our wording strategy is to not modify the normative Contracts wording in [P2900R8] at all other than removing of the prohibition for `pre` and `post` to apply to a coroutine, consistent with the principle that coroutine-ness is an implementation detail. Instead, we clarify the wording in [dcl.fct.def.coroutine] in the necessary places such that the behaviour of `pre` and `post` on the declaration of a function that happens to be implemented as a coroutine simply follows from the existing specification.

The evaluation of both precondition and postcondition assertions should be semantically equivalent whether a function actually *is* a coroutine or not. We believe this is the case in the current intent of the coroutine wording, but we believe clarifications are needed for that purpose:

— Precondition assertions should be evaluated after the normal function parameters are initialised (just as with any other function call) and prior to any coroutine-specific evaluations such as allocating space for the coroutine state (using a possibly user-defined coroutine state frame allocator[5]), making copies of the original parameter objects and moving their values into these copies, or evaluating anything in the replacement function body defined in [dcl.fct.def.coroutine] paragraph 5. To clarify this, we introduce the term *replacement body* so we can directly refer to it, and add "as part of the replacement body" to descriptions of both the allocating function invocation and the making of copies of the parameters.

— Postconditions should be evaluated when the coroutine returns to its caller (but not a resumer). The return to the caller can correspond to the first suspension[6] of a coroutine; all other suspension points return to an alternate resumer (i.e. not to the ramp). Returns that correspond to a suspension do *not* destroy local variables because suspending is not considered leaving the relevant scope within the coroutine. To achieve the effects we want, we recommend a minor rearranging of the wording in [P2900R8] to describe postcondition evaluation along with precondition evaluation in [expr.call] and not tie it to the return statement (leaving just a note in the [stmt.return] and in [dcl.fct.def.coroutine] describing when postconditions are expected to be evaluated.)

— To make it ill-formed to odr-use parameters in the predicate of `post`, we need to clarify that a coroutine does not just rewrite its body but also rewrites its declaration to not have top-level *cv*-qualifiers on its parameter declarations (though the copies do retain those *cv*-qualifiers). This is already implied by the existing wording for the generated ramp function body, but is currently not specified clearly enough and contains a misleading note. We suggest removal of the note as well as other drive-by fixes of the preceding wording.

The proposed wording is relative to [P2900R8].

Modify [dcl.contract.func], paragraph 6:

> A ~~coroutine ([dcl.fct.def.coroutine]),~~ a deleted function ([dcl.fct.def.delete])~~,~~ or a function defaulted on its first declaration ([dcl.fct.def.default]) may not have a *function-contract-specifier-seq*.

Modify [dcl.fct.def.coroutine], paragraph 5:

> A coroutine behaves as if the top-level *cv*-qualifiers in all *parameter-declarations* in the declarator of its *function-definition* were removed, and its *function-body* were replaced by the following *replacement body*:
>
> ```
> {
>   promise-type promise promise-constructor-arguments ;
>   [...]
> ```

Modify [dcl.fct.def.coroutine], paragraph 9:

[5]Note that the coroutine state frame allocator has access to the original parameter objects

[6]It need not correspond to a suspension in general — the coroutine could run synchronously to completion, or be destroyed in response to some interaction before reaching any active suspension.

An implementation may need to allocate additional storage for a coroutine. This storage is known as the *coroutine state* and is obtained by calling a non-array allocation function ([basic.stc.dynamic.allocation]) as part of the replacement body. The allocation function's name is looked up by searching for it in the scope of the promise type.

— If the search finds any declarations, overload resolution is performed on a function call created by assembling an argument list. The first argument is the amount of space requested, and is a prvalue of type `std::size_t`. The lvalues $\mathtt{p}_1 \ldots \mathtt{p}_n$ with their original *cv*-qualifiers are the successive arguments. If no viable function is found ([over.match.viable]), overload resolution is performed again on a function call created by passing just the amount of space required as a prvalue of type `std::size_t`.

Modify [dcl.fct.def.coroutine], paragraph 13:

When a coroutine is invoked, ~~after initializing its parameters ([expr.call])~~at the beginning of the replacement body, a copy is created for each coroutine parameter. For a parameter whose original declaration was of type *cv* `T`:

— If `T` is a reference type, the copy is a reference of type *cv* `T` bound to the same object as the parameter,

— Otherwise, the copy is a variable of type *cv* `T` with automatic storage duration that is direct-initialized from an xvalue of type `T` referring to the parameter. [*Note:* An identifier that names one of these parameters refers to the created copy and not the original paramter ([expr.prim.id.unqual]) — *end note*]

~~[*Note:* An original parameter object is never a const or volatile object ([basic.type.qualifier]). — *end note*]~~

Modify [intro.execution], paragraph 11:

[11] When invoking a function `f` (whether or not the function is inline), every argument expression and the postfix expression designating `f` are sequenced before every precondition assertion of ~~`f`~~the function call ([expr.call]), which in turn are sequenced before every expression or statement in the body of `f`, which in turn are sequenced before every postcondition assertion of the function call. Several contexts in C++ cause evaluation of a function call, even though no corresponding function call syntax appears in the translation unit.

Add a new paragraph after [expr.call], paragraph 8:

When control is transferred back to this function call ([stmt.return], [expr.await]), all postcondition assertions of the function call are evaluated in sequence ([dcl.contract.func]). [*Note:* This in turn is sequenced before the destruction of any function parameters. — *end note*]

Modify the new paragraph added after [stmt.return], paragraph 3:

~~All postcondition assertions ([dcl.contract.func]) of the function call ([expr.call]) are evaluated in sequence. The destruction of all local variables within the function body is sequenced before the evaluation of any postcondition assertions.~~

[*Note:* Postcondition assertions of the function call ([expr.call]) are evaluated in sequence after the destruction of any local variables in scopes exited by the return statement, and are, in turn, sequenced before the destruction of function parameters. — *end note*]

Acknowledgements

Thanks to Andrzej Krzemieński and Lewis Baker for their thorough review of the paper; to Gabriel Dos Reis, Gor Nishanov, and Lisa Lippincott for fruitful discussions of the proposal; and to Ville Voutilainen for his helpful remarks regarding caller-side checking.

Bibliography

[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.

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

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

[P2900R8] Joshua Berne, Timur Doumler, and Andrzej Krzemieński. Contracts for C++. https://wg21.link/p2900r8, 2024-07-26.

[P2932R3] Joshua Berne. A Principled Approach to Open Design Questions for Contracts. https://wg21.link/p2932r3, 2024-01-15.

[P2957R0] Andrzej Krzemieński and Iain Sandoe. Contracts and coroutines. https://wg21.link/p2957r0, 2023-08-15.

[P2957R1] Andrzej Krzemieński and Iain Sandoe. Contracts and coroutines. https://wg21.link/p2957r1, 2024-01-13.

[P3098R0] Timur Doumler, Gašper Ažman, and Joshua Berne. Contracts for C++: Postcondition captures. Manuscript in preparation.

[P3119R1] Joshua Berne. Tokyo Technical Fixes to Contracts. https://wg21.link/p3119r1, 2024-05-09.

[P3173R0] Gabriel Dos Reis. P2900R6 May Be Minimal, but It Is Not Viable. https://wg21.link/p3173r0, 2024-02-15.

[P3228R1] Timur Doumler. Revisiting side effects, elision, and duplication of contract predicate evaluations. https://wg21.link/p3228r1, 2024-05-21.

[P3251R0] Peter Bindels. C++ Contracts and Coroutines. https://wg21.link/p3251r0, 2024-04-23.

[P3267R1] Peter Bindels and Tom Honermann. C++ contracts implementation strategies. https://wg21.link/p3267r1, 2024-05-22.

[P3321R0] Joshua Berne. Contracts Interaction With Tooling. https://wg21.link/p3321r0, 2024-07-12.