A natural syntax for Contracts

Jens Maurer (jens.maurer@gmx.net)
Timur Doumler (papers@timur.audio)

Document #: P2961R0
Date: 2023-09-17
Project: Programming Language C++
Audience: SG21

Abstract

We propose a syntax for Contracts that naturally fits into existing C++, does not overlap with the design space of other C++ features such as attributes or lambdas, is intuitive, lightweight and elegant, and designed to aid readability by emphasising the primary information. The proposed syntax removes the weaknesses of attribute-like and closure-based syntax while maintaining full compatibility and extensibility in line with the SG21 requirements for a Contracts syntax.

1 Motivation

SG21 is currently working on standardising a first version of a Contracts facility — the so-called *Contracts MVP*. According to our roadmap [P2695R1], the last remaining major design decision is the choice of syntax. The syntax proposal currently under consideration is the so-called attribute-like syntax ([P2935R0]). While attribute-like syntax has its strengths — existing implementation experience in GCC (see [P1680R0]) and the possibility to lean on existing standard attribute grammar rather than inventing new grammar — it also has considerable weaknesses.

Attribute-like syntax uses `[[ ... ]]` delimiter tokens around every contract-checking annotation. This has been called a "heavy" syntax and is perceived as "ugly" by some users. It also makes contract-checking annotations look like standard attributes, even though they are not attributes and do not behave as such, creating confusion. For example, a contract-checking annotation can create an entirely new code path out of a function (e.g. via a throwing violation handler), for assertions even out of the middle of a function body, something that standard attributes were never designed to do. [P2487R1] lists many other differences between contracts and standard attributes.

The syntactic position of a contract-checking annotation using attribute-like syntax is problematic. If we wish to re-use the existing standard attribute grammar, then preconditions and postconditions need to go before any trailing return type, virtual specifiers, and a `requires` clause. This goes against the natural reading order of a function declaration and requires delayed parsing of postconditions. Assertions need to appertain to a null statement, which means they cannot be used freely as an expression and cannot serve as a complete drop-in replacement for C `assert`.

On the other hand, if we place attribute-like preconditions and postconditions at the end of a function declaration, and/or allow attribute-like assertions to be expressions rather than statements, we need to place attribute-like entities at a novel syntactic position that is not currently supported for standard attributes, and that we do not have implementation experience with, throwing away the main advantages of attribute-like syntax.

Further, attribute-like syntax places delimiter tokens around the whole contract-checking annotation, but does not syntactically separate any of its parts other than by a single colon before the predicate. This makes it difficult to visually distinguish the primary information (the predicate, contract kind, and name for the return value) from secondary information (such as labels that could be added post-MVP). Even worse, because all parts of the contract-checking annotation except the predicate occupy the same syntactic space (immediately preceding the colon), multiple parsing ambiguities arise, in particular with possible post-MVP extensions.

For example, there is an ambiguity between a label and the name of the return value before the colon (as both are identifiers); adding a new standard label can break existing code. One suggested workaround is to require extra parentheses around the return value, but that makes it syntactically ambiguous with an argument of the preceding label. As another example, there is an ambiguity between a capture and a structured binding before the colon (as both use square brackets). The suggested workarounds are similarly awkward: allowing only init-captures but not default captures and requiring extra parentheses around the structured binding.

Finally, we cannot have standard attributes appertaining to an attribute-like contract-checking annotation or specify certain other post-MVP extensions such as procedural interfaces without introducing novel and awkward grammar for attributes nested inside attributes.

In this paper, we propose a new natural, lightweight, and intuitive syntax for Contracts that solves all of the above problems.

2 Prior work

The first proposed alternative to attribute-like syntax was closure-based syntax [P2461R1]. Closure-based syntax remedies many of the issues with attribute-like syntax, but creates other issues of its own. It places the predicate between curly braces, which is awkward: normally, in C++ we place statements between braces but expressions between parentheses, and the predicate is an expression. Furthermore, it makes a contract-checking annotation look very much like a lambda, even though the two features have almost nothing in common.

The second proposed alternative was condition-centric syntax [P2737R0]. It was not a complete proposal, as it did not consider any post-MVP features such as captures, `requires` clauses, and labels on contract-checking annotations. But it was the first proposal to use the basic syntax structure that we use here as well:

```
contract-kind ( predicate )
```

Along with this basic syntax structure, [P2737R0] proposed a series of other design choices orthogonal to the choice of syntax, in particular:

— to rename "assertion" to the newly coined term "incondition",

— to use `precond`, `postcond`, and `incond`, instead of `pre`, `post`, and `assert`, respectively,

— to make all three of the above *full* keywords rather than contextual keywords,

— to use a predefined identifier `result` for the return value of a function instead of letting the user introduce their own name.

The above design choices have been poorly received in SG21. Instead of abandoning these additional design choices and instead focusing on the basic syntax structure, which was received with interest, the author chose to abandon the whole proposal.

In this paper, we improve upon both closure-based and condition-centric syntax. We adopt many of the ideas of closure-based syntax, in particular the lack of delimiter tokens around the contract-checking annotation and the syntax for captures. However, instead of using the problematic curly braces, we use parentheses around the predicate, following the basic syntax structure of the condition-centric syntax [P2737R0] but without adopting any of the other design choices from that paper.

Building on these ideas, we developed a complete syntax proposal that is fit for purpose in the Contracts MVP and accommodates all relevant post-MVP extensions such as captures, `requires` clauses, and labels on contract-checking annotations.

The author of the closure-based syntax has reviewed our proposal and decided to discontinue the closure-based syntax in favour of our proposal as it subsumes the ideas of the closure-based syntax and improves upon it. We are therefore left with attribute-like syntax and our proposal as the only two still active proposals for a Contracts syntax.

3 Design goals

We focus on the following design goals, which we believe are not sufficiently met by attribute-like or closure-based syntax:

— The syntax should fit naturally into existing C++. The intent should be intuitively understandable by users unfamiliar with contract-checking annotations without creating any confusion.

— A contract-checking annotation should not resemble an attribute, a lambda, or any other pre-existing C++ construct. It should sit in its own, instantly recognisable design space.

— The syntax should feel elegant and lightweight. It should not use more tokens and characters than necessary.

— To aid readability, the syntax should visually emphasise the primary information, that is, the contract predicate and the parts of a contract-checking annotation that may affect how the predicate is parsed: the contract kind, the name for the return value, and (post MVP) the captures. These should be syntactically separated from secondary information about the contract, such as (post MVP) labels to control the contract semantic.

At the same time, we maintain all the other desirable properties that attribute-like and closure-based syntax offer, such as compatibility (no parsing ambiguities, no breakage or change in meaning of existing C++ code) and extensibility (a natural path for evolution in the post-MVP directions that SG21 considers relevant).

4 The proposal

4.1 Grammar

We propose the following additions to the C++ grammar for the Contracts MVP:

init-declarator:
  declarator initializer$_{opt}$
  declarator requires-clause
  declarator requires-clause$_{opt}$ *pre-or-post-condition-seq*

member-declarator:
  declarator virt-specifier$_{opt}$ *pure-specifier*$_{opt}$ *pre-or-post-condition-seq*$_{opt}$

declarator requires-clause
declarator requires-clause$_{opt}$ *pre-or-post-condition-seq*
declarator brace-or-equal-initializer$_{opt}$
identifier$_{opt}$ *attribute-specifier-seq*$_{opt}$ `:` *brace-or-equal-initializer*$_{opt}$

function-definition:
attribute-specifier-seq$_{opt}$ *decl-specifier-seq*$_{opt}$ *declarator virt-specifier-seq*$_{opt}$
pre-or-post-condition-seq$_{opt}$ *function-body*
attribute-specifier-seq$_{opt}$ *decl-specifier-seq*$_{opt}$ *declarator requires-clause*
pre-or-post-condition-seq$_{opt}$ *function-body*

lambda-declarator:
lambda-specifier-seq noexcept-specifier$_{opt}$ *attribute-specifier-seq*$_{opt}$
trailing-return-type$_{opt}$ *pre-or-post-condition-seq*$_{opt}$
noexcept-specifier attribute-specifier-seq$_{opt}$ *trailing-return-type*$_{opt}$ *pre-or-post-condition-seq*$_{opt}$
trailing-return-type$_{opt}$ *pre-or-post-condition-seq*$_{opt}$
`(` *parameter-declaration-clause* `)` *lambda-specifier-seq*$_{opt}$ *noexcept-specifier*$_{opt}$
attribute-specifier-seq$_{opt}$ *trailing-return-type*$_{opt}$ *requires-clause*$_{opt}$ *pre-or-post-condition-seq*$_{opt}$

unary-expression:
postfix-expression
unary-operator cast-expression
`++` *cast-expression*
`--` *cast-expression*
await-expression
`sizeof` *unary-expression*
`sizeof (` *type-id* `)`
`sizeof ... (` *identifier* `)`
`alignof (` *type-id* `)`
noexcept-expression
new-expression
delete-expression
assert-expression

pre-or-post-condition:
`pre` *contract*
`post` *contract*

pre-or-post-condition-seq:
pre-or-post-condition pre-or-post-condition-seq$_{opt}$

assert-expression:
`assrt` *contract*

contract:
contract-condition // *can be expanded post-MVP, see section 5*

contract-condition:
`(` *return-name*$_{opt}$ *conditional-expression* `)`

return-name:
identifier `:`

4.2 Natural syntax for preconditions and postconditions

To add a precondition (or postcondition) to a function declaration, we simply write `pre` (or `post`), followed by the predicate in parentheses:

```
float sqrt(float x)
  pre (x >= 0);
```

This is a very natural syntax, as it is using parentheses in the same way as other language constructs that have a predicate: `if (`*`expr`*`)`, `while (`*`expr`*`)`, etc.

To introduce a name for the return value of a function, you write it immediately before the predicate, followed by a colon:

```
int f(int x)
  post (r: r > x);
```

Here, `pre` and `post` are contextual keywords. As we will see in section 4.3 below, it is fine to use them as an identifier in all other parts of the function declaration, therefore not breaking any existing code. Preconditions and postconditions are the last part of a function declaration, immediately before the semicolon (or the opening brace if the declaration is a definition):

```
template <typename T>
auto f(T x) -> bool
  requires std::integral<T>
  post (x > 0);
```

This order is consistent with the natural order of reading a function declaration: typically, the reader will first want to see the function signature and whether it is virtual, then any compile-time constraints (the `requires` clause), and finally any runtime constraints (the contract-checking annotations).

4.3 No parsing ambiguities

It has been suggested that the syntax proposed here might create parsing ambiguities with the other parts of a function declaration, such as a trailing return type or `requires` clause, if `pre` or `post` are used as identifiers for variables, functions, or types; but this is not actually the case. The grammar for a trailing return type is `->` *type-id*, and we can unambiguously tell when that *type-id* ends and a *pre-or-postcondition* begins:

```
auto f() -> pre pre(a);    // OK, pre is the return type, pre(a) the precondition
auto g() -> pre<post> pre(a);  // OK, pre<post> is the return type, pre(a) the precondition
```

Further, note that `requires` clauses use a restricted grammar where the expression following the `requires` keyword must be a *primary-expression* or a sequence of *primary-expressions* combined with the `&&` or `||` operators. Any other type of expression, such as a mathematical expression, a cast, or a function call, must be surrounded by parentheses, otherwise the program is ill-formed:

```
template <typename T>
void g() requires pre(a);    // ill-formed today

template <typename T>
void h() requires (pre(a));    // OK

template <typename T>
void j() requires (b)pre(a);    // ill-formed today

template <typename T>
void k() requires ((b)pre(a));    // OK

template <typename T>
void l() requires a < b > pre(c);    // ill-formed today
```

```
template <typename T>
void m() requires (a < b > pre(c));   // OK
```

Therefore, just like with the trailing return type, we can unambiguously tell when the expression ends and a *pre-or-postcondition* begins:

```
template <typename T>
void f() requires (b) pre(a);   // OK, pre(a) is the precondition

template <typename T>
void g() requires a < b > pre(c);   // OK, a<b> is a variable template, pre(c) is the precondition
```

There are further no parsing ambiguities when any given precondition (or postcondition) ends and the next one begins, as the predicate must always be surrounded by parentheses. Therefore, it is also OK to use `pre` and `post` as identifiers inside the predicate. They are parsed as keywords only in the syntactic place where they act as such, everywhere else the usual grammar rules apply:

```
void f(bool pre, bool post)
  pre(pre) pre(post);   // OK
```

4.4 Assertions and the `assert` name clash

We propose to use the same natural syntax, ***`contract-kind`*** `(` ***`predicate`*** `)`, also for assertions. This syntax will look instantly familiar to C++ developers as it is the same basic syntax that one would use today to write a macro-based assertion (but without any of the limitations of a macro-based solution). However, since assertions appear at block scope, unlike `pre` and `post`, we need to claim a full keyword for the contract kind. We therefore need to solve the issue that the most natural keyword, `assert`, would create a name clash with the existing `assert` macro from header `cassert`:

```
void f() {
  int i = get_i();
  assert(i >= 0);  // identical syntax for contract assert and macro assert
}
```

There are in principle three ways to resolve this name clash while keeping the natural syntax:

1. Remove support for header `cassert` from C++ entirely, making it ill-formed to `#include` it;
2. Do not make `#include <cassert>` ill-formed (perhaps deprecate it), but make `assert` a keyword rather than a macro, and silently change the behaviour to being a contract assertion instead of an invocation of the macro;
3. Use a keyword other than `assert` for contract assertions to avoid the name clash.

Option 1 seems too draconian as it would break too much existing code, including code shared between C and C++.

Option 2 seems worth exploring, as the default behaviour of macro `assert` is actually identical to the default behaviour of a contract assertion: print a diagnostic, then terminate the program. Contract-specific extensions like a user-defined violation handler will not affect pre-existing code. Replacing macro `assert` with a full keyword would also solve all the current issues with `assert` being a macro: it cannot be exported from a Standard Library module; it does not work with matched brackets syntax for brackets other than parentheses, such as `<...>`, `{...}`, and `[...]`, and as a result, many C++ constructs such as `assert(X{1, 2})` or `assert(Y<int, int>)` are ill-formed today; etc. (see also [P2264R5] and [P2884R0]).

However, there are several problems with this approach. The first problem is that the behaviour of macro `assert` is tied to whether `NDEBUG` is defined. To maintain compatibility and avoid checking predicates (potentially terminating the program) that were not checked before in an existing program, a compiler would have to apply the *enforce* semantic when `NDEBUG` is not defined, and the *ignore* semantic when `NDEBUG` is defined, to all contract assertions in the program by default (unless something else has been specified by the user). The status quo in the Contracts MVP is that the contract semantic of any contract-checking annotation is implementation-defined; the recommended default contract semantic is *enforce* regardless of whether `NDEBUG` is defined, and this recommendation is not normative (see [P2877R0]).

The second problem is that even if we tie the default contract semantic to `NDEBUG` in this way, contract assertions would still not be a backwards-compatible replacement for the `assert` macro. The *ignore* contract semantic is different from `assert` with `NDEBUG` defined because an ignored contract still fully parses and ODR-uses the entities in its predicate, while an ignored `assert` macro does neither: it simply throws away all tokens inside the macro argument. There are many different ways in which switching from the latter to the former could break existing code (the macro contained invalid tokens; template instantiation triggers a `static_assert`; etc.) or silently change the behaviour of an existing program (it is possible to SFINAE on whether an entity is being ODR-used). Changing ignored contracts to mimic the behaviour of the `assert` macro with `NDEBUG` defined is not an option either as it would go against an important design principle for contracts: whether a contract is checked or ignored should never lead to different code paths being taken, as this could lead to non-portable code and bugs disappearing or appearing when you turn contract checking on and off (see [P2834R1]).

Furthermore, if we claim `assert` as a full keyword, this would also break any existing C++ code that uses the identifier `assert` for a user-defined entity (such as a function, variable, type, or enumerator). We would also have to rename the enumerator `std::contracts::contract_kind::assert` which we adopted into the Contracts MVP via [P2811R7].

Considering the above problems, we propose Option 3. Possible alternative keywords include:

```
ass
asrt
assrt
assertion
co_assert
contract_assert
```

We have picked `assrt` for now, but we are happy with any other choice if this increases consensus. Such a keyword may look weird initially, but just like with `co_yield` and friends, users will get used to it quickly.

4.5 Assertions are expressions

With the natural syntax, it is straightforward to specify assertions as expressions, not statements. Consequently, with our syntax, assertions can be used not only as statements inside a function body, but actually anywhere one could use an expression, and in particular, anywhere one could use an `assert` macro today:

```
class X {
  int* _p;
public:
  X(int* p) : _p((assrt(p), p)) {}  // works
};
```

Therefore, contract assertions as proposed here can act as full drop-in replacements for `assert` macros, and that replacement is easily toolable (search and replace the keyword).

5 Post-MVP extensions

Our proposed syntax provides a natural path for evolution into all of the post-MVP directions that have been suggested so far. In this section, we discuss several possible post-MVP extensions that are of interest to SG21 according to the electronic poll results in [P2885R2] or that have been brought up in discussion since the poll results were published.

5.1 Captures

The contracts grammar proposed here can be extended as follows to allow captures on contract-checking annotations:

> *pre-or-post-condition:*
>   `pre` *contract*
>   `post` *contract*
>
> *assert-expression:*
>   `assrt` *contract*
>
> *contract:*
>   *contract-capture*$_{opt}$ *contract-condition*
>
> *contract-capture:*
>   `[` *capture-list* `]`

Here is a code example:

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

Note that with our syntax, a contract-checking annotation with a capture looks very similar to closure-based syntax, except that the predicate is in parentheses instead of braces. This is the natural choice and avoids making the contract-checking annotation look like a lambda (an entirely different construct). Instead, the syntax looks exactly like the thing that it is: a capture followed by a predicate using that capture. It is a new syntax for a new type of construct, yet it immediately looks familiar and intuitive.

Note further that with our syntax, we have the same design freedom as closure-based syntax [P2461R1] to allow the full capture syntax from lambdas, including default-captures:

```
int min(int x, int y)
  post [x, y] (r: r <= x && r <= y );    // possible with our proposal
```

But we could also choose to restrict ourselves to init-captures as [P2935R0] does.

5.2 `requires`-clauses on contracts

The contracts grammar proposed here can be extended to allow a `requires` clause that appertains to an individual contract-checking annotation. There are at least two possible syntactic positions for such a `requires` clause. We could place it at the end, after the predicate:

> *contract:*
>   *contract-capture*$_{opt}$ *contract-condition* *requires-clause*$_{opt}$

In code:

```
template <typename T>
void f(T x)
  pre (x > 0) requires std::integral<T>;
```

Alternatively, if we want to make the `requires` clause more visually prominent, we could place it at the beginning of the contract-checking annotation, right after the `pre` or `post` contextual keyword:

contract:
  requires-clause$_{opt}$ *contract-capture*$_{opt}$ *contract-condition*

In code:

```
template <typename T>
void f(T x)
  pre requires std::integral<T> (x > 0);
```

Note that neither option creates any parsing ambiguities, for the same reasons as discussed in section 4.3. Note further that both options allow for `requires` clauses appertaining to individual contract-checking annotations to coexist with a `requires` clause appertaining to the function itself:

```
template <typename T>
void f(T x)
  requires std::copyable<T>
  pre (x > 0) requires std::integral<T>;
```

or, alternatively,

```
template <typename T>
void f(T x)
  requires std::copyable<T>
  pre requires std::integral<T> (x > 0);
```

5.3 Attributes appertaining to contracts

Although not covered in [P2885R2], it has been argued that any new syntactic construct in C++, including contract-checking annotations, should allow for the possibility of standard attributes appertaining to it. Some meta-annotations that might be added to contracts post MVP could potentially be expressed as attributes appertaining to a contract-checking annotation.

Support for attributes appertaining to a contract-checking annotation is easy to accommodate with our proposed syntax. Since attributes are optional, ignorable information and are thus not part of a contract's primary information, we believe that it makes most sense to place them at the end of the contract-checking annotation:

contract:
  contract-capture$_{opt}$ *contract-condition* *attribute-specifier-seq*$_{opt}$

In code:

```
template <typename T>
void f(T x)
  pre (x > 0) [[deprecated]];
```

However, just like with `requires` clauses, it is also possible to place the *attribute-specifier-seq* at the beginning, right after the `pre` or `post` contextual keyword in case a more prominent syntactic position is desired.

Note that in either case, there is no grammar ambiguity with the *attribute-specifier-seq* appertaining to any other part of the function declaration, such as the function itself, the function type, or the trailing return type, because all other possible positions for the *attribute-specifier-seq* precede the *pre-or-postcondition*.

5.4 Labels

It has been suggested that post-MVP, we will need meta-annotations on a contract, so-called *labels*, that should not be spelled as attributes because they are not ignorable. The only currently known use case for this is to specify or constrain the possible contract semantic (observe, ignore, enforce) for a given contract; other use cases might be discovered in the future. There are many ways in which our proposed syntax could be extended to accommodate such labels.

If we want to consider such labels secondary information, we can place them at the end of the contract-checking annotation. In order to keep the grammar unambiguous, we need to surround the sequence of labels with delimiter tokens. We cannot use `[[ ... ]]` because these are reserved for attributes (see section 5.3), but we can use pretty much any other set of delimiter tokens, such as `[ ...]`, `<...>`, `{...}`, and so forth, or mark the labels by special characters such as `@`, depending on SG21's preference:

```
void f(int x)
  pre (x > 0) [audit];   // or <audit>, or {audit}, or [{audit}], or @audit ...
```

On the other hand, if we want to consider such labels primary information, we can place them at the beginning of the contract-checking annotation, right after the `pre` or `post` contextual keyword. In this case, we cannot use `[...]` as the delimiters anymore, as it would be ambiguous with the *contract-capture* (see section 5.1), but we can use any of the other options:

```
void f(int x)
  pre <audit> (x > 0);    // or {audit}, or [{audit}], or @audit ...
```

We could also allow both the leading and the trailing position. Our syntax places no restrictions on the internal grammar for these labels. They can be specified to be any kind of token sequence, depending on the design direction we choose post MVP.

One interesting possibility is to specify that the label, or set of labels, shall be a constant expression that evaluates to a compile-time value defining the desired per-contract configuration, perhaps to a value of some new type `std::contract_traits` similar to `std::coroutine_traits`. Such a grammar opens up the power of constant expressions (i.e. almost the full language) for abstracting the computation of the per-contract configuration. The syntax with labels in leading position, right after `pre` or `post` and delimited by `<...>`, seems appealing for this design direction, as the contract-checking annotation will resemble a template that is "templated" on its configuration (which acts as a non-type template parameter), and the constant expression acts as a template argument that "instantiates" (configures) the contract check. The grammar for this could look as follows:

contract:
  contract-eval-specifier$_{opt}$ *contract-capture*$_{opt}$ *contract-condition*

contract-eval-specifier:
  < *constant-expression* >

However, this is only one possible direction. With our proposal, we are not cutting off any other directions. The main difference to labels in attribute-like syntax is that in our syntax, the label sequence goes between delimiter tokens, whereas in attribute-like syntax it goes between the `pre` or `post` keyword and the colon. Arguably, our proposal actually leaves more syntactic freedom for labels than attribute-like syntax does. In attribute-like syntax, the label sequence can syntactically clash with anything else that goes between the `pre` or `post` keyword and the colon, such as the name for the return value. On the other hand, in our syntax, labels are guaranteed to not clash with anything else because they are separated from all other parts of the contract-checking annotation by their own delimiter tokens.

Note also that with our proposal, we can support both standard attributes and non-attribute labels appertaining to the same contract-checking annotation simultaneously, for example:

```
void f(int x)
  pre <audit> (x > 0) [[ deprecated ]];
```

5.5 Destructuring the return value

We can easily and naturally extend the syntax proposed here to add support for destructing the return value of a function with a structured binding when specifying a name for that value:

contract-condition:
  `(` *return-name*$_{opt}$ *conditional-expression* `)`

return-name:
  identifier `:`
  `[` *identifier-list* `]` `:`

This can be very useful in the postcondition of a function that returns a value of a tuple-like type:

```
std::tuple<int, int, int> f()
  post ([x, y, z] : x != y && y != z);
```

5.6 Procedural interfaces

With procedural interfaces, we can express a much richer set of contracts than with preconditions and postconditions alone. The idea was first published by Lisa Lippincott in her paper [P0465R0]. More recently, [P2885R2] and [P2935R0] mentioned the idea of integrating such procedural interfaces into a Contracts facility post-MVP.

With our proposal, we can support procedural interfaces with an interface block delimited by curly braces. This is the natural syntax in C++ for a block containing a list of statements, and very close to Lisa Lippincott's original notation in [P0465R0]. Here is a code example in this syntax — a procedural interface expressing the contract that a function should not throw an exception:

```
void f(int x)
interface {
  try {
    implementation;
  }
  catch (...) {
    assrt(false);
  }
};
```

6 Comparison with attribute-like syntax

In this section we compare different Contracts MVP and post-MVP code examples written in attribute-like syntax as proposed in [P2935R0], side-by-side with the syntax proposed in this paper, and discuss the different tradeoffs. Where appropriate, we mention different possible alternatives.

6.1 MVP functionality

6.1.1 Basic preconditions and postconditions

Shown below is a comparison of the two syntaxes for the most basic usage of preconditions and postconditions:

```
// P2935R0:                                    // This paper:

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

6.1.2 Assertion as a statement

Assertions make it rather obvious that contracts are in fact not attribute-like at all. A contract assertion creates an entirely new code path out of the middle of a function body (for example, via a throwing violation handler), which is something standard attributes were never designed to do.

In attribute-like syntax, an assertion at block scope takes the shape of an attribute appertaining to a null statement; in our proposal, it looks like a regular statement, resembling a function call or the invocation of an assert macro:

```
// P2935R0:                                    // This paper:

void f() {                                     void f() {
  int i = get_i();                               int i = get_i();
  [[ assert: i > 0 ]]                            assrt(i > 0);
  use(i);                                        use(i);
}                                              }
```

6.1.3 Assertion as an expression

The left-hand side of this code example is taken directly from [P2935R0]. Note that in attribute-like syntax, this would require a novel grammar that does not exist for attributes today and that we do not have implementation experience with. Attributes today need to appertain to another entity such as a declaration or a statement, and cannot be used on their own as an expression. At the same time, if an assertion is not an expression, it cannot be a full drop-in replacement for C `assert` as there would be places where a C `assert` is legal but a contract assert is not. With our proposal, using an assertion as an expression just works:

```
// P2935R0:                                    // This paper:

struct S2 {                                    struct S2 {
  int d_x;                                       int d_x;
  S2(int x)                                      S2(int x)
    : d_x( [[ assert : x > 0 ]], x )               : d_x( assrt(x > 0), x )
  {}                                             {}
};                                             };
```

6.1.4 Position inside more complex function declarations

The left-hand side of this code example is taken directly from [P2935R0]. In attribute-like syntax, preconditions and postconditions are placed in the same location where attributes that would appertain to the function's type would be located, i.e. before any trailing return type, virtual specifiers such as `override` and `final`, and a `requires` clause (see [P2935R0]). This has the benefit that we can re-use the existing standard attribute grammar. However, the resulting position is awkward and goes against the natural reading order of a function declaration; it also requires delayed parsing of postconditions (as the predicate may depend on the trailing return type).

By contrast, in our proposal, preconditions and postconditions are placed at the very end of a declaration, avoiding all of the above problems:

```
// P2935R0:

struct S1
{
  auto f() const & noexcept
    [[ pre : true ]] -> int;

  virtual void g()
    [[ pre : true ]] override = 0;

  template <typename T>
  void h()
    [[ pre : true ]] requires true;
};
```

```
// This paper:

struct S1
{
  auto f() const & noexcept -> int
    pre(true);

  virtual void g() override = 0
    pre(true);

  template <typename T>
  void h() requires true
    pre(true);
};
```

Placing preconditions and postconditions at the very end of a declaration is in principle also possible with attribute-like syntax, and this is discussed in [P2935R0]. But this is not covered by existing standard attribute-grammar, thus requiring a novel grammar that we do not have implementation experience with, throwing away the main advantage of attribute-like syntax.

6.1.5 Lambda with trailing return type

The same issue with the syntactic order also exists for lambdas, aggravated by the fact that a trailing return type is even more common here:

```
// P2935R0:

auto x = [] (int x)
  [[pre: x > 0]] -> int
{
  return x * x;
};
```

```
// This paper:

auto x = [] (int x) -> int
  pre(x > 0)
{
  return x * x;
};
```

6.2 Post-MVP functionality

6.2.1 Captures

In attribute-like syntax, captures look awkward as they involve square brackets inside double square brackets. Additionally, in attribute-like syntax they are ambiguous with square brackets for a structured binding (see 6.2.2). [P2935R0] suggests to only allow init-captures (but not default captures) and to surround structured bindings with an additional pair of parentheses for disambiguation. In our syntax, neither is necessary as the two features have different syntactic positions that arise naturally from our proposed grammar:

```
// P2935R0:

// no support for default captures


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

```
// This paper:

int min(int x, int y)
  post [x, y] (r: r <= x && r <= y);

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

6.2.2 Destructuring the return value

As mentioned above, in attribute-like syntax a structured binding is ambiguous with a capture unless we disallow default captures or require an extra pair of parens; with our syntax, it is not:

```
// P2935R0:

std::tuple<int, int, int> f()
  [[ post [x, y, z] : x != y && y != z ]];

// or, if needed to disambiguate from capture:

std::tuple<int, int, int> f()
  [[ post ([x, y, z]) : x != y && y != z ]];
```

```
// This paper:

std::tuple<int, int, int> f()
  post ([x, y, z] : x != y && y != z);
```

6.2.3 requires clause on the contract-checking annotation

In attribute-like syntax, the only possible syntactic position for a `requires` clause appertaining to the contract-checking annotation is the same as for all other extensions: immediately preceding the colon. In our syntax, we can choose a much more natural position: either at the end of the contract-checking annotation, or immediately after the `pre` or `post` keyword, depending on what SG21 prefers.

```
// P2935R0:

template <typename T>
void f(T x)
  [[pre requires(std::integral<T>) : x > 0]];
```

```
// This paper, option 1:

template <typename T>
void f(T x)
  pre (x > 0) requires std::integral<T>;

// This paper, option 2:

template <typename T>
void f(T x)
  pre requires std::integral<T> (x > 0);
```

6.2.4 requires clauses on both the contract and the function itself

In [P2935R0], if we wish to re-use the existing standard attribute syntax, the `requires` clause appertaining to the function itself comes after the contract-checking annotation, whereas in our syntax it comes before, making the declaration more readable.

Possible option: `requires` clause at the end

```
// P2935R0, option 1: attribute position

template <typename T>
void f(T x)
  [[pre requires(std::integral<T>): x > 0]]
  requires std::copyable<T>;

// P2935R0, option 2: final position

template <typename T>
void f(T x)
  requires std::copyable<T>
  [[pre requires(std::integral<T>): x > 0]];
```

```
// This paper, option 1:

template <typename T>
void f(T x)
  requires std::copyable<T>
  pre (x > 0) requires std::integral<T>;

// This paper, option 2:

template <typename T>
void f(T x)
  requires std::copyable<T>
  pre requires std::integral<T> (x > 0);
```

6.2.5 Attributes

In [P2935R0], a standard attribute appertaining to a contract-checking annotation (which is itself attribute-like) inevitably leads to nested double square brackets, which looks awkward, is hard to read, and requires a novel attribute grammar that we do not have implementation experience with (the existing grammar does not allow attributes appertaining to other attributes). On the other hand, in our proposal, a standard attribute appertaining to a contract-checking annotation looks as natural as a standard attribute appertaining to any other declaration:

```
// P2935R0:

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

```
// This paper:

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

Note that with our proposal, a standard attribute appertaining to a contract-checking annotation is never ambiguous with a standard attribute appertaining to the function itself, the function type, or the trailing return type (see section 5.3).

6.2.6 Labels

As discussed in section 5.4, with the syntax proposed here we can choose to place labels either at the beginning or at the end of a contract-checking annotation; the choice will depend on whether we consider these labels primary or secondary information. We can also choose between different delimiter tokens or special characters to separate them from the rest of the contract-checking annotation:

```
// P2935R0:

void search(range rg)
  [[ pre audit: is_sorted(rg) ]];
```

```
// This paper (two possible options shown):

void search(range rg)
  pre (is_sorted(rg)) [audit];

void search(range rg)
  pre <audit> (is_sorted(rg));
```

In any case, the different parts of a contract-checking annotation are much more clearly visually distinguishable in our syntax than in attribute-like syntax, which places everything except the predicate, including labels, into the same syntactic position immediately preceding the colon. The latter leads to a number of possible parsing ambiguities between labels and the name for the return value in attribute-like syntax that require awkward workarounds. No such issues arise in our syntax as the label, return value, and other parts of the contract all have their own unambiguous syntactic place:

```
// P2935R0:

int f(int x)
  [[ post foo: x > foo ]];
  // is foo label or return value?

int f(int x)
  [[ post (foo): x > foo]];
  // using extra parens; foo is return value

int f(int x)
  [[ post foo (bar): x > bar ]];
  // using extra parens; is bar return value
  // or label argument?
```

```
// This paper:

int f(int x)
  post <foo> (x > foo);
  // foo is label

int f(int x)
  post (foo: x > foo);
  // foo is return value

int f(int x)
  post <foo> (bar: x > bar);
  // bar is return value

int f(int x)
  post <foo(bar)> (x > bar);
  // bar is label argument
```

6.2.7 Procedural interfaces

To spell procedural interfaces, [P2935R0] resorts to an interface block delimited with double square brackets, which leads to nested double square brackets. On the other hand, with our proposal, we can adopt a much more natural-looking syntax that uses braces for the block of statements. This syntax is also much closer to the notation in Lisa Lippincott's original paper on procedural interfaces, [P0465R0]:

```
// P2935R0:

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

```
// This paper:

void f(int x)
interface {
  try {
    implementation;
  }
  catch (...) {
    assrt(false);
  }
};
```

7 Requirements from P2885

Our proposal satisfies all *must-have*, *important*, and *nice-to-have* requirements for a Contracts syntax from [P2885R2], except the requirement for implementation experience. If this proposal generates interest, we hope that someone will be able to help us with implementing this syntax in a C++ compiler to satisfy this requirement as well.

Below we list all these requirements and discuss how our syntax satisfies them.

7.1 Basic requirements

7.1.1 Aesthetics [basic.aesthetic]

We believe that our syntax is more elegant and readable than either attribute-like or closure-based syntax.

7.1.2 Brevity [basic.brief]

Our syntax uses the least amount of tokens and characters possible.

7.1.3 Teachability [basic.teach]

We believe that this syntax is easy to learn and teach, and more self-explanatory and intuitive than either attribute-like or closure-based syntax.

7.1.4 Consistency with existing practice [basic.practice]

We believe that this syntax is more consistent with existing practice than either attribute-like or closure-based syntax. Today, contracts facilities are implemented using macros, using the syntax `MACRO_NAME(predicate)`. We use the exact same basic syntax, also placing the predicate in parentheses. The only differences are that instead of a macro name, we use a contextual keyword, preconditions and postconditions are placed onto declarations instead of inside the function body, and the user can additionally name the return value in a postcondition, a feature that is not possible with macros.

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

We believe that this syntax is more consistent with the rest of the C++ language than either attribute-like or closure-based syntax. We do not make contract-checking annotations look like attributes, and we do not place predicates (which are expressions) between curly braces. In C++ today, expressions go between parentheses, while statements go between curly braces.

7.2 Compatibility requirements

7.2.1 No breaking changes [compat.break]

As long as we use a keyword other than `assert` for assertions (see discussion in section 4.4), our syntax does not break or alter the meaning of any existing C++ code.

7.2.2 No macros [compat.macro]

Our syntax does not require the use of macros or the preprocessor to be used effectively.

7.2.3 Parsability [compat.parse]

To our best knowledge the syntax we propose does not introduce any parsing ambiguities; see detailed discussion in section 4.3.

7.2.4 Implementation experience [compat.impl]

Unfortunately, we do not yet have any implementation experience with the syntax proposed here in a C++ compiler.

7.2.5 Backwards-compatibility [compat.back]

According to the SG21 electronic poll in [P2885R2], this is an *irrelevant* requirement.

7.2.6 Toolability [compat.tools]

In order for a C++ tool to implement meaningful functionality for Contracts, the tool needs to be able to not only recognise the contract-checking annotation itself, but also be capable of correctly parsing most parts of a C++ function declaration. We do not see any reason why this should be any more difficult with our proposed syntax than with attribute-like or closure-based syntax.

7.2.7 C compatibility [compat.c]

We do not know what the C committee intends to do and whether they are interested in standardising Contracts for C. We should not block progress on C++ Contracts while waiting for that decision to form. However, we do not see any reason why our proposed syntax should create more hurdles for adopting it in C than the other syntax proposals do.

7.3 Functional requirements

7.3.1 Predicate [func.pred]

Requirement satisfied.

7.3.2 Contract kind [func.kind]

Since with the natural syntax, `assrt(expr)` is an expression, `assrt` (or whichever other keyword we choose) must be a full keyword, not a contextual keyword. We can therefore not use the same keyword for the enum value in `std::contracts::contract_kind` corresponding to assertions. We can keep using the identifier `assert` for this purpose as in the current MVP, or choose any other identifier.

7.3.3 Position and name lookup [func.pos]

Requirement satisfied.

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

Requirement satisfied.

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

Requirement satisfied.

7.3.6 Multiple pre/postconditions [func.multi]

Requirement satisfied.

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

Requirement satisfied.

7.3.8 Return value [func.retval]

Requirement satisfied.

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

According to the SG21 electronic poll in [P2885R2], this is a *questionable* requirement. We decided not to satisfy it because we believe that letting the user define their own name for the return value is the better approach.

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

Requirement satisfied.

7.4 Future evolution requirements

7.4.1 Non-const non-reference parameters [future.params]

Requirement satisfied via captures.

7.4.2 Captures [future.captures]

The syntax proposed here can naturally be extended to support captures; see section 5.1 for discussion.

7.4.3 Structured binding return value [future.struct]

The syntax proposed here can naturally be extended to support destructuring the return value; see discussion in section 5.5 and comparison with attribute-like syntax in 6.2.2.

7.4.4 Contract reuse [future.reuse]

According to the SG21 electronic poll in [P2885R2], this is a *questionable* requirement. Joshua Berne suggested that this idea might be better addressed by introducing some kind of hygienic macro. We therefore decided not to consider this requirement further.

7.4.5 Meta-annotations [future.meta]

The syntax proposed here can naturally be extended to support labels and meta-annotations, offering the same syntactic freedom as attribute-like syntax; see section 5.4 for discussion.

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

There is nothing specific to the syntax proposed here that precludes this direction.

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

There is nothing specific to the syntax proposed here that precludes this direction.

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

There is nothing specific to the syntax proposed here that precludes this direction.

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

There is nothing specific to the syntax proposed here that precludes this direction.

7.4.10 Primary vs. secondary information [future.prim]

We believe that our syntax satisfies this requirement much better than attribute-like syntax.

7.4.11 Invariants [future.invar]

The syntax proposed here can be naturally extended to a *invariant* contract kind that can be declared at class scope, should SG21 decide to pursue this direction further. However, the syntactic space at class scope is somewhat crowded so we will most likely need to reserve a full keyword for this purpose.

7.4.12 Procedural interfaces [future.interface]

The syntax proposed here can be naturally extended to support procedural interfaces as proposed in [P0465R0]; see discussion in section 5.6 and comparison with attribute-like syntax in 6.2.7.

7.4.13 requires clauses [future.requires]

The syntax proposed here can be naturally extended to support *requires* clauses on individual contract-checking annotations; see discussion in section 5.2.

7.4.14 Abbreviated syntax on parameter declarations [future.abbrev]

According to the SG21 electronic poll in [P2885R2], this is the lowest-ranked *nice-to-have* requirement. We therefore did not dedicate any time considering this requirement in detail. However at first glance there does not seem to be anything specific to this proposal that precludes this direction.

7.4.15 General extensibility [future.general]

As we have shown above, the syntax proposed here can be naturally extended to a wide range of known ideas for future features. We therefore believe that it offers a high degree of extensibility also for future features not yet discussed.

Acknowledgements

We would like to thank Ville Voutilainen, Joshua Berne, Peter Brett, Gašper Ažman, and Andrzej Krzemieński for their helpful feedback on an earlier draft of this paper.

References

[P0465R0] Lisa Lippincott. Procedural function interfaces. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0465r0.pdf, 2016-10-16.

[P1680R0] Andrew Sutton and Jeff Chapman. Implementing Contracts in GCC. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1680r0.pdf, 2019-06-17.

[P2264R5] Peter Sommerlad. Make assert() macro user friendly for C and C++. https://wg21.link/p2264r5, 2023-09-13.

[P2461R1] Gašper Ažman, Caleb Sunstrum, and Bronek Kozicki. Closure-Based Syntax for Contracts. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2461r1.pdf, 2021-11-15.

[P2487R1] Andrzej Krzemieński. Is attribute-like syntax adequate for contract annotations? https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2487r1.html, 2023-06-11.

[P2695R1] Timur Doumler and John Spicer. A proposed plan for Contracts in C++. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2695r1.pdf, 2023-02-09.

[P2737R0] Andrew Tomazos. Proposal of Condition-centric Contracts Syntax. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2737r0.pdf, 2021-11-15.

[P2811R7] Joshua Berne. Contract-violation handlers. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2811r7.pdf, 2023-06-27.

[P2834R1] Joshua Berne and John Lakos. Semantic Stability Across Contract-Checking Build Modes. https://wg21.link/p2834r1, 2023-05-15.

[P2877R0] Joshua Berne and Tom Honermann. Contract Build Modes, Semantics, and Implementation Strategies. https://wg21.link/p2877r0, 2023-06-09.

[P2884R0] Alisdair Meredith. `assert` Should Be A Keyword In C++26. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2884r0.pdf, 2023-05-15.

[P2885R2] Timur Doumler, Gašper Ažman, Joshua Berne, Andrzej Krzemieński, Ville Voutilainen, and Tom Honermann. Requirements for a Contracts syntax. https://wg21.link/p2885r2, 2023-08-29.

[P2935R0] Joshua Berne. An Attribute-Like Syntax for Contracts. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2935r0.pdf, 2023-08-14.