Contracts for C++: Postcondition captures

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

Document #: P3098R1
Date: 2024-12-11
Project: Programming Language C++
Audience: SG21, EWG

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. The ability to do this is needed to specify many basic postconditions that are inexpressible in the current Contracts MVP [P2900R11], for example the postcondition of `push_back` that the size of the container is incremented by one. In addition, postcondition captures provide a way to use 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

1 Motivation... 3
2 History and context...3
2.1 Referring to "old" values..3
2.2 Procedural interfaces.. 4
2.3 Non-reference parameters.. 4
2.4 Closure-based syntax..5
3 Overview.. 6
4 Discussion.. 7
4.1 Contract assertions vs. lambda expressions...7
4.2 Kinds of captures and kinds of contract assertions...................................9
4.2.1 Init-capture in postcondition specifiers... 10
4.2.2 Explicit capture-by-copy in postconditions...................................... 10
4.2.3 Default capture-by-copy in postconditions.......................................11

4.2.4 Capture-by-reference in postconditions....11
4.2.5 Capture `this` and `*this` in postconditions....12
4.2.6 Captures for preconditions and assertions....12
4.3 Other static properties....14
4.3.1 `const`-ification....14
4.3.2 Capturing parameter packs....15
4.4 Evaluation model....15
4.4.1 Initialisation of captures relative to precondition assertions....15
4.4.2 Destruction of captures relative to postcondition predicate....16
4.4.3 Initialisation and destruction relative to other captures....17
4.4.4 Side effects, elision, and duplication....18
4.4.5. Destructive side effects in captures....20
4.4.6 Interaction between postcondition captures and the function body....21
4.4.7 Lifetime extension of temporaries....22
4.5 Failure to evaluate a postcondition capture....23
4.5.1 Constructing the capture throws....23
4.5.2 Destroying the capture throws....24
4.5.3 Other run-time failure modes....25
4.5.4 Compile-time failure modes....25
5 Proposed wording....25
Acknowledgements....26
References....26

Revision history

R1

Design presented during the SG21 (Contracts) telecon 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

Original version published in October 2024 mailing.

1 Motivation

With the current Contracts MVP proposal [P2900R11], many basic postconditions are inexpressible, for example the postcondition that `push_back` increments the size of the container by one. The reason is that in the Contracts MVP, by the time a postcondition predicate is checked – when the function returns – there is no way to refer to the state of the program at the time the function was called, such as the "old" value of a parameter.

As a consequence of this limitation, [P2900R11] 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) may not be a coroutine. These restrictions 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 these restrictions, postcondition assertions could spuriously pass where they should fail, or vice versa, with no apparent reason (for examples, see [P2900R11] Section 3.4.4, and [P3484R2] Section 1), and tools could not reason about the postcondition at the call site without inspecting the function body (which might be arbitrarily complicated and/or inaccessible due to being in a different translation unit). However, they require inelegant (and in some cases, undeployable) workarounds to express postconditions that depend on a parameter value – a need that arises frequently when writing postcondition assertions.

This paper is proposing a post-MVP extension to [P2900R11] that addresses all of the above limitations. We believe that this extension provides an important piece of functionality that improves the usability of Contracts. It is our hope that this extension can be approved for the same ship vehicle as [P2900R11]. However, it is not a *necessary* extension – the Contracts MVP is sufficiently complete, useful, and viable without it.

2 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 is such a basic 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.

2.1 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.

2.2 Procedural interfaces

When precondition and postcondition specifiers 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, in the Contracts MVP, we follow 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 the Contracts MVP.

We might add procedural interfaces as a post-MVP extension some time in the future (see discussion of post-MVP extensions in [P2755R0], [P2885R3], and [P2961R2]). However, the ability to refer to the state of the program at the time the function was called when checking its postconditions is too important to be delayed until procedural interfaces are adopted or to have as verbose an interface for common patterns.

2.3 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, the Contracts MVP 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 en explanation why requiring `const` on the definition only, and why the `const` cannot be 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, with [P2900R11], 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 Contracts – or to not add the postcondition assertion at all.

2.4 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 Contracts MVP syntax, meant to replace the attribute-like Contracts syntax at the time. As a post-MVP extension, [P2461R1] proposed adding a lambda-like capture to a contract assertion. For a postcondition, the capture is initialised when the function is called, while the predicate is evaluated when the function returns.

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 peculiarity also made the Contracts syntax too similar to lambda expressions, an unrelated C++ feature.

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, the Contracts MVP has been explicitly designed to seamlessly accommodate this extension. With the Contracts MVP now design-complete, the time has come to propose this extension.

3 Overview

We propose to add a new, optional syntactic construct to postcondition specifiers, called *postcondition captures*. Postcondition captures are placed immediately after the `post` contextual keyword and before the predicate:

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

This is the same syntax that was originally proposed in [P2961R2], except that the predicate is in parentheses instead of braces.

Postcondition captures are spelled in the same fashion as lambda captures, except that the only allowed captures are explicit captures-by-copy of function parameters as well as init-captures (see Section 4.2 for discussion).

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);
```

Non-reference function parameters captured by a postcondition capture can be used in a postcondition assertion without being declared `const`:

```
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

int f(int i)
  post [&i = i] (r: r >= i);  // OK, init-capture by reference
```

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:

```
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

generator<int> sequence(int from, int to)
  pre (from <= to)
  post [&from=from, &to=to] (g : g.size() == to - from + 1);
  // OK, use original parameter objects (probably not a good idea!)
```

With this proposal, when odr-using a non-reference parameter in a postcondition assertion, the user can now explicitly choose between three options:

- Guarantee that the parameter cannot be modified in the function body by declaring it `const` on all declarations of the function and all its overrides, and not defining the function or any of its overrides as a coroutine;
- Capture the parameter by copy, meaning that the postcondition assertion is guaranteed to observe the unmodified value even if the parameter is modified in the function body, at the cost of an extra copy;
- Capture the parameter by reference via an init-capture, meaning that the postcondition assertion will observe the same value as the function body, with all the caveats that this entails (see Section 4.2.4 for a discussion why we allow capture-by-reference only via init-captures).

Thus, postcondition captures not only enable the programmer to express postcondition assertions that refer to the state of the program at the time the function was called, but by extension also provide a comprehensive solution to the problem of odr-using non-reference parameters in postcondition assertions, a design issue that has plagued C++ Contracts proposals since their inception.

Syntactically, postcondition captures can only appear on postcondition assertions, not precondition assertions or assertion statements (see Section 4.2.6 for rationale). Further, unlike lambda captures, postcondition captures appear syntactically *after* attributes appertaining to the contract assertion (see grammar productions in Section 5):

```
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 [P2961R2], which was designed from the start to accommodate postcondition captures.

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. 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 postconditions – 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 */);
  }
};
```

Due to this difference, 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
}();
```

4.2 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.

4.2.1 Init-capture in postcondition specifiers

The ability to write init-captures on postcondition specifiers is the "must-have" minimal feature, as it enables writing postcondition predicates that are inexpressible in the current Contracts MVP. As we will see in later sections, all other capture semantics can be expressed either on top of this minimal feature or even with the existing Contracts MVP (albeit at the cost of more verbose syntax).

The init-capture of a postcondition is constructed when the function is called (see Section 4.4 for a discussion of the exact order of evaluation). Thus, we can compare not only "old" and "new" values of parameters, data members, and member function calls such as `size()`, but in fact "old" and "new" results of arbitrary function calls and expressions, for example:

```
// Process data 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 specifiers are given in [P2461R1]. We allow the full range of init-captures on postcondition specifiers that is provided for lambdas, in particular also the ability to init-capture by reference, as this is required for referring to the original parameter object of a function in case that parameter is not `const` (see Section 3).

4.2.2 Explicit capture-by-copy in postconditions

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

For the common case of capturing parameters by copy, 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.

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-local i
  int f2(int j) post [j] (r: r > j); // OK
};
```

4.2.3 Default capture-by-copy in postconditions

We might go a step further and allow default captures by copy:

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

However, even for parameters, the benefit-cost ratio for allowing this variation 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, on the flip side, it becomes 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.

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

4.2.4 Capture-by-reference in postconditions

Allowing capture-by-reference in contract assertions (whether explicit or implicit) other than via an init-capture does not seem beneficial and we therefore propose to make this ill-formed as well.

For parameters, capture-by-reference would allow the postcondition predicate to see parameter values that might have been modified in the function body by the time the postcondition predicate is checked. Such captures are occasionally useful, but the resulting predicates are much harder for human readers, compilers and tools to reason about, therefore we should not make it easy to spell such captures. With our proposal, capturing a parameter by reference can only be done via an init-capture (see Section 4.2.1):

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

For non-parameters, capture-by-reference would essentially do nothing: due to the name lookup rules for contract predicates (see Section 3), 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&`, which does not seem useful. This is due to the fundamental difference between lambdas and contract predicates discussed in Section 3.1. So again, we only allow such capture-by-reference via an init-capture (which allows the user to pick a different identifier for the captured object).

4.2.5 Capture `this` and `*this` in postconditions

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 3), 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 3.1.

Further, capturing `*this` by copy does not make sense 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`. However, again the desired effect can be achieved with an init-capture (Section 4.2.1):

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

4.2.6 Captures for preconditions and assertions

So far we have only considered postcondition specifiers. It has been suggested that captures could be added to precondition specifiers 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 with the Contracts MVP using either immediately-invoked lambda expressions or do-expressions [P2806R2]:

```
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.

The following table provides an overview over the possible syntactic combinations that we could consider adopting from lambda capture syntax. The blue (must-have) and green (important) cases are being proposed. The yellow (not important), orange (questionable) and red (does not seem useful) cases are *not* being proposed.

	`post`	`pre / contract_assert`
`[x = x, &y = y]`	Must-have; allows to express postconditions inexpressible in the Contracts MVP	Can be spelled with an immediately-invoked lambda in the predicate; not important
`[x]`	Can be spelled with init-captures; familiar shorthand for the most common use case, will be expected to work by users. Proposed only for function parameters.	Can be spelled with an immediately-invoked lambda in the predicate; not important
`[=]`	Can be spelled with explicit captures; questionable because implicit form makes it hard to reason about whether an *id-expression* in the predicate refers to a captured entity	Can be spelled with an immediately-invoked lambda in the predicate; not important
`[&x]`, `[&]`	Can be spelled with init-captures; does not seem useful as it essentially does nothing (except a subtle effect on `decltype`)	Does not seem useful as it essentially does nothing (except a subtle effect on `decltype`)
`[this]`	Can be spelled with init-captures; does not seem useful as it essentially does nothing	Does not seem useful as it essentially does nothing

4.3 Other static properties

4.3.1 `const`-ification

To discourage destructive side effects in contract predicates (see also Section 4.4.5), the Contracts MVP makes *id-expression*s that refer to objects declared outside of the contract assertion `const` by default (see [P3071R1] and [P3261R2]). However, objects declared inside the contract assertion do not have the `const` applied, as they are not observable outside of the cone of evaluation of the contract assertion, and modifying such objects is highly unlikely to be a destructive side effect.

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 specifiers will never suffer from that: each set of postcondition captures is initialised exactly once and then used exactly 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 specifiers.

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`
```

4.3.2 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 Evaluation model

While evaluating a postcondition assertion in [P2900R11] simply consists of evaluating its predicate (or not), evaluating a postcondition assertion with a postcondition capture now involves three distinct steps:

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

In this section, we discuss how these steps 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 key design principle of our proposed evaluation model is that, even though evaluating a postcondition assertion now involves these three distinct steps, the postcondition assertion should 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 (see Section 4.4.1), whether it is elided or duplicated and any side effects are observed (see Section 4.4.4), etc.

4.4.1 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, [P2900R11] 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. Therefore, the question 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, we could follow the lexical order and copy-construct the postcondition capture x from the corresponding parameter *before* evaluating the precondition assertion. On the other hand, it seems that checking all preconditions first, and only then constructing the postcondition captures, is the safer option because this way we could avoid bugs due to violating preconditions that could manifest themselves in the postcondition captures. This also matches the original proposal in [P2461R1], which says that no postcondition capture is executed before all preconditions are checked. We follow this proposal here. In the code example above, the parameter `i` is copied in the postcondition capture *after* the precondition assertion checking that the parameter is positive was evaluated.

Note that this order only extends to precondition and postcondition assertions on the *same* declaration. In a virtual function call, the caller-facing and callee-facing contract assertion sequences are still evaluated separately. Consider:

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

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

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

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

4.4.2 Destruction of captures relative to postcondition predicate

Since a postcondition capture will typically be used in the associated postcondition predicate expression, its destruction must be sequenced *after* the evaluation of that expression. However, we propose that the destruction of captures should immediately follow the evaluation of that expression and in fact be considered part of evaluating the predicate, similar to how the destruction of temporary objects created during evaluation of the predicate expression is also considered part of evaluating the predicate.

For this reason, if the predicate expression evaluates to `false` or its evaluation exits via an exception, the postcondition captures will be destroyed *before* the contract-violation handler is called (see Section 4.5 for a detailed discussion of contract-violation handling with postcondition captures). Effectively, evaluating the predicate expression and destroying the associated postcondition captures should be thought of as an atomic operation.

A counter-argument could be made that, for security reasons, we may not want any user-defined code (such as the destructor of the capture) to run after a violation of a contract predicate has been established but before the associated call to the contract-violation handler. In fact, we decided against adopting [P3417R0] for this reason: adopting that paper would have involved running copy constructors of exceptions in after a violation of a contract predicate has been established but before the associated call to the contract-violation handler.

However, this case seems different since the user explicitly opted into destructors of the captured objects being part of the predicate evaluation by explicitly writing the capture, similar to how destructors of temporary objects are already part of the predicate evaluation because the user explicitly wrote that expression creating those temporary objects. Therefore, executing the destructors of captured objects as part of predicate evaluations seems reasonable.

4.4.3 Initialisation and destruction relative to other captures

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

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

It would be extremely surprising if the construction order within a single capture list would not match the lexical order. Therefore, a must be constructed before b, and c must be constructed before d.

Further, a, b, c, and d are effectively local variables with automatic storage duration. As a general rule, local variables should always be destroyed in the reverse order in which they were constructed. Doing anything else means breaking fundamental assumptions about the lifetimes of C++ objects being nested, rather than overlapping, in a given scope. Therefore, if a is constructed before b, then a must be destroyed after b. Further, if a and b are constructed before (after) c and d, then a and b must be destroyed after (before) c and d.

It follows from the above constraints that it is logically impossible to satisfy the following three requirements simultaneously – we can only satisfy at most two of them but must give up one:

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 associated with each postcondition predicate immediately after evaluating that predicate: destroying b and then a immediately after evaluating (1) and destroying d and then c immediately after evaluating (2).

We propose to give up requirement A and to construct the captures for each postcondition assertion in reverse lexical order of these postcondition assertions. With this proposal, the sequence of events in the above code example is as follows:

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

Giving up requirement B and reversing the order in which postcondition predicates are checked, would require a breaking change to [P2900R11]. It also seems highly unintuitive and surprising. Consider:

```
int* f(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.

Likewise, giving up requirement C and *not* destroying the captures immediately after use in the predicate evaluation would be surprising and go against the usual behaviour of automatic variables in C++ that are destroyed when the scope in which they are declared is exited. Further, doing so would preclude treating any failure to destroy the capture as a failure of the associated predicate (see Section 4.5.2) and would introduce discontinuities in destruction order between caller-side and callee-side checked contract assertion sequences (see Section 4.4.4), precluding an efficient implementation.

On the other hand, construction of the closure is an independent operation temporally separated from evaluation of the associated postcondition predicate, and failure to construct the closure is a distinct kind of contract violation (see Section 4.5.1). It seems less important to be able to rely on a particular order of these constructions across different postcondition assertions than to be able to rely on the other properties of the above evaluation sequence.

4.4.4 Side effects, elision, and duplication

We do not propose any changes to the rules for side effects, elision, and duplication of contract assertion evaluations in [P2900R11]. 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). Therefore, the program must observe either all or no side effects resulting from the entire evaluation; for example, it 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. Similarly, 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; it is not allowed to selectively elide or duplicate some parts of that evaluation but not others.[1] 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.

Further, we need to specify how constructing postcondition captures fits into the rules for sequences of evaluations and repeated evaluations specified in [P2900R11]. 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 first example in this section:

```
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.

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 rules in [P2900R11] and the ones 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

[1] Note that this rule already exists in [P2900R11] today: the implementation is not allowed to selectively elide or duplicate some subexpressions of the 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 [P2900R11] to define the sequence of evaluations for virtual function calls.

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 callee-side with the *ignore* semantic,
- The postcondition capture is constructed callee-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 as specified in [P2900R11]. 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.

4.4.5. Destructive side effects in captures

As described in [P2900R11] 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.

Since postcondition captures may perform a copy, they create 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.

4.4.6 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.

4.4.7 Lifetime extension of temporaries

The lifetime of temporaries created during initialisation of the postcondition captures deserves special consideration as it is desirable to avoid gratuitous undefined behaviour due to dangling references. Consider a function that returns a temporary object:

```
std::vector<std::string> getStrings();  // returns a temporary
```

If the above function is used to initialise a postcondition capture-by-copy, everything is fine:

```
std::string f()
  [vec = getStrings()] (ret : ret == vec.at(0));  // OK
```

However, the attempt to bind a reference to that temporary is ill-formed, just like it is ill-formed today for a lambda capture:

```
std::string f()
  [&vec = getStrings()] (ret : ret == vec.at(0)); // error
```

Further, we need to consider the case where the postcondition capture is initialised by an lvalue, but that lvalue itself contains a reference to a temporary object created during the initialisation. Again, if the capture is by copy, everything is fine:

```
std::string f()
  [str = getStrings().at(0)] (ret: ret == str);  // OK
```

However, if the capture is by reference, we need to introduce a special rule to extend the lifetime of any temporary created during evaluation of the capture initialiser until the end of the evaluation of the associated postcondition predicate, in the same fashion as this is done for range-based for loops (see [P2012R2]):

```
std::string f()
  [&str = getStrings().at(0)] (ret: ret == str);  // OK, lifetime of
                                                  // temporary is extended
```

Note that the above lifetime-extension rule is another difference between postcondition captures and lambda captures: lambdas do not lifetime-extend temporaries in this fashion when capturing by reference. This difference in semantics is a consequence of the difference in the underlying model: while lambda captures can be thought of as data members of the compiler-generated closure type, postcondition captures should instead be thought of as local variables introduced into the scope of the postcondition assertion.

4.5 Failure to evaluate a postcondition capture

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).

In this section, we discuss the different possible failure modes in more detail.

4.5.1 Constructing the capture throws

Construction of a postcondition capture happens when a function is called and is temporally separated from evaluating the postcondition predicate and destroying the capture.

Therefore, if constructing a postcondition capture exits via an exception, it is considered a new kind of contract violation. If this occurs, and the evaluation semantic is *observe* or *enforce*, the contract-violation handler will be called with the enumeration value `detection_mode::evaluation_exception` returned from the function `contract_violation::detection_mode()`, and the new enumeration value `assertion_kind::post_capture` returned from the function `contract_violation::kind()`. The latter is introduced so the contract-violation handler can distinguish this case from the evaluation of the *predicate* exiting via an exception.

If 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 (2) are evaluated – `d`, `e`, and `f` get constructed,
- 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 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 (`a` , `f`, `e`, `d`).

If constructing a postcondition capture throws an exception and the constructor is `noexcept(true)`, `std::terminate` is called as usual, regardless of the evaluation semantic.

4.5.2 Destroying the capture throws

Unlike construction, *destruction* of a postcondition capture happens as part of evaluating the associated postcondition predicate and immediately after evaluating the predicate expression; effectively, evaluating the predicate expression and destroying the associated postcondition captures is a single atomic operation.

Following this model, if the predicate expression evaluates to `true` but the destructor of a capture exits via an exception, it is considered a contract violation of the associated predicate; the contract-violation handler will be called with the enumeration value `detection_mode::evaluation_exception` returned from the function `contract_violation::detection_mode()` and the enumeration value `assertion_kind::post` returned from the function `contract_violation::kind()`.

We already discussed in Section 4.4.2 that, if the predicate expression itself evaluates to `false` or exits via an exception, the associated postcondition captures are destroyed *before* the contract-violation handler is called. We now see another reason why it must be so:

otherwise, a single evaluation of one postcondition predicate could trigger an unbounded amount of contract violations (one for the predicate failure and one for each failure to destroy a capture) which would be very unintuitive and make the proposal significantly more complicated.

If the function itself exits via an exception and destroying a postcondition capture throws another exception, `std::terminate` is called as usual. If destroying a postcondition capture throws an exception and the destructor is `noexcept(true)`, `std::terminate` is called as usual. If evaluation of the postcondition predicate calls the contract-violation handler, and the handler exits via an exception causing stack unwinding, the postcondition capture is destroyed as part of that stack unwinding.

4.5.3 Other run-time failure modes

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 rules for evaluation of the predicate.

4.5.4 Compile-time failure modes

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

We are aware of the fact that the evaluation model proposed above will need to be modified before it can be approved by SG21. Therefore, we do not yet propose wording for it until the design has stabilised. That said, the syntax part of the proposal seems less controversial, and precisely specifying the proposed *grammar* extension seems useful. Below, we provide wording for that part of the proposal only.

The proposed changes are on top of the wording proposed in [P2900R11].

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*
> `&` `...`$_{opt}$ *identifier initializer*

Acknowledgements

Many thanks to Andrzej Krzemieński and Andrei Zissu for reviewing this paper and providing valuable feedback.

References

[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

[P2900R11] Joshua Berne, Timur Doumler, and Andrzej Krzemieński: "Contracts for C++". 2023-10-12

[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

[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

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