Return object semantics in postcondition specifiers

Timur Doumler (papers@timur.audio)
Andrzej Krzemieński (akrzemi1@gmail.com)
Joshua Berne (jberne4@bloomberg.net)

Document #:	P3007R0
Date:	2023-12-11
Project:	Programming Language C++
Audience:	SG21

Abstract

In this paper, we propose a set of semantics for the named return object in a postcondition specifier in the Contracts MVP. We consider its value category, type, address, and the circumstances under which it is well-defined behaviour to modify this object in a precondition predicate. It remains to be decided whether this object should be implicitly `const` inside the postcondition predicate to make such modifications ill-formed.

1 Introduction

The Contracts MVP ([P2900R2]), which is currently in development and targeting C++26 (as per SG21's roadmap in [P2695R1]), provides syntax to name the return object in a postcondition specifier so it can be used in the postcondition predicate:

```
// This function returns a positive integer
int f()
  post (r: r > 0);
```

However, the exact semantics of the identifier `r` in the above declaration — which we will call the *return-name*, following the grammar term introduced in [P2932R2] — were never specified. The only piece of wording that talks about it can be found in the pulled C++20 proposal [P0542R5], and later in the Contracts MVP wording section in [P2388R4]:

> A postcondition may introduce an identifier to represent the glvalue result or the prvalue result object of the function.

This wording is very vague. It does not say what value category `r` actually has: is it a prvalue, an xvalue, or an lvalue? It also does not say what type it has: is it a reference to the return object? If it is not a reference, what exactly is it, given that the return object itself is never declared and has no name, and what should `decltype(r)` be? What guarantees can we provide about the address of `r`? Under which circumstances is it well-defined behaviour to modify the return object in a postcondition predicate? Should we add a provision to make such modifications ill-formed, given that they almost always constitute an unintended bug? And if yes, how could we achieve that?

In this paper, we discuss all of these questions and propose a set of semantics to answer them.

2 Discussion

2.1 Value category of ***return-name***

In order to specify how the name for the return object behaves, we first need to specify what value category it has. Is it a prvalue, an xvalue, or an lvalue?

The first option is to make the return name a prvalue, similar to `this`, which is also a magic name for a special object available only in a certain context, and is also a prvalue. However, upon closer inspection, this does not work. We need to be able to do useful things with the return name, for example to call member functions on it:

```
X f()
  post(r: r.is_valid());
```

However, a prvalue does not represent an actual object, rather it represents a potential object — a value that can be used to initialise an object. As such, one cannot call a member function on a prvalue; first, it has to be materialised into a temporary object, which is a glvalue, and then the member function will be called on that. This implicit materialisation works for `this` because `this` is a prvalue to a pointer — anything useful we can do with `this` gets the value of the pointer by implicitly materialising it into a pointer temporary, dereferencing that, and giving us an lvalue for the object itself. This does not work for an object of arbitrary type, for example `std::unique_ptr`:

```
// This function returns a non-null pointer
std::unique_ptr<T> f()
  post (r: r);
```

If `r` was a prvalue here, it would be used to materialise a `std::unique_ptr` object in the predicate, thus using up the value and making it unavailable to actually be returned from the function.

Making the return name an xvalue is not an adequate choice either. When we refer to the return object in the postcondition, it is not immediately expiring like an object returned from a function call or cast returning an rvalue reference. We do not want overload resolution to choose a move constructor as it would for an rvalue reference as this may result in unintended and unintuitive behaviour.

It therefore follows that the return name in the postcondition predicate must be an lvalue.

2.2 Type of ***return-name***

First of all, note that the *return-name* cannot be a copy of the actual return value, because the function's return type might be non-copyable. Furthermore, it has to work for a function relying on guaranteed copy elision:

```
struct M {
  M(M&&) = delete; // non-copyable, non-movable
  M(int _i) : i{_i}  {}
  int i;
};

M getM()
  post(r: r.i > 0)
{
  return M{3}; // no M moved or copied
}

int main() {
  M m = getM(); // This is the only M created in this program
}
```

In the program above, the `r` in `post` needs to refer to the object `m` inside `main`, which needs to be already initialised by the time the postcondition of `getM()` is evaluated.

Therefore, we propose a model where the *return-name* is not actually a variable or a reference, but a name, very similar to the identifiers in structured bindings. The *return-name* introduces an identifier that is the name of an lvalue referring to the unnamed return object of the function, similar to how we specify in [dcl.struct.bind] that the i-th identifier in a structured binding's *identifier-list* is the name of an lvalue referring to the element i of the unnamed object or array introduced by the structured binding. For a *return-name* `r`, `decltype(r)` behaves in the same way as for structured bindings: for a function returning a value of type `T`, it should just be `T`, which is the least surprising to the user; `decltype((r))` would be `T&`, following the usual rules for `decltype` of expressions.

Alternatively, we could make *return-name* a reference (or possibly a `const` reference) to the return object. This would mostly behave like the solution we propose, except that it would make `decltype(r)` a reference type, even if the function returns a non-reference type, and no reference has been declared anywhere. We do not see a compelling reason to introduce the `decltype` inconsistency. The `decltype` of *return-name* should match the return type of the function, whether that type is a reference or not.

2.3 Address of the object referred to by *return-name*

An interesting subtlety arises for trivially copyable types, values of which may be returned in registers. For such types, if the *return-name* referred directly to the return object of the function, adding a postcondition to a function might be an ABI-breaking change: the *return-name* is an lvalue, which means one can take its address, which an object in registers does not have, therefore the object could no longer be returned in registers.

This would be an unfortunate outcome. However, there is a workaround. To allow passing return objects of trivially copyable types in registers, there is a provision in [class.temporary] p. 3 that implementations are permitted to create a temporary object to hold the result. We can therefore specify that for a function returning an object of trivially copyable type, *return-name* may refer to this temporary object instead of referring to the actual return object.

The only user-observable consequence of this workaround is that in code that uses the address of the *return-name*, for example,

```
int f(int* ptr)
  post(r: &r == ptr)
{
  return 42;
}

int main() {
  int i = f(&i);
}
```

If the return type is trivially copyable, like `int` here, then there would be no guarantee that the address of `r` matches the address of the object being initialised by the return value, and therefore the postcondition check above may fail. Conversely, if we change the return type to a user-defined type that is not trivially copyable, the above postcondition check is guaranteed to succeed.

To make this work correctly, we need to be rather precise in our specification. Postconditions must be evaluated after a normal return from a function, after *some* object `o` is initialised with the result of the function, after local variables and temporaries are destroyed, but before function parameters are destroyed; the *return-name* is the name of an lvalue that refers to that object `o`; and finally, that `o` may be the return object or, for trivially copyable types, a temporary created by the implementation to hold the result, and may have been initialised prior to evaluation of the return statement (to account for the guaranteed copy elision case).

The above also implies that even if `o` is not the actual return object, which is `i`, there must be a guarantee that `i` will be initialised from `o`, such that, if `o` has been modified while evaluating the postcondition annotation, those modifications will be seen when reading the value of `i` after control has been returned to the call site. This is important because, while an implementation may elide any side effects of a checked contract predicate (see [P2751R1]), it can only elide all such side effects or none of them. If a postcondition modifies the return value and simultaneously something else in the same predicate, eliding the first but not the second could break invariants and would make it very hard to reason about the code, so we should not allow this to happen.

2.4 Modifying `const`-qualified return objects

Since the type of *return-name* matches the return type of the function, it will be `const` if the return type of the function is `const`. Therefore, the following code is ill-formed:

```
const int f()
  post(i: ++i);   // error: cannot modify i with const-qualified type const int
```

It follows that attempt to circumvent the `const`-ness of the return object via `const_cast` is undefined behaviour as per [dcl.type.cv] p. 4 — "Any attempt to modify a `const` object during its lifetime results in undefined behaviour":

```
const int f()
  post(i: ++const_cast<int&>(i));   // undefined behaviour: modifying a const object
```

So far, so good. However, what should happen if the return type of the function is not `const`-qualified, but the return object is declared `const` at the call site? At least for non-trivially-copyable types, we said that the *return-name* always refers to that return object at the call site, after it has been initialised. This implies that the following code would be undefined behaviour as well:

```
struct S {
  S();
  S(const S&);        // not trivially copyable
  int x = 0;
};

S f()                 // return type is not const-qualified
  post(r: r.x = 5)    // oops, wrote = instead of ==
{
  return S();
}

int main() {
  const S s = f();    // ???
}
```

If the code above were undefined behaviour, it would mean that evaluating the postcondition predicate of `f` might be undefined behaviour depending on the call site of the function `f`, which might be located in an entirely different component of the codebase. Such an outcome would be very unfortunate.

The correct mental model here is that the `const` should not take effect until the initialiser of the object completes evaluating; modifying the object is only undefined behaviour from that point on. During the evaluation of the initialiser, which includes evaluating the postcondition specifiers of any functions invoked to evaluate that initialiser, modifications of the value of `s` should be allowed.

These semantics are actually not specific to evaluating postconditions, but also cover cases like an object declared `const` and initialised by a function call modifying the object in its body prior to returning it via guaranteed copy elision, or a delegating constructor modifying an object after it has been constructed by the delegated-to constructor. These cases should not be undefined behaviour either. We need a general clarification of the wording around initialising objects declared `const`.

With these semantics, the code above is well-defined behaviour, unless the return type of `f()` is `const`-qualified. The value of `s.i` inside `main` will be either `0` or `5` depending on whether the implementation decides to elide the side effects of the postcondition predicate evaluation according to the rules in [P2751R1] which we adopted for the Contracts MVP.

2.5 Make *return-name* implicitly `const`?

In the previous two sections, we have considered postcondition predicates that modify the return value. There are some rare cases where the user might actually want to do this, for example changing a value and then changing it back may save an allocation in certain cases. Apart from such rare cases however, modifying the return object in a postcondition will be an unintentional bug, and sometimes such bugs can be hard to find. The last code example contained a typical scenario how such a bug could be introduced: the user might accidentally write `=` instead of `==`. If the return type is implicitly convertible to `bool`, such a predicate will compile and run, but instead of checking the desired postcondition it will modify the return value of the function (subject to side effect elision) if contract checks are enabled.

We could make it harder to write such bugs by statically preventing the modification of the return object in a postcondition:

```
int f()
  post(i: ++i); // should this be ill-formed?
```

First, note that in general the compiler cannot actually prove whether any particular predicate will modify the return object, as function definitions may not be available, and even if they are, doing so would be equivalent to solving the Halting problem:

```
void g(int& i);

int f()
  post(i: g(i)); // will g(i) actually modify i?
```

Therefore, the only way to statically prevent modification of the return object in a postcondition is via the type system. Instead of saying that *return-name* is an lvalue that refers to the return object, we would have to say that it is a `const` lvalue. The effect would be that inside a postcondition, the *return-name* would behave like a non-`const` data member of type `T` in a `const` member function, or like a captured object of type `T` in a non-`mutable` lambda expression. The actual type of the object is still `T`; for a *return-name* `r` referring to it, `decltype(r)` is `T` — this would be the most consistent choice — but `r` is otherwise treated as an object of type `const T`, and `decltype((r))` is `const T&`. This means that `const` overloads will be called; modifying non-`mutable` members or calling non-`const` member functions on the return object or passing non-`const` references to it to other functions from the postcondition predicate are all ill-formed.

This approach would be very effective at preventing accidental modification of the return object in a postcondition. Intentional modification would still be possible — albeit more verbose — via a `const_cast`. Such a `const_cast` would not lead to undefined behaviour as long as the return type of the function is not `const`-qualified (see section 2.4 above). Likewise, modification of `mutable` data members of the return object would still be possible when calling `const` member functions on the object.

There is no consensus between the authors of this paper whether we should pursue this direction. Therefore, we limit ourselves to presenting some arguments both for and against this direction, and leave it up to SG21 to make the decision of whether the benefits of catching bugs due to unintended modifications of the return value outweigh the drawbacks of this direction.

2.5.1 Breaking the symmetry between `pre` and `post`

The first argument against this approach is that the return value in postconditions would behave differently from the actual return type of the function, which is observable through `const` overloads being taken instead of non-`const` overloads. Even more surprisingly, the behaviour of the return value in postconditions would become inconsistent with the behaviour of parameters in preconditions:

```
struct X {};

bool p(X& x)       { return true; }
bool p(const X& x) { return false; }

X f(X x)
  pre(p(x))         // returns true
  post(r : p(r));   // returns false
```

Breaking this symmetry feels conceptually unpalatable; such an inconsistency might make precondition and postcondition annotations harder to teach. It might also make it harder for static analysis of chained function calls to reason about the postcondition of a function call being equivalent to the precondition of a subsequent function call. On the other hand, one might argue that unlike the function parameters, the *return-name* is a new syntax, introducing a way to access an object that was previously inaccessible; unlike the function parameters, it is not declared by the user (neither to be `const` nor to be non-`const`); and therefore there is no real need for the return object in `post` to be consistent with parameters in `pre`, and we are free to pick the safer semantics for it. One might further argue that overload sets where the `const` version of a predicate gives a different answer to the non-`const` version, such as the above, are highly dubious and will not appear in practice.

2.5.2 Interfacing with legacy code

Another argument against the approach of making the *return-name* `const` is that it makes it harder to interface with legacy code. Consider:

```
struct X {
  bool is_valid();
};

X f()
  post(r: r.is_valid());
```

In the code above, the author of `struct X` forgot to make `is_valid()` a `const` function (or perhaps, it is an old C API that takes a non-`const` raw pointer to `X`, etc.), and we do not own that code so we cannot modify it. Yet we know for certain that `is_valid()` will not modify the object, so we should be able to tell the compiler to trust us and compile the code above.

The counterargument here is that if we made `r` implicitly `const`, one can still achieve the above by writing:

```
X f()
  post(r: const_cast<X&>(r).is_valid());
```

One can argue that the extra verbosity needed for such cases does not outweigh the benefits of catching bugs due to unintended *actual* modifications of `r`. It does, however, mean that contract annotations cannot serve as a full drop-in replacement for existing assert macros.

2.5.3 Making other entities in contract annotations implicitly `const`

Finally, we could consider fixing the consistency issues between `pre` and `post` by making function parameters inside `pre` also implicitly `const`. Arguably, this would catch even more bugs because modifications of function parameters inside preconditions will also almost always be bugs.

However, it is then not clear why function parameters should be implicitly `const` inside `pre`, but not inside `contract_assert`; it would be surprising if a contract predicate would behave substantially different in a `pre` on the function declaration than in a `contract_assert` as the first statement in the function body.

But if function parameters are implicitly `const` inside `contract_assert`, why should local variables not also be implicitly `const` inside `contract_assert`? Presumably, a correct predicate would not want to modify those either.

This line of thinking leads to the design proposed in [P3071R0]: that variables with automatic storage duration should be implicitly `const`[1] within the predicates of all contract annotations. This would be a rather drastic design change of the current Contracts MVP, at a late stage in the design process. We are not proposing such a design change here. However, should [P3071R0] get approved for the Contracts MVP, making parameter names and other variables of automatic storage duration implicitly `const` within contract predicates, then the only reasonable choice will be to follow suit amd make the return object implicitly `const` as well.

3 Summary

In this paper, we propose a set of semantics for the named return object `r` in a postcondition specifier `post (r: ...)` in the Contracts MVP, where this area is currently underspecified. We propose that `r` should be an lvalue naming the return object of a function, similar to how the identifiers introduced by a structured bindings are names, not references; `decltype(r)` should be the return type of the function.

We propose a special rule to ensure that adding a postcondition specifier does not break ABI if the return type is trivially copyable, and thus an object can be returned in registers. The observable effect of this rule that in these cases, the address of `r` might not be equal to the address of the return object but may instead be the address of a temporary introduced to hold the return value.

We further propose a clarification ensuring that modifying the return object in a postcondition predicate does not inadvertently introduce undefined behaviour if the return object is declared `const` at the call site, unless the return type of the function is itself `const`-qualified. The `const` should not take effect until the initialiser of the object completes evaluating, which includes evaluating the postcondition specifiers of any functions invoked to evaluate that initialiser.

One open question remains: whether the return object `r` in a postcondition specifier should be implicitly `const` to statically prevent bugs due to unintended modification of `r`, for example when the user accidentally types `=` instead of `==`. There is no consensus between the authors of this paper whether the benefits of this direction outweigh the drawbacks; we have presented some arguments for and against and are leaving the decision on this particular question up to SG21. However, should [P3071R0] get approved for the Contracts MVP, making parameter names and other variables of automatic storage duration implicitly `const` within contract predicates, then the return object should be implicitly `const` as well.

Acknowledgements

We would like to thank Jason Merrill, Brian Bi, Tom Honermann, Ville Voutilainen, Jens Maurer, Gašper Ažman, and Lewis Baker for their valuable contributions to the SG21 reflector discussion that led to this paper.

[1]...with the caveat that this `const` is necessarily shallow and can be cast away, unless we go for an even more drastic solution: the one proposed in [P2680R1], which statically enforces that contract predicates do not have any side effects at all. This direction has already been considered, and twice rejected, by SG21.

References

[P0542R5] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, and B. Stroustrup. Support for contract based programming in C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0542r5.html, 2018-06-08.

[P2388R4] Andrzej Krzemieński and Gašper Ažman. Minimum Contract Support: either *No_eval* or *Eval_and_abort* contracts. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2388r4.html, 2021-11-15.

[P2680R1] Gabriel Dos Reis. Contracts for C++: Prioritizing Safety. https://wg21.link/p2680r1, 2022-12-15.

[P2695R1] Timur Doumler and John Spicer. A proposed plan for Contracts in C++. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2695r1.pdf, 2023-02-09.

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

[P2900R2] Joshua Berne, Timur Doumler, and Andrzej Krzemieński. Contracts for C++. https://wg21.link/p2900r2, 2023-11-11.

[P2932R2] Joshua Berne. A Principled Approach to Open Design Questions for Contracts. https://wg21.link/p2932r2, 2023-11-14.

[P3071R0] Jens Maurer. Protection against modifications in contracts. https://wg21.link/p3071r0, 2023-12-10.