Postconditions odr-using a parameter of dependent type

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

Document #: P3489R0
Date: 2024-11-07
Project: Programming Language C++
Audience: SG21, EWG

Abstract

This paper considers the case where a non-reference parameter of dependent type is odr-used in a postcondition assertion. The Contracts MVP [P2900R10] specifies that the program is ill-formed unless the parameter is declared `const` on all declarations of the function. However, the parameter may be of dependent type, and we might not know whether its type is `const` until the template is instantiated. [P2900R10] is currently ambiguous about what should happen in this case; we propose two alternatives for how to resolve the ambiguity.

This paper is the third 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 The problem

The Contracts MVP [P2900R10] specifies that if a non-reference function parameter is odr-used in the postcondition of a function, it must be declared `const` on all declarations of that function, otherwise the program is ill-formed.

However, whether or not a function parameter is indeed `const` is not immediately visible if the function in question is a template (a function template, a member function of a class template, etc.) and the type of the parameter is a dependent type. Consider:

```
template <typename T>
void f(T t) post(t > 0);
```

This function template may be instantiated with a type that is `const`-qualified, or a type that is not; this may or may not involve type deduction. However, this is not known when parsing this function template, as the variable `t` does not have a visible `const` specifier on it. It is therefore not immediately obvious whether the above template declaration is ill-formed or not. [P2900R10] does not explicitly specify this case, i.e., we have a design hole that needs to be fixed.

It is clear that the program should be ill-formed if the template above is instantiated with a type `T` that is not `const`:

```
int main() {
  int i = 1;
  f<int>(i);  // error
}
```

However, it is less clear what should happen if the template above is instantiated with a type `T` that *is* `const`:

```
int main() {
  int i = 1;
  f<const int>(i);  // OK?
}
```

2 Possible solutions

We are aware of two possible options for resolving the ambiguity:

D1. Require the parameter declaration to have an explicit `const` qualifier, i.e., make the above template declaration ill-formed regardless of whether and how the template is instantiated;

D2. Allow the `const` qualifier to be part of the dependent type, i.e., do not make the above template declaration ill-formed, but make it ill-formed to instantiate the template with a type that is not `const`.

We enumerated the options with a "D" prefix (for "dependent"), to distinguish them from the options from [D3484R1] that have a "V" prefix (for "virtual") and the options from [D3487R0] that have a "R" prefix (for "registers").

Below we discuss the tradeoffs of each option.

Option D1

Option D1 is the more conservative option. It forces the user to express their intent directly by declaring the parameter `const` explicitly. It also catches a defect due to a missing `const` sooner, as the error will be triggered already when the template is declared, and not when it is instantiated, which may happen much later and in a different component of the program. Finally, it also prevents the user from writing brittle templates with postcondition assertions that might or might not compile depending on the template parameter. When deducing a template argument from a by-value parameter, `const` is not deduced as part of that type. So the parameter will only be `const` when specified explicitly in the template argument list. It does not seem to be useful to have templates that either do or do not compile depending on whether that `const` on the template argument list is there.

The tradeoff is that Option D1 makes programs ill-formed that do not contain a defect, such as the last example above. The parameter type of `f<const int>` is actually `const`, and [P2900R10] normally allows a parameter of such a type to be odr-used in a postcondition assertion; nevertheless, this program would be rejected.

Option D2

Option D2 is the more permissive option. It only rejects programs where, after the template is instantiated, the parameter type is actually found to not be `const` and therefore may be modified in the function body; the `const` does not need to be explicit at the point of declaration.

The tradeoffs are the inverse of Option D1: if the user got it wrong, the defect will be caught later rather than earlier, and this approach can lead to brittle templates with postcondition assertions that might or might not compile depending on the template parameter. It is not clear whether being more permissive here actually gains anything significant or useful.

3 Proposal

We believe that both options are worth considering and the tradeoffs of each option are sufficiently clear. We therefore propose both options, to determine which option has more consensus in SG21.

Note that Choosing Option D1 would leave the door open to adopting Option D2 at some point in the future, whereas the opposite is not true.

Note further that [D3487R0] — which deals with a different problem regarding parameters in postconditions — proposes the more extreme options R1 (removing postcondition assertions from [P2900R10] altogether), R2 (disallowing odr-use of *any* parameters in postcondition assertions), and R3 (disallowing odr-use of *non-reference* parameters in postcondition assertions). Options R1 — R3 from [D3487R0] would remove the issue discussed in this paper and should therefore be considered alongside Options D1 and D2 as possible solutions.

4 Wording

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

4.1 Option D1

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`~~ all declarations of that parameter 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]). — *end note*] [*Example:*
>
> ```
> int f(const int i)
> post (r: r == i);
> int g(int i)
> post (r: r == i); // error: i is not declared const.
> int f(int i) // error: i is not declared const.
> {
>   return i;
> }
>
> int g(int i) // error: i is not declared const.
> {
>   return i;
> }
> ```

```
template <typename T>
void f(T t) post(t > 0);  // error: parameter not declared const but odr-used in postcondition
```

— *end example*]

4.2 Option D2

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~~ the type of that parameter shall be `const` and shall not be an array or function type. [*Note:* This requirement applies even to declarations that do not specify the *postcondition-specifier*. The `const` qualifier of the parameter may be part of a dependent type. Arrays and functions are still usable when declared with the equivalent pointer types ([dcl.fct]). — *end note*] [*Example:*

```
int f(const int i)
post (r: r == i);
int g(int i)
post (r: r == i); // error: i is not declared const.
int f(int i) // error: i is not declared const.
{
  return i;
}

int g(int i) // error: i is not declared const.
{
  return i;
}
template <typename T>
void f(T t) post(t > 0);

int main() {
  int i = 1;
  f<int>(i);        // error: non-const parameter odr-used in postcondition
  f<const int>(i);  // OK
}
```

— *end example*]

Acknowledgements

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.

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