Contracts for C++: Postcondition captures

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

Document #: P3098R0
Date: 2024-10-14
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 compare them to the values at the time when the function returns and the postcondition predicate is checked. The ability to do this is needed to specify many basic postconditions, for example the postcondition of `push_back` that the size of the container is incremented by one. Yet, such postconditions are inexpressible in the current Contracts MVP.

Contents

1 Motivation

In the current Contracts MVP proposal [P2900R8], many basic postconditions are inexpressible, for example the postcondition that `push_back` increments the size of a container by one. The reason is that in the Contracts MVP, by the time a postcondition is evaluated (when the function returns), there is no way to refer to the "old" value of a parameter or other entity (the value it had when the function was called). The goal of this paper is to add a way to express such postconditions.

We believe that this extension provides an important piece of functionality that improves the usability of Contracts. While we do not propose to merge this proposal directly into the Contracts MVP [P2900R8], and instead propose it as a post-MVP extension, it is our hope that this extension can be approved in the same timeframe as the Contracts MVP itself, and thus be part of the initial release of the standard C++ Contracts facility. That said, this extension is not *necessary* for the Contracts MVP to be viable. We consider the Contracts MVP sufficiently complete, valuable, and releasable without it.

2 Prior work

The need to refer to "old" values in a postcondition is such a basic requirement that it was supported even in the very first C++ Contracts proposal from 2004, [N1613]. Since then, many different Contracts proposals provided the ability to do so, through various means. They can be broadly categorised into three buckets.

2.1 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 essentially the approach taken in [N1613], which was following the design of Contracts in the D programming language (resulting in a syntax 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]), but the ability to express "old" values should not have to wait this long.

2.2 `oldof` operator

A different approach that works for predicate-based contract assertions is that of 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 data member or parameter `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.

2.3 Captures

A generic and complete solution to this problem was first proposed in [P2461R1]: an extension that allows adding a capture to a contract assertion, similar to a lambda capture. For a postcondition, the capture would be evaluated when the function is called, but the predicate would be evaluated when the function returns. The proposal allowed the full syntactic freedom of lambda captures and thus provided the required generic functionality.

Primarily, [P2461R1] was a proposal for a Contracts syntax (to replace the attribute-like syntax which had been the only candidate syntax for the Contracts MVP until then), and only mentioned captures as a post-MVP extension. However, with the Contracts MVP now feature-complete, the time has come to formally propose adding captures. The "natural syntax" [P2961R2] that we ended up adopting for the Contracts MVP has been explicitly designed to seamlessly accommodate this extension. The only difference to the original postcondition captures syntax proposed in [P2461R1] is that the contract predicate is now surrounded by parentheses, not by curly braces:

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

Our proposal makes the above code well-formed. In the next section, we explore the semantics of this extension in more detail.

3 Discussion

3.1 Differences between lambdas and contract predicates

Contract assertions with captures have some similarities to lambdas, but also some important differences. These differences impact how such captures would work. The first such 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 (`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.

The second, related difference is that inside the body of a lambda, you cannot odr-use any variables with automatic storage duration from the enclosing scope, nor would it make sense to do so, as they would possibly no longer be alive by the time the lambda is called; you can only odr-use parameters and captured variables:

```
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, any variables with automatic storage duration that are alive at the time the contract capture is evaluated are still alive when the contract predicate is evaluated, 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 */);
  }
};
```

Therefore, when adding a capture to a contract assertion, instead of making an identifier accessible in the predicate that would otherwise not be accessible (as for a lambda), we might end up changing the meaning of an identifier that is already accessible in the predicate (a situation that does not occur for a lambda without an implicit capture).

Finally, the third 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` and then evaluated.

Due to these differences, we should not just apply the semantics of lambda captures "as is" to contract assertions; we need to carefully explore the available design space with these differences in mind.

3.2 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 value, 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.

3.2.1 Init-capture in postcondition specifiers

The ability to write init-captures on postcondition specifiers is the "must-have" 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 with the existing Contracts MVP, albeit at the cost of more verbose syntax.

The init-capture of a postcondition would be evaluated when the function is called, immediately after evaluating any preconditions and lexically-preceding postcondition captures, and would behave as init-captures on lambdas; the predicate of the postcondition would be evaluated when the function returns control to the caller, with multiple postcondition predicates being evaluated in lexicographical order. 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:

```
void sleep_for(duration sleep_duration)
  post [starttime=gettime()] (gettime() - starttime >= sleep_duration);
```

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

3.2.2 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);
```

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.

Note that the ability to write postcondition captures allows us to lift the restriction in the Contracts MVP that non-reference parameters odr-used in the postcondition specifier must be `const`. With our proposal, a non-`const` non-reference parameter can still not be directly odr-used in the postcondition predicate, but it can be captured to in the postcondition capture, and then the predicate will refer to that capture. In either case, the value of the object is the same: it is the value passed into the function. The difference is that when the parameter is not captured, it must be `const`, so we know the value did not change; however, when the parameter is captured, the predicate will refer to a copy taken at the time the function was called. In either case, this proposal, just like the Contracts MVP, does not allow accessing parameter values in the postcondition predicate that might have been modified in the body – unless the user writes an explicit init-capture-by-reference, at which point they probably know what they are doing (see Section 3.2.4).

Generally speaking, the two main use cases for postcondition captures are: to be able to compare the "old" value of something to its "new" value after a function has run, and to be able to use the value of a non-`const` non-reference parameter in the predicate without having to make the parameter `const`, in cases where this would lead to a less efficient implementation because you would have to take a copy in the function body instead of in the postcondition capture. Init-captures can address both use cases, but are more suited for the first, while capturing parameters by copy is a more convenient syntax for the second.

3.2.3 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, 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.

3.2.4 Capture-by-reference in postconditions

Allowing capture-by-reference in contract assertions (whether explicit or implicit) 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 predicate is evaluated, something that we explicitly decided not to allow because it leads to predicates that both humans and static analysers cannot reason about.

For non-parameters, capture-by-reference would essentially do nothing: due to the name lookup rules for contract predicates, 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.

If the user still wants to capture an entity by reference in a postcondition specifier, they can use explicit init-capture-by-reference, because we do not restrict the functionality of init-captures (see section 3.2.1):

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

3.2.5 Capture `this` in postconditions

Capturing `this` in contract assertions would have no effect: due to the name lookup rules for contract predicates, any member or member function that is accessible in the body of the function in question is *already* accessible in the contract predicate as well. 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. However, capturing `*this` is allowed because this is just a regular explicit capture-by-copy that would create a local copy of the entire object accessible in the predicate.

3.2.6 Captures for preconditions and assertions

So far we have only considered postcondition specifiers. In precondition specifiers and assertion expressions, captures would behave rather differently. At first, it seems 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, because for `pre` and `contract_assert`, the predicates of these contract assertions are evaluated immediately, so the above is just a shorthand syntax for the following predicate that you can already write in the Contracts MVP:

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

At the same time, 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 is just a shorthand for an immediately-invoked lambda), and therefore the meaning of such predicates, can be confusing and not immediately obvious to users. For example, the following captures would behave fundamentally differently:

```
void f(auto&& x)
  pre [_ = std::scoped_lock(x)] (is_uniquely_owned(x));

void g(auto&& x)
  post [_ = std::scoped_lock(x)] (is_uniquely_owned(x));
```

The first capture, on the precondition of `f`, locks `x` only for the duration of the precondition check, while the second capture, on the postcondition of `g`, locks `x` for the execution of the entire body of `g` including the postcondition check. Introducing captures for `pre` and `contract_assert` requires us to teach these differences, at the relatively small benefit of saving a few characters because we do not have to spell the predicate with an immediately invoked lambda. Again, the benefit-cost ratio does not seem to be very favourable. We therefore propose to make captures on any contract assertions other than `post` ill-formed.

The following table provides an overview over the syntactic options and their priorities. Blue (must-have) and green (important) are being proposed; yellow (not essential) and red (appears to be useless) are not being proposed:

	`post`	`pre / contract_assert`
`[x = x, &y = y]`	Must-have part of this proposal; allows to express postconditions inexpressible in the Contracts MVP	Can be spelled with an immediately-invoked lambda in the predicate; does not seem essential
`[x]`	Can be spelled with init-captures; familiar shorthand for the most common use case, will be expected to work by users	Can be spelled with an immediately-invoked lambda in the predicate; does not seem essential
`[=]`	Can be spelled with explicit captures; 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; does not seem essential
`[&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

3.3 Point of evaluation for the capture

Given the following function:

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

When should the postcondition capture be evaluated and the local copy of x be created: before or after the precondition is checked?

On the one hand, checking preconditions and evaluating the postcondition captures are both things that need to happen just before evaluating the function body, so we could consider doing both things in an intermingled fashion, following the lexical order in which the precondition and postcondition specifiers, respectively, are declared (meaning that the local copy of x would be created before the precondition is checked).

On the other hand, it seems that checking preconditions first, and only then evaluating 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.

3.4 Lifetime of captured objects

In which order should captured objects be constructed and destroyed? In C++, local objects 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, given the following precondition and postcondition specifiers:

```
int f(int x)
  post [a = get_a(), b = get_b()] (r: a >= r & r >= b)
  pre (x > 0);
```

the order of events should be as follows: first, the parameter x is initialised; then the precondition predicates are evaluated in lexical order; then a is constructed; then b is constructed; then the body of f executes; then the postcondition predicate is evaluated and the return object is initialised; then b is destroyed; then a is destroyed; then, x is destroyed.

It becomes even more complicated if we have multiple postconditions. In which order should a, b, c, and d be constructed and destroyed in the following function:

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

Given that postcondition predicates in [P2900R8] are evaluated in lexical order, captures must be destroyed sometime *after* the corresponding predicate is checked, and all objects must be destroyed in reverse order of construction. This gives us two possible options:

- **Option 1:** construct a, b, c, d; execute body of f; check predicate #1; check predicate #2; destroy d, c, b, a.
- **Option 2:** construct c, d, a, b; execute body of f; check predicate #1; destroy b, a; check predicate #2; destroy d, c.

On the one hand, Option 1 seems more intuitive, because all captures are constructed in the lexical order in which they are defined. On the other hand, Option 2 has the property that the captures are destroyed immediately after the corresponding predicate using them is evaluated, so the objects do not live longer than they have to. This property can be important, for example if the capture maintains a lock or some other resource.

It is possible to achieve both of these properties simultaneously, with the tradeoff that we would need to modify the current design of [P2900R8] such that postcondition assertions are evaluated in *reverse* lexical order. This modification yields the following third option:

- **Option 3:** construct a, b, c, d; execute body of f; check predicate #2; destroy d, c; check predicate #1; destroy b, a.

Options 2 and 3 have another desirable property: each postcondition specifier can be independently mapped to a procedural interface (and interpreted as "syntactic sugar" for one).

With Option 2, the above function declaration is equivalent to the following (using again the hypothetical syntax for procedural interfaces from [P2961R2] Section 6.7):

```
void f()
interface {  // interface equivalent of postcondition #2
  c = get_c();
  d = get_d();
  implementation;
  contract_assert(c == d);  // predicate of #2
}
interface {  // interface equivalent of postcondition #1
  a = get_a();
  b = get_b();
  implementation;
  contract_assert(a == b);  // predicate of #1
};
```

Note that with Option 2, these two procedural interfaces have to be written in reverse order compared to the version using `post` because if the statement `implementation;` in the first interface calls the second interface, and the statement `implementation;` in the second interface calls the function's implementation, then the two interfaces need to be in reverse order in order for the postconditions #1 and #2 to be checked in the lexical order in which they are written in the version using `post`.

With Option 3, if we are willing to give up on evaluating postcondition specifiers in lexical order, and instead evaluate them in *reverse* lexical order, the above function declaration maps even more directly and intuitively to a function declaration expressed with procedural interfaces, as the lexical order of the postcondition assertions now matches the lexical order of the equivalent functional interfaces:

```
void f()
interface {  // interface equivalent of postcondition #1
  a = get_a();
  b = get_b();
  implementation;
  contract_assert(a == b);  // predicate of #1
}
interface {  // interface equivalent of postcondition #2
  c = get_c();
  d = get_d();
  implementation;
  contract_assert(c == d);  // predicate of #2
};
```

On the other hand, if we choose Option 1 for the lifetime of captured objects, we cannot express the postconditions #1 and #2 with two equivalent procedural interfaces, and therefore a postcondition specifier does not, in general, correspond to an equivalent procedural interface. Instead, the combination of the two postcondition specifiers will be equivalent to a single combined procedural interface:

```
void f()
interface {
  a = get_a();
  b = get_b();
  c = get_c();
  d = get_d();
  implementation;
  contract_assert(a == b);  // predicate of #1
  contract_assert(c == d);  // predicate of #2
};
```

The authors recommend against choosing Option 1. Mapping postconditions to procedural interfaces makes for a simpler ABI, and allows for an easier implementation of contracts on virtual functions as well as for future extensions for "private contracts".

Consider the following example:

```
class Base {
  virtual int f(int b)
    post [pb = make_unique<int>(b)] (*pb > 0)
  = 0;
};

class Derived : Base {
  virtual int f(int d) override
    post [pd = make_unique<int>(d)] (*pd > -1)
  { return d; }
};

Base& get_some_base();

int main() {
  Base& base_actually_derived = get_some_base();
  base_actually_derived.f(5);
}
```

With Option 1, the destructor of d (in `Derived`) would have to be scheduled *after* the evaluation of the predicate `*pa > 0`. (in `Base`). It is difficult to see how this could be achieved in the case where the contract on `Base::f` is caller-checked and the one on

`Derived::f` is callee-checked. On the other hand, the implementability of options 2 and 3 is obvious: the procedural interface fragment for `Derived::f` is inlined into the body, and the procedural interface fragment for `Base::f` is inlined into the caller. A similar scenario will have to be true for "private" contracts, or contracts differing across a dynamic loading boundary in cases where long-term ABI stability is desired. Option 1 seems unimplementable in such cases.

Further, the authors recommend against Option 3. While appealing on its own, it would require a breaking change to [P2900R8] at a late stage in the design process, which could destabilise the entire proposal. Furthermore, even if that aspect would not matter, users might perceive it as unintuitive and surprising if postcondition assertions were evaluated in *reverse* lexical order, especially since this would have to affect *all* postcondition assertions, regardless of whether they have postcondition captures or not.

Our recommended (and proposed) option for the lifetime of captured objects in postcondition captures is therefore Option 2.

The table below visualises the tradeoffs of Options 1, 2, and 3.

	Option 1	Option 2	Option 3
Postconditions evaluated in their lexical order	✅	✅	❌
Captures constructed in their lexical order	✅	❌	✅
Captured destroyed immediately after use	❌	✅	✅
Mapping to procedural interfaces possible	❌	✅	✅
Mapping preserves order of postconditions	❌	❌	✅
No currently known implementability concerns	❌	✅	✅
Pure extension, no modification of [P2900] required	✅	✅	❌

3.5 Mutability of captured entities

Contract annotations are supposed to observe the state of the program, not change that state, exceptions such as logging from a contract predicate notwithstanding. Evaluating a contract predicate should not have any observable, "destructive" side effects that could alter the program logic, in particular by mutating any objects visible outside of the predicate. For this reason, the Contracts MVP makes identifiers that name objects of automatic storage duration `const` by default (see [P3071R1]). Further, for lambdas, identifiers that name captured entities are by default implicitly `const` in the lambda body (which can be disabled by marking the lambda as `mutable`).

The question arises whether we should also apply `const` by default for entities captured by a postcondition capture. This would mean that postcondition predicates such as the following would be ill-formed without a `const_cast` around the captured object:

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

However, this restriction seems unnecessary for contract predicates. Since we only allow init-captures and explicit capture-by-copy, the captured objects are always copies that are local to the predicate and not observable outside; therefore, no destructive side effects can arise by mutating them. Further, a contract predicate is just a single expression converted to `bool` and therefore much simpler than a lambda body, so any possible mutations will be more obvious. If a developer decided 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.

Considering all of the above, we follow the original proposal in [P2461R1] and propose that unlike lambda captures, entities captured by a postcondition capture should be mutable by default.

3.6 Capturing non-local variables

In C++ today, capture-by-copy for lambdas can only copy local variables with automatic storage duration. Global variables, variables at namespace scope, and variables with static storage duration cannot be captured by-copy; instead they can just be used in the lambda body:

```
namespace X {
  int i = 0;
  void test() {
    int j = 0;
    static int k = 0;

    auto f1 = [i] { return i; };  // error: cannot capture non-local `i`
    auto f2 = [] { return i; };   // OK

    auto f3 = [j] { return j; };  // OK
    auto f4 = [] { return j; };   // error: must capture `j` before use

    auto f5 = [k] { return k; };  // error: cannot capture non-local `k`
    auto f6 = [] { return k; };   // OK
  }
}
```

For lambdas, the motivation for this rule is that local variables might no longer be alive at the time the lambda is called, so being able to use them in the lambda body without capturing them would be extremely error-prone.

This motivation does not apply to postcondition captures: the postcondition predicate will be evaluated when the function returns, before parameters are destroyed. Therefore, any variable that is within its lifetime when the postcondition capture is evaluated will still be within its lifetime when the postcondition predicate is checked. So we could theoretically diverge from the lambda model.

However, with postcondition predicates there is another concern that does not exist with lambdas: every capture-by-copy introduces a name to the predicate context that shadows another name which otherwise *would* be available in the postcondition predicate. This makes it somewhat more difficult to reason about the predicate.

The only *local* variables affected by this shadowing in `post` are the parameters of the function. This is not too big 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, non-local variables will be 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. Further, allowing to capture non-local variables (that is, non-parameters) would mean that the set of captures allowed for postconditions would no longer be a subset of the captures allowed for lambdas, giving C++ users more things to learn and be surprised by.

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

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

3.7 Capturing parameter packs

The grammar for lambda captures allows capturing parameter packs, both as init-captures and as explicit capture-by-value. 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
```

3.8 Capturing parameters of coroutines

SG21 has approved adding support for `pre` and `post` on coroutines (see [P2957R2] and [P3387R0]) to the Contracts MVP. At the time of writing, this proposal is waiting to be approved by EWG and merged into [P2900R8].

With this proposal, if a function has a postcondition specifier that odr-uses any function parameter, it is ill-formed for such a function to be a coroutine, even if this parameter is declared `const` by the user:

```
generator<int> sequence(const int from, const int to)
  pre (from <= to)
  post (g : g.size() == to - from + 1);

generator<int> sequence(const int from, const int to) {
  int i = from;
  while (true)
    co_yield from++;  // Error: cannot define function as coroutine:
}                     // odr-using a function parameter in post
```

This restriction is more severe than that on non-coroutine functions, for which odr-using a parameter in post is only ill-formed if the parameter is not declared const on all declarations of the function. This more severe restriction is necessary because a coroutine creates copies of all parameter objects for the coroutine state and moves the values of the original parameter objects into these copies (the copies are necessary to give the coroutine state, which may outlive the function call, access to the parameters). This move-from happens even if a parameter is declared `const` (see [dcl.fct.def.coroutine]/13). Allowing access to such a parameter in post would result in reading potentially moved-from values, which can lead to unexpected behaviour and unintended bugs, and exposes coroutine internals to the interface of a function.

Our proposal for postcondition captures naturally removes this restriction by letting the user be explicit about whether they wish to refer to copies of the parameters, or to the original parameter objects – which may be moved from:

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

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

In addition, just like with non-coroutine functions, the parameters no longer have to be declared `const` in order to be used in this way.

4 Summary

We propose to allow adding *postcondition captures* to postcondition specifiers. Syntactically, these postcondition captures are placed immediately after the `post` contextual keyword, before the predicate, and spelled in the same fashion as lambda captures:

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

Postcondition captures enable us to write postcondition predicates that refer to the values of parameters and other entities at the time when a function is called – after its precondition assertions have been evaluated and before the function body is executed – and compare them to the values at the time when the function returns and the postcondition predicate is checked. This extension is required to express many basic postconditions, such as the one above, which is not possible with the current Contracts MVP.

Unlike for lambda captures, the only allowed captures are init-captures and explicit capture-by-copy. Implicit captures, capture-by-reference, and capturing `this` are all ill-formed because unlike for lambdas, they provide little to no benefit for contract predicates. Further, these captures are only allowed on `post`, but not on `pre` or `contract_assert`; for the latter two, such captures would not add any new functionality but would just be a shorthand for an immediately-invoked lambda inside the predicate while at the same time significantly complicating the proposal. Finally, while any init-capture is allowed on a postcondition specifier, only the parameters of the function can be captured by-copy.

The captured entities are visible only inside the predicate of the postcondition specifier to which the capture applies, possibly shadowing other names visible there.

Regarding the order of construction and destruction of captured objects, the requirement is that all such objects are destroyed in the reverse order of construction to avoid overlapping object lifetimes and comply with the usual expectations on C++ code. There are three options to achieve this: either, these objects are constructed in the order in which the captures are declared (Option 1), or they are destroyed immediately after the relevant postcondition predicate has finished evaluating (Option 2), or both of these are true but the postconditions themselves are checked in reverse lexical order (Option 3). Option 1 would preclude efficient implementation strategies for contracts on virtual functions as well as known desired extensions omitted from this paper. Furthermore, Option 3 would require reversing the evaluation order of postconditions – a breaking change to the Contracts MVP [P2900R8]. Therefore, our recommended solution is Option 2.

Unlike the captured entities in a lambda, the captured entities in a postcondition specifier are mutable inside the predicate by default. The main concern with mutability inside a contract annotation is that the predicate should not have any observable, "destructive" side effects that could alter the program logic, in particular by mutating any objects outside of the cone of evaluation of the contract predicate. Since we only allow init-captures and explicit capture-by-copy, the captured objects are always copies that are local to the predicate and not observable outside; therefore, no destructive side effects can arise by mutating them, and no implicit `const`-ification of *id-expression*s referring to these objects is necessary.

5 Proposed wording

The following changes on top of the wording in [P2900R8] specify the proposed grammar for postcondition captures.

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 [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*

The remainder of the wording will be added once the proposed design has been approved by SG21.

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

[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

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

[P2900R8] Joshua Berne, Timur Doumler, and Andrzej Krzemieński: "Contracts for C++". 2023-09-27

[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

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