Postconditions odr-using a parameter modified in an overriding function

Timur Doumler (papers@timur.audio)
Joshua Berne (jberne4@bloomberg.net)

Document #: P3484R2
Date: 2024-11-14
Project: Programming Language C++
Audience: SG21, EWG

Abstract

This paper considers the case where an overridden function odr-uses a non-reference function parameter in its postcondition assertion, and then an overriding function drops `const` on the declaration of that parameter, rendering the postcondition assertion in the overridden function significantly less useful. We discuss several possible directions for how to address this problem in the Contracts MVP [P2900R10] and propose the ones we consider viable.

This paper is the first part of a trilogy of papers dealing with known issues in the Contracts MVP [P2900R10] regarding postconditions odr-using non-reference function parameters:

— [D3484R2] Postconditions odr-using a parameter modified in an overriding function;

— [P3487R0] Postconditions odr-using a parameter that may be passed in registers;

— [P3489R0] Postconditions odr-using a parameter of dependent type.

These issues should be considered together, and ideally resolved in a consistent way.

1 Background

[P2900R10] Section 3.4.4 specifies that if a non-reference function parameter is odr-used in a postcondition assertion, that function parameter must be `const` on *all* declarations of that function, otherwise the program is ill-formed. Importantly, this requirement on all declarations of the function includes its *defining* declaration.

The rationale for this rule is that it allows the assumption that a function will not modify the value of a non-reference parameter odr-used in a postcondition before that postcondition is checked. Human readers, compilers, and static analysis tools can rely on that property. Allowing such modification would render that check significantly less useful as it would be impossible to reason about the postcondition without taking into account what happens in the function definition, and would lead to surprising failures. Consider the following function declaration:

```
// Returns: a number guaranteed to be greater or equal to the number passed in
Integer f(const Integer i) post (r: r >= i);
```

Now, if we could drop the `const` on the defining declaration of `f`, we could write a dummy implementation of `f` as follows:

```
Integer f(Integer i) {  // i is not const
  i = 0;
  return i;
}
```

Such an implementation blatantly fails to satisfy the postcondition of `f`, and is therefore incorrect. However, this defect cannot be caught by checking the postcondition assertion of `f`, because its predicate is now `0 >= 0` which evaluates to `true`:[1]

```
void test() {
  Integer j = f(3); // no contract violation detected
  // postcondition does not hold — j is now 0, which is smaller than 3!
}
```

In the above scenario, the postcondition fails to detect the bug because the implementation of `f` explicitly modifies the parameter. A possibly even more dangerous variant of this scenario occurs when such modifications are performed *implicitly*. For example, if the parameter is not `const` on the defining declaration of `f`, it may be returned by value, which will perform an implicit move, i.e. the postcondition assertion would observe the parameter object in a moved-from state:

```
std::string g(std::string p) post (r: starts_with(p)) {
  return p;
}
```

This would lead to a spurious failure of the postcondition check for no reason obvious to the reader of the above code. To avoid such failures, [P2900R10] makes the above implementation of `f` ill-formed: `const` cannot be dropped from any function parameter odr-used in a postcondition assertion.

Further, [P2900R10] Section 3.4.4 specifies that if a non-reference function parameter is odr-used in a postcondition assertion, and that function is implemented as a coroutine, the program is ill-formed, *even if* all such function parameters have been declared `const` by the user. The rationale for this rule is that a coroutine will move-from its function parameters to initialise the parameter copies in the coroutine frame. Therefore, the function parameters of a coroutine are effectively never `const`, even if declared as such by the user (see [P3387R0]). This case is thus notionally similar to the previous case where `const` was dropped from the parameter declaration in the implementation.

Together, these rules provide a static guarantee that, if a parameter is odr-used in a postcondition assertion, it *will not be modified* between the call to a function and the evaluation of its postcondition assertions. Another imporant benefit of this design is that static analysis can now reason about the value of a function parameter in a postcondition without having to take into account the function definition, which will often be either too complicated to analyse or inaccessible at the call site.

Reference parameters are excluded from the above restrictions because references refer to objects declared elsewhere, and the value of those objects when the function call completes are still relevant and available to the caller because those objects can still have outside references known to the caller. There is no expectation that the value will remain unchanged after the function body has executed, and many functions that pass an object through a modifiable reference do so with the exact intention

[1] There is an additional complication if the parameter type has at least one eligible copy or move constructor and each such constructor is trivial, and a trivial or deleted destructor, which would be the case in the example above if the parameter type were `int`. In that case, the parameter would be eligible to be passed via registers. SG21 chose option R7 from [P3487R0], which allows a postcondition check to observe the pre-temporary copy version of a parameter object passed via registers. This is required to enable caller-side checking of postconditions and make contract checks on virtual function calls as specified by [P2900R10] implementable. It follows that, if the code above were allowed, the postcondition check could actually observe either the value `3` or the value `0` for the parameter `i`, depending on inlining, optimisation flags, etc. To separate these concerns, we will use parameters and return values of type `Integer` throughout this paper, where that is a user-defined type that wraps an `int`, with all of the usual operator overloads, and most importantly with a user-provided non-trivial copy constructor.

of modifying that object; therefore, such parameters would not be used in a postcondition assertion with that expectation in mind.

2 The problem

Let us now slightly modify the example above by making `f` a virtual function:

```
struct Base {
  virtual Integer f(const Integer i) post (r: r >= i);
};
```

Note that if we override a virtual function, C++ allows dropping `const` from the parameter declaration in the overriding function, and [P2900R10] currently does not have any provision to make such an override ill-formed:

```
struct Derived : Base {
  Integer f(Integer i) override; // OK
};
```

This means that we can implement `Derived::f` such that it modifies the value of the parameter:

```
Integer Derived::f(Integer i) {
  i = 0;
  return i;
};
```

[P2900R10] Section 3.5.3 specifies the semantics of precondition and postcondition assertions on virtual functions: in a virtual function call, the function contract assertions of both the statically called function `Base::f` and the final overrider `Derived::f` are checked (see [P3097R0] for discussion). However, if we now call `Derived::f` through a reference to `Base`, we are in for an unpleasant surprise:

```
void test(Base& b) {
  Integer j = b.f(3);   // no contract violation detected
  // precondition does not hold — j is now 0, which is smaller than 3!
}

int main() {
  Derived d;
  test(d);
}
```

In the program above, even if the postcondition assertion of `Base::f` is checked, the fact that the implementation of `Derived::f` does *not* satisfy the postcondition of `Base::f` is not caught, because the parameter `i` has been modified in `Derived::f`, rendering the postcondition assertion of `Base::f` meaningless.

The more dangerous variant of this scenario that a parameter might be modified *implicitly*, for example moved-from when returned by value, is also possible and well-formed:

```
struct Base2 {
  virtual std::string g(const std::string p) post(r : r.starts_with(p));
};

class Derived2 : public Base2 {
  std::string f(std::string p) override {
    return p;
  }
};
```

In the above example, returning `p` from `Derived2::f` results in an implicit move from `p`, leading to the postcondition being checked with `p` being in a moved-from state and thus spuriously failing.

The specification in [P2900R10] Section 3.4.4 that requires `const` on all declarations including the definition — which was added to the design that evolved into [P2900R10] long ago by SG21 — is meant to prevent such scenarios, however it fails to do so for the override case. This hole in the design needs to be plugged before the Contracts MVP can progress further.

The challenge here is that, unlike in other cases that may lead to a `const` parameter being modified in the function body (hen `const` is dropped on a subsequent declaration of the *same* function, or when the function is implemented as a coroutine — which are both ill-formed in [P2900R10]), in this case `Base::f` does not and cannot know whether there are any functions overriding it, or how the corresponding parameter in those overriding functions is declared.

3 Possible solutions

We are aware of six possible approaches to dealing with this problem. These are, from most to least restrictive:

V1. Disallow odr-using any non-reference function parameter in a postcondition assertion that applies to a virtual function, regardless of whether that parameter is declared `const`, unless that function is marked `final` or is a member function of a class marked `final`.

V2. Require that if a non-reference parameter is odr-used in a postcondition assertion on a virtual function, that parameter must also be declared `const` in every declaration of every overriding function.

V3. Require that if a non-reference parameter is odr-used in a postcondition assertion on a virtual function, that parameter must also be declared `const` in the definition of every overriding function. The `const` may still be dropped in any non-defining declarations of any overriding functions.

V4. Require that if a non-reference parameter is odr-used in a postcondition assertion on *any* function, that parameter be declared `const` in the definition of that function and any overriding functions. The `const` may be dropped in any non-defining declarations of that function (which is a relaxation of the current rule in [P2900R10]) and any overriding functions.

V5. Allow overriding functions to drop `const` from a non-reference function parameter, with no special provision, i.e., if an overriding function modifies that parameter, "you get what you get" (in the virtual function call above, which invokes `Derived::f`, the postcondition check on `Base::f` will succeed even though `Derived::f` does not satisfy the postcondition of `Base::f`).

V6. Allow overriding functions to drop `const` from a non-reference function parameter, but make it undefined behaviour to actually modify a parameter object in an overriding function if that parameter is a non-reference parameter declared `const` in an overridden function.

Below we discuss the tradeoffs of each option. We enumerated the options with a "V" prefix (for "virtual"), to distinguish them from the options from [P3487R0] that have an "R" prefix (for "registers") and the options from [P3489R0] that have a "D" prefix (for "dependent").

Note that the options R1, R2, and R3 from [P3487R0], even though proposed in the context of a different problem, are also possible alternatives to the options V1 — V6 for the problem described in this paper. They remove the problem by removing the entire feature: R1 proposes to remove postcondition assertions from [P2900R10] entirely, R2 proposes to remove the ability of postcondition assertions on *any* function to odr-use *any* parameters, and R3 proposes to remove the ability of postcondition assertions on any function to odr-use any *non-reference* parameters. However, all three of these rather drastic options were already rejected by SG21 when we reviewed and polled [P3487R0], at which point the problem described in this paper was already known. For this reason, we do not consider the options R1, R2, and R3 from [P3487R0] any further.

Option V1

Option V1 would prevent the bug in the example above and is the most conservative choice. This choice is consistent with the choice we made for postcondition assertions on coroutines in [P2900R10]: if a function odr-uses a non-reference parameter in its postcondition, and that parameter is declared `const`, but there might be some other reason why the implementation of the function may modify that parameter anyway, the program is ill-formed. One such reason is that the function is a coroutine and thus the parameter will be modified by the underlying coroutine machinery. Another such reason is that the function is a non-`final` virtual function and thus an overriding function could modify the parameter.

Just like in the coroutine case, for non-`final` virtual functions the workaround would be to use a postcondition capture, once this post-MVP feature becomes available (see [P3098R0]):

```
struct Base {
  virtual Integer f(const Integer i) post (r: r >= i);      // error: cannot odr-use
};                                                          // parameter i here

struct Base {
  virtual Integer f(const Integer i) post [i] (r: r >= i); // OK: explicitly capturing
};                                                          // parameter i by copy
```

However, Option V1 has several downsides.

First, having to capture any parameter in order to odr-use it in the postcondition assertion means that we have to pay the cost of the copy. For coroutines, making that copy is the *only* way to get access to the pre-moved-from value of a parameter in the postcondition; on the other hand, for virtual functions, the copy will be unnecessary in most cases, i.e. we would be paying for the freedom to modify that parameter in an override, but we will most likely not make use of that freedom.

Further, unlike coroutines, any modifications to the parameters must be done explicitly in an overriding function and they will not happen implicitly as part of non-obvious language machinery. This suggests that there is a much lower risk of the user accidentally getting it wrong. For coroutines, there is no way the user could write an implementation that avoids the parameter modification; on the other hand, for virtual functions, there is a very simple way: just do not drop `const` from the parameter declaration on overriding functions, and do not modify that parameter in the function's implementation. This is what most users will do anyway, and we should not add any new obstacles for these users.

Finally, in today's C++ ecosystem, virtual functions are more pervasive than coroutines, and postcondition assertions have more known use cases for virtual functions than for coroutines (the usefulness of postcondition assertions on coroutines is fundamentally limited due to the nature of coroutines in C++). Entirely disallowing the ability to odr-use parameters in the postcondition assertion of a non-`final` virtual function in the first version of Contracts for C++ that we ship could noticeably hamper the usability of the feature. Shipping postcondition captures as proposed in [P3098R0] in the same version could somewhat mitigate but not fully remove the friction.

Overall, Option V1 might therefore be a disproportionately harsh measure for a relatively rare problem.

Option V2

Option V2 would also prevent the bug in the example above and is a less restrictive choice than Option V1: it would make only cases ill-formed where the parameter can actually be modified (when the overriding function actually drops the `const`). This option makes the behaviour for overriding functions consistent with the behaviour of subsequent declarations of the *same* function: if we redeclare a function, *or* an overload of that function, and drop `const` on a parameter in that declaration, the program is ill-formed. Option V2 thus seems more appealing than Option V1.

However, the tradeoff is that Option V2 could also lead to remote code breakage, which directly violates Design Principle 15 of [P2900R10], "No Client-Side Language Break". In particular, adding a postcondition to a virtual function that odr-uses a `const` parameter would remotely break any client code that overrides that function and yet has not added the `const` to all declarations of the override, including the definition.[2]

We anticipate that with virtual functions, such remote code breakage would be a frequent problem. Today, hardly anyone adds `const` to the declaration of a non-reference parameter as it has essentially no meaning; with [P2900R10], anyone who wishes to odr-use a parameter in a postcondition will *have* to add the `const`, breaking any overriding functions.

The crucial difference to the coroutine case is that providing a definition for a given function that makes the function a coroutine is local, not remote code, whereas overriding a function can happen in an entirely different component of the program. The possibility of such remote code breakage due to the introduction of Contracts could hamper their adoption and make releasing low-level libraries with newly introduced function-contract assertions significantly more difficult.

Option V3

Option V3 would also prevent the bug in the example above, while inflicting a smaller number of required changes to derived types than Option V2: only the defining declaration of the override will need the `const` added to the parameter declaration, not *all* declarations of the override. Option V3 is also likely to inflict a smaller number of remote code breakage, as only the translation unit that contains that defining declaration will be affected by the breakage. A library that makes use of a derived class with an override affected by the change will itself still compile without changes.

More notably, for users attempting to continue to link in binaries built in days past, as long as the function itself does not modify the function parameters in question (which is the case in the vast majority of cases, even when the parameters are not actually declared `const`), clients can continue to be rebuilt and linked against those binaries without issue or any need to access and update the source for the derived classes.[3]

Note that Option V3 does *not* remove the requirement that the parameter be declared `const` on all declarations of the function that has the postcondition odr-using that parameter. This property of Option V3 is also arguably its downside: it creates an inconsistency between the rules for the function that has the postcondition and the rules for function overriding it — the former needs to have `const` on the parameter declaration on *all* declarations of the function, while the latter need to have `const` on the parameter declaration only on the *defining* declaration of the function.

[2]As a special case of an override definition that does not declare a parameter `const`, such an override could also be implemented as a coroutine, in which case the parameter is also effectively not `const` (and may be moved from), even if declared `const` by the user on all declarations of that override including the definition). Defining such an override should be prevented.

[3]Technically, linking together code that was originally compiled with different declarations of the same function is an odr violation and therefore IFNDR, however the whole point of ABI compatibility is that we know such cases work in practice and we do not want to break that as it could constitute a serious obstacle to adoption of Contracts.

The other extreme for this alternative to lift the requirement of having `const` on all declarations is Option V4, which we discuss below.

Option V4

Option V4 is a further relaxation of Option V3, and in fact a relaxation of the current rule in [P2900R10]. With Option V4, `const` needs to be present only on the *definition* of the function that defines the postcondition itself, and if that function is a virtual function, also any overriding functions. The requirement that the parameter needs to be declared `const` on any non-defining declaration of a function would be dropped entirely, including for non-virtual functions. The consequence of such a design change would be that the following code, which is ill-formed in [P2900R10] today, would become well-formed (note that there are no virtual functions present in this code):

```
int f(int i) post (r: r >=  i);   // OK: const no longer needed here

int f(const int i) {   // const still needed here, ill-formed if missing
  return i;
}
```

Just like all the preceding options, this option would also prevent the bug in the example above: any attempt to modify the parameter in the implementation of the function itself or any of its overrides would be ill-formed. It would inflict the same amount of remote code breakage in overriding clients as Option V3 (and less than Option V2). Unlike Option V3, it would also not introduce any inconsistencies between the requirements on the declarations of the overridden function and its overriding functions, and in fact remove any inconsistencies between virtual and non-virtual functions. The rule for all functions would be the same: the `const` just needs to be on the definition.

At first glance, Option V4 seems appealing for the above reasons. However, for functions that *do* have postconditions that odr-use a non-reference parameter, we would need to address the ramifications of allowing an inconsistency between the `const`-qualifiers on a parameter in the declaration and in the definition.

One might, initially, believe that this inconsistency, which exists today, causes problems that we already deal with acceptably. For instance, it is possible to use the `const`-ness of function parameters in default arguments, `noexcept`-specifiers, and even the definitions of later parameters or trailing return types. Default parameters can have only one definition in a translation unit and may not odr-use function parameters. Elsewhere within a declaration the mechanism of choosing elements of a function signature does not matter — only the result does. It follows that with today's rules, all of the following declarations declare the same function:

```
int f(int i, int j);
int f(int i, std::conditional_t<std::is_const_v<decltype(i)>, long, int> j);
int f(const int i, std::conditional_t<!std::is_const_v<decltype(i)>, long, int> j);
auto f(const int i, int j)
-> std::conditional_t<std::is_const_v<decltype(i)>, int, long>;
```

For function-contract assertions, however, we expect the predicate to be evaluable by both a caller that sees only the declaration of a function and the function itself. Therefore, we must specify what happens when the expressions themselves are parsed with different understandings of whether or not a particular non-reference parameter of the function is declared `const`. To address this concern, we can consider three possible approaches:

V4a. Within a function declaration, if a non-reference parameter is odr-used by a postcondition, that parameter shall be implicitly treated as if it is `const`.[4]

[4]This option has been considered and rejected in the past by SG21 when reviewing [P2829R0]. Since this option was first discussed, [P3071R1] has been adopted into the Contracts MVP, often reffered to as `const`-ification. With

V4b. Apply the one definition rule to the evaluation of any preconditions or postconditions, making it ill-formed, no diagnostic required if the mismatch in `const` qualifiers on any parameters causes *any* changes in the meanings of the predicates.

V4c. Allow contract assertions to be evaluated with the parameter declarations from their associated declaration, including *cv*-qualifiers.

For a variety of reasons that we explore below, all of these options are highly problematic or unimplementable.

The primary problem arises from the cognitive load on a reader of an API attempting to understand it. Such readers would be forced to, when reading a postcondition, understand that parameters will be treated as `const` in the function body without actually seeing that information in the declaration. Unlike `const`-ification, this is not about an expression itself not modifying parameters, but the promise that the value a parameter is initialised with when a function is called is still that object's value when the function is returning back to the caller.

The next problem to consider is whether we want the evaluation of function-contract assertion predicates to evaluate the same functions regardless of where the code for it is generated. In general in C++, any time the same expression might be interpreted differently, we identify such cases as violations of the one definition rule and make them ill-formed, no diagnostic required (IFNDR). We apply the same rules to determine the equivalence of function-contract assertions (see [P2900R10] Section 3.3.1). Therefore, the following example of two declarations of a function attempting to repeat the preconditions is ill-formed (or IFNDR if the declarations are in separate translation units):

```
void f(int i) pre( std::is_same_v<decltype(i), int> );
void f(const int i) pre( std::is_same_v<decltype(i), const int> );
```

In general, any precondition or postcondition whose interpretation depends on the `const` qualifiers of a parameter would be similarly IFNDR if the definition repeated those function-contract assertions. The various flavours of Option V4, therefore, come into play when the function-contract assertions are *not* repeated on the definition. There, we must consider what gets invoked when the code for function-contract assertions is generated at a call site or within the function body. Consider a function with a precondition that uses a function template passed a parameter through a forwarding reference, and a postcondition that odr-uses that parameter:

```
template <typename T>
bool g(T&& t);

int f(int i)  // i does not need to be const here, despite being odr-used in post
  pre(g(UNCONST(i)))    // does this now call g<int&> instead of g<const int&>?
  post(r : r > i && g(UNCONST(i)) );
```

Here, `UNCONST` is a convenience macro added to access the underlying variable of a `const`-ified name in a contract assertion predicate, with its declared type which is not affected by `const`-ification (this macro is described in more detail as a useful escape hatch from `const`-ification in [P3261R1]):

```
// A macro to access underlying variable of const-ified names
#define UNCONST(x) const_cast<std::add_lvalue_reference_t<decltype(x)>>(x)
```

Without the postcondition, `g<int&>` in the example above is instantiated and called in all circumstances. With the postcondition, our different options will produce different results, none of which seem viable.

that change, it becomes harder to incorrectly depend on the `const`-ness of a function parameter, but it will always remain visible with `decltype`, which — just like in other `const`-ification contexts such as class members inside `const` member functions — is unaffected by `const`-ification and returns the original, non-`const`-ified parameter type. Changing the result of `decltype` is not an option as that removes any escape hatch from `const`-ification and breaks any code attempting to do computations with the type of a parameter instead of its value.

With Option V4a, a serious implementability issue arises because the compiler can now no longer correctly parse the preceding parts of the function declaration until it knows whether a parameter is odr-used in a postcondition. The call to `g` would have to be to `g<const int&>`, as the use of `i` in the postcondition would implicitly make `i const`. However, this is unknowable to the compiler until it parses the postcondition, which occurs after the precondition has been parsed and `g<int &>` has already been instantiated. The compiler would therefore have to somehow rewind and un-parse that first attempt to handle the precondition. This requires somehow having kept all side effects of instantiated templates — including initialisation of static class members — marked in such a way that they can be undone and then re-done with the correct template instantiation once the type of the function parameter is finally known. Such rollbacks of template instantiation do not exist in the language today for very good reasons.

Now, let us consider the function body of `f`:

```
int f(const int i) { // const required because of postcondition
  return i+1;
}
```

When generating code for this function, and in particular when generating code to check the precondition in this function, we would be doing so for a precondition that was originally parsed where `i` was of type `int`, yet the local variable we are applying it to is of type `const int`. Either we must force the other declaration to apply `const` to this function parameter, or we must ignore a top-level `const` qualifier, or we must make this case ill-formed (or IFNDR).

Option V4b would make this example IFNDR, as the function-contract assertions would have different interpretations if attached to different declarations.

Option V4c would pass this `const int` object to `g<int &>`. This falls apart if `g<int &>` actually does modify its parameter, as then the modification would be happening to a variable whose declaration has a top-level `const`-qualifier, which is undefined behaviour. Worse, even though such parameters are unlikely to be placed in read-only storage, the soundness of the postcondition relies on that value not changing.

Option V4a has even more direct problems if a `const_cast` is used without the use of `decltype`, which is the most straightforward escape hatch for `const`-ification:

```
int f(int i)
  pre(++const_cast<int&>(i))  // modifying i is now UB here...
  post( r : r > i);  // ...because post is odr-using i here!
```

Option V4a is further flawed due to the fact that even the fact that a postcondition odr-uses a parameter is itself something that can be dependent on the `const`-ness of the parameter, leading to paradoxical situations:

```
void f(int i)
  post([&]() {
    if constexpr (std::is_const_v<decltype(i)>) {
      return true;
    } else {
      return i != 0;
    }
  }());
```

It is our current view that the above problems render all of these approaches to dropping the `const` requirement on declarations that have a postcondition not clearly specifiable and not practicably implementable.

Option V5

Option V5 is the status quo[5] in [P2900R10]. It avoids the usage limitations imposed by Option V1 and the remote code breakage imposed by Option V2 or Option V3 and does not suffer from the implementation issues of Option V4. However, this option too has tradeoffs.

One downside of Option V5 is that, unlike Options V1 — V4, it would not actually prevent the bug, as it would allow a program to modify the parameter value in the implementation of an override. As a partial remedy, an implementation could easily issue a *warning* if an overriding function drops the `const` on a parameter odr-used in the postcondition assertion of an overridden function. The crucial difference to the coroutine case is that the implementation *knows* that the function is virtual at the point of declaration. While we cannot normatively mandate such a warning, we can add a non-normative recommended practice note to the wording that such a warning be issued.

Another downside of Option V5 compared to Options V1 – V4 is that it would mean abandoning the static guarantee of the current [P2900R10] design that, if a parameter is odr-used in a postcondition assertion, that parameter object *will not be involved in any non-`const` operations* and, for many types, will thus *not be modified* between the call to a function and the evaluation of its postcondition assertions.

It follows that neither humans nor static analysis tools would be able to reason about the value of a function parameter in a postcondition, at least not without taking into account which function will be selected by virtual dispatch and what the body of that function does (both of which are typically unknowable at the call site). This greatly reduces the amount of useful information that a static analysis tool could extract from a postcondition assertion at the call site of a virtual function.

Option V6

Option V6 is the option chosen in the past by C++2a Contracts [P0542R5]. However, we consider it completely unviable. Not only would this option fail to prevent the bug in the example above, but it would make the situation even worse: not only would there be a broken postcondition, but the program would also have undefined behaviour. This approach is therefore actively user-hostile and violates Design Principle 13 of [P2900R10], “Explicitly Define All New Behaviour”.

4 Proposal

We believe that Options V1 — V3 as well as V5 are worth considering, whereas Options V4 and V6 are unviable for the reasons explained above: a proposal that is not implementable or adds new undefined behaviour to the language does not seem to be worth spending more time on. We therefore propose Options V1 — V3 and V5 to determine which option has more consensus in SG21.

Note that choosing Option V1 would leave the door open to adopting V2, V3, or V5 without breaking changes at some point in the future, while Option V2 could only be evolved towards Option V3 or Option V5, Option V3 could only be evolved towards Option V5, and Option V5 could not be evolved towards either of the other options without breaking changes.

Table 1 summarises the main tradeoffs of all options V1 — V6 discussed in the paper.

[5]Since the current specification in [P2900R10] does not contain any special rules for parameter declarations on overriding functions, dropping `const` in such declarations is currently allowed and “you get what you get”.

	V1	V2	V3	V4	V5	V6
Allows non-reference parameters to be odr-used in `post` on virtual functions	❌	✅	✅	✅	✅	✅
Prevents modification of non-reference parameters odr-used in `post` and associated bugs	✅	✅	✅	✅	❌	❌
Adding `post` to a virtual function does not break derived classes that override it	✅	❌	❌	❌	✅	✅
Adding `post` to a virtual function does not break users of a derived class that overrides it	✅	❌	✅	✅	✅	✅
Does not introduce new inconsistencies between overriding and non-overriding functions	✅	✅	❌	✅	✅	✅
Clearly specifiable and practically implementable	✅	✅	✅	❌	✅	✅
Does not introduce new sources of undefined behaviour	✅	✅	✅	✅	✅	❌

Table 1: Main tradeoffs of proposed options V1 — V6.

5 Wording

The proposed wording changes are relative to the wording proposed in [P2900R10]. Note that we do not explicitly call out the case of an overriding function being implemented as a coroutine, however as per the wording proposed in [P2900R10], a coroutine behaves as if the top-level *cv*-qualifiers in all parameter-declarations in the declarator of its defining declaration were removed.

Option V1

Modify [dcl.contract.func] as follows:

> If the predicate of a postcondition assertion of a function odr-uses ([basic.def.odr]) a non-reference parameter of that function, all declarations of that parameter shall have a `const` qualifier and shall not have array or function type; if the function is virtual it shall be marked with the *virt-specifier* `final` (see [class.virtual]) or it shall be a a member function of a class with the *class-virt-specifier* `final` (see [class.pre]). [*Note:* This requirement applies even to declarations that do not specify the *postcondition-specifier*. Arrays and functions are still usable when declared with the equivalent pointer types ([dcl.fct]). — *end note*] [*Example:* `[...]` — *end example*]

Option V2

Modify [dcl.contract.func] as follows:

> If the predicate of a postcondition assertion of a function *f* odr-uses ([basic.def.odr]) a non-reference parameter of ~~that function~~*f*, all declarations of that parameter and the corresponding parameter on any functions that override *f* shall have a `const` qualifier and and shall not have array or function type. [*Note:* This requirement applies even to declarations that do not specify the *postcondition-specifier*. Arrays and functions are still usable when declared with the equivalent pointer types ([dcl.fct]). — *end note*] [*Example:* `[...]` — *end example*]

Option V3

Modify [dcl.contract.func] as follows:

> If the predicate of a postcondition assertion of a function *f* odr-uses ([basic.def.odr]) a non-reference parameter of ~~that function~~*f*, all declarations of that parameter shall have a `const` qualifier and shall not have array or function type; the corresponding parameter declaration in the definition of any function *g* that overrides *f* shall have a `const` qualifier and shall not have array or function type. [*Note:* This requirement applies even to declarations that do not specify the *postcondition-specifier*. Arrays and functions are still usable when declared with the equivalent pointer types ([dcl.fct]). For overrides that are coroutines, this requirement applies to the — *end note*] [*Example:* `[...]` — *end example*]

Option V5

Modify [dcl.contract.func] as follows:

> If the predicate of a postcondition assertion of a function odr-uses ([basic.def.odr]) a non-reference parameter of that function, all declarations of that parameter shall have a `cons` qualifier and shall not have array or function type. [*Note:* This requirement applies even to declarations that do not specify the *postcondition-specifier*. Arrays and functions are still usable when declared with the equivalent pointer types ([dcl.fct]). — *end note*] [*Example:* `[...]` — *end example*]
>
> *Recommended practice:* Implementations should issue a diagnostic when an overriding function omits `const` from any declaration of a non-reference parameter whose corresponding parameter in an overridden function is odr-used in a postcondition assertion of that overridden function.

Acknowledgements

Thanks to John Lakos and Oliver Rosten for their review of an earlier draft of this paper.

Revision history

— **R0**, 2024-10-31: Original version

— **R1**, 2024-11-07: Incorporated SG21 feedback; explained relationship of this paper to companion papers [P3487R0] and [P3489R0]; prefixed proposal numbers with “V” to be distinct from companion papers; various minor fixes

— **R2**, 2024-11-14: Introduced Option V3 to only require `const` on definitions of overrides and Option V4 to only require `const` on definitions of any functions; added deeper explanation of remote breakage scenarios; added table summarising tradeoffs; changed parameter type in examples from `int` to [**?**] with a non-trivial copy constructor to separate the concern discussed in this paper from the concern of passing parameters via registers

Bibliography

[D3484R2] Timur Doumler and Joshua Berne. Postconditions odr-using a parameter modified in an overriding function. https://wg21.link/d3484r2, 2024-11-09.

[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.

[P2829R0] Andrew Tomazos. Proposal of Contracts Supporting Const-On-Definition Style. https://wg21.link/p2829r0, 2023-02-27.

[P2900R10] Joshua Berne, Timur Doumler, and Andrzej Krzemieński. Contracts for C++. https://wg21.link/p2900r10, 2024-10-12.

[P3071R1] Jens Maurer. Protection against modifications in contracts. https://wg21.link/p3071r1, 2023-12-17.

[P3097R0] Timur Doumler, Joshua Berne, and Gašper Ažman. Contracts for C++: Support for Virtual Functions. https://wg21.link/p3097r0, 2024-04-15.

[P3098R0] Timur Doumler, Gašper Ažman, and Joshua Berne. Contracts for C++: Postcondition captures. https://wg21.link/p3098r0, 2024-10-14.

[P3261R1] Joshua Berne. Revisiting const-ification in Contract Assertions. https://wg21.link/p3261r1, 2024-10-10.

[P3387R0] Timur Doumler, Joshua Berne, Iain Sandoe, and Peter Bindels. Contract assertions on coroutines. https://wg21.link/p3387r0, 2024-10-09.

[P3487R0] Timur Doumler and Joshua Berne. Postconditions odr-using a parameter that may be passed in registers. https://wg21.link/p3487r0, 2024-11-07.

[P3489R0] Timur Doumler and Joshua Berne. Postconditions odr-using a parameter of dependent type. https://wg21.link/p3489r0, 2024-11-01.