Contracts for C++: Postcondition captures

Timur Doumler (papers@timur.audio)
Gašper Ažman (gasper.azman@gmail.com)
Joshua Berne (jberne4@bloomberg.net)

Document #: P3098R2
Date: 2026-05-12
Project: Programming Language C++
Audience: EWG, LEWG

Abstract

We propose the addition of captures to postcondition specifiers. With these *postcondition captures*, postcondition predicates can refer to values of parameters and other entities at the time when a function is called, and use those values when checking the postcondition predicate at the time when the function returns. Postcondition captures allow the user to assert common postconditions inexpressible in C++26, e.g., that `push_back` increments the size of the container by one. In addition, they enable using the value of a non-reference function parameter in a postcondition assertion without having to declare that parameter `const` on all declarations of the function and of all functions overriding it.

Contents

Revision history

R2 (May 2026 mailing):

- Removed capture-by-reference
- Changed capture initialisation and destruction ordering following SG21 guidance
- Rebased paper onto C++26 DIS
- Rewritten and expanded up-front overview section
- Various minor fixes, additions, and editorial changes

R1 (December 2024 mailing, as presented to SG21 on 2024-12-05):

- Rebased paper onto [P2900R11]
- Added discussion of name lookup, side effects, elision and duplication, interaction between postcondition captures and the function body, lifetime extension of temporaries, and failure to evaluate a postcondition capture
- Clarified `const`-ification rules in the initialiser of the postcondition capture
- Rearranged subsections to improve the logical flow of the paper
- Various other editorial changes

R0 (October 2024 mailing):

- Original version

1 Motivation

C++26 postcondition assertions, which are checked when a function returns, provide no way to refer to the state of the program at the time the function was called, such as the "old" value of a function parameter or the "old" state of the object of a member function. Yet, such postconditions are relatively common in C++ libraries including the C++ standard library — for example, the postcondition of `push_back` that, if the function returns normally, the size of the container is incremented by one. Such postconditions are inexpressible in C++26.

Further, C++26 places restrictions on using non-reference parameters in postcondition assertions. Such parameters must be declared `const` on all declarations of the function and any overriding functions, and the function (or any overriding functions) cannot be a coroutine:

```
int f(int i)
  post (r: r >= i); // error in C++26: `i` is not const
```

These restrictions are necessary to allow human readers, compilers, and static analysis tools to reason that a function will not modify the value of a non-reference parameter odr-used in a postcondition before that postcondition is checked, without having to inspect the function body (which might be arbitrarily complicated and/or inaccessible due to being in a different translation unit). Without these restrictions, postcondition assertions could spuriously pass where they should fail, or vice versa, with no apparent reason (for examples, see [P2900R14] Section 3.4.4, and [P3484R2] Section 1).

However, requiring `const` on non-reference parameters as above is awkward and unintuitive, as users have long been taught that such `const` qualifiers on non-defining declarations have no observable effect. It also makes it impossible to use the value of a parameter in a postcondition assertion if, for whatever reason, that `const` cannot be added in all places required (for example, if there are overriding functions whose declaration and/or definition cannot be modified), or if the function must be implemented as a coroutine).

This paper is proposing an extension to C++26 contract assertions that addresses all of the above limitations.

2 Overview

We propose to add a new, optional syntactic construct to postcondition assertions, called *postcondition captures*. Postcondition captures are spelled in the same fashion as lambda captures. They are placed immediately after the `post` contextual keyword and before the predicate:

```
post [captures] (predicate)
```

Postcondition captures are variables constructed when the function is called. They are visible only to the predicate of the postcondition assertion they are associated with. With a

postcondition capture, postconditions that refer to the state of the program at the time the function was called — such as the postcondition of `push_back` that the size of the container is incremented by one — can be expressed as follows:

```
void push_back()
  post [old_size = size()] (size() == old_size + 1);
```

The passed-in value of a non-reference function parameter can be used in a postcondition assertion without declaring the parameter `const`, by capturing that parameter instead (which results in a copy):

```
int f(int i)
  post (r: r >= i);           // error: `i` is not const

int f(int i)
  post [i] (r: r >= i);       // OK, capture by copy
```

Further, postcondition captures on an overridden function remove the need to add `const` to any overriding functions and thus avoid remote code breakage. Finally, postcondition captures make it possible to odr-use non-reference parameters in the postcondition assertions of a coroutine (which is ill-formed in C++26 because coroutine parameters will be moved-from on construction of the coroutine frame regardless of whether they are declared `const`):

```
generator<int> sequence(int from, int to)
  pre (from <= to)
  post [from, to] (g : g.size() == to - from + 1);
  // OK, use copies of the parameters made when function is called
```

However, we do not allow capture-by-reference. This would introduce several specification and implementation challenges (see Section 4.3.4) while not being all that useful in practice:

```
int f(int i)
  post [&i] (r: r >= i);       // error
```

We also do not allow default captures (see Section 4.3.3), or capturing `this` or `*this` (see Section 4.3.5), as these are less useful on postcondition assertions than they are on lambdas. We also do not allow captures on `pre` or `contract_assert` (Section 4.3.6).

Since postcondition captures are variables declared inside a contract assertion, not outside of it, they are not subject to `const`-ification, following the existing C++26 rule (see Section 4.4.1):

```
void increment (Iterator& iter)
  post [iter_old = iter] (++iter_old == iter); // OK, no const_cast needed
```

The proposed evaluation model (see Section 4.5) for postcondition captures is as follows. Evaluating a postcondition assertion with a postcondition capture involves three distinct phases:

- Constructing the postcondition captures;
- Evaluating the predicate of the postcondition;
- Destroying the postcondition captures.

While these phases happen at different points in time, they are part of the same postcondition-assertion evaluation, and thus should be thought of as a *single unit*:
.

- All three phases of a given postcondition-assertion evaluation must be evaluated with the same evaluation semantic (*ignore*, *observe*, *enforce*, or *quick-enforce*).
- All three phases of a given postcondition-assertion evaluation must be checked either caller-side or callee-side (or not at all).
- If the postcondition-assertion evaluation is evaluated multiple times, each phase will be evaluated multiple times, and the side effects of each phase will be observed the that amount of times.
- If the result of the predicate can be predicated without evaluating any of it, and thus the check can be elided, all three phases need to be elided.

Further, our proposed model considers all postcondition captures related to a particular sequence of function-contract assertions as local variables in a scope that contains that sequence. All precondition assertions are evaluated and captures are initialised as the scope begins, then the function is invoked. After the function returns, all postcondition assertions are evaluated, and finally the local variables (the captures) are destroyed when the scope exits.

When precondition assertions and postcondition assertions are interleaved, these evaluations happen in lexical order (see Section 4.5.1):

```
int f(int i, int j)
  post [i] (r: r != i)
  pre (i > 0);
```

In this example, the postcondition capture is constructed *before* the precondition assertion is evaluated because it lexically appears before it.

After all postcondition assertions have been evaluated, destruction of the postcondition captures happens in reverse lexical order of declaration (see Section 4.5.2). It follows that, if any postcondition-predicate evaluations trigger calls to the contract-violation handler, this happens before any destruction of postcondition captures.

If constructing or destroying the capture itself exits via an exception, the contract-violation handler is called; calling `detection_mode()` on the passed-in contract-violation object will return `detection_mode::evaluation_exception` as usual, and calling `kind()` will return `assertion_kind::post_capture`, a new proposed enum value (see Section 4.6.1). If a

postcondition capture would copy or destroy an object that is not copyable or destructible, the program is ill-formed; if the postcondition capture is evaluated during constant evaluation but constructing or destroying it is not a core constant expression, a compile-time contract violation occurs as usual (see Section 4.6.2).

3 History and context

The need to refer to the state of the program at the time the function was called when checking the postcondition of that function arises frequently. It is such a common requirement that many different C++ Contracts proposals, starting with the very first one [N1613] over two decades ago, attempted to tackle this problem in various ways.

3.1 Referring to "old" values

One possible approach is to define an operator which, when applied to a named entity, yields the value that this entity had at the time the function was called. Such an operator was proposed many times: as a magic function `std::old` in [N1669], as a keyword `__old` in [N1773], as a keyword `oldof` in [N1866], and as a keyword `pre` (standing for "previous", not for "precondition") in [N4110]. If size is a data member of the container class, the postcondition of `push_back` that we have been using as an example can be expressed with such an operator as follows:

```
void push_back()
  post (size == oldof(size) + 1);
```

However, if there is no parameter or variable called `size`, and the value has to be obtained via a function call `size()` instead, the operator approach does not provide a way to write this postcondition. It is therefore not a generic solution – a different approach is needed.

3.2 Procedural interfaces

When precondition and postcondition assertions are written as a block containing a sequence of statements, "old" values can be stored in a variable that is declared by the user inside that block, initialised before the function is called, and checked afterwards. In the hypothetical syntax for procedural interfaces [P0465R0] as shown in [P2961R2] Section 6.7, the aforementioned postcondition of `push_back` could be expressed as follows:

```
void push_back()
interface {
  auto old_size = size();
  implementation;
  assert (size == old_size + 1);
}
```

This is the approach taken in [N1613], except that it was following the design of Contracts in the D programming language and thus the proposed syntax was slightly different from the above example.

However, C++26 follows a different design, where each precondition and postcondition is specified separately by a *predicate* – a single expression that evaluates to `true` if the contract is satisfied. In this design, there is no place where one could declare the variable `old_size`, so this approach does not work for C++26 contract assertions.

We might add procedural interfaces as an extension to C++ some time in the future, but this is not currently planned in the C++29 timeframe (see [P3850R0]). The ability to refer to the state of the program at the time the function was called when checking its postcondition assertions is too important to be delayed until procedural interfaces are adopted or to have as verbose an interface for common patterns.

3.3 Non-reference parameters

The problem of breaking postcondition assertions by modifying parameter values in the function body was also already recognised in [N1613]. In that proposal, a parameter was *implicitly* `const` if it was odr-used in a postcondition. In C++2a Contracts [P0542R5], modifying a parameter odr-used in a postcondition was instead specified to be undefined behaviour.

From today's perspective, neither of these options are viable. Instead, C++26 requires the parameter to be explicitly declared `const` on *all* declarations of the function, as well as all declarations of *all overrides* of the function (for an explanation why it is problematic to require `const` on the definition only or make the `const` implicit, see discussion of Option V4 in [P3484R2]). Further, a function that odr-uses a non-reference parameter in a postcondition assertion cannot be a coroutine, because coroutines modify their parameters even if they are declared `const` by the user (see [P2957R2] and [P3387R0]).

Unfortunately, all of these restrictions have significant tradeoffs. Having to declare a non-reference parameter `const` on all declarations is an unusual restriction: in today's C++, the `const` has no meaning on a non-defining declaration and is almost never written.

Further, extending the `const` requirement to all declarations of all overrides can lead to remote code breakage: adding a postcondition assertion to a virtual function will break any overrides of that function and any code using those overrides (see discussion of Option V2 in [P3484R2]).

Finally, adding such a postcondition assertion to a coroutine requires making the function a non-coroutine, for example by wrapping the original coroutine implementation into a non-coroutine wrapper (see example in [P3387R0] Section 2), and making an explicit parameter copy in that wrapper.

There are cases where none of the above limitations and workarounds are acceptable. However, in C++26, the only alternatives to the above are to make the parameter a

reference parameter – which will often make the API worse and prevent optimisations, discouraging the introduction of contract assertions – or to not add the postcondition assertion at all.

3.4 Closure-based syntax

A generic and complete solution to this problem was first proposed in the closure-based syntax proposal, [P2461R1]. That paper was primarily a proposal for a basic contract-assertion syntax, meant to replace the attribute-like syntax (`[[pre: x]]` and `[[post: x]]`) that was the main proposal at the time. The ability to add a lambda-like capture to a contract assertion was included in [P2461R1] as a follow-up extension.

The closure-based syntax proposal had a design issue: it placed the contract predicate inside braces `{...}`, even though C++ usually surrounds expressions with parentheses `(...)` and statements with braces `{...}`, and the predicate is an expression. This made a predicate look like the body of a lambda expression — an entirely different kind of construct.

Therefore, after long deliberation, the "natural syntax" [P2961R2], which places the contract predicate inside parentheses, was adopted instead. However, the idea of postcondition captures from [P2461R1] was recognised as a very desirable feature (see [P2885R3]). Therefore, C++26 contract assertions have been explicitly designed to seamlessly accommodate this extension, and we now propose to include it in C++29.

4 Discussion

4.1 Contract assertions vs. lambda expressions

Postcondition captures are conceptually similar to lambda captures, which allows us to leverage a familiar syntax. However, contract assertions and lambda expressions themselves are distinct C++ features with a number of important differences.

These differences exist because lambda expressions are objects that have their own lifetime and are designed to be passed around, while contract assertions are not — they are correctness checks attached to a function declaration. While lambda captures are effectively data members of that object, postcondition captures should instead be thought of as local variables introduced into the scope of the contract assertion. Due to these differences, we would be ill-advised to simply apply the semantics of lambda captures "as is" to contract assertions.

The first and most obvious difference is that the body of a lambda is a block that can contain an arbitrary sequence of statements, and the value of evaluating the lambda is determined by its `return` statement. On the other hand, the "body" of a contract assertion, its predicate, is not a statement but a single expression that is contextually converted to `bool`.

The second difference is that the definition of a lambda does not say anything about if and when the lambda will be called; it might happen at any later time, in any other context, and on any other thread. On the other hand, it is defined very precisely when a contract assertion is evaluated: when the function is called (`pre`), when control flow reaches the contract assertion (`contract_assert`), or when the function returns control to the call site (`post`). Therefore, if we were to consider contract assertions with captures as a special kind of lambda, `pre` and `contract_assert` would be akin to always-immediately-invoked lambdas, whereas `post` would always be like a lambda invoked at the end of the block in which it is declared.

Finally, the third difference is related to the second: inside the body of a lambda, variables with automatic storage duration from the enclosing scope cannot be used directly (only captured), nor would it make sense to do so, as they would possibly no longer be alive by the time the lambda is called:

```
int a = 0;       // global
class X {
  int b = 1;     // member
  void test() {
    int c = 2;   // uncaptured local
    int d = 3;   // captured local
    auto f = [d] {
      /*  you can refer to a here, but not to b or c;
      referring to d refers to the captured variable,
      not the one in the outer scope */ };
  }
};
```

On the other hand, for a contract assertion, such lifetime issues cannot arise, not even for postcondition assertions – all variables that are alive when the function is called are still alive when the postcondition is checked. Therefore, there is no reason to "hide" variables from a contract assertion predicate, and name lookup in the predicate sees the same entities as the function body does:

```
int a = 0;
class X {
  int b = 1;
  void test()
    pre (/* you can refer to a and b here */)
    post (/* you can refer to a and b here */)
  {
    int c = 2;
    int d = 3;
    contract_assert (/* you can refer to a, b, c, and d here */);
  }
};
```

4.2 Name lookup

Due to the fundamental differences between lambdas and contract assertions described in Section 4.1, the interaction between postcondition captures and name lookup in the predicate will necessarily have to work differently from the interaction between lambda captures and name lookup in the function body. In particular, postcondition captures introduce new variables into the scope of the predicate that can now shadow variables from outer scopes that would otherwise be found in the predicate by name lookup. Consider:

```
bool b = true;

void f1() post (b) {        // (b) refers to outer b, evaluates to false
  b = false;                // modifies outer b
}

void f2() post[b = b] (b) { // (b) refers to capture, evaluates to true
  b = false;                // modifies outer b
}
```

On the other hand, within the postcondition capture itself, variables are not shadowed by other captures:

```
int i = 1;
int j = 2;

void f() post [i = 3, j = i] (j == 1) {} // `j == 1` evaluates to true
```

This is identical to how captures work for lambdas:

```
int i = 1;
int j = 2;

int k = [i = 3, j = i] {
  return j;  // returns 1, not 3
}();
```

4.3 Kinds of captures and kinds of contract assertions

For our proposal, we need to decide which subset of lambda captures (init-captures, explicit and implicit captures by copy, explicit and implicit captures by reference, capturing `this`) makes sense for which kinds of contract assertions (`pre`, `post`, `contract_assert`). We explore this question in this section.

4.3.1 Init-captures on postcondition assertions

The ability to write init-captures on postcondition assertions is the "must-have" minimal feature to provide the functionality missing in C++26. These init-captures are allowed to

capture arbitrary entities, not just function parameters, which enables a large range of use cases.

For example, we can record the input size of a container rather than having to copy the whole container. We can store the underlying pointer value of a smart pointer, rather than having to copy the whole pointer (which may not be possible, e.g. `std::unique_ptr`). We can also record any other program state at the time of the function call, such as current values of shared variables, the current time, etc.:

```
// process must complete within a given deadline
void process(duration max_process_time)
  post [starttime=gettime()] (gettime() - starttime >= max_process_time);
```

Many other motivating use cases for such init-captures on postcondition assertions are given in [P2461R1].

4.3.2 Explicit capture-by-copy on postcondition assertions

The ability to capture explicitly by copy is not strictly necessary, because the same effect can be achieved with an init capture. However, it is a familiar shorthand syntax and users will expect it to work:

```
// Capture-by-copy syntax
int min(int x, int y)
 post [x, y] (r: r <= x && r <= y);
```

```
// Equivalent init-capture syntax
int min(int x, int y)
 post [x=x, y=y] (r: r <= x && r <= y);
```

We do not see any benefit in making the syntax on the left-hand side ill-formed, as that would simply impose more typing on the user to achieve the same result. We therefore propose that it should be allowed.

The next question is whether we want to allow capturing variables other than function parameters by copy with the above syntax. However, every capture-by-copy introduces a name to the predicate context that shadows another name which otherwise *would* be available in the postcondition predicate, and unlike for lambdas, this shadowing also happens for *local* variables. This makes it more difficult to reason about the meaning of the predicate.

For parameters, this is not too much of a concern, because the name that is being shadowed is right there both in the function parameter list and in the capture, and the object being shadowed will go out of scope anyway after the postcondition predicate is evaluated and the function returns. However, all other variables are defined farther away. Shadowing them by capture would therefore make it more difficult to reason about what name is being shadowed in the predicate, what it means, and which object is being accessed where.

Therefore, we propose that only function parameters should be captured-by-copy in a postcondition capture:

```
namespace X {
  int i = 0;
  int f1()      post [i] (r: r > i); // error: cannot capture non-param i
  int f2(int j) post [j] (r: r > j); // OK
};
```

4.3.3 Default capture-by-copy on postcondition assertions

We propose that default captures on contract assertions should be ill-formed:

```
int min(int x, int y)
  post [=] (r: r <= x && r <= y);  // error
```

The benefit-cost ratio for allowing this variation, even for function parameters, does not seem quite as favourable as for explicit capture-by-copy. Yes, we get an even shorter syntax for operating on copies of non-`const` non-reference parameters in the postcondition predicate; however, this syntax makes it more difficult to read and interpret the predicate correctly.

If the capture is explicit (either as an init-capture or as an explicit by-copy capture), the name of every copied object appears in the capture immediately left of the predicate, which is easy to see. However, if the capture is implicit, it is no longer immediately obvious which variables appearing in the predicate refer to the original object and which are copies local to that predicate. Remember that for lambdas, this ambiguity can never arise: if the variables in question were not captured, they would not be accessible in the body of the lambda at all.

4.3.4 Capture-by-reference on postcondition assertions

We propose to make all forms of capture-by-reference in contract assertions (whether explicit or implicit) ill-formed:

```
void f(int i)
  post [&i] (r: r > i);    // error
  post [&i=i] (r: r > i);  // error
```

If we allowed capture-by-reference of parameters, we would have to guarantee that the caller side and the callee side always refer to the same parameter object. This does not work well with parameter types that can be passed via registers (see [P3487R0]): despite holding a reference to the parameter object, the postcondition assertion is not going to consistently observe changes that the function body does to its value, as it might be operating on a temporary copy ([class.temporary] paragraph 3).

Further, if the function is implemented as a coroutine, a reference to a parameter in a postcondition assertion would refer to a moved-from object ([dcl.fct.def.coroutine] paragraph 14), which is arguably surprising and hostile to the user.

Finally, we would be left with two choices — both suboptimal — regarding the relationship of these references to the lifetime of the parameter objects. The first would be to require that the referenced object's lifetime always extend past the end of the evaluation of the

postcondition assertion. While standard C++ does guarantee that today (postcondition assertions are evaluated before parameter destruction, see [expr.call]), that guarantee does not always hold with compiler extensions such as types with the `[[clang::trivial_abi]]` attribute (e.g., libc++'s `std::unique_ptr`) which may be relocated (ending the lifetime of the original parameter object). This restriction would then hinder adoption of such advancements on top of potential future trivial relocation proposals — but only when postcondition assertions with such by-reference captures are present. This would lead to an ABI change resulting from the presence of a contract assertion, a clear violation of the Contracts Prime Directive ([P2900R14] Section 3.1.1). The other alternative is to introduce no such restrictions, which would lead to dangling references in the same situations — a new potential footgun in the language that does not seem to carry its weight.

For non-parameters, the above issues do not arise, but their utility is questionable. Capture-by-reference of an *id-expression* would essentially do nothing: due to the name lookup rules for contract predicates (see Section 4.2), an *id-expression* referring to an object other than the parameters that would be accessible in the postcondition capture for capturing it by reference is *already* accessible in the postcondition predicate directly without having to capture it. The only effect that the capture-by-reference would have is that for any identifier `x` of non-reference type captured in this way, `decltype(x)` would change from `T` to `T&`. This does not seem useful.

Capture-by-reference of a temporary or rvalue reference is equally problematic, as lifetime extension rules are already complicated enough and might not apply intuitively. Capturing an lvalue reference returned by an expression — such as the results of a lookup into a container— might be useful if that lookup would produce a different result after the function executes, but the same semantics can be achieved by capturing a pointer to that object instead of a reference:

```
void f(int key, std::map<int, int>& m)
   pre (m.find(key) != m.end())
   post [&ref=*m.find(key)] (&*m.find(key) == &ref) // error: ref capture
   post [ptr=&*m.find(key)] (&*m.find(key) == ptr); // OK, address stable
```

This would not limit functionality while making it more clear that the address being captured is a relevant component of what is being checked and when it is being evaluated.

4.3.5 Capture `this` and `*this` on postcondition assertions

In most cases, capturing `this` in a postcondition assertion assertions would have no effect whatsoever: due to the name lookup rules for contract predicates (see Section 4.2), any member or member function that is accessible in the body of the function in question is *already* accessible in the postcondition predicate as well. The only exception to that is the naming of member variables in the postcondition assertion of a destructor, where using the value of such a variable will lead to undefined behaviour (see [P3510R1]). We certainly do not want to make such use easier via postcondition captures.

We therefore propose to make it ill-formed to capture `this` in contract assertions. Note that this is again a consequence of the difference between lambdas and contract predicates discussed in Section 4.1.

Capturing `*this` by copy does not make sense either because that would make it hard to reason about whether members named in the postcondition predicate refer to the captured or the current version of `this`. Therefore, we propose to make capturing `*this` ill-formed as well. Again, the desired effect can be achieved with an init-capture (Section 4.3.1):

```
class DeckOfCards {
  // …
  void shuffle()
    post [old = *this] (std::ranges::is_permutation(old, *this));
};
```

4.3.6 Captures on precondition assertions and assertion statements

So far we have only considered postcondition assertions. It has been suggested that captures could be added to precondition assertions and assertion statements as well. However, for these other kinds of contract assertions, captures would necessarily behave rather differently.

At first, captures on `pre` and `contract_assert` seem useful for generating local copies that can be accessed only in the predicate, for example if the predicate check requires modification of a parameter value but we do not want to modify the original object:

```
void (Iter begin, Iter end)
  pre [begin, end] (begin != end && ++begin != end);
```

However, for `pre` and `contract_assert`, the predicates of these contract assertions are checked immediately after such a capture is constructed; there is no temporal separation between the two as there is for `post`. Therefore, captures on `pre` and `contract_assert` would not actually provide any new functionality; they would just be a shorthand syntax for something we can already write in C++26 using immediately-invoked lambda expressions:

```
void (Iter begin, Iter end)
  pre ([begin, end] { return begin != end && ++begin != end; }());
```

We conclude that captures for `pre` and `contract_assert` are non-essential syntactic sugar that would complicate specification and implementation. In addition, the difference in behaviour between captures for `post` (which "saves" values for later, something that can otherwise not be done) and captures for `pre` and `contract_assert` (which do no such thing) would likely be not immediately obvious to users. We therefore propose to make captures on any contract assertions other than `post` ill-formed.

4.4 Other static properties

4.4.1 `const`-ification

To discourage destructive side effects in contract predicates, C++26 makes *id-expression*s that refer to objects declared outside of the contract assertion `const` by default. However, this `const`-ification does not apply to objects declared inside the contract assertion, as they are not observable outside of the cone of evaluation of the contract assertion and thus highly unlikely to cause destructive side effects.

Postcondition captures declare objects that are only observable within the predicate of the postcondition. Following the logic above, we therefore do *not* apply `const` by default for the captured entities, following the original proposal in [P2461R1]. For example, the following code is well-formed, without requiring a `const_cast` around `iter_old` in the predicate:

```
void increment (Iterator& iter)
  post [iter_old = iter] (++iter_old == iter);  // OK
```

The above semantics seem at odds with lambda captures, which do have `const` applied by default unless the lambda is declared mutable. However, as discussed above, a contract predicate is rather different from a lambda.

First, it is just a single expression converted to `bool` and therefore much simpler than a lambda body, so any possible mutations will be more obvious.

Further, if a developer decides to explicitly capture an object for local use inside the predicate, it is very likely that any mutation of this local object will be intentional.

Finally, part of why a lambda's call operator is `const` is to make it not relevant whether it is evaluated on a lambda or a separate copy of a lambda: when storing callbacks, this can be the source of subtle bugs and surprises when copies of the same function are stored in different places. Postcondition assertions will never suffer from that: each set of postcondition captures is initialised exactly once and then used at most once, in pairs together. Therefore, `const` would not prevent any of the lambda-specific concerns that lead to lambda call operators being `const`, nor are they relevant for postcondition assertions.

While the postcondition captures themselves are not `const`-ified, objects declared outside of the contract assertion are still `const`-ified as usual even if they appear in the initialiser of the postcondition capture:

```
int i = 0;
void f()
  post [j = ++i] (j);  // error: cannot modify const lvalue `i`
```

4.4.2 Capturing parameter packs

The grammar for lambda captures allows capturing parameter packs, both as init-captures and as explicit capture-by-copy. We see no good reason to disallow this for postcondition

captures, with the same grammar. This avoids unnecessary inconsistency and can be occasionally useful:

```
template <typename... Args>
void test(Args... args)
  post [args...] (r: ((r < args) && ...));  // capture-by-copy

template <typename... Args>
void test(Args... args)
  post [...x=args] (r: ((r < x) && ...));   // init-capture
```

4.4.3 Ordering with respect to attributes

Unlike with lambda captures, postcondition captures appear syntactically *after* attributes appertaining to the contract assertion:

```
int f(int i)
  post [[vendor::always_enforce]] [i] (r: r >= i);
```

The rationale is that the capture relates to the predicate, and affects the meaning of the predicate expression, so it should be as close as possible to it, while the attribute relates to the entire contract assertion, and may affect properties unrelated to the predicate expression such as the evaluation semantic or the error message (see [P3088R1]) so it should be as close as possible to the introducing keyword (`pre`, `post`, or `contract_assert`). These syntactic rules follow the direction set by the C++26 contract assertion syntax [P2961R2], which was designed from the start to accommodate postcondition captures.

4.5 Evaluation model

While evaluating a captureless C++26 postcondition assertion only involves evaluating its predicate (or not), evaluating a postcondition assertion with a postcondition capture now involves three distinct phases:

- Constructing the postcondition captures;
- Evaluating the predicate;
- Destroying the postcondition captures.

In this section, we discuss how these phases should be sequenced relative to each other and relative to the evaluation of other precondition and postcondition assertions and other evaluations in the program. The proposed evaluation model follows two key design principles.

The first design principle is that, even though evaluating a postcondition assertion now involves these three distinct phases, the postcondition assertion should still be thought of as a *single unit* with regards to properties such as: what evaluation semantic it is evaluated with (*ignore*, *observe*, *enforce*, or *quick-enforce*), whether it is checked caller-side or callee-side, whether it is elided or duplicated and any side effects are observed, etc.

The second design principle is that all postcondition captures related to a particular sequence of function-contract assertions are considered to be akin to *local variables* in a scope that contains that sequence. As the scope begins, all precondition assertions are evaluated and postcondition captures are initialised, then the function is invoked. The function may then move on to another scope containing another sequence of function-contract assertions, for example, when passing from caller-side checks to callee-side checks. After the function returns, all postcondition assertions are evaluated, and finally the local variables (the captures) are destroyed when the scope exits.

These two principles make the proposed feature easy to teach and understand. Its exact semantics, described in the remainder of this paper, all fall out from them.

4.5.1 Initialisation of captures relative to precondition assertions

When a function is called, the precondition assertions are evaluated and the postcondition captures are constructed. We need to specify the relative order of the two. In particular, C++26 allows interleaving precondition and postcondition assertions, i.e. having postcondition assertions that lexically precede precondition assertions:. This allowance exists because precondition and postcondition assertions often come in semantically-related pairs or groups and the user might want to group all assertions that belong to such a group together (possibly even wrapped by a macro).

The question then arises what the order of evaluation should be if a postcondition assertion with a capture lexically precedes a postcondition assertion? Consider:

```
int f(int i)
  post [i] (r: r != i)
  pre (i > 0);
```

Should the postcondition capture be constructed before or after the precondition is checked?

On the one hand, it seems that checking all precondition assertions first, and only then constructing the postcondition captures, is the more defensive choice as it avoids committing to constructing any postcondition captures prior to checking all of the precondition assertions of the function. This also matches the original proposal in [P2461R1], which says that no postcondition capture is constructed before all precondition assertions are checked.

On the one hand, following the lexical ordering gives a clearer intuition for when evaluations will happen, including initialising captures, and so we follow that lexical ordering here. The initialisation of a postcondition-assertion capture happens as part of evaluating the precondition assertions of the function, in the same lexical ordering that the postcondition assertion is in relation to the precondition assertions of that function. This maintains the concept that precondition assertions (when enforced) will guard against evaluating code that follows them while not introducing surprising reorderings on top of that.

Note that this order also impacts how captures will be initialized in virtual function calls. In a virtual function call — assuming the adoption of `pre` and `post` on virtual functions as proposed in [P3097R2] — the caller-facing and callee-facing contract-assertion sequences

are still evaluated separately with captures initialized as part of one group or the other. Consider:

```
struct X {
  virtual int f(int i)
    post [i] (r: r != i)  // (1)
    pre (i > 0);          // (2)
};

struct Y : X {
  int f(int i) override
    post [i] (r: r > i)   // (3)
    pre (i % 2);          // (4)
};
```

In the above example, the evaluation order is as follows:

- The *caller*-facing postcondition capture at (1) is constructed,
- The *caller*-facing precondition at (2) is evaluated,
- The *callee*-facing postcondition capture at (3) is constructed.
- The *callee*-facing precondition at (4) is evaluated.

4.5.2 Initialisation and destruction relative to other captures

In which order should postcondition captures be constructed and destroyed relative to each other and to other evaluations? Consider:

```
void f()
  post [a = get_a(), b = get_b()] (a == b)   // (1)
  post [c = get_c(), d = get_d()] (c == d);  // (2)
```

From a language design perspective, the following four properties seem desirable:

A. Construct the captures in their lexical order: `a`, `b`, `c`, `d`;
B. Check the postcondition predicates in their lexical order: (1), then (2);
C. Destroy the captures in the reverse order of construction;
D. Destroy the captures associated with each postcondition predicate immediately after evaluating that predicate.

However, it turns out to be logically impossible to satisfy all four properties simultaneously. We can only satisfy at most three of them, but must give up one.

Giving up property A seems highly unintuitive and surprising to users. Clearly, within a *single capture list* the initialisation order must match the lexical order — `a` must be constructed before `b`, and `c` must be constructed before `d`. Relaxing this requirement across *different* postcondition assertions — initialising `c` and `d` in that order, but before `a` and `b`, in order to satisfy all of the properties B, C, and D — seems somewhat less bad, and was considered in a previous revision of this paper, but was explicitly rejected by SG21.

Giving up property B — i.e., reversing the order in which postcondition predicates are checked — seems equally unintuitive and surprising. Consider:

```
int* g(int* p)
  pre (p)
  pre (*p > 0)
  post [p] (*p > 0)         // (1)
  post (r: r)               // (2)
  post [p] ((*r / *p) > 1); // (3)
```

In the above example, it is reasonable to expect that the postcondition predicate at (1) will be checked before the predicate at (3) since (3) depends on (1). Having to write these postcondition assertions in the opposite lexical order to achieve the desired effect would introduce a massive new footgun. Furthermore, it would require a breaking change to C++26. Therefore, this option is out of the question.

Giving up property C would also be unfortunate because this would break fundamental assumptions about the lifetimes of C++ objects being nested, rather than overlapping, in any given scope. Since `a`, `b`, `c`, and `d` are effectively local variables with automatic storage duration, they should always be destroyed in the reverse order in which they were constructed, whatever that order happens to be.

Thus, we propose to give up property D, and to evaluate the postcondition-assertion sequence of function `f` at the beginning of this section in the following order:

- Construct `a`, then `b`;
- Construct `c`, then `d`;
- Execute the body of `f`;
- Check the postcondition predicate at (1);
- Check the postcondition predicate at (2);
- Destroy `d`, then `c`;
- Destroy `b`, then `a`.

This evaluation order fits seamlessly into our proposed programming model, which considers the postcondition captures to be local variables in a scope that encompasses the entire sequence of function-contract assertions that the given postcondition assertion belongs to, rather than a scope that encompasses only the given postcondition assertion itself.

4.5.3 Side effects, elision, and duplication

We do not propose any changes to the rules for side effects, elision, and duplication of contract assertion evaluations. However, construction of the captures, evaluation of the predicate, and destruction of the captures are all considered parts of the *same* postcondition assertion evaluation (and not independent evaluations, even though they are separated in time). The program must observe either all or no side effects resulting from the entire

evaluation; it is not allowed to selectively elide or duplicate some parts of that evaluation but not others.[1]

Concretely, the program is not allowed to observe only the side effects of constructing and destructing the postcondition capture but not those of the associated postcondition predicate or vice versa. If the implementation decides to elide or duplicate the evaluation of a postcondition predicate, it must then also elide or duplicate the construction and destruction of the associated postcondition captures. Conversely, if the postcondition predicate is a provably pure expression that can be trivially elided, but the compiler cannot prove the same property for the constructor and destructor of the capture, and the evaluation semantic is not *ignore*, it must perform construction and destruction the capture. Consider:

```
int i = 0;
bool f() { ++i; return true; }

struct X {
  X() { f(); }
  ~X() { f(); }
};

void g() post [x = X{}] (f()) {}

int h() {
  g();
  return i;
}
```

In the above program, `h()` may return 0, 3, 6, 9 … but not any integer indivisible by 3.

Finally, we need to specify how constructing postcondition captures fits into the rules for sequences of evaluations and repeated evaluations. This is necessary to clearly specify the behaviour for all possible combinations of precondition and postcondition assertions being checked caller-side or callee-side, respectively.[2] Consider again the example from Section 4.5.1:

```
int f(int i)
  post [i] (r: r != i)  // (1)
  pre (i > 0);          // (2)
```

Now, let us consider a highly unusual configuration: the engineer configures the build in such a way that the precondition assertion at (2) is checked callee-side, but the postcondition assertion at (1) is checked caller-side. Since the postcondition capture at (1) is part of the same postcondition assertion as the predicate at (1), the capture must be constructed caller-side as well.

---

[1] Note that this rule already exists in C++26 today: the implementation is not allowed to selectively elide or duplicate some subexpressions of a contract-assertion predicate but not others.

[2] Note that the concept of *caller-side* and *callee-side* checks relates to where the actual checks are laid down by the compiler. This is entirely different from the concept of *caller-facing* and *callee-facing* contract assertions used in [P3097R2] to define the sequence of evaluations for virtual function calls.

At first, it seems that an order of evaluation where the capture at (1) is constructed *after* the check at (2) would be impossible to implement. However, upon deeper inspection, this situation is perfectly compatible with the C++26 rules and those proposed here. The key is to realise that constructing postcondition captures is *a part of the precondition assertion sequence* of a function call, and to remember that whenever the evaluation of a function contract assertion sequence is repeated, an evaluation with the *ignore* semantic also counts as an evaluation. Once we realise this, it becomes clear that when the program is configured as described above, what is actually happening is the following:

- The precondition assertion is evaluated caller-side with the *ignore* semantic,
- The postcondition capture is constructed caller-side with the *enforce* semantic,
- The precondition assertion is evaluated again callee-side with the *enforce* semantic,
- The postcondition capture is constructed again callee-side with the *ignore* semantic,

Thus, this case is just one possible variation of function contract assertion sequence duplication. In practice, the postcondition capture will end up being constructed *before* the precondition is checked, not after, since the precondition evaluation preceding the postcondition capture is an *ignored* one. This might seem surprising, but it is exactly what the user asked for when they configured their program to be compiled in this peculiar way.

4.5.6. Destructive side effects in captures

As described in [P2900R14] Section 3.1 "Design Principles", contract assertions are supposed to observe the state of the program, but not change it, particularly in a way that could influence the correctness of the program that the contract assertions are supposed to assert. Contract assertions that violate these principles are said to have *destructive side effects* and are not a correct use of the Contracts facility.

Destructive side effects are not synonymous with *side effects* in the core language sense (modifying objects and performing I/O). A contract predicate can have destructive side effects even if it has no side effects in the core language sense; conversely, a contract predicate can have side effects in the core language sense, even those that are observable outside of the cone of its evaluation, and yet not have destructive side effects. Consider:

```
void f(const std::string& message)
   pre([]{
     std::ostringstream todiscard;
     return parse(message, &todiscard);
   }() == SUCCESS);
```

Evaluating the above contract predicate may cause a dynamic memory allocation, lock a mutex, interact with `errno`, etc., which is perfectly fine as long as the developer deems those side effects not destructive for their particular program.

A postcondition capture will perform a copy, which creates a new opportunity to introduce destructive side effects to a program. Consider:

```
struct Widget {
  Widget() { /* … */ }
  Widget(const Widget& other) { /* … */ }  // (1)
  ~Widget() { /* … */ }                    // (2)
  // …
};

Widget f(Widget w)
  post [w] (ret: ret.id() == w.id());
```

In this program, it is up to the developer to ensure that the copy constructor (1) and destructor (2) of `Widget` will not have any destructive side effects when evaluated as part of the postcondition capture mechanism.

4.5.6 Interaction between postcondition captures and the function body

Even if the postcondition capture itself does not have any destructive side effects, it can interact with program state outside of the postcondition assertion, which in turn can lead to surprising behaviour. For example, the copy created in the postcondition capture can observe modifications of program state that happen in the *function body*, i.e., in between evaluation of the capture and evaluation of the associated predicate.

In particular, this will happen when attempting to capture an object of a non-regular type that appears to provide value semantics but actually provides reference semantics. Consider a type `Box` that encapsulates a value stored on the heap:

```
template <typename T>
class Box {
  std::shared_ptr<T> _storage = std::make_shared<T>();

public:
  // Default/copy/move ctor, dtor, assignment all compiler-generated
  Box(T value) { *_storage = value; }

  // Getters/setters
  T getValue() const     { return *_storage;  }
  void setValue(T value) { *_storage = value; }
  operator T() const     { return getValue(); }

  // To make an actual (deep) copy of the object:
  Box makeCopy() {
    Box copy;
    copy.setValue(getValue());
    return copy;
  }
};
```

It is questionable whether such irregular data types represent good design, but they are quite common in practice. If such a type is captured-by-copy in a postcondition capture, and then the underlying data is modified in the function body, the postcondition predicate will observe the modified data, even though it operates on a copy of the original parameter object:

```
void append(Box<CharArray> str, Char c)
  post [oldStr = str] (str == oldStr + c);  // will spuriously fail
```

Such behaviour subverts the intent of postcondition captures, but we cannot really do anything about it as the regularity of a type is not a compile-time-checkable property. It gets even more tricky if the `Box` class defined above is used in a generic piece of code:

```
template <IntegerLike T>
int half(IntegerLike  i)
  pre(i >= 0)
  post(r : r >= 0)
  post [i] (r: r <= i) // Capture `i` because I want to modify it below
{
  int h = i / 2;
  i.setValue(0);       // Now I can modify `i`, right?
  return h;
}

int main() {
  Box<int> n(42);
  auto h = half(n);
}
```

In this example, the postcondition assertion will see the modified value of `i` and will therefore spuriously fail. Yet, the source of the problem is not immediately apparent, and it is not immediately clear who is responsible: the author of `Box`, the author of `half`, or the user who used one in conjunction with the other.

This property of postcondition captures is therefore a caveat that needs to be taught explicitly: capturing a function parameter by copy will copy-construct the object that the postcondition predicate will observe and "you get what you get" from that copy-constructed object, including false positives and false negatives if you are using a type with irregular copy semantics.

Note that `const`-ification of the capture, if we were to propose it (we do not — see Section 4.4.1), would not help in such cases.

4.6 Failure to evaluate a postcondition capture

In this section, we discuss the different possible failure modes of postcondition captures and how they are handled in our proposed model.

4.6.1 Constructing or destroying the capture fails at runtime

Constructing and destroying postcondition captures is part of evaluating the associated postcondition assertion. Therefore, if any such construction or destruction fails, this should be treated as a contract violation, and in particular a violation of *that* postcondition assertion (just like a failed predicate check is a violation of that postcondition assertion).

If constructing or destroying a postcondition capture exits via an exception, and the evaluation semantic of that postcondition assertion is *observe* or *enforce*, the contract-violation handler is called, and the function `contract_violation::detection_mode()` returns the enumeration value `detection_mode::evaluation_exception` as usual.

To help the user distinguish between violations originating from evaluating the capture and violations originating from evaluating the predicate, we propose that the function `contract_violation::kind()` should return a new enumeration value `assertion_kind::post_capture` rather than `assertion_kind::post` in this case.

If the violation occurred while constructing a postcondition capture, the contract-violation handler returns normally, and the evaluation semantic is *observe*, any already constructed postcondition captures for *that* postcondition assertion are destroyed before execution continues, and further evaluation of that postcondition assertion is abandoned, i.e., its predicate will not be checked on function exit as performing the check would in fact be impossible (it may access the value of capture objects that have never been constructed). Consider:

```
void f()
  post [a = get_a(), b = get_b(), c = get_c()] (pred(a, b, c))   // (1)
  post [d = get_d(), e = get_e(), f = get_f()] (pred(d, e, f));  // (2)
```

In the example above, if all contract assertions are evaluated with the *observe* semantic, and `get_b()` exits via an exception, the order of evaluation is as follows:

- The captures at (1) are evaluated – `a` gets constructed, constructing `b` throws;
- The contract-violation handler is called; assuming that it returns normally,
- `a` is destroyed,
- The captures at (2) are evaluated – `d`, `e`, and `f` get constructed,
- the body of `f` is executed,
- the postcondition predicate at (1) is skipped,
- the postcondition predicate at (2) is evaluated,
- `f`, `e`, and `d` are destroyed,
- control is returned to the caller of `f`.

If constructing `b` throws and subsequently the contract-violation handler itself exits via an exception, the postcondition captures that have been constructed are destroyed as part of stack unwinding as normal, in reverse order of construction.

If constructing or destroying a postcondition capture throws an exception and the constructor is `noexcept(true)`, `std::terminate` is called as usual. If the function itself exits via an exception and destroying a postcondition capture throws another exception, `std::terminate` is called as usual.

If constructing or destroying a postcondition capture exits via an exception and the evaluation semantic is *quick-enforce*, the program is terminated immediately as usual; no further constructors or destructors of postcondition captures are called.

If constructing or destroying a postcondition capture results in `longjmp` being called, program termination, the thread being suspended indefinitely, etc., then those effects happen as normal, consistent with the existing C++26 rules for evaluating contract assertions.

4.6.2 Constructing or destroying the capture fails at compile time

If a postcondition capture would copy or destroy an object but it is not copyable or destructible, the program is ill-formed, irrespective of the evaluation semantic chosen by the implementation:

```
void f(std::unique_ptr<int> ptr)
  post [ptr] (ptr);  // error: unique_ptr is not copyable

template <std::movable T>
T select(T x, T y)
 post [x, y] (r: r == x || r == y);  // error on template instantiation
                                     // unless T is also std::copyable
```

Postcondition captures can also be evaluated during constant evaluation, following the usual rules. If such an evaluation encounters an expression that is not a core constant expression (for example, a copy constructor or destructor that is neither `constexpr` nor `consteval`), a compile-time contract violation occurs. If the evaluation semantic is *observe*, a diagnostic is issued; if the evaluation semantic is *enforce* or *quick-enforce*, the program is ill-formed.

5 Proposed wording

In this revision, we provide wording for the proposed grammar only. The remaining parts of the wording will be provided in a later revision once EWG has approved the proposed design. The proposed changes are relative to the C++26 DIS.

Modify [dcl.contract.func] as follows:

postcondition-specifier:
`post` *attribute-specifier-seq*$_{opt}$ *contract-capture-clause*$_{opt}$
`(` *return-name*$_{opt}$ *conditional-expression* `)`

Add a new subsection "Postcondition captures" [dcl.contract.capture] after [dcl.contract.res]:

contract-capture-clause:
  [*contract-capture-list*]

contract-capture-list:
  contract-capture
  contract-capture-list , *contract-capture*

contract-capture:
  contract-simple-capture
  contract-init-capture

contract-simple-capture:
  identifier ...$_{opt}$

contract-init-capture:
  ...$_{opt}$ *identifier initializer*

Acknowledgements

Many thanks to Andrzej Krzemieński and Andrei Zissu for reviewing this paper and providing valuable feedback. Many thanks to Lisa Lippincott for the various ideas and discussions that influenced the design of this proposal.

Bibliography

[N1613] Thorsten Ottosen: "Proposal to add Design by Contract to C++". 2004-03-29

[N1669] Thorsten Ottosen: "Proposal to add Contract Programming to C++" (revision 1). 2004-09-10

[N1773] D. Abrahams, L. Crowl, T. Ottosen, J. Widman: "Proposal to add Contract Programming to C++ (revision 2)". 2005-03-04

[N1866] Lawrence Crowl and Thorsten Ottosen: "Proposal to add Contract Programming to C++ (revision 3)". 2005-08-24

[N4110] J. Daniel Garcia: ""Exploring the design space of contract specifications for C++". 2014-07-06

[P0465R0] Lisa Lippincott: "Procedural function interfaces". 2016-10-16

[P2012R2] Nicolai Josuttis, Victor Zverovich, Filipe Mulonde, and Arthur O'Dwyer: "Fix the range-based for loop, Rev 2". 2020-09-29

[P2461R1] Gašper Ažman, Caleb Sunstrum, and Bronek Kozicki: "Closure-based Syntax for Contracts". 2021-10-14

[P2755R0] Joshua Berne, Jake Fevold, and John Lakos:"A Bold Plan for a Complete Contracts Facility". 2023-09-23

[P2806R2] Bruno Cardoso Lopes, Zach Laine, Michael Park, Barry Revzin: “do-expressions”. 2023-11-16

[P2885R3] "Timur Doumler, Joshua Berne, Gašper Ažman, Andrzej Krzemieński, Ville Voutilainen, and Tom Honermann: "Requirements for a Contracts syntax". 2023-10-05

[P2900R14] Joshua Berne, Timur Doumler, and Andrzej Krzemieński: "Contracts for C++". 2025-02-13

[P2957R2] Andrzej Krzemieński and Iain Sandoe: "Contracts and coroutines". 2024-09-26

[P2961R2] Jens Maurer and Timur Doumler: "A natural syntax for Contracts". 2023-09-17

[P3071R1] Jens Maurer: ""Protection against modifications in contracts". 2023-12-17

[P3088R1] Timur Doumler and Joshua Berne: "Attributes for contract assertions". 2024-02-13

[P3097R2] Timur Doumler and Joshua Berne: "Contracts for C++: Virtual functions". 2026-05-11

[P3261R2] Joshua Berne: "Revisiting `const`-ification in Contract Assertions". 2024-11-25

[P3387R0] Timur Doumler, Joshua Berne, Iain Sandoe, and Peter Bindels" Contract assertions on coroutines". 2024-10-15

[P3417R0] Gašper Ažman and Timur Doumler: "Improve the handling of exceptions thrown from contract predicates". 2024-10-16

[P3484R2] Timur Doumler and Joshua Berne: "Postconditions odr-using a parameter modified in an overriding function". 2024-11-14

[P3487R0] Timur Doumler and Joshua Berne: "Postconditions odr-using a parameter that may be passed in registers". 2024-11-07

[P3510R1] Nathan Myers and Gašper Ažman: "Leftover properties of this in constructor preconditions". 2024-11-20

[P3850R0] Timur Doumler and Joshua Berne: "A proposed plan for extending Contracts in C++29". 2026-05-12