

Closure-Based Syntax for Contracts

Document #: P2461R0
Date: 2021-10-14
Project: Programming Language C++
Audience: WG21 SG21 (Contracts)
Reply-to: Gašper Ažman
<gasper.azman@gmail.com>
Caleb Sunstrum
<calebs@edg.com>
Bronek Kozicki
<brok@spamcop.net>

Contents

1	Introduction	2
1.1	On Extensions and Viability	2
2	Proposal	2
2.1	Example	2
2.2	Proposed syntax	3
2.2.1	MVP Restrictions	3
3	Semantics	4
3.1	Evaluation order	4
3.1.1	Assertions	4
3.1.2	pre- and post -conditions	4
3.2	post -condition reference-capture limitations in the MVP	5
3.3	Side-effect elision	5
3.3.1	An alternative model (from P2388)	5
4	Future Extensions (not a proposal)	5
4.1	Explicit Captures	5
4.1.1	Capturing views by value	6
4.1.2	Checking whether a call didn't exceed its time budget	6
4.1.3	Checking a call didn't leak memory	6
4.1.4	Grabbing only the interesting part of an input	6
4.1.5	Mutation semantics checking	6
4.2	Destructuring the return value	7
4.3	Attributes appertaining to contract annotations	7
4.4	"trust" annotations (new)	7
4.5	"cost" annotations	7
4.6	Multithreaded usage / locking	8
4.7	Testing and fuzzing checks	8
4.8	Summary	9
5	Comparison tables with attribute-based syntax	9
5.1	Referencing function arguments in postconditions	9
5.2	Introducing the return variable	10
5.3	Preconditions and assertions that need copies	10
5.4	Postconditions that need destructuring [when lambda-captures get it]	11

5.5 Summary	11
6 Mutation and Static Analyzers	11
6.1 Capture design space	11
7 C-compatibility	12
8 Considered and rejected syntaxes	13
8.1 Abbreviated lambdas	13
8.2 Semicolon separators in the body (a-la requires blocks)	14
9 Proposed Wording	14
10 Acknowledgements	14
11 References	14

1 Introduction

The attribute-derived syntax for contracts is limiting and steps on the shared space between C and C++. This paper explores an alternative syntax that should offer an easier extension path.

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*.

The main non-syntactic difference between the two papers is the manner of spelling future extensions.

Due to the way this paper models annotations, it may leave fewer things undefined compared to [P2388R2], despite the fact that it does not propose 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.

Note: WG14 has communicated that their vendors don't have a blocking problem with the attribute-like syntax, though they have reservations; in addition, WG21 members have expressed difficulties with teaching the *: means it's not an attribute* intricacies.

1.1 On Extensions and Viability

The authors believe that **any MVP must clearly show plausible syntax for all known extensions**. This does not mean *propose*. It means *show*. Specifying precise semantics for the entire extension space is not in the spirit of a *minimum viable product*, but *viability* implies that all desired features can at some point be supported. This means there must be syntax, so syntax we show.

2 Proposal

2.1 Example

We introduce three context-sensitive keywords: **pre**, **post**, and **assert**. **pre** and **post** are only keywords in the top level context of a function declarator.

pre and **post** can appear in function declarations after 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)
  }
  // r is as-if auto&&
  post (r) { r == (x + y) }
{
  assert { x > 0 }; // this is currently "valid" syntax,
                  // but we should reclaim it.
  auto cx = x;
  return cx += y;
}

```

One may note that this is strikingly similar to the syntax proposed in [N1962], way back in 2006. Our thanks to Andrzej Krzemiński for digging this up.

2.2 Proposed syntax

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

correctness-specifier:

precondition
postcondition
assertion

precondition:

pre *lambda-introducer*_{opt} *correctness-specifier-body*

postcondition:

post *lambda-introducer*_{opt} *return-value-decl*_{opt} *correctness-specifier-body*

assertion:

assert *lambda-introducer*_{opt} *correctness-specifier-body*

return-value-decl:

(*identifier*)

correctness-specifier-body:

{ *conditional-expression* }

For the MVP, the *lambda-introducer* is required to be omitted.

If the *lambda-introducer* is omitted, the *correctness-specifier-body* behaves as-if the *lambda-introducer* was [**&**] for free functions and [**&**, **this**] for member functions.

In a *postcondition*, the *return-value-decl*, if present, introduces the name for the prvalue or the glvalue result object of the function. This identifier is valid within the *correctness-specifier-body*.

All closures behave as-if their associated lambda body was declared **mutable**. This makes no difference to [**&**], but it does make a difference for by-value capture extensions.

2.2.1 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 }; // error, x and y are not const

int min(int const x, int const y)
  post (r) { r <= x && r <= y }; // ok

int min(int x, int y)
  post [x, y] (r) { r <= x && r <= y }; // OK (extension, explicit copy)
```

The closure definition works with this - the function arguments are captured by reference, which happens to be reference-to-const, given that they are const, which gives the exact semantics of [P2388R2].

3 Semantics

We specify the future in a somewhat more general manner than strictly required for the MVP, to indicate the inner workings of the future extensions.

3.1 Evaluation order

This section describes the order of evaluation *if contract checking is enabled*. If it's disabled, there is no evaluation, but the contract check bodies are still ODR-used.

3.1.1 Assertions

Any *assertion* is executed as if it was an immediately-invoked lambda expression.

3.1.2 pre- and post-conditions

We need to make preceding *precondition* s 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; their *bodies* are, obviously, evaluated after the function exits.

If a **pre**-condition *B* follows a **pre**-condition *A* in a function's declaration, then no part of *B* shall be executed before *A* has been proven;

If a **post**-condition *P* follows a **pre**-condition *A* in a function's declaration, then not even *P*'s *correctness-specifier-introducer* shall be executed before *A* is proven. This is to protect initialization from out-of-contract behavior.

No postcondition closure is executed before all preconditions are proven.

This means that the following execution orders are all OK:

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

3.2 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.

3.3 Side-effect elision

This MVP defines 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.

3.3.1 An alternative model (from P2388)

P2388 has a looser model, which the authors of this paper do not oppose. In that model, the compiler is allowed to elide any and all side-effects indiscriminately, from any subexpression.

That model effectively disallows lock/unlock pairs inside the body of a check, but is unclear about lock/unlock pairs inside closure initialization, which P2388 does not speak of.

4 Future Extensions (not a proposal)

4.1 Explicit Captures

The main task of this proposal is enabling explicit parameter captures in the future. The main consumer of that are postconditions.

```
auto plus(auto x, auto y) -> decltype(x + y)
  post [x, y] (r) {
    // capture x, y by value, *explicitly*, at point of call
    r == (x + y)
  }
{
```

```

    return x += y;
}

```

Modeling using lambda-captures allows us to explain why post-conditions can't refer to rvalue-reference arguments, and all the other possible implementation-limitations as well.

4.1.1 Capturing views by value

Explicit captures allow capturing views by value for later checking. No “magic” *oldof* implementation can do this - but closures do this easily.

```

template <typename T, size_t n>
void sort_contiguous(std::span<T, n> elems)
    post [on_entry = std::vector<T>(elems), elems] { is_permutation(on_entry, elems) }
    post [elems] { std::is_sorted(elems.begin(), elems.end()) }
{
    std::sort(elems.begin(), elems.end());
}

```

4.1.2 Checking whether a call didn't exceed its time budget

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 };

```

The authors thank Lisa Lippincott for this wonderful idea during a conversation in Aspen a few years ago.

4.1.3 Checking a call didn't leak memory

With a tracking allocator, we can check we didn't leak any memory:

```

int does_not_leak(allocator auto alloc)
    post [usage=alloc.usage(), &alloc] { usage == alloc.usage() }
{
    // do stuff that should not leak
    return 0;
}

```

4.1.4 Grabbing only the interesting part of an input

We can optimize contracts by “remembering” just the required properties of an input:

```

void append(auto& container, auto&& item)
    post [s=container.size()] { container.size() == s+1 }
{
    container.push_back(std::forward<decltype(item)>(item));
}

```

4.1.5 Mutation semantics checking

Sometimes we want to check that two operations are equivalent for the given inputs, because the algorithm is taking advantage of that.

```

auto cat(auto x, auto y)
    pre [cx=x] { (x+=y) == x+y }
{

```

```
return std::move(x += y);
}
```

4.2 Destructuring the return value

There have been rumours of a proposal for destructuring in function arguments. When the language gets that, we can just inherit that directly. A teaser that we *could* just adopt directly:

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

4.3 Attributes appertaining to contract annotations

The syntax allows for attributes on annotations. We could use those for vendor-defined control of execution.

```
int f(int * n)
  pre {n != nullptr}
  pre [[acme::audit]] {*n >= 0};
```

An alternative (although the authors are not completely sure this does not conflict with the *attribute-specifier-seq* that appertains to the function type):

```
int f(int * n)
  [[acme::audit]] pre {*n >= 0};
```

This one courtesy of Andrzej Krzemiński.

4.4 “trust” annotations (new)

Comparison from [P2388R2]/8:

Extension of this proposal	[P2388R2]
<pre>int f(int* p) pre new {*p > 0} ;</pre>	<pre>// after ; at end int f(int* p) [[pre: *p > 0; new]] ;</pre>
<pre>int f(int* p) pre new("call @me") {*p > 0} ;</pre>	<pre>// after ; at end int f(int* p) [[pre: *p > 0; new("call @me")]] ;</pre>

4.5 “cost” annotations

In the same way, we can add convenient cost annotations:

Extension of this proposal	[P2388R2]
<pre>void sort(auto first, auto last) post audit("really expensive") [s=vector(first, last)] { is_permutation({first, last}, s); }</pre>	<pre>void sort(auto first, auto last) [[post r: r > 0: audit("really expensive")]];</pre>
	<pre>void sort(auto first, auto last) [[post audit("really expensive") r: r > 0:]];</pre>

4.6 Multithreaded usage / locking

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 Testing and fuzzing checks

It's important to be able to test the actual precondition and postcondition specification independent of the function they are guarding.

This is due to

- enabling contracts should ideally not *introduce additional bugs*.
- contracts necessarily *narrow* from the *wide* contract specification; therefore the contract specification of the *check* is wider than the contract of the *function*.
- Contracts need to “expect the unexpected”. This makes them excellent candidates for fuzzing.

We could introduce a pair of reflection traits for functions:

```
// unchecked precondition: y != 0
int f(int x, int y)
  pre { x / y >= 0 }
  pre [x] { x % 3 != 0 }
  post (r) { r == x / y }
;

// all preconditions
std::preconditions<&f>(&x*/3, &y*/2); // false, x%3 == 0
// by-index
std::preconditions<&f>[0](3, 2); // true, 3 / 2 == 1, >= 0
```



```
std::preconditions<&f>[1](3, 2); // false, 3 / 2 == 1 , >= 0

// all postconditions
std::postconditions<&f>(<math>(*x)/3</math>, <math>(*y)/2</math>(<math>(*r)/1</math>); // true, 1 == 3/2
std::postconditions<&f>[0](3, 2)(1); // true, 1 == 3/2
std::postconditions<&f>[0](4, 2)(1); // false, 1 != 4/2

// fuzzer:
std::postconditions<&f>[0](3, 0)(1); // segfault, bugreport
```

Postconditions *have* to be fuzzable independently of their functions. If the algorithm is wrong (the very condition the postcondition is there to detect), the postcondition should absolutely not segfault, but instead report **false**.

Traits such as above absolutely must be independently executable.

We could also introduce names:

```
int g(int x)
  pre("nonnegativity") { x >= 0 }
;

std::preconditions<&g>["nonnegativity"](1); // true
```

4.8 Summary

- 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).
- It’s consistent with the rest of the language, instead of inventing a yet-another minilanguage.

5 Comparison tables with attribute-based syntax

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

5.1 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):

```
// Extension of 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 4: Another oldof example:

Extension of this proposal	P2388R2
<pre>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; } }</pre>	<pre>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; }</pre>

5.2 Introducing the return variable

Extension of this proposal	P2388R2
<pre>int f(int* i, array<int, 8>& arr) post [&i] (r) { r >= i };</pre>	<pre>int f(int& i, array<int, 8>& arr) [[post r: r >= i]]; </pre>
	<pre>// alternative int f(int& i, array<int, 8>& arr) [[post(r): r >= 0]] </pre>

5.3 Preconditions and assertions that need copies

Table 6: [P2388R2] has no answer for preconditions that need to mutate a copy:

Extension of this proposal	[P2388R2] Does not work
<pre>int f(forward_iterator auto first, forward_iterator auto last) pre { first != last } pre [first] { std::advance(first, 1), first != last };</pre>	<pre>int f(forward_iterator auto first, forward_iterator auto last) [[pre: first != last]] // ok [[pre: std::advance(first, 1), // nope first != last]]];</pre>

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

Table 7: Functions could conceivably have destructure-only APIs:

Extension of this proposal	[P2388R2]
<pre>auto returns_triple() post (r) { match(r) { [x, y, z] => x > y && y > z; - => false; }};</pre>	<pre>auto returns_triple() [[post r: match(r) { [x, y, z] => x > y && y > z; - => false; }]];</pre>

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.

5.5 Summary

- The closure-based syntax makes it obvious when values are captured, and even hints 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.

6 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.

6.1 Capture design space

There is design space in what kind of captures we allow of parameters of different kinds.

The case space spans between condition (**pre**, **post**) by parameter kind (value, lvalue, rvalue) by qualifier (non-const, **const**) by kind-of-capture (reference, value), which gives us $2 * 3 * 2 * 2 = 24$ cases.

The example is

```
auto f(param x)
  pre cap { x }
  post cap { x }
;
```

#	param	cap	cond	MVP	Ext
1	T	[&]	pre	y	y
2	T	[&]	post	(!)	(!)
3	T	[=]	pre		y
4	T	[=]	post		y
5	T const	[&]	pre	y	y
6	T const	[&]	post	y	y
7	T const	[=]	pre		y
8	T const	[=]	post		y
9	T &	[&]	pre	y	y
10	T &	[&]	post	y	y
11	T &	[=]	pre		y
12	T &	[=]	post		y
13	T const &	[&]	pre	y	y
14	T const &	[&]	post	y	y
15	T const &	[=]	pre		y
16	T const &	[=]	post		y
17	T &&	[&]	pre	y	y
18	T &&	[&]	post	y	y
19	T &&	[=]	pre		y
20	T &&	[=]	post		y
21	T const &&	[&]	pre	y	y
22	T const &&	[&]	post	y	y
23	T const &&	[=]	pre		y
24	T const &&	[=]	post		y

- **MVP**: this paper.
- **Ext**: this paper plus minimal “allow captures” extension.
- **(!)**: Capturing by value in postconditions makes no sense to the caller, therefore it’s not a well-behaved contract and we should diagnose (2).
- All the [=]-captures are missing for the MVP due to the lack of explicit captures, which means there is no way to spell that.

The reader is encouraged to specify their own “acceptability mask” if they want to limit this further, and propose it in discussion. However, the authors don’t really see how any further restriction from Ext couldn’t be circumvented by the user, and (2) is protection from Murphy, not Machiavelli.

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 Considered and rejected syntaxes

8.1 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 syntax). Notably, this was not the main proposed syntax, which was

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

In function declarations, this syntax ends up looking like

```
int f(auto x) pre => x > 0 pre => 42 % x == 0 post [x](r) => x * r == 42 {
    return 42 / x ;
}
```

It's *parsable*, but it doesn't scan well to the human eye. Contrast with the proposed

```
int f(auto x) pre { x > 0 } pre { 42 % x == 0 } post [x](r) { x * r == 42 } {
    return 42 / x ;
}
```

It has the same number of characters, but the visual terminator that `}` provides makes everything a lot more readable.

Reasons to reject:

- We do not want to wait for EWG to express a stance on abbreviated lambdas
- We want the closure to be optional, which is a departure
- The proposal does not consider making the closure `mutable` by default, which we want. The author said this was not considered at all and may be revisited in a future revision, but that does not exist at present.

8.2 Semicolon separators in the body (a-la requires blocks)

The authors also considered separating conditions with semicolons, similarly to `requires` blocks.

```
int f(auto x)
  requires {
    {x % 2} -> integral;
    {as_signed(x)} -> convertible_to<int>;
  }
  pre {
    x % 2 == 0;
    as_signed(x) == x;
  }
;
```

While it makes sense with `requires` blocks, the behavior of `;` in `pre`- and `post`-conditions is identical to the `&&`-operator, because we want short-circuiting. In addition, the lack of a `return` keyword in the body makes the individual conditions seem disembodied.

Requiring chaining using `&&` solves this comprehension issue – it behaves the way it looks, and the lack of a semicolon in the body hints at the fact that the thing between `{}` is an expression and not a statement.

In other words, we say what we mean, and we disallow the syntax that doesn't mean what it looks like.

9 Proposed Wording

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

10 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 Krzemiński*, who has been a steadfast integrator of opinion in P2338 - the MVP paper sequence. I've helped a bit, but he's been extraordinary, and also dug up more prior art and contributed examples. *I thrice presented him co-authorship, which he did thrice refuse.*
- *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.
- *Bengt Gustaffson*, for an amazingly long and thorough review of the R0.

11 References

- [N1962] L. Crowl, T. Ottosen. 2006-02-25. Proposal to add Contract Programming to C++ (revision 4).
<https://wg21.link/n1962>

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

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