Postconditions odr-using a parameter that may be passed in registers

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

Document #:	P3487R0
Date:	2024-11-07
Project:	Programming Language C++
Audience:	SG21, EWG

Abstract

This paper considers the case where a non-reference parameter is odr-used in the predicate of a precondition or postcondition assertion and is eligible to be passed via registers. To enable caller-side checking of preconditions and postconditions, we need to add a provision to the the Contracts MVP [P2900R10] that allows the check to observe either the caller-side or the callee-side version of the parameter object. However, for postconditions, this can lead to surprising behaviour. We propose several alternatives for how to address this problem.

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

— [D3484R1] Postconditions odr-using a parameter modified in an overriding function;

— [D3487R0] Postconditions odr-using a parameter that may be passed in registers;

— [D3489R0] Postconditions odr-using a parameter of dependent type.

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

1 Background

For efficiency reasons, the major ABIs used for implementations of C++ allow objects to be passed to a function and returned from a function via *registers* when the type of the object satisfies certain requirements. The C++ Standard accommodates such passing and returning via registers in [class.temporary]/3 as follows:

> When an object of class type `X` is passed to or returned from a function, if `X` has at least one eligible copy or move constructor ([special]), each such constructor is trivial, and the destructor of `X` is either trivial or deleted, implementations are permitted to create a temporary object to hold the function parameter or result object. The temporary object is constructed from the function argument or return value, respectively, and the function's parameter or return object is initialised as if by using the eligible trivial constructor to copy the temporary (even if that constructor is inaccessible or would not be selected by overload resolution to perform a copy or move of the object).

Effectively, such objects passed and returned via registers do not exist in memory and do not have an address; their value is instead accessed by materialising a temporary copy. In C++ today, this specification peculiarity causes no friction, because there is (with one minor exception[1]) no context where the pre-temporary copy versions of these objects could be directly observable by the user. The current wording does not say it explicitly, but there is an assumption within it that, once a temporary has been created to "hold the function parameter or result object", that temporary will be referred to whenever the name that denotes the object is used going forward.

In practice in C++ today, that is always the case. However, the Contracts MVP [P2900R10] adds function-contract assertions (precondition and postcondition assertions). Depending on how we specify them, they could create a new context where the pre-temporary copy versions of parameter objects and/or return objects are not only observable, but usable by name. We therefore must clarify exactly what semantics we want to allow in those cases, and what that means for implementations and for users.

2 Discussion

2.1 Caller-side checking

One of the design goals of [P2900R10] is to allow the implementation to perform precondition and postcondition checks either callee-side or caller-side. A discussion of implementation strategies can be found in [P3267R0] and [P3321R0]; a discussion of the motivation and use cases for both caller-side and callee-side checks can be found in [P3228R1], [P3264R1], [P3270R0], [P3321R0], and references therein. We provide a very brief summary below,.

For callee-side checks, the compiler would emit code to perform the precondition and postcondition checks when compiling the definition of the function. For caller-side checks, the compiler would instead emit code around the function call to perform the checks. Note that the precondition and postcondition assertions are part of the function declaration and thus known at every call site.

Callee-side checks can be emitted in all cases. On the other hand, caller-side checks cannot be emitted in certain cases. One such case is an indirect call (e.g., through a pointer to function or pointer to member function), since the compiler does not know at the call site which concrete function will be called. Another such case is an ABI that requires function parameters to be destroyed callee-side (e.g., the Microsoft ABI), which means that postconditions cannot be checked caller-side without an ABI break, as postconditions must be evaluated before destruction of function parameters.

Even though not all checks can be performed caller-side in all cases, the ability to perform at least *some* caller-side checks is important. With this ability, the user can enable contract checks to diagnose defects when working with a library that has been compiled with contract checks off (it can often be too costly or outright impossible to recompile the library with contract checks on and re-link the program). Caller-side checks also enable an implementation of contract checks on virtual functions as specified in [P2900R10]: the caller-facing contract (that of the statically called function) can be checked caller-side, while the callee-facing contract (that of the final overrider selected by virtual dispatch) can be checked callee-side. Note that only the caller knows the caller-facing contract in this case.[2]

[1]An object that meets the requirements to be passed in a register may still have a user-provide constructor that may observe its address through the use of `this`, and that address may then differ from the address seen for the parameter within the function body.

[2]How one could implement checking the caller-facing contract of a virtual function call on an ABI that requires function parameters to be destroyed callee-side, without forcing an ABI break, is currently still an open question, but this problem is only tangentially related to the problem discussed in this paper.

2.2 Preconditions and parameters

Precondition checks can only observe objects passed to a function, i.e., the function parameters, not objects returned from the function. [P2900R10] currently does not contain an explicit provision that would allow precondition checks to observe the pre-temporary copy parameter objects. It follows therefore from [class.temporary] that precondition checks must observe the same parameter object as the function body. This makes caller-side precondition checks unimplementable with the current specification.

We can fix this problem by adding an explicit provision to [P2900R10] that a precondition check is implicitly allowed to observe either the pre-temporary parameter object or the temporary copy. If the precondition check observes the pre-temporary object, it will do so before the copy is made to pass the object into a function. If it observes the temporary copy, it will observe the same object as the function body. In either case, there is no problem.

2.3 Postconditions and the return object

Unlike precondition checks, postcondition checks can observe both parameters and the return object. [P2900R10] contains an explicit provision that allows postcondition checks to observe either the caller-side or the callee-side version of the return object:

> If the implementation is permitted to introduce a temporary object for the return value ([class.temporary]), the result name may instead denote that temporary object.

This provision is directly observable. Consider:

```
class X { /* ... */ };

X f(const X* ptr) post(r: &r == ptr) {
  return X{};
}

int main() {
  X x = f(&x);
}
```

In the example above, if `X` is *not* a type eligible to be passed via registers, the postcondition check of `f` is guaranteed to pass, because `r` must denote the return object `x` in `main()`; however, if `X` *is* a type eligible to be passed via registers, the postcondition check may[3] fail, because `r` may instead denote a temporary object.

This behaviour may be surprising to a user not familiar with the rules for returning objects via registers, but there is no actual problem — this behaviour is an exact mirror image of parameters in postcondition assertions. Postcondition checks that refer to the return object may therefore be implemented either caller-side or callee-side with the current specification.

2.4 Postconditions and parameters

The third and last case to consider is a postcondition assertion odr-using a non-reference parameter:

```
X* ptr;

void f(const X x) post (ptr == &x) {
  ptr = &x;
}
```

[3]In practice, whether this check fails will depend on both optimization levels and whether `f` is inlined into `main()`.

If `X` is a type eligible to be passed via registers, is this postcondition guaranteed to pass or not?

Just like for precondition assertions, [P2900R10] currently does not contain an explicit provision that would allow postcondition checks to observe the pre-temporary copy parameter objects. It follows therefore from [class.temporary] that postcondition checks, like precondition checks, must observe the same parameter object as the function body; the postcondition assertion above is guaranteed to pass.

This makes caller-side precondition checks unimplementable with the current specification without an ABI break that changes the ABI to no longer pass parameters via registers (other reasons why they might be unimplementable, in particular an ABI that requires function parameters to be destroyed callee-side, notwithstanding).

Just like for the previous two cases, we could add an explicit provision to [P2900R10] that a postcondition check is implicitly allowed to observe either the pre-temporary parameter object or the temporary copy. However, unlike for preconditions, for postconditions the fact that the addresses of the object that the contract assertion sees and the object that the function body sees might be different (which in itself is harmless) is no longer the only observable effect of such a provision. In addition to that, we now also run into the problem that the temporary copy is made when the function is called, but the postcondition assertion is evaluated when the function returns. There is a period in between during which arbitrary code could be executed that can change the state of the parameter object.

[P2900R10] requires every parameter odr-used in a postcondition assertion to be declared `const` on all declarations of the function, and requires that function to not be a coroutine, which guarantees that the parameter object that the function body observes is not modified between the function call and its return. However, if the parameter is passed in registers, and the postcondition observes the pre-temporary copy parameter object, these two versions of the parameter object could diverge. This has surprising consequences and renders the postcondition's actual meaning significantly more difficult to reason about.

In particular, even if the parameter object is `const`, the function body could still modify any `mutable` subobjects of that object. If the postcondition assertion is allowed to observe the pre-temporary copy version of the object, it will not see such modifications.

Now, of course we should not write postconditions that directly depend on such mutable state anyway, and if we do, we have already shot ourselves in the foot. But the problem at hand is more subtle: we may have a type whose correctness is connected to that mutable state.

Consider a class `RandomInteger` holding a random integer value:

```
class RandomInteger {
  int _value = rand();
public:
  int value() const {
    return _value;
  }
};
```

The value is computed once when an object of type `RandomInteger` is created and does not change afterwards. This value is accessible via a public `value()` member function, which is marked `const` because it does not change the *observable* state of the object — it always returns the same value throughout its lifetime.

As an implementation detail, we might compute the value lazily when `value()` is called for the first time, and cache it afterwards, with no observable change in behaviour:

```
class RandomInteger {
  mutable bool _computed = false;
  mutable int  _value;
public:
  int value() const {
    if (!_computed) {
      _value = rand();
      _computed = true;
    }
    return _value;
  }
};
```

Note that our `RandomInteger` class, as defined above, has a trivial copy constructor and a trivial destructor and is therefore eligible to be passed via registers. This leads to a new footgun:

```
int f(const RandomInteger i)
post(r: r & i.value() == 0) {
  return ~i.value();
}
```

If there is no guarantee that the `i` in the postcondition assertion refers to the same object as the `i` in the function body, this code will break. The postcondition assertion will see a different integer value returned from `i.value()` than the body of the function, and thus fail where it should pass or vice versa, even though for any user reading this code without a deep understanding of objects being passed in registers will see nothing obviously incorrect with this code.

Such code works in C++ today because after the parameter object has been packed into registers and passed to the function, the original object will never be touched by anything again (remember that the type also needs to have a trivial or deleted destructor, not just an eligible trivial copy or move constructor). However, allowing a postcondition check to observe the original object — which is required to enable caller-side postcondition checks without an ABI break — changes that, which creates the footgun above.

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

R1. Remove postcondition assertions from [P2900R10] entirely.

R2. Make it ill-formed to odr-use *any* function parameter in a postcondition predicate.

R3. Make it ill-formed to odr-use any *non-reference* function parameter in a postcondition predicate.

R4. Add an explicit provision that, when a non-reference function parameter is odr-used in a postcondition predicate and the type of the parameter satisfies the requirements for being passed in registers, the corresponding *id-expression* may refer either to the same object as it does in the function body or to the temporary which had been created to initialize the parameter, thereby allowing caller-side checking of a postcondition predicate that odr-uses a non-reference function parameter without an ABI break. Make it ill-formed to odr-use a non-reference function parameter in a postcondition predicate unless the parameter is of scalar type.[4]

R5. Add the above provision. Make it ill-formed to odr-use a non-reference function parameter in a postcondition predicate if the type of the parameter satisfies the requirements for being passed via registers, unless it is of scalar type.

[4]Scalar types are arithmetic types, enumeration types, pointer types, pointer-to-member types, `std::nullptr_t`, and *cv*-qualified versions of these types. Arithmetic types are integral and floating-point types; integral types include character types and `bool`.

R6. Add the above provision. Make it ill-formed to odr-use a non-reference function parameter in a postcondition predicate if the type of the parameter satisfies the requirements for being passed via registers and has at least one `mutable` subobject (applies recursively to all member subobjects, base class subobjects, and array elements).

R7. Add the above provision and do nothing further. A postcondition may odr-use a non-reference function parameter of any type.

R8. Do not add the above provision. Instead, clarify that when a non-reference function parameter is odr-used in a postcondition predicate, the corresponding *id-expression* must refer to the same object as it does in the function body (status quo in [P2900R10]); caller-side checking of a postcondition predicate that odr-uses a non-reference function parameter remains impossible without an ABI break. A postcondition may odr-use a non-reference function parameter of any type.

We enumerated the options with an "R" prefix (for "registers"), to distinguish them from the options from [D3484R1] that have a "V" prefix (for "virtual") and the options from [D3489R0] that have a "D" prefix (for "dependent").

Below we discuss the tradeoffs of each option.

Option R1

Option R1 would be a rather drastic measure at this point. However, consider that postcondition assertions are significantly more difficult to specify than preconditions (see [P1323R2], [P3007R0], and [P3098R0]), and unlike preconditions, postcondition assertions have so far already generated several known issues that needed fixing in the specification of [P2900R10] (see [P3387R0], [P3460R0], [P3483R0], [D3484R1], and [D3489R0]). Option R1 would remove *all* known and unknown footguns from postcondition assertions by removing the feature itself.

We believe that [P2900R10] would still be viable and useful without postcondition assertions. Postcondition assertions have significantly fewer uses than precondition assertions, and their value can to a certain extent also be achieved with good unit test coverage.

For the record, the option of removing postcondition assertions from the Contracts MVP was once before polled in SG21:

SG21 Poll, Teleconference 2021-12-14

Postconditions should be in the MVP at this time.

SF	F	N	A	SA
1	7	3	4	1

Result: Marginal consensus (if at all).

Option R2

Option R2 is likewise less than ideal in our opinion, because it significantly limits the set of postconditions we can write, and thus significantly limits the usefulness of the feature, until we can add postconditions captures [P3098R0] to the Standard.

This option becomes more appealing if we include [P3098R0] in the first version of Contracts that we ship. Postcondition captures would never be referencing parameters in a place they cannot be referenced today, so they would not be impacted by this issue at all. However, capturing parameters for use in a postcondition predicate incurs the cost of a copy, which in many cases is not conceptually necessary, thus violating the “do not pay for what you do not use” design principle of C++ (see also [D3484R1] Option R1, which suffers from the same issue).

Option R3

Option R3 is similar to Option R2, except that it allows reference parameters, which are not affected by any of the issues surrounding copies of objects and are not affected by postcondition captures as proposed in [P3098R0].

However, allowing only reference parameters still significantly limits the set of postconditions we can write. In addition, it encourages users to pass parameters by-reference instead by-value as this would be the only way to make the postcondition assertion compile, which can lead to more error-prone and less efficient code than the normally recommended pass-by-value. We therefore do not consider Option R3 to be an improvement over Option R2.

Option R4

Option R4 allows by-value parameters of types that are not affected by the footgun and cannot be changed such that they would be affected by the footgun, i.e., scalar types. This would already enable many more useful postconditions compared to Options R1 — R3.

However, if we were to change the type of a parameter from a built-in type to a user-defined type, for example from `int` to `BigInt`, or from `double` to `std::complex<double>`, the postcondition would stop compiling, with no workaround available. Additionally, Option R4 would make it significantly harder to add postconditions to generic code, including any templates designed to work with both built-in and user-defined types (which includes the entire STL and vast amounts of other generic libraries).

Option R5

Option R5 is a relaxation of Option R4. It allows by-value parameters of scalar type (disallowing them would remove the ability to write many simple and useful postcondition assertions), and in addition, it allows by-value parameters of any type as long as they are *not* eligible to be passed via registers and therefore cannot not affected by the footgun.

One downside of this approach is that most users will not be familiar with the rules around types eligible to be passed in registers, which means that the compiler error they get will likely be very confusing to them. Worse, the definition of user-defined types can change over time, which makes this option brittle. That a particular type is *not* eligible to be passed via registers is not something that code should guarantee to its clients indefinitely in all cases; conversely, making a type trivially copyable and/or movable and trivially destructible should not break client code.

We made a similar choice to not discriminate on particular type traits in the postcondition assertions of a function when we decided that whether a type is trivially movable should not affect whether a non-reference parameter of that type can be odr-used in the postcondition assertion of a coroutine (i.e., when we rejected [P3387R0] Option 5b). Choosing Option R5 here would be inconsistent with that design choice.

Option R6

Option R6 is a further relaxation of Option R5. It carves out the narrowest possible prohibition on parameter types that can be odr-used in a postcondition predicate — any type is allowed as long as it does not have the exact property that can lead to the footgun: types that are eligible to be passed in registers *and* have `mutable` subobjects.

Option R6 is the least prohibitive option that both avoids the footgun and allows implementing caller-side postcondition checks without an ABI break. However, it suffers from the same problem as Option R5: a very specific and seemingly unrelated change to a type can lead to the postcondition no longer compiling. For Option R6, the error would be even less obvious for Option R5, as it would occur when the user decides to add a `mutable` member to a type that happens to be eligible to be passed in registers, e.g. when the user does a refactoring such as the one we did with `RandomInteger` above, which is a relatively common technique. The result is brittle code, a very hard-to-understand compiler error, and no good workaround.

Option R7

Option R7 makes the behaviour of postconditions with respect to objects passed to a function in registers consistent with the behaviour for objects returned from a function in registers, as well as with the behaviour of preconditions as proposed in Section 2.2. Option R7 is therefore arguably the optimal choice with respect to having a straightforward specification and implementation of the feature, enabling caller-side checking, avoiding ABI issues, not making any user code unnecessarily ill-formed, and not imposing any unnecessary run-time cost on the user.

However, the tradeoff of Option R7 is that it adds the above footgun to the C++ language. Note that the footgun only occurs in rare edge cases, in particular when a `const` object eligible to be passed via registers is used as a non-reference parameter and its type relies on mutable state for its correctness, and a postcondition assertion would break if it happens to observe an earlier version of that mutable state. Note further that such cases could potentially be mitigated by an implementation issuing a warning if a type eligible to be passed in registers has `mutable` subobjects, is used as the type of a non-reference function parameter, and that parameter is odr-used in a postcondition assertion of that function or another function that that function overrides.

Option R8

Option R8 is a confirmation of the status quo. It is the only solution that both avoids the above footgun and is not a breaking change to [P2900R10]: postconditions odr-using a `const` non-reference parameter of a non-coroutine function remain well-formed. Option R8 is therefore arguably the optimal choice with respect to the immediate user experience when dealing with code such as the above.

However, Option R8 also comes with a high cost: implementing caller-side postcondition checks remains impossible without an ABI break. The necessary ABI break to enable caller-side postcondition checks — and by extension, to implement full contract checks on virtual functions as specified by [P2900R10] — would consist of no longer passing function parameters via registers if they are odr-used in the postcondition assertion. However, imposing an ABI break on all users who wish to add postcondition assertions to their functions would arguably do significant harm to the adoption of Contracts in the C++ ecosystem, which is why one of the fundamental design principles of the Contracts MVP ([P2900R10] Principle 16) is to avoid such an ABI break.

3 Proposal

With regards to parameters odr-used in *preconditions*, we propose to add a provision to [P2900R10] that a precondition check is implicitly allowed to observe either the pre-temporary parameter object or the temporary copy, as discussed in Section 2.2.

With regards to parameters odr-used in *postconditions*, we believe that Options R1 — R8 are all worth considering, and propose all of them to determine which option has more consensus in SG21. A summary of the tradeoffs for each option can be found in Table 1.

Note that there are three requirements that are *impossible* to satisfy simultaneously — we need to choose two. These requirements are: allowing a postcondition predicate to odr-use non-reference parameters of *any* type; avoiding the footgun created by parameter objects with `mutable` subobjects; and allowing caller-side checking of postconditions that odr-use non-reference parameters without an ABI break that removes passing via registers. If we are willing to abandon the first requirement, we can choose between Options R1 — R6, which offer a spectrum between most restrictive and most brittle; if we are willing to abandon the second requirement, the optimal solution is Option R7; if we are willing to abandon the third requirement, the optimal solution is Option R8.

Options R1 — R3 do not offer any advantage over Options R4 — R6 with regards to the specific issues discussed in this paper. However, R1 — R3 may still be interesting because they would remove the source of the issues discussed in the two companion papers [D3484R1] and [D3489R0].

Note further that all options except Option R8 form a chain of successive relaxations of the previous option. Therefore, choosing Option R1 would leave the door open to adopting Options R2 — R7 without breaking changes at some point in the future; Option R2 could be evolved towards Options R3 — R7, but not R1, etc. On the other hand, Option R8 is mutually exclusive with any of the other options: an evolution from any of the other options towards Option R8, or in the other direction, is impossible without breaking changes.

Note finally that choosing Option R4 would be consistent with a possible relaxation of the rule for postconditions on coroutines to also accept non-reference parameters of scalar type.

	R1	R2	R3	R4	R5	R6	R7	R8
Allows postcondition assertions in general	❌	✅	✅	✅	✅	✅	✅	✅
Allows odr-using reference parameters	❌	❌	✅	✅	✅	✅	✅	✅
Allows odr-using non-reference parameters of at least scalar type	❌	❌	❌	✅	✅	✅	✅	✅
Allows odr-using non-reference parameters of any type	❌	❌	❌	❌	❌	❌	✅	✅
Avoids brittle discrimination based on certain type traits	✅	✅	✅	?	❌	❌	✅	✅
Makes it impossible to shoot yourself with the footgun	✅	✅	✅	✅	✅	✅	❌	✅
Allows caller-side postcondition checking without ABI break	✅	✅	✅	✅	✅	✅	✅	❌

Table 1: Main tradeoffs of proposed options R1 — R8. Discriminating on whether the parameter is of scalar type (R4) is marked with a question mark because it is significantly less brittle than discriminating based on more complicated and non-obvious type traits (R5 and R6).

4 Wording

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

Preconditions

Modify [dcl.contract.func] as follows:

> If the predicate of a precondition assertion of a function odr-uses ([basic.def.odr]) a nonreference parameter of that function, and the implementation is permitted to introduce a temporary object for the parameter object value ([class.temporary]), it is unspecified whether the corresponding *id-expression* in the predicate of the precondition assertion denotes that temporary object or the original parameter object. If the predicate of a postcondition assertion of a function odr-uses ~~([basic.def.odr])~~ a nonreference 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*]

Postconditions — Option R1

Remove all wording that relates to:

— The `post` identifier with special meaning;

— The *postcondition-specifier* and *result-name-introducer* grammar non-terminals;

— Postcondition assertions and result names;

— The `post` enumeration value in `std::contracts::assertion_kind`.

The exact wording diff is not provided here due to its length.

Postconditions — Option R2

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~~the program is ill-formed. [*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*]

Postconditions — Option R3

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~~the program is ill-formed. [*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*]

Postconditions — Option R4

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~~have scalar type ([basic.types.general]). [*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*]

If the implementation is permitted to introduce a temporary object for the parameter object value ([class.temporary]), it is unspecified whether the corresponding *id-expression* in the predicate of a postcondition assertion denotes that temporary object or the original parameter object.

Postconditions — Option R5

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 parameter has a type for which the implementation is permitted to create a temporary object to hold the function parameter ([class.temporary]), it shall have scalar type ([basic.types.general]). [*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*]

If the implementation is permitted to introduce a temporary object for the parameter object value ([class.temporary]), it is unspecified whether the corresponding *id-expression* in the predicate of a postcondition assertion denotes that temporary object or the original parameter object.

Postconditions — Option R6

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 parameter has a type for which the implementation is permitted to create a temporary object to hold the function parameter ([class.temporary]), it shall not have any `mutable` subobjects ([dcl.stc]). [*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*]

If the implementation is permitted to introduce a temporary object for the parameter object value ([class.temporary]), it is unspecified whether the corresponding *id-expression* in the predicate of a postcondition assertion denotes that temporary object or the original parameter object.

Postconditions — Option R7

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*]
>
> If the implementation is permitted to introduce a temporary object for the parameter object value ([class.temporary]), it is unspecified whether the corresponding *id-expression* in the predicate of a postcondition assertion denotes that temporary object or the original parameter object. [*Note:* It follows that, for objects that can be passed in registers, the postcondition assertion might not see any modifications of `mutable` subobjects ([dcl.stc]) of the parameter object performed by the function or a function overriding it. — *end note*]

Postconditions — Option R8

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]). An *id-expression* that denotes a non-reference parameter in the predicate of a postcondition assertion denotes the same object as in the function body, even if the implementation is permitted to introduce a temporary object for the parameter object value ([class.temporary]). — *end note*] [*Example:* `[...]` — *end example*]

Acknowledgements

Thanks to Alisdair Meredith, John Lakos, Jens Maurer, and Mungo Gill for their helpful remarks during drafting of this paper. Thanks to Oliver Rosten for his review of the paper.

Bibliography

[D3484R1] Timur Doumler and Joshua Berne. Postconditions odr-using a parameter modified in an overriding function. https://wg21.link/d3484r1, 2024-11-01.

[D3487R0] Timur Doumler and Joshua Berne. Postconditions odr-using a parameter that may be passed in registers. https://wg21.link/d3487r0, 2024-11-01.

[D3489R0] Timur Doumler and Joshua Berne. Postconditions odr-using a parameter of dependent type. https://wg21.link/d3489r0, 2024-11-01.

[P1323R2] Hubert Tong. Contract postconditions and return type deduction. https://wg21.link/p1232r2, 2019-02-20.

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

[P3007R0] Timur Doumler, Andrzej Krzemieński, and Joshua Berne. Return object semantics in postcondition specifiers. https://wg21.link/p3007r0, 2023-12-11.

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

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

[P3264R1] Ville Voutilainen. Double-evaluation of preconditions. https://wg21.link/p3264r1, 2024-05-17.

[P3267R0] Peter Bindels. C++ contracts implementation strategies. https://wg21.link/p3267r0, 2024-05-22.

[P3270R0] Joshua Berne and John Lakos. Repetition, Elision, and const-ification With Regard to `contract_assert`: A Principled Analysis. https://wg21.link/p3270r0, 2024-05-22.

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

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

[P3460R0] Eric Fiselier, Nina Ranns, and Iain Sandoe. C++ Contracts Implementers Report. https://wg21.link/p3460r0, 2024-10-16.

[P3483R0] Timur Doumler and Joshua Berne. Contracts in C++: Pre-Wrocław technical clarifications. https://wg21.link/p3483r0, 2024-10-31.