Contracts for C++: `const` parameters in postconditions of overridden functions

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

Document #:	P3484R0
Date:	2024-10-31
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 essentially meaningless. We propose several alternatives for how to address this problem in the Contracts MVP [P2900R10].

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 includes the *defining* declaration of the function; thus, the implementation of the function can be assumed to not modify the value of that parameter.

The rationale for this rule is that allowing the implementation to modify a parameter that is later used to check the postcondition would render that check meaningless. Consider:

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

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

```
int f(int 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 always evaluates to `true`:

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

For this reason, [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, and therefore the function parameters of a coroutine are effectively not `const`, even if declared as such by the user (see [P3387R0] for discussion). This case is thus notionally similar to the previous case where `const` was dropped from the parameter declaration in the implementation.

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 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 int f(const int 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 {
  int f(int i) override; // OK
};
```

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

```
int Derived::f(int 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) {
  int 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` (something that `Base::f` has no control or visibility over), rendering the postcondition assertion of `Base::f` meaningless. This is exactly the scenario that the specification in [P2900R10] Section 3.4.4 seeks to avoid; nevertheless, the code above is well-formed according to the current specification.

3 Possible solutions

We are aware of four possible options for dealing with this problem. These are, from most to least restrictive:

1. 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`.
2. 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.
3. Allow overriding functions to drop `const` from a non-reference function parameter, with no special provision, i.e., "you get what you get": in the virtual function call above, which invokes `Derived::f`, the postcondition check on `Base::f` succeeds even though `Derived::f` does not satisfy the postcondition of `Base::f`.
4. 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.

Option 4 is the option that was chosen in the past by C++2a Contracts [P0542R5]. However, we can immediately discard it as unviable. Not only would this option fail to catch the bug in the example above, but it is also the most user-hostile option: the mere act of adding a postcondition assertion to a program would have the potential to introduce new undefined behaviour, even if its predicate were written correctly. This directly violates Design Principle 13 of [P2900R10], "Explicitly Define All New Behaviour".

Option 1 would catch 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 int f(const int i) post (r: r >= i);      // error: cannot odr-use parameter i
};
```

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

However, Option 1 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.

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 1 might therefore be a disproportionately harsh measure for a relatively obscure problem. Option 2 would also catch the bug in the example above and is a less restrictive choice than Option 1: 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 2 thus seems more appealing than Option 1.

However, the tradeoff is that Option 2 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`, or any override that is implemented as a coroutine. 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 vastly more difficult.

Option 3 does not suffer from the downsides of Options 1 and 2. Option 3 is also the status quo in [P2900R10] and therefore does not require any design changes to Contracts for C++. Since the current specification in that paper does not contain any special rules for parameter declarations on overriding functions, dropping `const` in such declarations is currently allowed in [P2900R10] and "you get what you get". This is also the downside of Option 3 compared to Options 1 and 2: Option 3 does not actually catch the bug.

However, there is a remedy available: if an overriding function drops the `const` on a parameter odr-used in the postcondition assertion of an overridden function, an implementation can easily issue a *warning*. 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 non-normatively recommend it.

We believe that Options 1 — 3 are all worth considering and the tradeoffs of each option are sufficiently clear. We therefore propose all three options (which are mutually exclusive) to determine which option has more consensus in SG21.

Note that choosing Option 1 would leave the door open to adopting Option 2 or Option 3 without breaking changes at some point in the future, while Option 2 could only be evolved towards Option 3 but not Option 1, and Option 3 could not be evolved towards either of the other options.

4 Proposed wording

The proposed wording changes are relative to the wording proposed in [P2900R10].

Option 1

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, that parameter shall be declared `const` 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 2

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*, that parameter and the corresponding parameter on any functions that override *f* shall be declared `const` 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 3

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, that parameter shall be declared `const` 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 the declaration of a non-reference parameter whose corresponding parameter in an overridden function is odr-used in a postcondition assertion of that overridden function.

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.

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

[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.

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