Requirements for a Contracts syntax

Timur Doumler (papers@timur.audio)
Gašper Ažman (gasper.azman@gmail.com)
Joshua Berne (jberne4@bloomberg.net)
Andrzej Krzemieński (akrzemi1@gmail.com)
Ville Voutilainen (ville.voutilainen@gmail.com)
Tom Honermann (tom@honermann.net)

Document #: P2885R1
Date: 2023-08-15
Project: Programming Language C++
Audience: SG21, EWG

1 Motivation

As SG21 (the Contracts study group) moves along towards our plan [P2695R1] to build consensus on a minimal viable product (MVP) for a Contracts feature for C++, the final major piece still missing from the design is the choice of syntax for contract-checking annotations. Three proposals for a Contracts syntax have been formally proposed so far: *attribute-like* [P2388R4], *closure-based* [P2461R1], and *condition-centric* [P2737R0]. More syntax designs have been proposed informally or might still appear in the future. We currently do not have a good framework to compare and evaluate all these syntax designs.

This paper attempts to establish such a framework by collecting requirements for a Contracts syntax. In addition to listing requirements, we also discuss how different syntax designs (both proposed and hypothetical) compare under these requirements to some extent. However, as the main goal of this paper is to establish requirements, not to conduct an in-depth analysis of concrete syntax proposals, the proposal comparisons here are necessarily incomplete and somewhat superficial, and mainly serve to understand the requirements themselves. Deeper analysis of concrete syntax proposals should be conducted in dedicated papers.

While SG21 does not have a process to approve a set of requirements such as the one presented in this paper (we can only approve actual proposals), it is our hope that this paper can serve as a comprehensive reference to help SG21 make an informed decision about which syntax to choose for the Contracts MVP targeting C++26.

A future revision of this paper will also feature the results of an SG21 electronic poll set up to determine the priority of the requirements listed here, as well as a classification into objective and subjective requirements.

2 Methodology

The requirements for a Contracts MVP syntax listed here are drawn from multiple sources: the "use cases" paper [P1995R1] which collected requirements for a C++ Contracts facility more generally; all the functionality that has been adopted into the Contracts MVP since then (see [P2521R4] and references therein) and that needs to be supported by the new syntax; as well as additional requirements that were suggested on the SG21 reflector.

There is further a known need for the MVP to be able to viably evolve further to support a wider range of use cases than we are initially targeting, i.e., the full breadth of use cases captured by [P1995R1] or identified since that effort completed. Any choice of syntax must provide a path for such evolution. The requirements for a Contracts syntax therefore also aim to satisfy the needs of these possible post-MVP extensions. We considered a wide range of sources for these future evolution requirements, including ideas from published papers ([P0465R0], [P1607R1], [P2461R1]), discussions on the SG21 reflector, as well as a post-MVP Contracts design developed internally by Bloomberg, which we expect to be released in the form of an SG21 proposal at a later date.

We labelled all requirements in this paper with stable identifiers, such as e.g. "[basic.teach]", for ease of reference both within this paper and from other papers.

Some requirements in this paper may contradict each other. We do not expect a Contracts syntax proposal to materialise that fully satisfies all requirements (in fact, such a syntax is probably not possible); instead, our goal is to make the different tradeoffs visible, in the hope that this will help SG21 make an informed choice.

Further, not all requirements in this paper have the same priority. We can prioritise all requirements on a spectrum ranging from *must-have* (do not ship a Contracts syntax that does not fulfil this requirement), *important*, *nice-to-have*, *questionable*, all the way to *irrelevant* (we should not spend any time on evaluating whether the Contracts syntax fulfils this requirement).

However, such a prioritisation of the requirements is itself subjective, so we do not attempt to assign priorities in this paper. Instead, to have data more representative of SG21 consensus, we will set up an electronic poll for SG21 members to rank all requirements listed in this paper on a scale from 5 (must-have) to 1 (irrelevant), and publish the poll results in a future revision of this paper. We do provide information in this paper that will help with this ranking, for example where appropriate we point out whether it is possible to work around a particular requirement not being met.

Finally, the requirements can be placed on a spectrum between *objective requirements* (we can clearly and unambiguously evaluate whether a given syntax proposal meets the requirement in question) and *subjective requirements* (we cannot unambiguously evaluate whether a syntax meets the requirement, for example because it might be judged differently depending on personal preference, or because this depends on unknown factors). Classifying a requirement as objective or a subjective one is itself subjective, so we do not attempt such a classification in this paper; instead, we will make this classification a part of the aforementioned electronic poll.

As the next step, we suggest that authors of previously discussed syntax proposals should provide papers offering a deeper analysis of how well their proposed syntax satisfies the requirements in this paper, or why they choose not to satisfy them. One such paper has already been published, [P2935R0] for attribute-like syntax; others may or may not materialise later. Using these papers, and the results of the electronic poll, we will then be able to rank the different syntax proposals by how well they meet the requirements SG21 considers important.

3 Proposed syntaxes

In this section, we briefly summarise the three syntax designs that have been previously discussed for the Contracts MVP, as they will be referenced throughout this paper. However, we strongly recommend the reader to familiarise themselves with the actual proposals (referenced below) and the design rationale described therein, before working with this paper. As we said above, each of these proposals probably needs a new revision to clearly indicate that its authors wish to move forward with it as a candidate for the final Contracts MVP, and to provide an analysis of how it satisfies the requirements in this paper.

Note that proposals to add Contracts to C++, and therefore proposals for a Contracts syntax, have a very long history that we cannot possibly summarise here; published proposals go at least as far back as 2004 ([N1613]; for an overview over early work on C++ Contracts, see [P0147R0] and references therein).

3.1 Attribute-like syntax

This is the syntax from the original C++20 Contracts proposal [P0542R5] (first proposed in [N4415] and in a slightly different form in [N4435]). During the C++20 development cycle, this syntax has gained consensus in EWG and plenary and was included into the C++20 Working Draft, before Contracts were removed from C++20 due to reasons entirely unrelated to syntax. This syntax has been successfully implemented in GCC (see [P1680R0]) and has been re-proposed for the Contracts MVP; see [P2388R4] section 7 for a detailed design rationale. More recently, [P2935R0] offers an updated proposal of attribute-like syntax, taking into account the requirements listed in R0 of the present paper and providing a better picture of possible future evolution of this syntax.

Applied to the Contracts MVP (which lacks some additional syntactic features from [P0542R5] such as the `` `default` ``, `` `audit` ``, and `` `axiom` `` labels), the attribute-like syntax proposes to spell a contract-checking annotation as follows:

```
[[ contract_kind : predicate ]];
```

where *`contract_kind`* is one of `` `pre` `` (for preconditions), `` `post` `` (for postconditions), and `` `assert` `` (for assertions), and *`predicate`* is an arbitrary C++ expression contextually convertible to `bool`.

Note that this differs from *actual* standard attribute syntax, which only allows the following syntax for standard attributes (see [dcl.attr.grammar] in the Standard):

```
[[ attribute ]]
[[ attribute ( argument_clause ) ]]
```

Therefore, due to the presence of a colon in a position other than in an *attribute-using-prefix*, the attribute-like Contracts syntax is not actually valid attribute syntax (see also [compat.back]), and will be rejected by a compiler that does not implement Contracts (for a more in-depth discussion of incompatibilities between attribute-like Contracts syntax and actual standard attribute syntax, see [P2487R1]).

The other noteworthy innovation of the attribute-like syntax apart from the colon is that the return value of a function is referenced in a postcondition through a user-declared identifier that immediately precedes the colon:

```
int f(int i)
  [[pre: i >= 0]]
  [[post r: r >= 0]];  // `r` names the return value of `f`
```

3.2 Closure-based syntax

Closure-based syntax [P2461R1] proposes to spell a contract-checking annotation as follows:

```
contract_kind { predicate }
```

which is strikingly similar to the syntax proposed in [N1962] back in 2006. The return value of a function is referenced in parentheses after the `contract-kind`:

```
int f(int i)
  pre { i >= 0 }
  post (r) { r >= 0 };  // `r` names the return value of `f`
```

The motivation for this proposal is twofold.

First, this syntax entirely avoids the attribute design space (which is also shared with the C language) and the associated problems (see [P2487R1] and discussion of requirement [basic.lang] below).

Second, the claim of the authors is that this syntax is better suited than the attribute-like syntax for various post-MVP extensions. In particular, it allows the user to refer to non-`const` parameters in postconditions (see [future.params]), or even capture any kind of value at the beginning of the function and then refer to that value in the postcondition at the end of the function (see [future.captures]), by writing a capture similar to a lambda capture:

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

or even an arbitrary init-capture:

```
void vector::push_back(const T& v)
  post [old_size = size()] { size() == old_size + 1 }; // init-capture
```

This is a powerful technique. See [P2461R1] for many use cases of closures in contract-checking annotations, detailed code examples, and comparisons with attribute-like syntax.

3.3 Condition-centric syntax

Condition-centric syntax [P2737R0] is the most recently published proposal. In this paper, contract-checking annotations are spelled as follows:

```
contract_kind ( predicate )
```

This approach, like the closure-based syntax, also entirely avoids the attribute design space. Further, placing the predicate into parentheses is more consistent with existing practice in C++ to place statements inside curly braces and expressions inside parentheses (see [basic.lang]). Further, [P2737R0] proposed a series of other syntactic design choices:

- Changing the standard term for a contract "assertion" to the newly coined term "incondition", and changing the identifiers for the three contract kinds from `pre`, `post`, and `assert` to `precond`, `postcond`, and `incond`, respectively;
- making all these three identifiers full keywords;
- For naming the return value of a function in a postcondition, removing the syntactic place for a user-defined identifier, and replacing it with a predefined identifier `result`.

These design choices were not favourably received on the SG21 reflector. In fact, all of these choices can actually be decoupled from the choice of primary syntax, `*contract_kind* ( *predicate* )`, which we believe is promising and worth considering on its own. Therefore, when we talk about "condition-centric syntax" in the remainder of this paper, we mean just this choice of primary syntax – placing the predicate in parentheses – and not the other three syntactic design choices that [P2737R0] proposed alongside it (which we will consider separately). Thus, a contract-checking annotation with condition-centric syntax would look as follows:

```
int f(int i)
  pre (i >= 0);
```

A weak point of [P2737R0] is that it is not concerned with post-MVP extensions. It is therefore an open question how well-suited the condition-centric syntax would be to such extensions. We can derive some preliminary conclusions from section 7 of this paper, but we would like to see a future paper conducting a proper analysis of this syntax design direction.

3.4 Hypothetical syntaxes

In order to explain some of the requirements listed below, in a few cases we "invent" hypothetical syntaxes on the fly to make a point. These are *not* to be interpreted as serious proposals; they serve merely as an aid to understand better how we could evaluate possible future syntax proposals under the requirements in this paper.

For example, if we want to consider a syntax where contract-checking annotations are clearly delimited by special tokens from other C++ code (which is currently the case only in attribute-like syntax), but at the same time avoid the attribute design space, we might "invent" a hypothetical syntax, for example:

```
[{ contract_kind : predicate }];
```

or perhaps:

```
@( contract_kind : predicate );
```

We do not investigate such hypothetical syntaxes beyond the amount necessary to properly define a given requirement; we leave such investigation to future papers.

4 Basic requirements

This section contains basic requirements that are not specific to the Contracts MVP but apply to syntax design in general.

4.1 Aesthetics [basic.aesthetic]

The syntax should be elegant. It should be easily readable and at the same time not too obtrusive.

From informal conversations with developers in the wider C++ community, we have anecdotal evidence that many users consider the attribute-like syntax `[[pre: x > 0]]`, to be too "heavy" or "ugly", and dislike it for that reason. At the same time, some users consider the particular way in which the attribute-like syntax stands out visually to be a benefit, as it creates a clear separation between contract-checking annotations and other C++ code. When the reader sees `[[pre: x > 0]]`, they immediately know this is a precondition. On the other hand, `pre{x > 0}` could also be part of an init-declaration, and `pre(x > 0)` might be a function invocation; identifying these token sequences as a precondition requires context.

Note that anecdotal evidence about user perception of the attribute-like syntax is by no means robust data. Note also that the closure-based and condition-centric syntaxes are less known outside of SG21, and we therefore have no data whatsoever on the perception of these syntaxes in the wider C++ community.

4.2 Brevity [basic.brief]

The syntax should be succinct. It should not use more tokens and characters than what is useful.

The attribute-like syntax uses more tokens and more characters than either closure-based and condition-centric syntax:

```
[[ pre: x > 0 ]]   // attribute-like:    6 tokens (8 chars) + predicate
pre {x > 0}        // closure-based:     3 tokens (5 chars) + predicate
pre (x > 0)        // condition-centric: 3 tokens (5 chars) + predicate
```

One could therefore argue that the attribute-like syntax satisfies this requirement less well than closure-based or condition-centric syntax. On the other hand, one could also argue that a human reader typically does not perceive `[[` as two separate tokens (we do not usually write `[ [`, for instance), and might perceive `pre:` as a single syntactic unit as well, making a comparison based on the technical definition of "token" less useful.

4.3 Teachability [basic.teach]

The syntax should be easy to learn and teach. It should be self-explanatory, intuitive to the user and easy to remember. It should not introduce unnecessary complexity.

Evaluating syntax proposals under this requirement is not straightforward. However note that, for example, a hypothetical syntax `[{ pre: x > 0 }]` as an alternative to attribute-like syntax does not seem to satisfy this requirement very well, since it would introduce an entirely novel token sequence that users are not familiar with and might find unintuitive (are the curly braces on the inside or on the outside?).

Note further that there is overlap with [basic.lang]: a syntax design inconsistent with the existing C++ language is necessarily less accessible to learners.

4.4 Consistency with existing practice [basic.practice]

The syntax should correlate with existing literature and community knowledge about how to make use of contract-checking frameworks.

Evaluating syntax proposals under this requirement is equally not straightforward and is left for future papers. To give an example, we note that the proposal in [P2737R0] to rename the term "assertion" to "incondition", and to use the token `incond` instead of `assert` to mark this kind of contract, goes against this requirement as it replaces an established term of art with a newly coined term.

4.5 Consistency with the rest of the C++ language [basic.cpp]

The syntax should fit naturally into the existing C++ language. It should not create any ambiguity, confusion, or inconsistencies with other language features. It should "feel like" C++.

It seems that closure-based syntax satisfies this requirement less well than condition-centric syntax, because closure-based syntax places the contract predicate inside curly braces, whereas condition-centric syntax places the contract predicate inside parentheses. In C++, we usually place statements inside curly braces and expressions inside parentheses; the contract predicate is an expression.

Notably, the attribute-like syntax might not satisfy the requirement of being consistent with existing rules for standard attributes. [P2487R1] provides a thorough analysis of the differences between attribute-like syntax for Contracts and standard attribute syntax in existing C++. It comes to the conclusion that while contract-checking annotations seem compatible with the notion of semantic ignorability in C++ as per [dcl.attr.grammar] Note 5 (see also [P2552R3]), they are not compatible with the notion of syntactic ignorability for attributes in C (see also [compat.c]); moreover, they completely disregard the existing syntactic rules for appertainment, aggregation, and comma-separation of standard attributes.

Further, even if current MVP Contracts are compatible with semantic ignorability of attributes in C++, some of the proposed or planned post-MVP extensions (see Section 7) might not be. For example, if we introduce a label post-MVP (see [future.meta]) that forces a particular contract-checking annotation to be checked (either with "enforce" or with "observe" semantics), such an annotation is semantically ignorable only in the absence of a contract violation (note that side effects of predicate checks are not guaranteed to occur, see [P2751R0]), but not if a contract violation occurs. Therefore, a guaranteed-to-be-checked contract cannot be semantically ignorable, unless we decide to apply the ignorability rule only to programs with no contract violations. However, note that a program with a contract violation still has perfectly well-specified behaviour; this property is central to a Contracts facility that features violation handling.

How much inconsistency with attributes there actually is is to some degree a matter of interpretation and ongoing debate. The usual counterargument is that attribute-like contract-checking annotations are not *actually* standard attributes (due to the presence of a colon in a position other than in an *attribute-using- prefix*), and therefore the inconsistencies with attributes do not matter. While this is formally correct, it at least presents a significant teachability burden (see [basic.access]): C++ developers are usually not concerned with such language-lawyer level details and perceive anything in `[[ ... ]]` as an attribute.

5 Compatibility requirements

This section contains technical requirements that address compatibility with the existing C++ language and ecosystem, as well as possibly the C language.

5.1 No breaking changes [compat.break]

The syntax should not break any existing C++ code or change the semantics of any existing C++ code.

For example, the syntax should avoid the use of the `` `assert` `` identifier followed by a `` `(` `` token, which is recognized as a macro and expanded if the standard `` `<cassert>` `` header has been included (that is, unless we explicitly decide to change the existing meaning of `` `assert` ``, for example as proposed in [P2884R0]). This is a problem particularly for condition-centric syntax:

```
void f() {
  // code...
  assert(errno == 0);  // does not work as a contract syntax!
}
```

As another example, the syntax should not claim identifiers commonly found in code today, such as `` `pre` `` and `` `post` ``, as full keywords:

```
void f() {
  int pre = 0;  // must work
}
```

To our knowledge, all current proposals for a Contracts MVP syntax avoid such breakage, with the exception of [P2737R0], which claims full keywords for condition-centric syntax, although it mitigates the impact by renaming `` `pre` ``, `` `post` ``, and `` `assert` `` to `` `precond` ``, `` `postcond` ``, and `` `incond` ``, respectively, which are much less likely to be used as identifiers in existing C or C++ code.

5.2 No macros [compat.macro]

The syntax should minimise the use of macros and the preprocessor.

Note that we expect codebases might wrap the whole contract-checking annotation with macros like `` `#define EXPECTS(x) [[pre: x]]` ``, for engineering reasons; but it should be possible to use all the features of Contracts without any use of the preprocessor, both in terms of functionality (like switching semantics in the build system) and in terms of readability.

While all current proposals for a Contracts MVP syntax satisfy this requirement, a historical example of a proposal that does not is [P1429R3]. In this proposal, the contract semantic (ignore, enforce, etc.) could be specified explicitly in the contract-checking annotation using an identifier (`` `ignore` ``, etc.). Providing new ways to switch semantics at compile time required the use of macros that expanded to these identifiers, which might have been one of the reasons the proposal was rejected at the time.

5.3 Parsability [compat.parse]

The syntax should not create any new hurdles when parsing C++ and should not unnecessarily complicate the existing C++ grammar.

We have not thoroughly analysed the current proposals for a Contracts MVP syntax under this requirement; however we noticed in Section 7 that there is some potential for violations of this requirement to arise when trying to satisfy certain future extension requirements with a particular choice of syntax (see for example [future.meta.param] and [future.requires]).

In general, syntax designs where the whole contract-checking annotation is clearly delimited by special tokens from other C++ code (such as attribute-like, which uses `[[ … ]]` as delimiter tokens, or some other hypothetical syntax that uses such delimiter tokens, e.g. `[{ … }]` or `@( … )`) seem to be less prone to violating this requirement than syntax designs where this is not the case (such as closure-based or condition-centric), because without delimiting tokens, contract-checking annotations are competing for space in the C++ grammar with all other possible C++ constructs where they can appear. This is even more true if we use contextual keywords for the contract kind rather than full keywords.

5.4 Implementation experience [compat.impl]

We should have implementation experience with the proposed syntax in a C++ compiler.

At present, we are aware of only one proposed Contracts syntax for which there is actual implementation experience, namely the attribute-like syntax, which has been successfully implemented in GCC (see [P1680R0]). None of the other syntax proposals currently even contain any statements from implementers about implementability.

Implementation experience is important for making sure that requirements like [compat.parse] and [func.pos] are actually satisfied. Otherwise, it is all too easy to miss edge cases in the design phase that can cause problems further down the road. As an example, consider the case of a contract-checking annotation on a function returning a pointer to an array shown in [func.pos]. As another example, consider the various issues with the spaceship operator that were discovered after having standardised it without implementation experience.

5.5 Backwards-compatibility [compat.back]

Contract-checking annotations should be backwards-compatible, i.e. we should be able to add them to existing C++ code, and a legacy compiler that does not yet support Contracts would simply ignore them, rather than diagnosing a syntax error.

Note that it is possible to work around this requirement in the same way we work around any syntactically new feature not supported by legacy compilers: by wrapping the new syntax in a macro and providing a feature-test macro to test for its availability (and this does not constitute a violation of requirement [compat.macro]).

It is a common misunderstanding that plain attribute-like syntax as proposed in the original C++20 Contracts proposal [P0542R5] satisfies this requirement because the Standard normatively specifies that any attribute not recognised by the implementation is ignored (see [dcl.attr.grammar] paragraph 6). But the attribute-like Contracts syntax is not actually valid attribute syntax, due to the presence of a colon in a position other than in an *attribute-using-prefix*. Therefore, any compiler that does not implement the attribute-like Contracts syntax will issue an error that this code is ill-formed.

In fact, no proposed Contracts syntax satisfies this requirement. The only way to satisfy it would be to make contract-checking annotations actual standard attributes that follow the current standard attribute grammar (see [dcl.attr.grammar] in the C++ Standard):

```
[[ contract_kind (argument-clause) ]]
```

A straightforward attempt at such a Contracts syntax would be to make the contract predicate the attribute argument clause:

```
int f(int i)
  [[ pre (i >= 0) ]];
```

This approach does not seem viable for at least the following reasons:

- It would be impossible to have anything but a single *attribute-token* (`pre`, `post`, or `assert`) before the parentheses without breaking the backwards-compatibility requirement. With this syntactic restriction, it is unclear how we could introduce an identifier for the return value ([func.retval]) or add any kind of labels or meta-annotations post-MVP ([future.meta]).
- The syntax would be identical to attribute `assume` [P1774R8], which is not a contract-checking annotation (and is orthogonal to Contracts by design).

[P2935R0] proposes a more elaborate way to satisfy this requirement via an extension allowing optional parentheses after the `pre`, `post`, or `assert` and around the rest of the contents of a contract-checking annotation, without changing or restricting the syntax otherwise:

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

However, this approach, and in fact *any possible approach* to satisfy this requirement, still suffers from the following issues:

- There would still be a backwards-compatibility issue because all entities in a contract predicate are ODR-used, even if the contract semantic is "ignore", whereas entities in an unsupported attribute are not, and ODR-use can trigger observable behaviour changes both at compile time and at runtime via template instantiations and lambda captures,

- Contracts could not have any functionality that is not semantically ignorable as per [dcl.attr.grammar] Note 5. This might be true for the basic MVP functionality, but not for the proposed or planned post-MVP extensions. This is already arguably a problem for the proposed attribute-like syntax (see discussion in [basic.lang]), because contract-checking annotations with this syntax are very similar to attributes and overwhelmingly *perceived* as attributes, but the problem would be exacerbated even further if Contracts were *actual* attributes.

5.6 Toolability [compat.tools]

Contracts are useful as input not only for compilers and human readers, but also for other tools: static analysers, linters, IDEs, etc. The Contracts syntax should therefore be friendly to such tools. Note that many such tools do not have a fully-fledged C++ frontend.

To our knowledge, whether and to what extent the current Contracts syntax proposals satisfy this requirement has not yet been studied. This could be looked at as part of an overall review of the Contracts MVP proposal by SG15 (the Tooling study group).

5.7 C compatibility [compat.c]

The Contracts syntax should also be standardisable for the C programming language.

Note that a Contracts syntax adopted by WG14 might be a subset of the syntax adopted by WG21, or an alternative spelling for the same constructs that is supported by both languages. Note also that we do not have a mechanism for judging what syntax WG14 would accept without a proposal for that committee reaching consensus, and that we do not currently plan to block progress on Contracts in order to wait for WG14. We can however make sure that the Contracts syntax does not break any existing C code, does not change the semantics of any existing C code, and does not conflict with existing constructs in C in any other way as far as we can see.

An analysis of how well the current syntax proposals would satisfy this requirement has not yet been conducted; we defer such analysis to future papers. Such analysis should consider the benefit to C++ of a compatible syntax (or costs for an incompatible syntax) with C that can be used in system headers, standard library implementations, and other C libraries. Consider also that we can work around the C/C++ compatibility issue by wrapping syntax that C does not understand into macros, as we already do with other C++ language features.

6 Functional requirements

This section contains requirements for a Contracts syntax that are needed to support the current functionality in the Contracts MVP.

6.1 Predicate [func.pred]

The syntax should allow for the predicate to be an arbitrary C++ expression contextually convertible to `bool`. Contracts should not place additional syntactic restrictions on this expression.

This requirement is satisfied by all three currently proposed syntaxes.

6.2 Contract kind [func.kind]

The syntax should clearly and easily distinguish between the three kinds of contract-checking annotations: preconditions, postconditions, and assertions. The names for the enumerators in `std::contracts::contract_kind` need to be consistent with any eventually chosen syntax.

If this means that these names need to be different from the ones we already adopted from [P2811R7] into the Contracts MVP – ``pre``, ``post``, and ``assert`` – new names for these enumerators should be proposed along with the syntax.

This requirement can be satisfied in a straightforward manner by all three currently proposed syntaxes by choosing appropriate identifiers.

6.3 Position and name lookup [func.pos]

Preconditions and postconditions should appear in an unambiguous syntactic position as part of a function declaration; on the other hand, assertions should appear syntactically as statements inside a function definition. The syntax for pre/postconditions should allow for name resolution in the predicate to be performed as if it were placed at the start of the function body; for postconditions, name resolution also includes names related to return value (see also [func.retval]). The syntax for assertions should allow for name resolution in the predicate to be performed as if the predicate were an expression at that point in the lexical flow of code.

For obvious cases, this requirement is satisfied by all three currently proposed syntaxes. However, as a more complicated case, consider a contract on a function that returns a pointer to an array:

```
int i = 0;
int (*f(char i) [[ pre: i != 0 ]] )[sizeof i];
```

In code like this, it must be clear where the contract annotation goes (after the function declarator? after the array declarator?), it must be possible to refer to either the global ``i`` or the parameter ``i`` in the contract predicate, and it must be unambiguous which one the predicate refers to. The example above is using the syntax that works with the current GCC implementation; the current syntax proposals still need to show how they will handle cases like this.

6.4 Pre/postconditions after parameters [func.pos.prepost]

More specifically than the general requirement [func.pos], preconditions and postconditions should go somewhere after the parameter clause to avoid complications with reading and parsing predicates that refer to parameters.

This requirement is satisfied by all three currently proposed syntaxes. An example of a hypothetical syntax that would not satisfy this requirement is the so-called "piano syntax" which was brought up as an idea by Andrew Tomazos on the SG21 reflector. In this syntax, whether a contract-checking annotation is a precondition or a postcondition would be determined by whether it appears before the function signature or after it:

```
[? i >= 0 ?]   // precondition
[? j >= 0 ?]   // precondition
int select(int i, int j)
[? @ >= 0 ?];  // postcondition; `@` refers to the return value
```

Such a syntax would probably be implementable (requiring multiple parsing passes), but would not satisfy this requirement because preconditions are using parameter names but are syntactically positioned before the parameter declarations.

Note that this requirement does not specify whether pre/postconditions should go before or after the trailing return type, as the trailing return type cannot introduce a name and is therefore not needed for name lookup (even though the type must be known if the postcondition refers to the return value). It also does not specify whether pre/postconditions should go before or after other parts of the declaration that are orthogonal to preconditions and postconditions and irrelevant for name lookup in the predicate, namely any virtual specifiers such as `final` and `override`, and any `requires` clause.

Attribute-like syntax usually places pre/postconditions *before* any trailing return type, virtual specifiers, and `requires` clause, as it re-uses the syntactic position of standard attributes appertaining to a function type. This is the case in the C++20 Contracts proposal [P0542R5], in the latest paper [P2935R0], as well as in the GCC implementation. Other papers such as [P2521R4] assume the position of pre/postconditions to be at the very end of a declaration, immediately before the semicolon (or the opening brace of the function body if the declaration is a definition). [P2935R0] offers some analysis of the tradeoffs between these two choices.

6.5 Assertions anywhere an expression can go [func.pos.assert]

More specifically than the general requirement [func.pos], assertions should be syntactically able to appear anywhere that any other evaluable expression might be able to appear.

The basic use case covered by [func.pos] is that assertions are simply statements within the function definition. But the choice of syntax should not prevent the ability to consider putting assertions anywhere an expression with a void type might appear, for example as the left hand side of a comma operator used within the member initialiser list of a constructor:

```
class X {
  int* _p;
public:
  X(int* p) : _p((ASSERTION, p)) {}
};
```

Such an assertion might for example check that `p != nullptr`. Note that the `assert` macro from header <cassert> can be used in any such syntactic position; if Contracts are supposed to be a better replacement for <cassert>, Contracts assertions ought to be usable in all places in which the `assert` macro can be used.

The basic requirement of having assertions as statements inside the function definition ([func.pos]) is satisfied by all three currently proposed syntaxes. However, the extended requirement that assertions should be able to appear anywhere that any other evaluable expression might be able to appear is not considered by any of these proposals – it might work, or it might not. Note that the extended requirement places additional restrictions on the choice of syntax, because a contract-checking annotation that can appear in any position where an expression might appear must not break existing code ([compat.break]) or introduce parsing ambiguities ([compat.parse]).

6.6 Multiple pre/postconditions [func.multi]

The syntax should allow for multiple separate pre/postconditions on the same function declaration.

Having to combine multiple predicates into a single pre/postcondition would have the disadvantage that they cannot be addressed separately for violation handling, or given different contract semantics.

This requirement is satisfied by all three currently proposed syntaxes.

6.7 Mixed order of pre/postconditions [func.mix]

The syntax should allow for pre/postconditions to be intermingled in any order, as opposed to forcing the user to write preconditions first and postconditions after.

On the one hand, in many cases the user would want to write preconditions first and postconditions after, because postconditions are evaluated after. On the other hand, there are cases where one might choose to group them differently for stylistic reasons. For example, one might want to place all the audit-level checks ([future.meta]) at the end, or pre/postconditions related to specific parameters grouped next to one another, or perhaps have more control over the order in which captures are initialised ([future.captures]).

This requirement is satisfied by all three currently proposed syntaxes. However, as an example of a hypothetical syntax that would not satisfy this requirement, consider the "piano syntax" mentioned in [func.pos.prepost].

6.8 Return value [func.retval]

For postconditions, the syntax should provide *some* way to refer to the return value of the function within the postcondition's predicate.

This requirement is satisfied by all current proposals for a Contracts MVP syntax, albeit in different ways. The attribute-like syntax and closure-based syntax both propose a syntactic place for the introduction of a user-defined variable name for the return value:

```
int f(int& i, array& arr)
  [[post r: r >= i]];  // attribute-like: before the colon

int f(int& i, array& arr)
  post [&i] (r) { r >= i };  // closure-based: inside parentheses
```

On the other hand, [P2737R0] proposes to have a predefined identifier with a special meaning to refer to the return value, such as `result`:

```
int f(int& i, array& arr)
  post ( result >= i );
```

Note that [P2737R0] ties the design choice of providing a predefined name to the choice of primary syntax (in this case, condition-centric), but actually these aspects of the syntax are orthogonal. There is however currently no proposal on how to fit the first approach (user-defined name for the return value) into condition-centric syntax.

Note further that in whichever way the syntax lets the user refer to the return value, the lack of such an ability is something for which there is no general workaround. If this requirement is not satisfied, the user would have to manually define a variable to hold the return value, which is error-prone and could potentially disable RVO and have other observable effects on the program.

6.9 Predefined name for return value [func.retval.predef]

More specifically than the general requirement [func.retval], the syntax should provide a predefined identifier for the return value of a function, which can be used within a function's postcondition predicate.

As described in [func.retval] above, [P2737R0] provides the identifier `result` for this purpose, while the other syntax proposals provide no such predefined identifier and *require* the user to define their own name.

On the one hand, for a very simple postcondition like "the return value is nonnegative", predefining an identifier such as `result` for the return value minimises the amount of tokens that the user needs to write for the predicate, as they do not have to define their own name:

```
float abs(float x)
  [[post : result >= 0]];
```

On the other hand, with this approach, name clashes have to be dealt with (see [P2737R0]). Also, an identifier such as `result` might be too long for certain expressions, especially if the return value appears multiple times in the same predicate. Both issues could potentially be remedied by reserving a special character such as `@` to represent the return value that cannot clash with any user-defined identifiers.

6.10 User-defined name for return value [func.retval.userdef]

More specifically than the general requirement [func.retval], the syntax should allow the user to specify their own name for the return value of a function within a function's postcondition predicate.

A user-defined variable name is handy, for example, in a more complex mathematical expression where the return value appears multiple times (shown here with attribute-like syntax):

```
float cube_root(float x)
  [[post r : approx_eq(x, r * r * r)]];
```

Allowing the user to pick a single-letter name keeps the expression short and readable. In addition, engineers and mathematicians often expect the usage of certain letters in certain equations.

The attribute-like syntax and closure-based syntax both satisfy this requirement, while condition-centric does not; to our knowledge, no suggestions have been made so far how this could be achieved with condition-centric syntax.

7 Future evolution requirements

This section contains requirements for a Contracts syntax that are not necessary to support the current functionality in the Contracts MVP, but should be satisfied to allow for future evolution of the C++ Contracts facility.

Note that we are *not* proposing to add any of the new features that motivate the requirements in this section, we merely intend to reserve the syntactic space for them in case we decide to add them post MVP (which we might or might not end up doing).

Note also that there are differences in plausibility for these future extensions. For example, [future.meta] is probably very plausible, because the C++20 Contracts proposal [P0542R5] already had such meta-annotations, and many concrete use cases for them are known; on the other hand, requirements like [future.reuse] or [future.invar] are probably somewhat less plausible, because we do not yet fully understand how to implement such features, and do not yet have any experience with them.

7.1 Non-`const` non-reference parameters [future.params]

The Contracts syntax should be open to an extension for *some* mechanism to refer to a non-`const` non-reference function parameter in a contract predicate. In particular, in a postcondition, we should be able to refer to a copy of the parameter value that was passed in as well as to the value of the same parameter at the point when the predicate is evaluated.

In the C++20 Contracts proposal [P0542R5], referring to such a parameter in a postcondition was undefined behaviour; in the current Contracts MVP, it is ill-formed. Allowing it post-MVP enables use cases like referencing a moved-from object in a postcondition or performing a destructive operation such as advancing a forward iterator in a predicate.

Closure-based syntax satisfies this requirement by proposing an extension that allows to name such parameters with a capture (see [P2461R1] section 3.2.1); this simultaneously also satisfies the more general requirement [future.captures]. With this syntax, we can write preconditions or assertions that need to mutate a copy of a parameter as follows:

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

According to [P2935R0], attribute-like syntax could satisfy this requirement via init captures (see [future.captures.init] below), but not captures-by-value as above, making the attribute-like syntax version more verbose. No attempt has been made to satisfy this requirement with condition-centric syntax.

It is conceivable that this requirement could also be satisfied through techniques other than captures, although no such technique has been proposed to date.

7.2 Captures [future.captures]

Beyond the ability to refer to non-const non-reference parameters ([future.params]), the Contracts syntax should be open to an extension for capturing arbitrary values for use in a contract predicate.

For preconditions and postconditions, such captures would be initialised at function invocation time; for assertions, they would be initialised when control flow reaches the assertion. Note that for postconditions, the predicate will be evaluated when the function returns. This requirement is more powerful than [future.params] because it allows capturing any value, not just parameters.

The closure-based syntax was designed specifically to satisfy [future.params] and [future.captures] with the same technique that works analogous to lambda captures. There are many use cases for such captures (see [P2461R1] section 5.1) where the user may want to capture values other than a parameter value. Consider for example:

```
void vector::push_back(const T& v)
  post [old_size = size()] { size() == old_size + 1 };
```

Note also that this implies a syntactic requirement that these captures need to be in front of the predicate, even if other possible meta-annotations might be placed after the predicate (see [future.meta]), because the captures may change how the predicate is parsed (see also [future.prim]).

[P2935R0] extends the attribute-like syntax to support init-captures, although they arguably do not look quite as natural and elegant as they do in closure-based syntax:

```
void vector::push_back(const T& v)
  [[ post [old_size = size()] : size() == old_size + 1 ]];
```

Note that unlike closure-based syntax (which supports the full breadth of captures: by value, by reference, init-captures, etc.), [P2935R0] suggests that we should only support init-captures in contract-checking annotations. The paper argues that supporting capture-by-value would encourage hiding the name of the function parameter with a captured copy. Whether it is sufficient to provide just init-captures in contract-checking annotations as in [P2935R0] or whether Contracts should provide the full breadth of capture syntax as in [P2461R1] is a question we leave to future papers.

Further, supporting capture-by-value in attribute-like syntax is not possible using the syntax `[x, y]` for the capture because this would be ambiguous with a structured binding for the return value ([future.struct]). Closure-based syntax provides two distinct syntactic places for the two, while attribute-like syntax does not.

No attempt has been made to satisfy this requirement with condition-centric syntax.

7.3 Structured binding return value [future.struct]

For postconditions, if the return type is compatible with structured bindings, the Contracts syntax should provide a syntactic space for a list of identifiers to destructure the return value and refer to the elements within the postcondition's predicate.

For example, we should be able to write a postcondition like this one:

```
auto returns_triple()
  post ([x, y, z]) { x > y && y > z };  // works in closure-based syntax
```

Note that a structured binding is itself just syntactic sugar. If the syntax does not satisfy this requirement, we can always work around it, with the tradeoff of more verbosity and potentially poorer readability.

Closure-based syntax satisfies this requirement in a straightforward way (see code above), because it has a separate syntactic place for the return value. [P2935R0] proposes how attribute-like syntax could support this requirement as well:

```
auto returns_triple()
  [[post [x, y, z]: x > y && y > z ]];
```

Note that attribute-like syntax would have to use nested `[]` both for [future.captures] and for [future.struct]. Because [P2935R0] proposes to support init-captures, but no other types of captures, there is no parsing ambiguity, and both can be supported at the same time, but the syntax arguably looks unnatural and hard to read:

```
Interval cropInterval(Interval& interval)
  [[ post [old_size = interval.size] [ptr, size]: size <= old_size ]];
```

[P2935R0] notes that we can introduce optional or mandatory parentheses around the structured binding in this case, to make the usage of nested `[]` somewhat less ambiguous to the reader, although this modification does not necessarily make the syntax any easier to read:

```
Interval cropInterval(Interval& interval)
  [[ post [old_size = interval.size] ([ptr, size]): size <= old_size ]];
```

No investigation has been conducted so far into how this requirement (or even the basic requirement of naming the return value; see [func.retval]) could be satisfied by condition-centric syntax.

7.4 Contract reuse [future.reuse]

For the case of a large set of functions all sharing the same set of preconditions and/or postconditions, the Contracts syntax should be open to an extension that allows the user to factor out this set of contract-checking annotations and re-use it, rather than having to repeat the same set over and over again.

For example (with a hypothetical, not proposed syntax):

```
template <typename T>
contract nonneg_noneq(T a, T b)  // declares a reusable contract
  [[ pre: a >= 0 ]]
  [[ pre: b >= 0 ]]
  [[ pre: a != b ]];

int f(int a, int b) [[nonneg_noneq(a, b)]];
int g(int a, int b) [[nonneg_noneq(a, b)]];
int h(int a, int b) [[nonneg_noneq(a, b)]];
```

It has been argued that such a reuse of a set of contracts can be achieved without a dedicated feature. We can factor out a set of preconditions into a regular function returning a `bool`, and then call that single function in the precondition of each of the functions in our set. However, such a workaround does not provide the ability for a Contracts facility to identify and diagnose the particular precondition or postcondition that failed, and for the user to handle these violations individually. This can only be achieved with a dedicated feature that allows a first-class contract declaration, as above.

Note that such a mechanism for declaring a named contract provides a means for syntax extensions similar to what is enabled by such declarations of concepts. For example, this would allow a contract name to be used in combination with a type specifier in a parameter declaration (see also [future.abbrev]).

An analysis of how this requirement could be realised with the current syntax proposals has not yet been conducted; we defer such analysis to future papers.

7.5 Meta-annotations [future.meta]

The Contracts syntax should offer a syntactic place to place a meta-annotation or label on a contract-checking annotation, to tell the compiler about Standard-defined properties of the contract-checking annotation which the compiler would not be able to determine for itself. It should be possible to add multiple such markers to the same contract-checking annotation as they might be orthogonal.

Use cases for such meta-annotations include:

- When checking the contract predicate would violate the complexity and/or performance guarantees of a function (for example, `audit` from the C++20 Contracts proposal [P0542R5]),
- When the contract predicate cannot be evaluated at runtime because it refers to functions that do not have a definition (for example, a magic function determining whether `r` is a valid range),
- When the contract predicate should not be evaluated at runtime because the information it conveys is useful for a standard analyser but not necessarily for the program itself,
- When the contract annotation is newly introduced into an existing production codebase,
- When the user wishes to specify an explicit contract semantic (ignore, observe, enforce, etc.) for the given contract-checking annotation.

Note that this requirement can be satisfied by adding Standard-defined single-identifier labels (such as `audit`, `axiom`, and `default` from the C++20 Contracts proposal [P0542R5]) in some appropriate syntactic place, but also through other means such as string tags:

```
// Hypothetical solutions with attribute-like syntax:
[[pre audit v22: x > 0]]                   // C++20 style
[[pre(x > 0), tag("audit"), tag("v22")]]  // string tags
[[pre<audit | v22> : x > 0 ]];             // inside angly brackets

// Hypothetical solutions with closure-based syntax:
pre<audit, v22> {x > 0};
pre audit v22 {x > 0};

// Hypothetical solutions with condition-centric syntax:
pre<audit, v22> (x > 0);
pre audit v22 (x > 0);
```

7.6 Parametrised meta-annotations [future.meta.param]

The Contracts syntax should offer a syntactic place to have parameters on such meta-annotations or labels.

Generic programming can require such parameters on labels as whether a particular label applied can be dependent on the result of a compile-time metaprogram, for example:

- Whether a precondition such as `std::distance(begin,end)` can be evaluated at runtime may be dependent on the iterator category of `begin` and `end`:

  ```
  template <typename Iter>
  void take3(Iter begin, Iter end)
    [[ pre checked<std::forward_iterator<Iter>> :
       std::distance(begin, end) >= 3 ]];
  ```

- Whether a contract-checking annotation is audit-level or not might be dependent on properties of a template parameter, such as its size:

  ```
  [[pre audit<(sizeof(T) > N)>: x > 0]];
  ```

- An `audit` label might be abstracted into a `cost` label with a numeric parameter, to provide more fine-grained information about the algorithmic and/or performance cost of checking a contract predicate:

  ```
  [[ pre cost<audit_cost> : x > 0 ]]   // equivalent to `audit`
  [[ pre cost<audit_cost/2> : x > 0 ]] // “half” the cost of `audit`,
                                       // enforced in more configurations
  ```

Note that this requirement precludes a Contracts syntax where both the label parameter and the predicate are placed inside parentheses, because this might introduce a parsing ambiguity (see also [compat.parse]):

```
pre audit(sizeof(T) > N); // is (...) the predicate or a param of `audit`?
```

Note further that unlike simple labels without parameters ([future.meta]), this requirement cannot be fully satisfied if the annotations are restricted to strings.

7.7 User-defined meta-annotations [future.meta.user]

The Contracts syntax should offer a syntactic place and form for users to provide meta-annotations or labels that they have declared themselves, in a fashion that will not conflict with standard-provided names.

Use cases for such user-provided labels on contract annotations include:

- Marking that the annotation was added in a specific library version. In this way, whether the annotation is "new" can be based on what the last stable version deployed was. For example, for a precondition introduced in library version 3.1.0:

  ```
  // Hypothetical solution with attribute-like syntax:
  void f(int x) [[ pre mylib::version(3,1,0) : x > 0 ]];

  // Hypothetical solution with condition-centric syntax:
  void f(int x) pre<mylib::version<3, 1, 0>> (x > 0);
  ```

- Marking that the annotation should be processed explicitly by a particular static-analysis tool;
- Marking that the annotation has a certain large/small cost of execution relative to the function being annotated, in case the user needs more granularity than the Standard-provided properties.

Note that this requirement implies the ability to put user-defined marks into a namespace or some similar mechanism.

We might want to consider a mechanism akin to a using-declaration for such namespaced marks to avoid repeating the namespace. This would place further restrictions on the possible syntax.

7.8 Meta-annotations re-using existing keywords [future.meta.keyword]

The Contracts syntax should allow to use existing C++ keywords as the identifiers for meta-information on a contract-checking annotation.

Use cases include:

- C++20 contracts [P0542R5] used the `default` keyword to identify contract-checking annotations with a runtime cost that was not abnormal,
- The `new` keyword is a natural term to use for contracts that have been newly added into an existing production codebase.

Note that this requirement seems to restrict the syntax to something where the whole contract-checking annotation is clearly delimited by special tokens from other C++ code, or at least parsing would be substantially more difficult otherwise.

```
[[ pre new: x > 0]];    // attribute-like syntax
[{ pre new: x > 0 }];   // hypothetical non-attribute-like with delimiters

pre new {x > 0};  // closure-based: probably does not work
pre new (x > 0);  // condition-centric: probably does not work
```

In addition, such labels could create an additional source of confusion because the same token might have a different meaning as a label and in an expression that labels are parameterised on. Consider:

```
[[ pre new<2> cost<sizeof(new int)>: x > 0 ]];
[[ pre template<(X < 4)>: x > 0]];
```

Depending on the syntax chosen, this would be more or less confusing:

```
[[ pre static_cast<v> : i > 0 ]];  // clear that the expression `i > 0` is
                                   // predicate and not argument of label

pre static_cast<v>(i > 0);         // unclear
```

Note that if the syntax does not satisfy this requirement, we can always work around it by choosing identifiers that are not keywords, with the tradeoff that these identifiers might not be a natural fit for cases like the ones listed above.

7.9 Non-ignorable meta-annotations [future.meta.noignore]

The Contracts syntax should allow for an extension to introduce meta-annotations or labels as discussed in [future.meta] which do not fit the Ignorability Rules for standard attributes established in [P2552R3].

Examples for such non-ignorable meta-annotations include:

- Any label that restricts the semantics available to a contract-checking annotation to a checked semantic ("observe" or "enforce") is not itself ignorable (see also discussion in [basic.lang]);
- Other features that could be attached to a label, such as allowing a label to identify a local contract-violation handler to use, would not be ignorable.

Note that this requirement means that using standard attribute syntax for *labels* would not be suitable (regardless of the primary syntax for contract-checking annotations – shown here with condition-centric syntax):

```
// Does not work:
void f(int x) pre (x > 0) [[ unchecked ]];
void f(int x) pre (x > 0) [[ mylib::use_secure_violation_handler ]];
```

Alternatively, we could use some other non-attribute-like syntax for labels that surrounds them with delimiter tokens (such as `[{ … }]` or `@( … )`), but depending on the choice of the primary syntax, this could end up looking rather messy and hard-to-read (see [basic.aesthetic], [basic.brief], [basic.access]), for example with attribute-like syntax:

```
void f(int x) [[ pre: x > 0]] [{ unchecked }];
```

Another way to satisfy this requirement would be to use standard attribute syntax for labels, but change the current Ignorability Rules for standard attributes in C++.

7.10 Primary vs. secondary information [future.prim]

Any secondary information on the contract-checking annotation, such as meta-annotations (see [future.meta]), should be clearly visible as secondary, and not impede the identification and readability of the primary information:

- the contract kind,
- the predicate,
- the identifier introduced to refer to the return object in a postcondition, if we choose to do that (see [func.retval.userdef]),
- any captures, if we choose to introduce them post-MVP (see [future.captures]).

In short, anything that impacts parsing the expression needs to be considered primary; conversely, things that only impact the evaluation of the contract-checking annotation, might be considered secondary.

As an example of a syntax that does not satisfy this requirement, consider:

```
std::vector<T> getSortedVector()
  [[ post audit r: is_sorted(r) ]];
```

Here, a piece of primary information – the introduction of `r` as the name for the function's return value – is not sufficiently highlighted compared to a piece of secondary information – a meta-annotation indicating that this is an `audit` level contract. Instead, both are placed right next to each other.

Notably, the original C++20 Contracts proposal [P0542R5] also did not satisfy this requirement, as it used almost the same syntax as above: in C++20 Contracts, this contract-checking annotation would have been spelled `[[ ensures audit r: is_sorted(r) ]]`.

Moreover, as described in [P2935R0], the label takes precedent over naming the return value when resolving an ambiguity, in a further violation of this requirement:

```
[[ post audit : true ]]     // `audit` is an audit label
[[ post audit : audit ]];   // Error, invalid expression
[[ post (audit) : audit ]]; // `audit` is the return value
```

It is currently an open question what the ideal way would be to separate the secondary information from the primary information in code like the above. This question needs to be answered by future papers. It might be putting the secondary information towards the very end or front of the contract annotation, additionally surrounding it with parentheses, curly braces, or square brackets, putting it into a separate attribute nested within the contract annotation, or associating it with a namespace or module interface rather than an individual contract annotation. We could also have only a predicate in the contract-checking annotation and have the secondary information in an attribute-like annotation that follows. Note that this could conflict with some of the other requirements, for example with [future.meta.noignore] if we choose standard attribute syntax for these annotations.

[P2935R0] suggests an attempt at satisfying this requirement with attribute-like syntax by optionally placing the secondary information at the end, following another colon:

```
std::vector<T> getSortedVector()
  [[ post r: is_sorted(r) : audit ]];
```

7.11 Invariants [future.invar]

The Contracts syntax should be open to an extension for expressing class invariants. It should therefore be chosen such that a new kind of contract-checking annotation, "invariant", can be placed at class scope.

Note that the syntactic space for declarations at class scope is a lot more crowded than that at the end of a function declaration. This might restrict the choice of Contracts syntax. Consider:

```
class X {
  [[ invariant: is_sorted() ]];  // attribute-like syntax
  // members and member functions...
};

class X {
  invariant {is_sorted()};       // closure-based syntax
  // members and member functions...
}

class X {
  invariant (is_sorted());       // condition-centric syntax
  // members and member functions...
}
```

The attribute-like syntax (or any other syntax with clear delimiter tokens around the contract-checking annotation) is not going to run into any problems here. For closure-based and condition-centric syntax, this is less obvious. At first glance, it seems that such annotations at least cannot represent valid code today; but this requires a deeper analysis.

Note that class invariants can be expressed in terms of preconditions and postconditions on member functions. The verbosity and repetition of such an approach could be somewhat reduced if we had a way to declare reusable contracts (see [future.reuse]). There is currently no formal proposal to add class invariants; it has been pointed out that making such annotations work in practice is rather non-trivial and requires a thorough analysis. We currently do not have any experience with such a facility in C++.

7.12 Procedural interfaces [future.interface]

The Contracts syntax should be open to an extension for supporting procedural interfaces as described in [P0465R0].

Procedural interfaces enable fully and clearly capturing a much richer set of functionality than can be accomplished with preconditions and postconditions alone. To give just one example use case, consider a contract check on the statement "this function will not throw" which cannot be expressed as a predicate that is evaluated only when the function returns normally. With procedural interfaces, this can be expressed as follows (for this example, we are using a hypothetical extension of the attribute-like syntax, instead of the hypothetical syntax from [P0465R0], which was not designed with a Contracts facility in mind):

```
void f()
  [[ interface :
    try {
      implementation;
    } catch (...) {
      [[ assert : false ]]; // contract violation
    } ]];
```

The Contracts syntax should therefore be chosen such that we can extend it with a new kind of contract annotation, "interface", or possibly a brace-delimited "interface" block, which can in turn contain nested contract annotations such as assertions, as well as statements such as the `` `implementation;` `` statement, and has the same functionality for integrating with other extensions that the existing contract kinds (pre, post, and assert) provide.

7.13 `requires` clauses [future.requires]

The Contracts syntax provide an appropriate syntactic position to apply, to each individual contract-checking annotation, a `requires` clause that controls when that annotation is considered.

The ability to add `requires` clauses would greatly simplify the writing of contract-checks in generic code. In generic programming a precondition or postcondition might only be in effect (or even expressible) for certain template parameters, and for others it will not even be syntactically well-formed. The implementation of a function itself may even stay the same, such as when precondition usability is based on the iterator category of a provided iterator.

While SFINAE or concepts can be used to provide different function templates with different contracts for this purpose, this can quickly lead to a combinatorial explosion of repeated function declarations and implementations. The Contracts syntax should therefore provide an appropriate syntactic position to apply, to each individual contract-checking annotation, a `requires` clause that controls when that annotation is considered.

Note that such a clause should not clash with a possible `requires` clause appertaining to the function itself, which places restrictions on the Contracts syntax. For example, the following syntax would probably not work (see also [func.pos.prepost]):

```
template <typename T>
void f(T x)
  pre (x != 0) requires std::is_integral_v<T>;
               // ambiguous: does this requires-clause appertain to the
               // precondition, or to `f` itself?
```

7.14 Abbreviated syntax on parameter declarations [future.abbrev]

Post-MVP, we might want to add an abbreviated *short-form* Contracts syntax that appertains to a parameter declaration as opposed to a function declaration. The Contracts MVP syntax (the *long-form* syntax) should be chosen such that we can later introduce a short-form syntax appertaining to parameters having some degree of consistency and cohesion with the long-form syntax.

Use cases for such short-form Contracts appertaining to parameters can be found in several contract-like features that exist today as non-standard compiler extensions, for example:

- Nullability attributes supported by Clang and GCC. The syntax uses keywords or attributes:

  ```
  void f1(int* _Nonnull p);             // p is not null.
  void f2(int* _Nullable p);            // p may be null.
  void f3(int* p [[gnu::nonnull]]);     // p is not null.
  int* f4 [[gnu::returns_nonnull]] () ; // return value is not null.
  ```

- A proposal for a bounds checking feature in Clang that uses annotations on parameters to specify range boundaries. This feature has already been deployed within Apple. The syntax uses keywords:

  ```
  void f5(int* __counted_by(n) p, size_t n);
      // p points to an array indexable from [p,p+n).
  void f6(void* __sized_by(s) p, size_t s);
      // p points to a buffer of size s.
  void f7(int* __ended_by(end) p, int *end);
      // p points to an array indexable from [p,end).
  ```

Note that if we choose attribute-like syntax, using `[[ … ]]` as delimiter tokens, a short-form syntax using the same delimiter tokens will conflict with standard attribute syntax:

```
void f(int p [[nonnull]]);  // does not work as short-form contract syntax
```

We would have to resort to alternative delimiter tokens such as `[{ … }]` or `@( … )` (for short-form syntax only, or for both short-form and long-form), or to spelling the short-form syntax without delimiter tokens, or perhaps to using another way to disambiguate the short-form syntax from attributes:

```
void f(int p @(nonnull));     // alternative delimiter tokens
void f(int p : nonnull);      // syntax with no delimiter tokens
void f(int p [[: nonnull]]);  // disambiguating colon
```

7.15 General extensibility [future.general]

This list of requirements, while large, cannot claim to be exhaustive of all things that might possibly be needed in the future from a Contracts syntax. There should be room for extensibility within the syntax that will not be hindered by the wide range of possible things that might be adjacent to a contract-checking annotation in the many places it might be used.

8 Conclusions

While the main goal of this paper is to establish requirements, not to conduct an in-depth analysis of the different proposed (or hypothetically possible) Contracts syntaxes, this exercise does provide some preliminary insights into the advantages, disadvantages, and existing gaps of the different design directions.

The attribute-like syntax is currently the only option with implementation experience. A strength of the attribute-like syntax is that it surrounds each contract-checking annotation with clear delimiter tokens (in this case, `[[ … ]]`) that separates them from all other C++ code. This is more friendly for certain post-MVP extensions: with delimiter tokens around the annotations, there is less danger of running into parsing ambiguities due to the already crowded C++ grammar, and more design freedom within the contract-checking annotations themselves to add various kinds of custom labels and meta-annotations using any kind of custom grammar (such as supporting the `default` label from C++20 Contracts which is a keyword outside of a contract-checking annotation).

The main weaknesses of the attribute-like syntax is that many people perceive it as too "heavy" and "ugly", and that it treads on design space currently occupied by standard attributes in both C++ and C. The latter creates potential teaching challenges and inconsistencies (although to what extent this is actually the case is a matter of ongoing debate). The overlap with attributes could be avoided by a hypothetical syntax using different delimiter tokens, say, `[{ … }]`, or `@( … )`, but depending on personal preference such designs could potentially be perceived as even more "ugly".

Closure-based syntax, on the other hand, does not seem to suffer from the perceived aesthetic issues of attribute-like syntax, and it does not overlap with attributes. It does not offer the full grammar freedom of token-delimited contract-checking annotations, but its capture syntax provides a very elegant solution for enabling certain other post-MVP extensions, in particular for writing captures for use in predicates, which is a very powerful technique. Attribute-like syntax allows this, too, but suffers from the fact that we end up with nested square brackets, making the resulting syntax for captures (and for other extensions such as destructuring the return value) look less natural than in closure-based syntax and harder to read.

A notable weakness of the closure-based syntax is that it places the predicate (i.e., an expression) between curly braces, which is awkward as in C++ statements normally go between curly braces while expressions go between parentheses.

Condition-centric syntax solves the problem by placing the predicate between parentheses, which some consider more elegant and more consistent with existing C++. However, condition-centric syntax at this point seems the most incomplete design: it is unclear how we would support naming the return value of a function in the postcondition, or how a number of post-MVP extensions could be supported by this syntax, due in part to various potential grammar ambiguities. In addition, it is particularly unfortunate that condition-centric syntax cannot use the identifier `assert` for assertions without breaking existing usage of the `assert` macro from header <cassert>.

If we wish to add secondary information such as meta-annotations to contract-checking annotations in the future, we may wish to make such information clearly visible as secondary, and not impede the identification and readability of the primary information, such as the contract predicate and the name for the return value. Here, another notable difference between the syntaxes arises. Attribute-like syntax places all of these on the same level between the same delimiter tokens, and only separates the predicate by a colon, making it more difficult to distinguish the primary information. On the other hand, condition-centric syntax clearly separates the predicate from other information by placing it between parentheses, and closure-based syntax clearly separates both the predicate and the name for the return value from any other information by placing them between braces and parentheses, respectively.

Document history

R0 → R1

- Added new requirements [func.mix] and [future.abbrev].
- Renamed "Accessibility" [basic.access] to "Teachability" [basic.teach].
- Merged [basic.lang] and [compat.cpp] into a single requirement [basic.cpp] as they were heavily overlapping and the latter was not very clearly separated from requirement [compat.break].
- Factored out the two proposed approaches for satisfying [func.retval] into separate, more specific requirements that can be voted on separately: [func.retval.predef] and [func.retval.userdef], respectively.
- Factored out the basic requirements on position and name lookup from [func.pos.prepost] and [func.pos.assert] into single requirement [func.pos]
- Made requirements [func.pos.prepost] and [func.pos.assert] more specific: pre/postconditions must go after parameter declaration, assertions might appear anywhere a C assert can appear.
- Added more rationale to [compat.back], [compat.impl], and [future.reuse].
- Rephrased [basic.aesthetic] and [basic.brief] to present a more balanced view on these requirements.
- Clarified the intent of [compat.c].
- Fixed inaccurate description of the syntactic position of pre/postconditions in attribute-like syntax in [func.pos].
- Added example of ambiguous position and name lookup for a function returning a pointer to an array in [func.pos.prepost].
- Added a mention of the "piano syntax" idea as a syntax violating [func.pos.prepost] and [func.mix].
- Added a code example for attribute-like syntax violating [future.prim].
- Slightly rewrote [future.captures] to clarify the difference between capturing parameters and capturing arbitrary values, and between allowing init-captures and allowing any kind of captures like in lambdas.

- Added mentions of the various possible extensions and design alternatives for attribute-like syntax discussed in [P2935R0] throughout the paper.
- Added a paragraph discussing primary vs. secondary information to summary.
- Mentioned differences in plausibility for future evolution requirements.
- Rewrote methodology section mentioning the upcoming SG21 electronic poll for determining the priority of requirements and classifying them as objective or subjective; removed such prioritisation or classification from the description of the individual requirements.

Acknowledgements

Thanks to Jens Maurer, Arthur O'Dwyer, Nathan Sidwell, Oliver Rosten, and Walter E. Brown for their very helpful feedback on this paper.

References

[N1613] Thorsten Ottosen: Proposal to add Design by Contract to C++. 2004-03-29
[N4415] Gabriel Dos Reis, J. Daniel Garcia, Francesco Logozzo, Manuel Fähndrich, and Shuvendu Lahiri: Simple Contracts for C++. 2015-04-12
[N4435] Walter E. Brown: Proposing Contract Attributes. 2015-04-09
[P0147R0] Lawrence Crowl and Thorsten Ottosen: The Use and Implementation of Contracts. 2015-11-08
[P0465R0] Lisa Lippincott: Procedural function interfaces. 2016-10-16
[P0542R5] Gabriel Dos Reis, J. Daniel Garcia, John Lakos, Alisdair Meredith, Nathan Myers, and Bjarne Stroustrup: Support for contract based programming in C++. 2018-06-08
[P1429R3] Joshua Berne and John Lakos: Contracts That Work. 2019-07-23
[P1607R1] Joshua Berne, Jeff Snyder, and Ryan McDougall: Minimizing Contracts. 2019-07-23
[P1680R0] Andrew Sutton and Jeff Chapman: Implementing Contracts in GCC. 2019-06-17
[P1774R8] Timur Doumler: Portable Assumptions. 2022-06-14
[P1995R1] Joshua Berne, Timur Doumler, Andrzej Krzemieński, Ryan McDougall, and Herb Sutter: Contracts – Use Cases. 2020-03-02
[P2388R4] Andrzej Krzemieński and Gašper Aźman: Minimum Contract Support: either *No_eval* or *Eval_and_abort*. 2021-11-15
[P2461R1] Gašper Ažman: Closure-Based Syntax for Contracts. 2021-11-15
[P2487R1] Andrzej Krzemieński: Is attribute-like syntax adequate for contract annotations? 2023-06-11
[P2521R4] Andrzej Krzemieński, Gašper Aźman, Joshua Berne, Bronek Kozicki, Ryan McDougall, and Caleb Sunstrum: Contract support — Record of SG21 consensus. 2023-06-15
[P2552R3] Timur Doumler: On the ignorability of standard attributes. 2023-06-14
[P2695R1] Timur Doumler and John Spicer: A proposed plan for Contracts in C++. 2023-02-09
[P2737R0] Andrew Tomazos: Proposal of Condition-centric Contracts Syntax. 2022-12-04

[P2751R0] Joshua Berne: Evaluation of *Checked* Contract-Checking Annotations. 2023-01-14

[P2811R7] Joshua Berne: Contract-Violation Handlers. 2023-06-27

[P2884R0] Alisdair Meredith: `assert` Should Be A Keyword In C++26. 2023-05-15

[P2935R0] Joshua Berne: An Attribute-Like Syntax for Contracts. 2023-08-14