Closure-Based Syntax for Contracts

Document #:	D2461R0
Date:	2021-10-11
Project:	Programming Language C++
Audience:	WG21 SG21 (Contracts)
Reply-to:	Gašper Ažman <gasper.azman@gmail.com>
	Caleb Sunstrum <calebs@edg.com>
	Bronek Kozicki <bkozicki@acm.org>

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 proposes almost the same semantics as [P2388R2].

The only significant change from [P2388R2] is the semantics of effect elision - this paper specifies it as all-or-nothing, per *correctness-annotation*.

Due to the way this paper models the thinking around annotations, it may leave fewer things undefined as [P2388R2] does, despite the fact that that we aren't allowing explicit closures yet.

Note: this paper is an exploration. The authors do 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 const x, auto const y) -> decltype(x + y)
    pre { x > 0 }
    pre {
        /* 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)
    }
    // ret is as-if auto&&
    post (ret) { ret == (x + y) }
{
    assert { x > 0 }; // this is currently "valid" syntax,
                      // but we should reclaim it.
    auto cx = x;
    return cx += y;
}
```

2.1.1 Example with future extensions

The above really just assumes the default capture is `[&]`, which we can introduce later.

Modeling using lambda-captures allows us to explain why post-conditions can't refer to rvalue-reference arguments.

Example: (extensions not proposed here)

```
auto plus(auto x, auto y) -> decltype(x + y) // no const
    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 potential extensions
        (x > 0 && y > 0 ? as_unsigned(x) + as_unsigned(y) > as_unsigned(x) : true) &&
        (x < 0 && y < 0 ? as_unsigned(x) + as_unsigned(y) < as_unsigned(x) : true)
    }
    pre [&y, x, z=x] {
        // 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, y] (ret) { // capture x, y by value, *explicitly*
        ret == (x + y)
    }
    post [&x, y] (ret) {
        // ill-formed, by-ref postconditions on value-params are meaningless for caller
        ret == (x + y)
    }
{ // just to show where try-blocks go
    assert { x > 0 };
    return x += y;
}
```

2.1.2 Capture syntax

All captures work like lambda captures, except that they are force-omitted in this MVP. If omitted, the *correctness-specifier-body* block acts as-if the capture is `[&]`.

For `post`-conditions, there is a `(` *identifier* `)` prefix to the

2.1.3 Proposed syntax

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$_{opt}$ *correctness-specifier-body*

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

return-value-decl:
  `(` *identifier* `)`

correctness-specifier-introducer:
  lambda-introducer return-value-id$_{opt}$

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

If the *lambda-introducer* is omitted, the *correctness-specifier-body* behaves as-if the *lambda-introducer* was `[&]`. (This omission is forced in the MVP).

If the *correctness-specifier-keyword* of the *correctness-specifier* is `post`, the *return-value-decl* must be present, and introduces the name for the prvalue or the glvalue result object of the function. This identifier is valid within the *correctness-specifier-body*.

2.1.4 MVP Restrictions

Naming a non-const value parameter in a post-condition is ill-formed for now. This can be lifted by allowing copy-capture later, when we allow the *lambda-introducer* to appear. This is to both prevent **referencing moved-from objects**, and to **allow the calling code to reason** about the properties of the result object, such as in the example:

```
int min(int x, int y)
    post (r) { r <= x && r <= y };
```

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 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 and both are of the same kind (`pre` or `post`), then no part of B shall be executed before A has been proven;

No postcondition closure is executed before all preconditions are checked.

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.

2.3 `post`-condition reference-capture limitations in the MVP

Capturing function parameters by mutable-reference in postconditions may cause difficulties for static analysis, as some expressions containing these will require interprocedural/inter-TU analysis, which may be beyond the capabilities of a compiler. Dedicated static analysis tools should still be able to handle these, however. [P2388R2] forbids mutating function arguments.

Example (courtesy of Tomasz Kamiński):

```
int pickRandom(int beg, int end)
    post [&] (r) {
        ret >= beg &&
        ret <= end
    };
```

Given that we don't know the function body, and we could have changed `beg` and `end`, this conveys no information for static analysis (you'd have to mark `beg` and `end const`).

We therefore have a choice of how to start out with this proposal:

— forbid capturing parameters by mutable reference
— forbid capturing parameters by reference altogether
— do nothing and just expect degraded static analysis performance (capture-by-mutable-reference is not a problem for runtime checking)

The stated goal of feature-bijection with [P2388R2] for this paper says we should forbid reference-capture for parameters in post-conditions and only allow capture-by-value in the MVP.

2.4 Side-effect elision specification

This MVP presupposes that for the purposes of optimization, the compiler is allowed to either execute, or not, entire correctness specifiers, together with their closures. Subexpression elimination is only permitted under the (stricter) as-if rule.

This is because, while it should not be lippincott-discernible to the program whether a specifier was actually executed, this might only actually be true if the specifier gets to clean up after itself. In other words, the sum of the parts is assumed "pure", the parts are not.

3 New good stuff if we pick closure-based semantics

— 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 obvious 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) { [&] { 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 envisaged by [P2388R2] and previous papers.

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

The best idea for where to put such markers is 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 attempting to prevent referencing modified arguments, requiring that 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 [i] (r) { r >= i }
  post [old_7=arr[7]] (r)
       { 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 [first] (r)
       { distance(first, r) >= 0u }
  post [&last] (r)
       { 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) { r >= i };
```

P2388R2

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

This proposal	P2388R2

	`// 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:

This proposal	[P2388R2] Does not work

`int f(forward_iterator auto first,`
`      forward_iterator auto last)`
`    pre { first != last }`
`    pre [first] { std::advance(first, 1),`
`                  first != last };`
	`int f(forward_iterator auto first,`
	`      forward_iterator auto last)`
	`    [[pre: first != last]] // ok`
	`    [[pre: std::advance(first, 1), // nope`
	`         first != last]];`

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

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

This proposal	[P2388R2]

`auto returns_triple()`
`    post (r) { match(r) {`
`         [x, y, z] => x > y && y > z;`
`         _         => false;`
`    }};`
	`auto returns_triple()`
	`    [[post r: match(r) {`
	`          [x, y, z] => x > y && y > z;`
	`          _         => false;`
	`      }]];`

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 Multithreaded Usage

Issue courtesy of Aaron Ballman:

A potential issue with P2388R2 that is carried over into D2461R0 is with side effect operations. Given that they're unspecified, does this mean there's no safe way to write a portable contract which accesses an object shared between threads? e.g., multithreaded program where a function is passed a mutex and a pointer to a shared object; can the contract lock the mutex, access the pointee, then unlock the mutex?

With closure-based semantics, we can avoid this:

```
void frobnicate_concurrently(auto&& x)
    // closures-are-a-future-extension.disclaimer
    pre [g=std::lock_guard(x)] { is_uniquely_owned(x); };
```

In this MVP, we allow the compiler to *assume there are no side-effects* to an expression for the purposes of optimisation, *but they can either all be omitted, or none may*, for a given statement, including the closure.

We therefore have a plausible RAII-based metaphor that people already understand.

4.7 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 immensly 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 What about abbreviated lambdas?

The `post`-condition syntax naturally looks like a shorthand lambda:

```
post [ closure ] ( identifier ) { conditional-expression }
```

This topic was explored in [P0573R2] by Barry Revzin, who proposed this syntax as point 2.3.

Alternative from the paper:

```
post [ closure ] ( identifier ) => conditional-expression
```

However, this doesn't work in function declarations because terminating expressions is difficult, so we like the earlier syntax better.

In this case, the default closure is `[&]`, the parameter-type is `auto&&` and we don't need to take a stance on the `noexcept` specifier or the return type.

However, given the above, EWG should take a stance on whether that looks like a plausible set of semantics for shorthand lambdas, because we shouldn't have a yet another set of semantics for those.

7 C-compatibility

C and C++ implementations often share a set of system headers, and there will naturally be a desire to add contracts to entities in those headers.

One of the motivating reasons behind the attribute-like syntax in [P2388R2] is that a C compiler can be reasonably updated to ignore the contracts unless/until C gets Contracts as well. It's worth noting that the proposed syntax in [P2388R2] is still ill-formed for attributes, and a properly conforming C compiler that has not been updated to handle (ignore) the contracts would still issue diagnostics.

There is some debate as to whether it'd be a *good* thing if a C compiler were to still accept code that has Contracts in it when the C compiler is unaware of Contracts, and it has been noted that some implementations may simply consume all tokens in an unrecognized attribute until reaching the closing `]]`, regardless of whether the internal structure of the attribute is properly conforming.

The syntax proposed in *this* paper, however, cannot be ignored by a C compiler that is unaware of Contracts - it is unarguably ill-formed C code.

This syntax lends itself easily to conditional compilation, especially with a feature-test macro:

```
int my_func(int x)
#if __cpp_contracts /* Perhaps just __contracts to allow C to easily opt-in? */
  pre { x > 0; }
#endif /* __cpp_contracts */
{
  /* ... */
}
```

This is not a motivating difference from [P2388R2] - conditional compilation can just as easily be used to guard Contracts there; the main difference in C-compatibility between these two proposals is that [P2388R2] has a greater potential of a Contracts-unaware C compiler ignoring any contracts without a meaningful diagnostic or programmer opt-in.

8 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.

9 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 Gašper.
— *Tomasz Kamiński*, for *also* pointing out the function parameter syntax for return values, and reminding the authors that reference-captures for non-const parameters render postconditions less useful for static analysis in the absence of the function body.
— *Ville Voutilainen*, for always connecting all of the weird bits of impact everything has on everything else.

10 References

[P0573R2] Barry Revzin, Tomasz Kamiński. 2017-10-08. Abbreviated Lambdas for Fun and Profit.
https://wg21.link/p0573r2

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