An Attribute-Like Syntax for Contracts

Document #:	P2935R1
Date:	2023-09-07
Project:	Programming Language C++
Audience:	SG21 (Contracts)

Reply-to: Joshua Berne <jberne4@bloomberg.net>

Abstract

SG21 is poised to decide upon a final syntax for the Contracts proposal it will produce for inclusion in the C++ Standard and has gathered the requirements such a syntax would ideally satisfy. This paper presents a syntax, based on the C++20 Contracts syntax, for the Contracts MVP. This syntax is one that is familiar, that is easily extensible, with which there is implementation experience, and which meets many of the requirements described in [P2885R1].

Contents

Revision History

Revision 1 (Discussed during 2023-09-07 SG21 telecon)

- Minor cleanups and clarifications

Revision 0

- Original version of the paper for discussion during an SG21 telecon

1 Introduction

An attribute-like syntax for contract-checking annotations (CCAs) was first proposed in 2015 via [N4415]. This syntax evolved, was eventually adopted into the C++ Working Draft in 2018 via [P0542R5], and switched to using the identifiers `pre` and `post` for preconditions and postconditions (instead of the original `expects` and `ensures`) via [P1344R0]. This approach has a long history of being analyzed, implemented, and considered, all in an attempt to determine how it can evolve to meet future needs. Papers such as [P2487R0] have explored some of the impacts of using this syntax, and [P2487R1] discusses in-depth the impact of choosing an attribute-like syntax.

SG21 has made significant progress, following the path proposed in [P2695R1], to produce a complete Contracts proposal, which will be consolidated into the draft currently in development, [P2900R0]. While several open questions about the behavior of Contracts remain to be answered, most are edge cases that are not central to the feature and will, we hope, be addressed thoroughly in the coming months. The primary remaining decision, however, is the syntax of the feature SG21 will propose to the WG21 community.

This paper proposes the use of the C++20 Contracts style of syntax for CCAs. Each annotation will take the following basic form:

contract-checking-annotation
`[ [` *contract-kind metadata-sequence* `:` *evaluable-code* `] ]`

For the contract *kinds* that are in the initial SG21 Contracts proposal, the *evaluable code* is always a normal C++ expression that is contextually converted to `bool`, i.e., a *conditional-expression*. The only use of the *metadata-sequence* in the initial proposal is for the optionally named return value in a postcondition, yet a vast number of future proposals can make effective use of this syntactic location.

This syntax can be used naturally for preconditions, postconditions, and assertions:

```
int f(const int x, int y)
  [[ pre : y > 0 ]]
  [[ post : fstate() == x ]]  // Parameters referenced in post must be const.
  [[ post r   : r > x ]]      // Postcondition may optionally name return value.
  [[ post (r) : r > y ]]      // Return value may have parenthesis.
{
  [[ assert : x > 0 ]];       // Assertions form a complete statement.
  return x;
}
```

For the basic Contracts facility, a few general points must be made to have a complete proposal.

- Preconditions and postconditions are placed in the same location where attributes that would appertain to the function's type (as opposed to the function) would be located, i.e., after the *cv-qualifiers*, *ref-qualifiers*, and *noexcept-specifier* on the function, but before any trailing return type, virtual specifiers such as `final` and `override`, or `requires` clause:

```
struct S1 {
  auto f() const & noexcept [[ pre : true ]] -> int;
  virtual void g() [[ pre : true ]] final = 0;
  template <typename T>
  void h() [[ pre : true ]] requires true;
];
```

  For a lambda expression, a CCA that appertains to the lambda expression or to its function call operator is considered to appertain to the type of the function call operator. Therefore CCAs may be added to lambda expressions with or without function parameter lists, and with trailing return types:

```
auto w = []         [[pre: true]]        { return 6; };
auto x = [] (int x) [[pre: true]]        { return 7; };
auto y = [] (int x) [[pre: true]] -> int { return 8; };
```

  Note that this location is where an attribute would appertain to a function's *type* and not to the function itself. The advantage of this locations is that the same syntax will be applicable if we ever choose to allow CCAs to be placed on function types:

```
using positiveIntFunction = int() [[ post r : r > 0 ]];
std::function<positiveIntFunction> f;
```

  The downside is that the CCAs on a function are not located in what would seem to be the most natural place:

  – For functions with a trailing return type, the return type is not yet known when reading or parsing a postcondition.

  – `requires` clauses, which provide a form of compile-time contract on template parameters, are syntactically *after* the CCAs, which are in terms of runtime values.

  – Other properties, such as virtual specifiers, can be very far removed from the major parts of the function decalration by verbose CCAs.

  We discuss possible alternatives in Section 4.2

- In postconditions, the return value's name must be introduced prior to the colon (`:`):

```
int f()
  [[ post r : r > 0 ]];
```

  To allow for future evolution that *might* be ambiguous with the name chosen for a return value, the return value can always have parenthesis added around it with no other change in meaning:

```
int f()
  [[ post (r) : r > 0 ]];  // same as above
```

- Where they occur, preconditions and postconditions can be of any number and in any order.[1] In various cases and for style or readability, alternating arbitrarily between preconditions and postconditions is beneficial, such as when there is a natural grouping based on the parameters:

```
std::pair<double,double> clamp2d(double x, double y,
                                 const double minx, const double maxx,
                                 const double miny, const double maxy)
  // Check the x-dimension range.
  [[ pre : minx <= maxx ]]
  [[ post r : r.first >= minx && r.second <= maxx ]]

  // Check the y-dimension range.
  [[ pre : miny <= maxy ]]
  [[ post r : r.second >= miny && r.second <= maxy ]];
```

- When new features allow additional identifiers in the space between the *kind* and the colon for (possibly user-defined) additions, the last identifier before the colon in a postcondition names the return value only when that identifier is parenthesized or would not qualify for another purpose:

```
int f( )
  [[ post audit   : true ]]   // OK, this is an audit label.
  [[ post audit   : audit ]]; // Error, invalid expression.
  [[ post (audit) : audit ]]; // OK, return value name is audit.
```

  Choosing this disambiguation is important to avoid important labels being applied to postconditions and then silently ignored, even when that label is *not* used in the postcondition's expression.

- Assertions form complete statements and thus appear at block scope only. The syntax for assertions, however, does not preclude relaxing this restriction to use assertions within other expressions:

```
struct S2 {
  int d_x;
  S2(int x)
  : d-x( [[ assert : x > 0 ]], x ) // error, but could be made valid later
  {}
};
```

  One could allow the assertion to appertain to a full statement or expression, but then the order of checking for that assertion would be unnatural compared to other assertions. Instead, an assertion would be treated as appertaining to an otherwise empty expression with a type of `void`, and would be usable anywhere such an expression is allowed.

[1] C++20 contracts allowed for such repetition and intermingling, as have other previous proposals such as [P2388R4] and [P2461R1].

2 Goals and Principles

The syntax choices made here and the choices for how this syntax can evolve to handle other use cases are managed by certain guiding principles.

1. **Make simple things simple.** — The simplest preconditions, postconditions, and assertions have only 8–11 characters of overhead on top of the predicate being checked.

   ```
   double sqrt(double x) [[ pre : x >= 0 ]];
   ```

2. **Meet the spirit of language attributes.** — Users have been taught that attributes are ignorable; attributes provide additional hints or restrictions on behavior, but a program with all instances of an attribute removed would have conforming behavior for the program with that attribute included. Well-written contracts are generally very similar: They do not form an integral part of the essential behavior of a system, but instead simply provide a mechanism by which the correctness of that essential behavior can be verified. While certain code that focuses on the actual contract-checking facility — such as tests of a function's precondition checks — does depend on the CCA's effect being well-defined and observed, that case is a rare and clear exception to the general rule that CCAs are purely defensive in nature and not an essential part of a program's behavior.[2]

3. **Have distinctive syntax with meaning.** — The code in a CCA will be evaluated in a different way from other code, and having the syntax clearly indicate that the CCA is not merely a function call is important. A distinct syntax also provides a framework for introducing other language features that relate to contracts.

4. **Basic use of even advanced features can remain simple.** — By supporting labels that are single identifiers with no need to express and understand the operators that combine them, the syntax simplifies taking advantage of even more advanced features. For example, we might choose to add within the CCA metadata sequence a label named `audit` — or even one using a name that is a keyword like `new` — and then mix and match use of these labels on CCAs:

   ```
   template <typename T>
   T* binary_search(const std::vector<T> &data, const T& value)
     [[ pre audit : is_sorted(data) ]]
     [[ post r : (r == nullptr) || (*r == value) ]]
     [[ post audit new r :
        (r == nullptr) ==  std::contains(data.begin(), data.end(), value) ]];
   ```

   Support for single-identifier labels that are provided by the Standard without the need for excess syntax also maximizes the ability for C to provide support for the same labels with the same meaning, even if C should choose not to provide support for more advanced customization of CCA behavior through additional user-defined labels.

5. **Maintain extensibility.** — There is ongoing work, such as [P2755R0], to design, implement, and propose features to build on the Contracts MVP to to satisfy many of the use cases gathered in [P1995R1]. Neither that list, nor the work being done to meet some of those needs, are exhaustive of all potential future needs. As a wider range of developers use Contracts in

[2] See [P1743R0] and [P2053R1].

an ever-expanding variety of situations, even more use cases are certainly likely to become apparent.

Providing a well-identified boundary between a CCA and the code that surrounds it (i.e., the opening `[[` and the closing `]]` on an attribute-like CCA) retains ample room to introduce additional features consistently across all CCAs and includes potential places a CCA might appear that would otherwise lack the ability to parse such a construct.

This flexibility to consider only the syntax that has already been adopted for CCAs within the enclosing tokens will help to guarantee that future proposals can be considered entirely on their merits and usefulness rather than rejected due to no acceptable syntactic approach to producing them.

6. **Metadata can be first.** — A CCA can be seen, in part, as a small function. The predicate is the function's body, while all of the other data contained within the CCA is its declaration. In general, the declaration and meta-information pertaining to an entity, all of which would be implemented by features that build on the Contracts MVP, come first in C++ and might guide the understanding of the body.

   Putting meta-information that is outside the expression first also improves certain fundamental characteristics.

   - Regularity: Metadata is often going to be fairly regular across groups of contracts. Coding styles might, for instance, dictate that noncostly checks should come first, followed by `audit`-level checks, followed by uncheckable CCAs that have no runtime-enforceable semantics. Being able to put the meta-information for all such checks in consistent locations relative to the tokens that introduce a CCA means the correctness can be more easily verified, and its regularity can improve readability.

     For example, a large codebase with extensive precondition checking that made regular use of the labels `audit`, `new`, and `unchecked` labels might mandate that such labels always be placed in the same columns, noting that `audit` and `unchecked` should be mutually exclusive:

     ```
     int *binarySearch(int *begin, int *end, int val)
        [[ pre  unchecked       : is_reachable(begin,end) ]]
        [[ pre  audit           : is_sorted(begin,end)    ]]
        [[ pre  audit     new   : is_monotonic(begin,end) ]]
        [[ post           new r : r == end || *r == val   ]]
        [[ post unchecked new r :    is_reachable(begin,r)
                                  && is_reachable(r,end)  ]];
     ```

     Even large expressions such as the newly added postcondition place the labels in visually easy to locate positions.

   - The ability to use `grep`: Searching for CCAs with certain labels becomes much simpler when the CCA kind and the CCA's labels are consistently adjacent, are in a particular order, or are at least on the same source lines. Tools such as `grep` make this form of search much easier, but even visual inspection is aided by avoiding the need to connect the start of a CCA with the end of its predicate to discover which CCAs have which labels.

- Bounded size: Contract predicate examples in presentations or papers are often very simple, along the lines of `p != nullptr` and `x > 0`. In practice, predicates are likely to expand to be potentially quite large, including complex nested logical operations and the invocation of verbosely named functions. Consider, for example, potential postconditions on a complex-valued `sqrt` that placed certain extra guarantees on inputs with no imaginary component, along with a general guarantee to prefer the solution in the positive-real half of the complex plane:

```
template <typename T>
complex<T> sqrt(const complex<T>& val)
  [[ post r : close_enough(r * r, val) ]]
  [[ post r : (val.imag() == 0)
              ? (val.real() < 0) ? (r.imag() > 0) && (r.real() == 0)
                                 : (r.imag() == 0) && (r.real() >= 0)
            : val.real() > 0 ]];
```

  As predicates grow longer, any information placed after the predicate in a CCA becomes significantly more difficult for a human to associate with that predicate.

  When all meta-information is placed before the predicate, only the volume of the meta-information itself increases the distance between the kind and that meta-information. If repeated use of verbose, custom labels becomes burdensome, the ability to provide user-defined labels should allow the effects of multiple labels to be easily combined into a single label, thus reducing large amounts of boilerplate code into a single token.

- Capturing intent and setting expectations: Introducing a CCA with the labels that are assigned to it, many of which will describe to users the character of the CCA's predicate, assists the user in understanding the importance and nuance of a CCA. A `new` label might warrant extra inspection when debugging, since violation of a newly introduced check might be a reason that a bug that had previously remained hidden is now being escalated. An uncheckable CCA is a good forewarning of potential misuses of a function, and such misuses will be hard to diagnose; such a CCA thus indicates that extra thought is needed when stumbling upon a function that appears to be behaving poorly in a production system.

- Impact on parsing: Any meta-information that might impact the meaning of the expression within a CCA likely needs to be lexically prior to the expression; otherwise, compilers must parse all the way to the potential location of that meta-information before being able to complete translation of the expression. This form of look-ahead parsing is onerous and error prone and often introduces cases — such as those needed in many dependent contexts — in which disambiguators must be manually introduced into expressions, e.g., when using template members of a dependent type.

7. **Metadata can be last.** — Despite all the arguments for putting metadata first, having a clear closing token at the end of a CCA facilitates extending the syntax to put some optional metadata *after* the expression, for example with a semicolon (`;`) or colon (`:`) separating the expression from the metadata that follows it:

```
[[ assert : X : audit ]];
[[ assert : X ; audit ]];
```

The additional colon might be hard for a reader to distinguish from half of a ternary operator but would be unambiguously parseable. Using a semicolon would validly separate an expression today but might conflict with some potential proposals for expression statements.

8. **Allow optional backward compatibility.** — Due to its similarity with the attribute syntax, the CCA's syntax could be extended to allow *optional* parentheses after the `pre` or `post` and around the rest of the contents of a CCA:

```
void f(int x)
  [[ pre(: x > 0) ]]
  [[ post(r : r < 0 ]];
```

   A compiler for an older version of C++ would then see this complete CCA as a regular attribute and would ignore it (often with a warning that we hope is suppressible). Modern C++ would need to introduce a disambiguation rule where anything beginning with `[[pre` or `[[post` would be treated as a CCA rather than as an attribute (and would thus be exempt from the design rules for attributes).

   Enabling such optional parentheses would give users a built-in choice that removes the need for any macros when writing code that is portable between platforms supporting different language standards.

3 Formal Syntax

The grammar definition of a CCA is as follows:

contract-checking-annotation :
    precondition-specifier
    postcondition-specifier
    assertion-specifier

Each possible annotation follows the same general structure:

precondition-specifier :
    `[ [ pre :` *conditional-expression* `] ]`

postcondition-specifier :
    `[ [ post` *postcondition-return-value-specifier*$_{opt}$ `:` *conditional-expression* `] ]`

postcondition-return-value-specifier :
    identifier
    `(` *identifier* `)`

assertion-specifier :
    `[ [ assert :` *conditional-expression* `] ]`

The grammar term for *attribute-specifier* is expanded to include precondition and postcondition specifiers:

attribute-specifier :
    `[ [` *attribute-using-prefix*$_{opt}$ *attribute-list* `] ]`
    alignment-specifier
    precondition-specifier
    postcondition-specifier

As with other elements of an *attribute-specifier*, the syntactic location where the specifier appears controls to what that construct appertains. In the case of preconditions and postconditions, the annotation must be placed where it would appertain to the function type of a function declaration.

Assertions are specified using a new statement grammar term:

assertion-statement :
    assertion-specifier ;

This grammar term is allowed as an alternative for the *statement* grammar term — allowing it to be used within compound statements such as function bodies or nested within other control flow structures.

3.1 Requirements Discussion

Various requirements that SG21 has expressed for a Contracts syntax have been expressed via [P2885R1].

- `[basic.aesthetic]`[3], `[basic.brief]`[4], and `[basic.teach]`[5] — We believe this syntax clearly identifies the Contracts syntax as distinct while being minimally intrusive. Many Contracts-related papers and presentations have been published, dating back to the times when C++20 Contracts were already in the Standard, and many readers made no negative comments about the syntax and understood it intuitively.

  Some have objected to the use of an attribute-like syntax which, to a nonexpert, seems to in all ways be an attribute while not necessarily meeting the guarantees expected of a full attribute as codified by [P2552R3]. Defensive checks, while not following the letter of that requirement, should not be part of the essential behavior of a typical function and thus do meet the spirit of the requirement on attributes. In our (admittedly limited and possibly biased) experience, this attribute-like syntax has not been a source of any confusion outside of WG21.

- `[basic.practice]`[6] — Extensive literature related to contract checking exists which makes use of the terms *precondition*, *postcondition*, and *assertion*. The use of the identifiers with special meaning — `pre`, `post`, and `assert` — builds on that historical precedent. Discussions within WG21 — and in particular within SG21 — have also almost universally used these three terms. The very strong consensus for [P1344R0] made it especially clear that these identifiers have strong support.

- `[basic.cpp]`[7] — That this syntax is largely similar to an existing language construct already used with a variety of flexible meanings is a good sign that this syntax too will fit in with other parts of the language with minimal worries about how much it might *feel* like C++.

- `[compat.break]`[8] — The syntax for assertions was carefully chosen so that it would *not* conflict with uses of `<cassert>`. The general syntax has had full formal wording already merged into

[3] See [P2885R1], Section 4.1, "Aesthetics" `[basic.aesthetic]`.
[4] See [P2885R1], Section 4.2, "Brevity" `[basic.brief]`.
[5] See [P2885R1], Section 4.3, "Teachability" `[basic.teach]`.
[6] See [P2885R1], Section 4.4, "Consistency with existing practice" `[basic.practice]`.
[7] See [P2885R1], Section 4.5, "Consistency with the rest of the C++ language" `[basic.cpp]`.
[8] See [P2885R1], Section 5.1, "No breaking changes" `[compat.break]`.

the C++ standard once, and developers have completely implemented this syntax, and no known incompatibilities or breakages have resulted.

- `[compat.macro]`[9]: The syntax allows for full use of the proposed features of the Contracts MVP with no need for extra preprocessor usage.

  As with all other syntaxes, unless the option to allow a backward-compatible use of this syntax is chosen, macros will be needed to place CCAs in code that intends to support C++ language versions that do and do not have support for Contracts.

  Other features, such as the selection of semantics through labels, might involve the use of macros depending on the level of functionality that they provide.

- `[compat.parse]`[10] — Using three bespoke grammar compositions for preconditions, postconditions, and assertions allows humans and compilers to easily read and identify CCAs with this syntax. The attribute delimiters, `[[` and `]]`, provide a highly unambiguous framework in which the entire CCA specification can freely live and evolve without parsing problems with other language constructs.

- `[compat.impl]`[11] — The attribute-like syntax presented here is the C++20 Contracts syntax which is available in GCC version 13.0, released in April 2023. A version with additional support for user-defined contract labels and other related features is available on `godbolt.org` with the name "x86-64 gcc (contract labels)".

- `[compat.back]`[12] — The attribute-like syntax is the only proposal known that has any possibility for having a backward-compatible subset. Some changes would need to be made to allow this.

  – Introduce the ability to have optional `()`s around the contents of an attribute-like CCA, enabling CCAs to be written that will be ignored by older compilers.

  – Disambiguate anything starting with `[[pre`, `[[post`, or `[[assert` in modern C++ as a CCA and not as an attribute.

  This flexibility is not part of this proposal but could be considered for a future addition.

- `[compat.tools]`[13] — Searching for only the attribute-like syntax CCAs is significantly easier than other context-sensitive syntax approaches would be, thus enabling the possibility of tooling that parses and acts on the presence of CCAs without the need to fully parse the entire C++ grammar.

- `[compat.c]`[14] — The C and C++ languages have the same grammar for attributes; hence, an attribute-like CCA is no more a valid C attribute than it is a valid C++ attribute. This leaves the same grammatical space available to C that is available to C++, enabling C to choose to adopt exactly the same syntax as proposed by this paper.

[9] See [P2885R1], Section 5.2, "No macros" `[compat.macro]`.
[10] See [P2885R1], Section 5.3, "Parsability" `[compat.parse]`.
[11] See [P2885R1], Section 5.4, "Implementation experience" `[compat.impl]`.
[12] See [P2885R1], Section 5.5, "Backwards-compatibility" `[compat.back]`.
[13] See [P2885R1], Section 5.6, "Toolability" `[compat.tools]`.
[14] See [P2885R1], Section 5.7, "C compatibility" `[compat.c]`.

Some features that build on the MVP proposal will have varying likelihood of adoption in C. Many, such as `requires` clauses and structured bindings for the return value, are unlikely needed by C functions and are thus irrelevant to the compatibility discussion. Others might be heavily influenced by a desire for C compatibility.

The primary place where overlap might exist in a need for functionality is in the labels the Standard Library might provide to influence CCA semantics. An expression-based label syntax[15] — i.e., `[[pre<audit | review> : true]]` —- would seem unlikely to be adopted by C since this syntax would heavily leverage the C++ semantics of the expression to define how the contents of the expression influence the behavior of the CCA. A label syntax that simply allows for a sequence of tokens would more readily allow for a C specification that supports a subset of those tokens — i.e., `[[pre audit new : true]]`.

- `[func.pred]`[16] — The attribute-like syntax allows for arbitrary C++ expressions as CCA predicates and easily extends to other kinds of expressions as well as block-style CCA bodies, such as on procedural interfaces.[17]

  The rules for name lookup and what can be used in a CCA expression specified here match SG21's expectations for being arbitrary C++ expressions that are convertible to `bool` and have the specified rules for name resolution — e.g., all function parameters are resolvable by name within CCA predicates.

- `[func.kind]`[18] — The values of the `contract_kind` enumeration adopted with [P2811R7] use the same identifiers as those used to introduce the three kinds of CCAs in this proposal. The identifiers associated with the `contract_kind` enumeration, therefore, do not need to change to remain consistent with this proposal.

- `[func.pos]`[19], and `[func.pos.prepost]`[20] — The grammar proposed does not place CCAs at the end of a function declaration since they then do not appertain unambiguously to a function's type. See Section 4.2 for a discussion of fully meeting this requirement. Motivations mentioned above (primarily implementation experience and the ability to consider CCAs on function types in the future) which conflict with this requirement.

- `[func.pos.assert]`[21] — The grammar for *assertion specifiers* could also easily be included as a potential grammar for arbitrary expressions if that was desired.

- `[func.multi]`[22] — All preconditions and postconditions in this proposal occur in a single sequence, with a well-defined location within function declarations, that may mix with other attributes or specifiers. This sequence allows for a well-defined, lexical ordering to those CCAs.

- `[func.mix]`[23] — The locations where preconditions and postconditions may be placed on a

[15] See Page 20 for more discussion of this possibility.
[16] See [P2885R1], Section 6.1, "Predicate" `[func.pred]`.
[17] See [P0465R0].
[18] See [P2885R1], Section 6.2, "Contract kind" `[func.kind]`.
[19] See [P2885R1], Section 6.3, "Position and name lookup" `[func.pos]`.
[20] See [P2885R1], Section 6.4, "Pre/postconditions after parameters" `[func.pos.prepost]`.
[21] See [P2885R1], Section 6.5, "Assertions anywhere an expression can go" `[func.pos.assert]`.
[22] See [P2885R1], Section 6.6, "Multiple pre/postconditions" `[func.multi]`.
[23] See [P2885R1], Section 6.7, "Mixed order of pre/postconditions" `[func.mix]`.

function, whether at the end or where attributes would appertain to the function type, puts no artifical restrictions on the ordering between preconditions and postconditions. In other words, they may be intermingled, by design, or grouped based on the developer's preferred style or desired order of evaluation. (Note that, should the ability to add a *capture list* to post conditions be adopted, the ordering of postconditions in relation to preconditions will be observable and potentially salient, as the captures will be initialized at the time of precondition evaluation in the lexical order of the CCAs.)

- `[func.retval]`[24], `[func.retval.predef]`[25], and `[func.retval.userdef]`[26] — This syntax provides a way to introduce a name for an identifier. A separate proposal to have an additional implicit name available within the conditional expression of a postcondition on a non-`void` returning function could be considered as an additive proposal. Postconditions that make use of that name could simply not name the return value explicitly.

  Note that the C++20 Contracts proposal explicitly chose not to include the implicit introduction of a bespoke name for the return value, though the idea was certainly known at the time.

4 Alternative Proposals

Two primary alternate approaches, either of which could be pursued should SG21 have consensus to do so, are presented here.

4.1 Non-Attribute Opening and Closing Delimiters

Concerns have been raised with the use of `[[` and `]]` in this attribute-like syntax.

- Although the syntax does not actually meet the grammatical requirements of being a C or C++ attribute, it so strongly resembles that syntax that anyone who isn't a language lawyer will see them as attributes and assume they follow all of the advertised rules of attributes — in particular, those rules expressed by EWG by adopting [P2552R3].
- Though the current minimal proposal always allows an implementation to choose the *ignore* semantic for attributes, many use cases require that the Contracts feature have nonremovable effects:
  - Allowing users to specify a particular semantic for a particular CCA or do anything that limits the choice of semantics to only checked semantics, i.e., some subset of *enforce* and *observe*.
  - Any functionality that evaluates CCAs independently of how they might be configured, such as features that could be added to allow for testing of CCAs
- The attribute syntax adds five total characters plus whatever whitespace a user considers stylistically appropriate, and some developers consider this addition to be overly heavy for a

[24] See [P2885R1], Section 6.8, "Return value" `[func.retval]`.
[25] See [P2885R1], Section 6.9, "Predefined name for return value" `[func.retval.predef]`.
[26] See [P2885R1], Section 6.10, "User-defined name for return value" `[func.retval.userdef]`.

syntax.[27]

- While one could argue that defensive checks are superfluous to the essential logic of a piece of software, knowing what conditions are expected to hold cannot be ignored. The high relevance of such expectations to all uses of a function conflicts in many minds with the spirit of ignorability generally associated with attributes.

The distinguishing factor of the attribute-like syntax proposed here is that the contents of a CCA are enclosed in a clearly identifiable pair of opening and closing tokens. The only property of the `[[` and `]]` delimiters upon which this syntax depends is that they are easily distinguishable from other syntactic constructs and naturally support being nested without any special need for lookahead rules; parsing of a CCA can be known to be complete as soon as the closing delimiter is processed.

Additional syntactic options that use a distinct set of outside delimiters can therefore be considered:

- The doubled square brackets can be replaced by a different set of delimiters inside square brackets without being similar to existing language constructs. Therefore, we would consider an alternative to the attribute-like syntax that begins with `[{` and ends with `}]`:

```
int f(const int x, int y)
  [{ pre : x > 0 }]
  [{ post : x > 0 }]
  [{ post r : x > 0 }]
{
  [{ assert : x > 0 }];
  return x;
}
```

- With the adoption of [P2558R2], we now have three new characters — `@`, `$`, and `` ` `` — available to consider. The `@` symbol is used for function decorators in languages such as Python, and we consider another alternative to the attribute-like syntax that uses an opening delimiter of `@(` and a closing `)`:

```
int f(const int x, int y)
  @( pre : x > 0 )
  @( post : x > 0 )
  @( post r : x > 0 )
{
  @( assert : x > 0 );
  return x;
}
```

Choosing alternative opening and closing delimiters but otherwise retaining the shape of the attribute-like syntax would retain certain fundamentals that other syntax choices often discard.

- Language constructs related to the Contracts facility would be easily identified as having special consideration applied to them due to the presence of `[{` and `}]` or `@(` and `)`.

[27] Some participants in the standardization process, however, consider this number of characters to be comparable to most other language constructs and can type sufficiently fast to be unbothered. These same participants tend to consider a clearly distinguished syntax for the Contracts facility to be a feature, not a bug.

- Within the bespoke syntax, we are free to give identifiers and keywords alternate meanings without needing special treatment.
- Unambiguous opening and closing delimiters allow for a wide range of future evolutionary paths that have not yet been considered.

Additionally, Contracts-related functionality would be free to make additional uses of `[{` and `}]` or `@(` and `)` without the need to find new syntactic constructs or navigate the complex, subtle, and often fraught discussions related to using attribute-like syntax choices.

4.2 Trailing Function Contract-Checking Annotations

The C++ function declaration syntax is crowded and confusing. The attribute-like syntax of C++20 Contracts led to choosing a location in function declarations for preconditions and postconditions that was the same as the location for attributes that would appertain to the function type. Being applicable to function types means we have an evolutionary path to include CCAs as part of a function type in general.

This location choice, however, led to CCAs being prior to trailing return types, virtual specifiers, and `requires` clauses. Many developers seem to find this unintuitive — so much so that sample code in papers, discussions within the committee, and presentations often show CCAs in the wrong place when mixed with these language features.

Instead of being part of the *atttribute-specifier-seq* that appertains to a function type, *contract-specifiers* could be defined within the grammar to occur at bespoke locations at the end of function declarations, or at the end of the declaration-like part of a function definition:

```
template <typename T>
auto f() const&& [[ function_type_attribute ]]  // #1
  -> int [[ function_return_type_attribute ]]
     requires requirement<T>
     override final
         [[ pre : X ]]  // #2
         [[ post : Y ]]
{ return 17; }
```

This positioning has some potential disadvantages:

- When not using a trailing return type, `requires` clause, or virtual specifier location `#1` and `#2` are adjacent and thus indistinguishable.
- There is no confusion about possibly mixing other attributes between CCAs, as CCAs live in a separate place in the grammar by themselves.
- When there is a trailing return type no `requires` clause or virtual specifier there is no visual separation between an attribute which appertains to the return type and a CCA:

```
void f() -> int
  [[ function_return_type_attribute ]]
  [[ pre : X ]]
  [[ post : Y ]];
```

This confusion might be alleviated more with a choice of delimiters other than `[[]]` as suggested above.

- Even without trailing return type, there are other (admittedly obscure) situations where the return type surrounds the function name and its parameters, such as a function that returns a pointer to an array:

```
int (*g(char i) [[ function_type_attribute ]])
  [17] // array length
  [[ return_type_attribute ]];
```

- This syntactic position is specific to the various forms of function declarations and would not naturally extend to applying to function types in the future. Choosing this syntax would preclude future extensions (for better or for worse) to capture CCAs when stating a function type.
- This is not a syntactic location with which we have implementation experience, there may be further surprises that need to be addressed.

Compelling reasons to consider this choice exist regardless:

- The end of the declaration does, in general, seem to be the most intuitive location for CCAs.
- This location is one where CCAs will be readily able to grow to be as specific as desired without unduly separating more bounded parts of the function declaration such as the return type or virtual specifiers.
- This location places runtime considerations after compile-time considerations such as `requires` clauses.

The option also exists to support both locations for CCAs - at the end of the declaration *or* where an attribute would appertain to the function type, but this would require careful study of the needed disambiguation rules and what their impact would be on future use cases.

5 Evolution

Both the attribute-like syntax and the alternative syntax proposals mentioned above provide two major mechanisms via which evolution may occur.

1. Additional contract *kinds* can be added that are clearly also part of the Contracts facility by simply using a different identifier while maintaining the attribute-like structure of the existing CCAs.
2. Between the introducing identifier (`pre`, `post`, `assert`, or others that might be added) and the colon (`:`) lies significant flexibility for introducing additional features that can be consistently used with any of the existing kinds of CCAs. Throughout this section, we will refer to this as the CCA-metadata sequence, which the basic syntax uses only as a location for the optional name for a return value in postconditions.

Among the various evolutionary paths for the syntax that are considered in Section 5.1, each element of the CCA-metadata sequence takes one of the following easily distinguishable forms.

- A single identifier — When this single identifier is the last element of the sequence on a postcondition and when the identifier does not qualify as any of the other available types (i.e., when name lookup to see if it is a valid label fails), this identifier is a name for the return type:

```
[[ assert audit : true ]];
```

- An identifier followed by a template parameter list (i.e., a sequence of types and values surrounded by `<>`s) or a function parameter list (i.e., a sequence of expressions surrounded by `()`s) — Some identifiers might be reserved for particular constructs (such as `requires`), and others might be used to identify user-defined or built-in contract labels:

```
[[ assert checked<std::is_random_access_iterator<ITER>> :
   std::distance(start,end) > 3 ]];
```

- A sequence of (comma-separated) init-captures, i.e. an identifier followed by an initializer, inside `[]`s would represent an init-capture list:

```
void swap(int& lhs, int& rhs)
  [[ post [ orig_lhs = lhs, orig_rhs = rhs ] :
     lhs == orig_rhs && rhs == orig_lhs ]];
```

  This feature would also be compatible with naming the return value:

```
T increment(T&& input)
  [[ post [ orig_input = input ]
     r : r == (orig_input + 1) ]];
```

- A sequence of (comma-separated) identifiers would be used as names for structured bindings initialized with the return value. Naturally this functionality would be mutually exclusive with naming the return value, but not with other features:

```
std::tuple<int,int,int> combine(const int a, const int b, const int c)
  [[ post [ra, rb, rc] : a == ra && b == rb && c == rc ]];
```

  If the visual ambiguity with this declaring structured bindings for return values and an init-capture list becomes concerning, parenthesis around this list are allowed or could even be required to avoid any confusion:

```
std::tuple<int,int,int> combine(const int a, const int b, const int c)
  [[ post ([ra, rb, rc]) : a == ra && b == rb && c == rc ]];
```

Note that none of the discussions in Section 5.1 are complete proposals, and they indicate only how possible future proposals might fit into this proposed syntax. This syntax, overall, also allows for other possible extensions within the bounds of the opening `[[` and closing `]]`, none of which are meant to be precluded by this proposal.

5.1 Requirements Discussion

The SG21 contracts syntax requirements paper, [P2885R1], also included requirements related to various potential future proposals that have been hinted at or considered. While none of these

enhancements are going to be part of the initial SG21 Contracts proposal, it is imperative that the choice of syntax must not close the door completely on features that might be essential to the scalable and robust use of the Contracts facility in the future.

- `[future.prim]`[28] — Contracts metadata should be capable of potentially altering the fundamental meaning of the predicate of a CCA. For this primary reason, Contracts metadata must precede that predicate; otherwise, compilers will need to either perform some form of look-ahead parsing to search for such metadata, or labels that alter the meaning will not be an option.

  In particular, we can envision wanting contract labels that a compiler understands and that would specify aspects of a predicate, such as

  – the predicate must be pure (and reference pure functions only)

  – the predicate must be a conveyor expression and invoke only conveyor functions[29]

  – the predicate must use only the public API of the enclosing class

  There might, however, be mechanisms to apply additional meta-information to CCAs within a scope, completely outside each individual CCA. Constructs enabling the application of labels to all CCAs within a scope or on a type might be provided to reduce the cognitive load of understanding the CCAs themselves

  In addition, we should expect that most projects will select a particular set of labels to apply uniformly with common code formatting to all CCAs across the project, while the predicates that make up each individual CCA might vary from tiny to quite large. Keeping the uniform part of the CCAs together at the start of the CCA makes searching for them (particularly with tools such as `grep`) and identifying them (particular with tools such as human eyeballs) easier.

  Put together, we can see that optional metainformation can be kept small and easily identified, occupying only the space between the kind and the `:`, leaving minimal chance of burying the predicate in syntactic noise in most common usages.

- `[future.struct]`[30] — Naming structured bindings to attach to a return value can be expressed intuitively by providing the list of names in enclosing `[]`s, optionally parenthesized, as part of the CCA-metadata sequence instead of a single name for the return value:

```
std::pair<int,int> minMax(int x, int y)
  [[ post [min,max] : min == std::min(x,y) && max == std::max(x,y) ]];
  [[ post ([min,max]) : min == std::min(x,y) && max == std::max(x,y) ]];
```

  Just as when a name is used for the return value, these names would be implicitly `const&` structured bindings initialized to reference the actual returned object. This naming of the return with a structured binding should be mutually exclusive with assigning an identifier to the return value as a whole and must immediately precede the colon (`:`).

[28] See [P2885R1], Section 7.10, "Primary vs. secondary information" `[future.prim]`.
[29] See [P2680R1].
[30] See [P2885R1], Section 7.3, "Structured binding return value" `[future.struct]`.

Note that this syntax is not incompatible with captures, though the two syntaxes are very similar, due to the assignment operators in an init-capture list which allow for disambiguation:

```
template <typename T>
std::pair<T,T> swapMembers(std::pair<T,T> &&input)
  [[ post [ orig_first = input.first, orig_second = input.second ]
     [first,second] : first == orig_second && second == orig_first ]];
```

Surrounding structured binding names with `()`s also allows for this form of disambiguation, and those parenthesis could be made non-optional.

- `[future.reuse]`[31] — Several methods are available for reusing expressions in different contexts in C++ that are not contract specific.
  - Macros are one possibility and do not require answering questions about doing name lookup on the same expression in different contexts; the meaning is always understandable once one understands how macro expansion occurs.
  - This potential use of macros could be better served should C++ adopt a form of hygienic macro such as is available in many other languages.
  - A new *kind* of CCA could be added, if strongly desired, that may declare a set of preconditions and postconditions that could then be named at a later point for reuse:

    ```
    struct S {
       int d_x;

       [[ function_contracts(xfamily) :
           [[ pre : x > 0 ]]
           [[ post r : r > x ]]
           [[ post : d_x == x ]] ]]

       int x1(int x) [[ function_contracts(xfamily) ]] { d_x = x; return x; }
       int x2(int x) [[ function_contracts(xfamily) ]] { d_x = x; return x*x; }
       int x3(int x) [[ function_contracts(xfamily) ]] { d_x = x; return x*x*x; }
    };
    ```

    This form of reusing the same CCAs (i.e., the predicates of the various CCAs within the defining `function_contracts` attribute) would, however, be quite novel due to needing to store the expressions as, essentially, an unparsed sequence of tokens until each individual use on a function. Without storing the CCAs as such *token soup*, reuse would become quite limited as a CCA would be unable to refer to function parameters or know the return type when attempting to reuse such batches of conditions.

- `[future.params]`[32] and `[future.captures]`[33] — Both storing the initial values of function parameters as a copy and capturing other values upon function invocation can be accomplished by adding support for a list of init-captures inside the CCA-metadata sequence:

[31] See [P2885R1], Section 7.4, "Contract reuse" `[future.reuse]`.
[32] See [P2885R1], Section 7.1, "Non-const non-reference parameters" `[future.params]`.
[33] See [P2885R1], Section 7.2, "Explicit captures" `[future.captures]`.

```
void swap(int& x, int& y)
  [[ post [ old_x = x, old_y = y ] : x == old_y && y == old_x ]];

template <class Rep, class Period>
void sleep(const std::chrono::duration<Rep,Period>& rel_time)
  [[ post [ start_time = std::steady_clock::now() ]
     : std::steady_clock::now() >= start_time + rel_time ]];
```

Note that we do not suggest supporting by-value capture by just naming the value to capture because that would encourage hiding the name of the function parameter with a captured copy. In addition, by not allowing by-value captures in this way, the syntax for destructuring a return value and the syntax for an init-capture list are distinguishable, thereby allowing them to be used together with no ambiguity.

Making such copies with the same name is, of course, still supported:

```
double clamp(double value, double low, double hi)
  [[ pre : low <= hi ]]
  [[ post [low=low,hi=hi] r : r >= low && r <= hi ]];
```

Note that use of a function parameter within the capture list does *not* force that parameter to be `const` since this use occurs during precondition evaluation before the function might be changing the value in its implementation. Similarly, the names declared by the capture list hide function parameter names if they match (although giving these captures distinct names, like `orig_low` and `orig_hi`, might be stylistically preferable).

- `[future.meta.param]`[34], `[future.meta.user]`[35], and `[future.meta.keyword]`[36] — Contract labels such as `default`, `audit`, or `new` can all be included unadorned as part of a CCA-metadata sequence:

  ```
  int* binary_search(int *begin, int *end, int val)
    [[ pre audit : std::is_sorted(begin,end) ]];
  ```

  The available identifiers for labels could be specified in any number of ways, from being built-in to the language or specified using some other form of declaration that makes these names available for the name lookup done when evaluating a CCA-metadata sequence.

  In a postcondition, a trailing identifier in the CCA-metadata sequence could potentially be a label or a name for the return value. When an ambiguity arises because resolution on the label name succeeded (i.e., the identifier is a valid label name), the label interpretation should be chosen.[37]

  The designers of user-defined or Standard-Library-defined labels might also want to parameterize their labels with additional arguments, which can be done with either a *template-argument*

[34] See [P2885R1], Section 7.6, "Parametrised meta-annotations" `[future.meta.param]`.

[35] See [P2885R1], Section 7.7, "User-defined meta-annotations" `[future.meta.user]`.

[36] See [P2885R1], Section 7.8, "Meta-annotations re-using existing keywords" `[future.meta.keyword]`.

[37] This disambiguation rule does mean that new labels provided by the Standard Library would potentially break existing code, but the fix for such users would be contained to the expression using the introduced return value name. Any such conflict is also easily fixed by choosing a different name for the return value or enclosing the return value name in `()`s.

list or a *parameter list*:

```
void f()
  [[ pre mylib::newinversion(5) : true ]]
  [[ pre mylib::checked<true> : true ]];
```

Depending on the mechanism chosen for introducing labels, either (or even both) of these syntaxes may be appropriate.

Due to labels existing within the CCA-metadata sequence, which is a wholly new context, even keywords (again, as mentioned in the `[future.meta.keyword]` requirement) should be available for use as labels:

```
void g()
  [[ pre default new : true ]];
```

Multiple labels, separated by white space, can be used on a CCA. The mechanism to declare user-defined labels, however, might provide a facility to specify that certain labels are mutually exclusive.

Labels with no namespace should be reserved for use by the Standard Library, and user-defined labels should require a namespace. Hence, the labels `audit`, `default`, and `new` in an earlier example would be defined by the Standard Library, and the labels prefixed with `mylib::` would be provided by a library, presumable named `mylib`.

The reservation of non-namespaced labels to the Standard Library also allows other constructs that start with different identifiers to be put in the CCA-metadata sequence without concern for conflicting with user-defined labels. Any unnamespaced element of the CCA-metadata sequence must be either a Standard Library label or another construct (such as a `requires` clause) described by the Standard.

Another direction in which this syntax can be extended is to provide a location for an arbitrary expression, to be evaluated at compile time, that produces a `constexpr` object whose type and value might be used to control properties of a CCA's evaluation. This functionality would be in lieu of allowing a sequence of user-defined labels that have a new name-lookup rule and distinct syntax.

```
void h()
  [[ pre<std::contracts::audit | std::contracts::review> : true ]];
```

Here we can see that this option could be used by the Standard to provide static constants that can be combined with overloaded operators to produce a single object that controls the behavior of a CCA. This approach would preclude the use of keywords for labels, conflicting with the desires expressed in the `[future.meta.keyword]` requirement. Such controls, however, should likely not be used to do anything that impacts the parsing and interpretation of a CCA's predicate, and thus both syntactic locations might be useful. Expressions like this might also benefit from choosing a Standard Library namespace, such as `std::contracts`, that is implicitly searched when parsing the expression so that the Standard can provide simple names for commonly recognized use cases. A form of `using namespace` declaration that applies to CCA control expressions such as this, which the Standard Library `<contracts>` header could then take advantage of, would be an even more general solution with this purpose.

Finally, since the meat of the CCA is its predicate, perhaps labels should be placed in a secondary position that comes *after* the predicate to thus minimize the amount of syntactic noise that must be visually parsed prior to getting to the predicate. For that purpose, labels could easily be added *after* the predicate in a number of ways with varying delimiters:

```
void a1()
  [[ pre : true : audit ]];
void a2()
  [[ pre : true ; audit ]];
void a3()
  [[ pre : true ]]<audit>;
```

Using a second colon (`:`) would possibly conflict with the syntax for statement labels, but those are not viable in an expression such as the predicate. Using a semicolon (`;`) would similarly cause difficulties if some of the terser proposals for expression statements were adopted. Placing metadata outside the closing brackets, `]]`, would require significant consideration related to where the CCA might be allowable in the grammar.

- `[future.meta.noignore]`[38] — None of the alternatives proposed for this syntax place user-defined labels or other meta-annotations within nested attributes or attributes that might be positioned to appertain to a CCA. This fundamental decision leaves open the full freedom to define behavior for such features which would innately conflict with the current consensus of what an attribute is allowed to do (see [P2552R3]).

- `[future.invar]`[39] — Class invariants are, fundamentally, a different *kind* of CCA that would be specified within a class and then applied as preconditions and postconditions to a subset of the functions of that class (and possibly elsewhere).

  The syntax for specifying such invariants is quite natural:

  ```
  class T {
    int d_x, d_y;

    [[ invariant : d_x < d_y ]];
  };
  ```

  Invariants are likely to need a fair bit of bespoke CCA-metadata to control to what functions each individual invariant should be applied. In particular, some invariants should apply to all functions, and others might apply to only public functions. From experience, attempting to apply most invariants to `const` member functions is folly (and often leads to circular invocations as invariants get written in terms of public accessors), so by default invariants might not apply to such functions, but metadata to override that default would seem advisable when invariants that relate to `mutable` members should be checked.

  Syntax to check the invariants of a certain form of a particular object would also benefit friend functions in opting in to verification that they have not inadvertently broken an object. Checks could be evaluated with another *kind* of CCA that, similar to an assertion, can be used within a function body:

[38] See [P2885R1], Section 7.9, "Non-ignorable meta-annotations" `[future.meta.noignore]`.
[39] See [P2885R1], Section 7.11, "Invariants" `[future.invar]`.

```
void f(T& lhs, T& rhs)
{
  // ...
  [[ check_invariants : lhs ]];
  [[ check_invariants : rhs ]];
}
```

This CCA's predicate is not a *conditional expression*, but an arbitrary expression whose return value is inspected for any invariants. In a nongeneric context, making this construct ill formed if the return value has no actual invariants could be considered.

Another option to evaluate checks would be to provide a core-language–provided label that identifies any precondition, postcondition, or assertion CCA as being a check of invariants, in which case the contract expression would be used to denote the object whose invariants should be checked:

```
struct S {
   friend void foo(const S&, const S&);
   // ...
};
void foo(const S& lhs, const S& rhs)
  [[ pre invariants_of : lhs ]]
  [[ pre invariants_of : rhs ]]
  [[ post invariants_of : lhs ]]
  [[ post invariants_of : rhs ]]
{
  // .. Do stuff.
}
```

- `[future.interface]`[40] — Procedural function interfaces, as introduced in [P0465R0], offer a powerful way to express many aspects of how a function expects to interact with control flows that come in and out of it. Unlike the simpler `pre` and `post`, these interfaces provide a robust way to specify complex, related conditions as well as checks related to exceptions.

  To provide this functionality, specifying interfaces could be done with a new *kind* of CCA, `interface`:

```
int f( int low, int hi)
  [[ interface :
     int old_low = low;
     int old_hi = hi;
     [[ assert : old_low < org_hi ]];
     auto rval = implementation;
     [[ assert rval >= old_low ]];
     [[ assert rval <= old_hi ]];
  ]]
```

  Obviously, semantic rules would be attached to this syntax.

[40] See [P2885R1], Section 7.12, "Procedural interfaces" `[future.interface]`.

– When evaluated, the identifier with special meaning, `implementation`, will invoke the associated function with the original function arguments. All control flows must evaluate `implementation` exactly once.

– The `implementation` expression produces a `const` lvalue reference to the return value of the function.

– Any code that potentially follows `implementation` must follow the rules of postconditions; in particular, only reference or `const` function parameters may be named.

– An exception that escapes the evaluation of `implementation` will be rethrown automatically if control flow reaches the end of the interface:

```
void f()
   [[ interface :
      try {
        implementation;
      } catch (...) {
      }
   ]];
```

This rethrowing prevents an interface from hiding exceptions thrown by the function when the interface is checked.

– Any exception that is thrown by the interface — other than the automatic rethrowing of an exception emitted by `implementation` or the use of `throw` when an exception emitted by the implementation is in flight — will be caught and will invoke the contract-violation handler.

– Two situations will result in the return value of `implementation` being returned to the original caller: a `return` statement with no arguments or reaching the end of the interface block. An interface is not intended to alter what is actually returned by a function invocation; therefore a `return` statement with an argument is ill formed.

As with all other CCA *kinds*, an interface would support labels and requires clauses in the same location:

```
template <typename T>
void g(T&& t)
  // audit-level check of strong exception-safety guarantee
  [[ interface audit requires(is_copyable<T>) :
     T old = t;
     try {
       implementation;
     } catch (...) {
       [[ assert : old == t ]];
     }
  ]];
```

- `[future.requires]`[41] — Placing a `requires` clause within the CCA-metadata sequence is relatively straightforward, although a normal `requires` clause is challenging to place when

[41] See [P2885R1], Section 7.13, "requires clauses" `[future.requires]`.

other functions or expressions might follow it. To alleviate that difficulty, we could introduce a new grammar form for a `requires` clause that has parentheses placed around the constraint:

contract-requires-clause :
`requires (` *constraint-logical-or-expression* `)`

These `requires` expressions could then be placed anywhere within the CCA-metadata sequence:

```
template <typename Iter>
void f(Iter lhs, Iter rhs)
  [[ pre requires(random_access_iterator<Iter>) : std::distance(lhs,rhs) < 17 ]];
```

When a `requires` clause is not satisfied, the rest of the CCA will be discarded, including any following capture lists and the CCA predicate.

This form of type-based conditional control over which preconditions and postconditions (or even assertions) are expressed can greatly aid the writing of thorough CCAs in generic code. Note that selective compilation such as this is hard to do within a postcondition expression itself, especially when features outside the expression, such as a postcondition's ability to capture values, is dependent on properties of the type:

```
template<typename T>
class myVector {
  // ...
  void push_back(T&& value)
    [[ post requires(std::is_copy_constructible_v<T>) [old_value = value]
      : back() == old_value ]];
  // ...
}
```

Here we see that we can check that the correct value is placed at the end of our `vector` by the `push_back` function *only* when we can capture a copy of that value prior to beginning to invoke `push_back`. When the element type is not copyable, this form of the check is clearly nonviable. Without this feature, separate functions could be defined with the distinct postconditions and identical implementations:

```
template<typename T>
class myVector {
  // ...
  void push_back(T&& value) requires (std::is_copy_constructible_v<T>)
    [[ post [old_value = value]
      : back() == old_value ]]
    { /* implementation */ }
  void push_back(T&& value) requires (!std::is_copy_constructible_v<T>)
    { /* implementation */ }
  // ...
}
```

Taking duplication of this sort to the extreme, however, leads to an eventual 2^n varieties of a function, all of which likely have exactly the same implementation, a problem we might hope to avoid.

- `[future.abbrev]`[42] — An abbreviated syntax for allowing a single-parameter function that returns `bool` to be used as a shorthand for preconditions on a single function parameter could easily be considered with no support for features beyond being an unadorned precondition check:

```
bool positive(double d);

double sqrt0(double x) [[ pre : positive(x) ]];
double sqrt1(double x : positive);
```

A less-abbreviated form that supports all features could also be considered:

```
double sqrt2(double x [[ pre : positive ]])
```

Here the `pre` could even be optional:

```
double sqrt3(double x [[: positive ]]);
```

The alternative syntaxes proposed that would not be hard to distinguish from an attribute could even be used without the seemingly stray colon (`:`), inferring the *kind* of the CCA based on the location and not the presence of the `pre` token:

```
double sqrt3(double x [{ positive }] );
double sqrt3(double x @( positive) );
```

The primary difference between an embedded and abbreviated precondition check, such as this and one declared at the end of the function declaration, is that, in an abbreviated precondition check, the expression would be implicitly treated as a function of one parameter and would be passed the function parameter. This transformation would be similar to that applied to Concepts in many situations, but specifics would need to be explored to apply this same form of transformation to functions instead.

- `[future.general]`[43] — The CCA-metadata sequence is well structured to add in other metadata or features that might be desired for CCAs in general. All of the *kinds* specified in this proposal have identical capabilities to include such a sequence of metadata.

  The attribute syntax in general for new *kinds* is also relatively easy to include in arbitrary locations in the language that have not already been foreseen; the `[[ ... ]]` that surrounds a CCA is ambiguous only with other attributes, and the *kind* makes disambiguating a CCA from another attribute simpler for compilers and humans. (We hope that the Standard will not introduce an attribute that uses the same token as a CCA kind.)

6 Conclusion

The C++20 Contracts facility was the product of enormous effort over many years, multiple organizations, and a variety of noteworthy individual contributors, culminating in a feature with a syntax that had sound reasoning and concrete benefits.

[42] See [P2885R1], Section 7.14, "Abbreviated syntax on parameter declarations" `[future.abbrev]`.
[43] See [P2885R1], Section 7.15, "General extensibility" `[future.general]`.

As the course laid out in [P2695R0] completes, the attribute-like syntax presented here is imminently suitable.

- CCAs with this syntax are clear, distinct, and recognizable as CCAs on function declarations.
- Ample flexibility exists in the syntax style to adopt new features applicable to all CCAs, such as labels or `requires` clauses.
- The attribute-like syntax is general enough to apply this syntactic style to additional contract *kinds*, such as interfaces and invariants, conveying commonality with the Contracts MVP *kinds* via a shared basic syntactic structure.
- Any other future extensions in this syntax, placed within the easily distinguished delimiters, is insulated from the rest of the grammar in its specification . E.g., such extensions may make novel use of keywords, or have lists of values without the need for commas to separate them.
- Implementation experience with this syntax shows that it is both viable and readable.

Taken together, we hope that this paper has demonstrated both the viability and the distinct benefits of adopting the attribute-like syntax for the Contracts feature that SG21 will propose for adoption.

Acknowledgements

Thanks to John Lakos, Jens Maurer, Timur Doumler, Lori Hughes, Andrzej Krzemieński, Gašper Ažman, and Thomas Köppe for early reading of this paper.

Thanks to Rostislav Khlebnikov for the suggestion of using the `@` symbol now that it has been made available by [P2558R2].

Bibliography

[N4415] Gabriel Dos Reis, J. Daniel Garcia, Francesco Logozzo, Manuel Fahndrich, and Shuvendu Lahri, "Simple Contracts for C++", 2015
http://wg21.link/N4415

[P0465R0] Lisa Lippincott, "Procedural Function Interfaces", 2016
http://wg21.link/P0465R0

[P0542R5] J. Daniel Garcia, "Support for contract based programming in C++", 2018
http://wg21.link/P0542R5

[P1344R0] Nathan Myers, "Pre/Post vs. Enspects/Exsures", 2019
http://wg21.link/P1344R0

[P1743R0] Rostislav Khlebnikov and John Lakos, "Contracts, Undefined Behavior, and Defensive Programming", 2019
http://wg21.link/P1743R0

[P1995R1] Joshua Berne, Andrzej Krzemieński, Ryan McDougall, Timur Doumler, and Herb Sutter, "Contracts — Use Cases", 2020
http://wg21.link/P1995R1

[P2053R1] Rostislav Khlebnikov and John Lakos, "Defensive Checks Versus Input Validation", 2020
http://wg21.link/P2053R1

[P2388R4] Andrzej Krzemieński and Gašper Ažman, "Minimum Contract Support: either No_eval or Eval_and_abort", 2021
http://wg21.link/P2388R4

[P2461R1] Gašper Ažman, Caleb Sunstrum, and Bronek Kozicki, "Closure-based Syntax for Contracts", 2021
http://wg21.link/P2461R1

[P2487R0] Andrzej Krzemieński, "Attribute-like syntax for contract annotations", 2021
http://wg21.link/P2487R0

[P2487R1] Andrzej Krzemieński, "Is attribute-like syntax adequate for contract annotations?", 2023
http://wg21.link/P2487R1

[P2552R3] Timur Doumler, "On the ignorability of standard attributes", 2023
http://wg21.link/P2552R3

[P2558R2] Steve Downey, "Add @, $, and ` to the basic character set", 2023
http://wg21.link/P2558R2

[P2680R1] Gabriel Dos Reis, "Contracts for C++: Prioritizing Safety", 2023
http://wg21.link/P2680R1

[P2695R0] Timur Doumler and John Spicer, "A proposed plan for contracts in C++", 2022
http://wg21.link/P2695R0

[P2695R1] Timur Doumler and John Spicer, "A proposed plan for contracts in C++", 2023
http://wg21.link/P2695R1

[P2755R0] Joshua Berne, Jake Fevold, and John Lakos, "A Bold Plan for a Complete Contracts Facility (forthcoming)", 2023

[P2811R7] Joshua Berne, "Contract-Violation Handlers", 2023
http://wg21.link/P2811R7

[P2885R1] Timur Doumler, Joshua Berne, Gašper Ažman, Andrzej Krzemieński, Ville Voutilainen, and Tom Honermann, "Requirements for a Contracts syntax", 2023
http://wg21.link/P2885R1

[P2900R0] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, "Contracts for C++", 2023