Closure-Based Syntax for Contracts

Document #: D2461R0
Date: 2021-10-29
Project: Programming Language C++
Audience: WG21 SG21 (Contracts)
Reply-to: Gašper Ažman
<gasper.azman@gmail.com>
Caleb Sunstrum
<calebs@edg.com>

Contents

1 Introduction

The attribute-derived syntax for contracts is limiting and has as of late come into scrutiny for stepping on the shared space between C and C++. This paper explores an alternative syntax that should hopefully be more powerul while being able to express the same semantics.

This paper does *not* intend to change any semantics from [P2388R2]. If something looks like a proposed change, that's either a hint at a future extension, or a bug in the paper, so please report it.

The author is actively looking for coauthors, so please don't be shy if you feel like you can contribute.

Note: this paper is an exploration. The author does not object to the currently agreed-upon syntax; but it does seem to present certain challenges that this paper tries to address.

2 Proposal

2.1 Syntax

We introduce three context-sensitive keywords: `pre`, `post`, and `assert`. `pre` and `post` are only keywords if at the end of a function declarator. In the future, we can put other keywords between the keyword and the colon (see example).

`pre` and `post` can appear in function declarations after the place for the optional trailing `requires` clause.

Example:

```
auto plus(auto x, auto y) -> decltype(x + y)
    requires(requires(){ {x + y} -> std::integral; }) // show where annotations go
    pre { x > 0 } // proposed here
    pre audit("slow for numeric code") new [&] { // (audit, new) are future extensions
        /* check for overflow - badly */
        (x > 0 && y > 0 ? as_unsigned(x) + as_unsigned(y) > as_unsigned(x) : true) &&
        // since these are conditional-expressions, use '&&' to combine them
        (x < 0 && y < 0 ? as_unsigned(x) + as_unsigned(y) < as_unsigned(x) : true)
    }
    pre [&y, &x, z=x] { // captures that aren't [&] are extensions
        // capture y by reference, x by value and copy x into z
        // check that += does the same thing as +
        (z+=y) == (x+y)
    }
    post [&, x, &ret=return] { /* option 1 */
        // we will be changing x, so need to capture x on entry - an extension
        ret == (x + y)
    }
    post [&, x] (auto const& ret) { /* option 2 */
        // we can also opt to specify the return value to be the only argument
        // of a postcondition.
        ret == (x + y)
    }
try { // just to show where try-blocks go
    assert { x > 0 };
    return x += y;
} catch (...) {
    throw; /* Did you expect something that makes sense? No? Neither did I. */
};
```

2.1.1 Capture syntax

All captures work like lambda captures, except that they can be omitted (we don't need the introducer) in which case the *correctness-specifier-body* block acts as-if the capture is `[&]`.

For `post`, the capture-expression may contain the keyword `return` (for non-`void` functions). In this context, `return` acts as the alias for the return variable, and `decltype(return)` is the same as the function return type.

2.1.2 Proposed syntax, option 1

Let's take a look at the generic syntax of a *correctness-annotation* (to use the term from [P2388R2]):

correctness-specifier:
  correctness-specifier-keyword correctness-specifier-introducer$_{\text{opt}}$_ *correctness-specifier-body*

correctess-specifier-keyword:
  `pre` | `post` | `assert`

correctness-specifier-introducer:
  lambda-introducer

correctness-specifier-body:
  `{` *conditional_expression* `}`

If the *lambda-introducer* is omitted, the *correctness-specifier-body* behaves as-if the *lambda-introducer* was `[&]`.

If the *correctness-specifier-keyword* of the *correctness-specifier* is `post`, any *initializer* of any *init-capture* in the *lambda-introducer* may be the keyword `return`, which behaves as an *id-expression* denoting the glvalue or the prvalue result object of the function.

For compatibility with [P2388R2], we are currently forcing the `_lambda-introducer_` for `pre` and `assert` to be either `[&]` or omitted. This gives us the exact semantics [P2388R2] specifies, but the future extension is obvious.

For `post`, we must allow the return capture.

2.1.3 Proposed syntax, option 2

We could allow the `post`-condition grammar to be

post-specifier-introducer:
  lambda-introducer `(` *parameter-declaration* `)`

This would separate the parts of the `post` expression evaluated with pre-conditions (the capture) and the parts evaluated at exit (the body).

This does mean that, due to the ability to omit the `[&]` capture, `post`-conditions would most commonly look like

```
post (auto const& ret) { ret == x + y }
```

which is longer than

```
post [&, &ret=return] { ret == x + y }
```

by exactly one character.

The author thinks this is not as desirable, as it overspecifies the type of the return-object, but it does offer a slightly more separated mental model.

2.2 Evaluation order

This section describes the order of evaluation *if contract checking is enabled.* If it's disabled, there is no evaluation.

2.2.1 Assertions

Assertions (any `assert`-based *correctness-specifier*) are executed as if they were immediately-invoked lambda expressions, and are therefore are not a problem.

2.2.2 pre- and post-conditions

We need to make preceding `pre`-conditions protect both the *lambda-introducer* and the *correctness-specifier-body* of any subsequent *correctness-specifier*.

Therefore, `pre`-conditions are executed as is obvious: first the *correctness-specifier-introducer* (if any), and then immediately their *correctness-specifier-body*.

`post`-conditions are evaluated slightly differently; their *correctness-specifier-introducer* is evaluated in-sequence along with `pre`-conditions, **except for any `return`-initializers** (option 1); the binding of those names is deferred until after function exit. This is so we can both capture initial values of parameters and still refer to the result object.

`post`-condition *bodies* are, obviously, evaluated after the function exits.

If a specifier B follows a specifier A in a function's declaration, then no part of B shall not be executed before A has been checked; This restriction is of course relaxed for a `pre`-condition appearing after a `post`-condition, though `post`-conditions still cannot be reordered between each-other.

This means that the following execution orders are all OK:

- A, B
- A, A, B, B
- A, B, A, B
- A, B
- (prove A at compile time), B
- (inherit proof of A from caller precondition), B

Consequence 1: If no `pre`-condition checks fail, the initializers of all initializers of contract blocks (including `post`-blocks) are evaluated at function-entry (except **`return`**-initializers).

Consequence 2: It is valid to run the initializers of all `post`-conditions after the bodies of all `pre`-conditions, since that satisifies the mandate that initializers of successor specifiers run after the bodies of preceding `pre`-conditions.

3 New good stuff

- We get improvements in lambda-capture grammar "for free". Once lambda-introducers get destructuring support, so do contracts, instead of inventing yet another minilanguage.
- We don't have to re-specify anything regarding pack expansions, etc; lambda-introducers get us that, too.
- We can check time/environment-based contracts (see example below).
- Proper support for using the return-value in initializer expressions, and the ability to copy the return value so it's not consumed.
- It's consistent with the rest of the language, instead of inventing a yet-another minilanguage.

3.1 Stateful contracts (extension)

A yet-unserved use-case is checking whether a realtime function actually runs in the time promised; this syntax makes it easy:

```
int runs_in_under_10us()
    post [start=gettime()] { gettime() - start <= 10us };
```

Or, perhaps check we didn't leak any memory:

```
int does_not_leak(allocator auto alloc)
    post [usage=alloc.usage(), &alloc] { usage == alloc.usage() };
```

Or, that `sort` actually returns a permutation:

```
void sort(auto first, auto last)
    post audit [&, input=to<vector>(first, last)]
               { is_permutation(input, {first, last}) };
```

The attribute-derived syntax does not suggest an obvious way to do this, since it doesn't have an obvius closure.

3.2 Destructuring the return value

We need to reach for an immediately-evaluated lambda expression because we don't have destructuring support in lambda-introducers, but that'll change, hopefully, and when it does, we should inherit the fixes.

```
auto returns_triple()
    post [&r=return] { [&] { auto [a, b, c] = r; return c > 0; }() }
{
    struct __private { int __a; int __b; int __c; };
    return __private{1, 2, 3};
}
```

4 Challenges with the attribute-derived syntax

This section explores future extensions as invisaged by [P2388R2] and previous papers.

4.1 Place for annotations like "axiom", "new", etc.

The best idea for where to put such markers it at the end, after a semicolon; from [P2388R2]/8:

This proposal	[P2388R2]

`int f(int* p)`
`    pre {p}`
`    pre new {*p > 0}`
`;`
	`// after ; at end`
	`int f(int* p)`
	`    [[pre: p]]            // stable annotation`
	`    [[pre: *p > 0; new]] // new annotation`
	`;`

`int f(int* p)`
`    pre audit("allows messages") {p}`
`    pre new("2021-09-27") {*p > 0};`
	`// after : at end`
	`[[post r: r > 0: new]]`

	`// in braces at start`
	`[[post{new} r: r > 0]]`

4.2 Referencing function arguments in postconditions

There are issues with arguments that change value during function evaluation and postconditions. They are described in [P2388R2]/6.4 and 8.1. [P2388R2] side-steps this issue by saying that referencing aruments which changed value is UB, which is why referenced arguments should be `const`-qualified (in definitions).

The ideas using the [P2388R2] syntax look like this (all from [P2388R2]/8.1):

```
// This proposal
int f(int& i, array<int, 8>& arr)
  post [&r=return, i] { r >= i }
  post [&r=return, old_7=arr[7]]
       { r >= old_7 }
```

```
// p2388r2 1)
int f(int& i, array<int, 8>& arr)
  [[post r, old_i = i: r >= old_i]]
  [[post r, old_7 = arr[7]: r >= old_7]];
```

```
// p2388r2 3)
int f(int& i, array<int, 8>& arr)
  [[post r: r >= oldof(i)]]
  [[post r: r >= oldof(arr[7])]];
```

```
// p2388r2 2)
int f(int& i, array<int, 8>& arr)
  [[post r: r >= oldof(i)]]
  [[post r: r >= oldof(arr[7])]];
```

Table 3: Another oldof example:

This proposal

```
template<class ForwardIt, class T>
ForwardIt find(ForwardIt first,
               ForwardIt last,
               const T& value)
  post [&r=return, first]
       { distance(first, r) >= 0u }
  post [&r=return, &last]
       { distance(r, last) >= 0u }
{
  for (; first != last; ++first) {
    if (*first == value) {
      return first;
    }
  }
  return last;
}
```

P2388R2

```
template<class ForwardIt, class T>
ForwardIt find(ForwardIt first,
               ForwardIt last,
               const T& value)
  [[post r: distance(oldof(first), r) >= 0u]]

  [[post r: distance(r, last) >= 0u]]

{
  for (; first != last; ++first) {
    if (*first == value) {
      return first;
    }
  }
  return last;
}
```

4.3 Introducing the return variable

This proposal

```
int f(int* i, array<int, 8>& arr)
    post [&i, r=return] { r >= i };
```

P2388R2

```
int f(int& i, array<int, 8>& arr)
  [[post r: r >= i]];
```

```
// alternative
int f(int& i, array<int, 8>& arr)
  [[post(r): r >= 0]]
```

4.4 Preconditions and assertions that need copies

Table 5: [P2388R2] has no answer for preconditions that need to mutate a copy:

<table>
<tr><th>This proposal</th><th>[P2388R2] Does not work</th></tr>
<tr><td><pre>
int f(forward_iterator auto first,
      forward_iterator auto last)
    pre { first != last }
    pre [&, first] { std::advance(first, 1),
                     first != last };
</pre></td><td><pre>
int f(forward_iterator auto first,
      forward_iterator auto last)
    [[pre: first != last]] // ok
    [[pre: std::advance(first, 1), // nope
           first != last]]];
</pre></td></tr>
</table>

4.5 Postconditions that need destructuring [when lambda-captures get it]

Table 6: Functions could concievably have destructure-only APIs:

<table>
<tr><th>This proposal (when lambdas can do this)</th><th>[P2388R2]-esque</th></tr>
<tr><td><pre>
auto returns_triple()
    post [&[x, y, z]=return]
         { x > y && y > z }
</pre></td><td><pre>
auto returns_triple()
    [[post [x, y, z]:
           x > y && y > z]];
</pre></td></tr>
</table>

This syntax kind-of works, but is not proposed, and there is nowhere to specify the binding type (reference or copy?) We haven't even solved this for lambda captures, but we will, and we want to inherit the language once we do.

4.6 Summary

- The closure-based syntax makes it obvious when values are captured, and even suggests an implementation - just put the closures on the stack before the function arguments.
- It doesn't invent another language for capturing values, which means the syntax will grow together with lambda captures.
- It makes it **obvious how to do stateful postconditions** that check before/after: the closure runs with `pre`, the body runs after return. This is far from obvious with the [P2388R2] syntax.

5 Mutation and Static Analyzers

Static analyzers should be able to handle limited mutation in order to analyze C++, and many contracts that describe function behaviour will require some mutation of a copy. Allowing copies to be made is therefore immensely useful in a contract facility.

We have assurances from at least some analyzer vendors they see no issue with allowing copies and mutation in contract annotations in the future.

6 Proposed Wording

TODO. Writing it will be an exercise, and the author wants to see if there is any enthusiasm for this at all before spending the time.

7 Acknowledgements

- *The entire WG21.* This is a huge effort, and since contracts were pulled from C++20, the group has been showing an extraordinary level of determination to get to consensus.
- *Andrzej Krzemieński*, who has been a steadfast integrator of opinion in P2338 - the MVP paper sequence. I've helped a bit, but he's been extraordinary.
- *Tom Honermann*, who saw the interplay with function try-blocks.
- *Phil Nash*, for quite a few insightful comments, and the function-parameter syntax for return values
- *Peter Brett*, for encouraging me to drop the complex sequence of ;-separated conditions and stick to a single condition (subconditions separated by `&&`).
- The *BSI* for reviewing this paper early.
- *Lisa Lippincott*, for her study of stateful function contracts and all the hours she's spent explaining the point and their shape to me.

8 References

[P2388R2] Andrzej Krzemieński, Gašper Ažman. 2021-09-10. Minimum Contract Support: either Ignore or Check_and_abort.
https://wg21.link/p2388r2