A natural syntax for Contracts

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

Document #:	P2961R2
Date:	2023-11-08
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 visually separating the different syntactic parts of a contract check. The proposed syntax removes the problems of attribute-like and closure-based syntax while maintaining full compatibility and extensibility in line with the SG21 requirements for a Contracts syntax.

Contents

Revision history

Revision 2, 2023-11-05:

— Chose keyword `contract_assert` for assertions

— Added more rationale why using keyword `assert` for contract assertions is not viable

— Added section about implementation experience

— Updated discussion of viability for the C language with results from SG22 meeting

Revision 1, 2023-10-09:

— Made *pre-or-postcondition* precede *pure-specifier* for consistency with `= default` and `= delete`

— Clarified that naming the return value in a postcondition is optional

— Added discussion of keyword alternatives to `assert`; changed proposed keyword alternative from `assrt` to a shortlist with two candidates: `contract_assert` and `assertexpr`

— Added discussion of viability for the C language

— Added discussion of a possible post-MVP syntax for class invariants

— Added discussion of three new syntax requirements from [P2885R3] raised after the SG21 electronic poll on syntax requirements was conducted

Revision 0, 2023-09-17:

— Initial version.

1 The issues with attribute-like syntax

SG21 is currently working on standardising a first version of a Contracts facility — the so-called *Contracts MVP* (see [P2900R0]). According to our roadmap [P2695R1], the last remaining major design decision is the choice of syntax (see [P2885R3]). The longest-standing proposal for a Contracts syntax is the so-called attribute-like syntax ([P2935R3]). While attribute-like syntax has its strengths — longer implementation experience (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 check. This has been called a "heavy" syntax and is perceived as "ugly" by some users. The `[[...]]` syntax was designed for attributes so that they can appertain to many different kinds of entities (functions, classes, variables, types, statements, etc.) in many different contexts. Contracts, on the other hand, have a well-defined syntactic place and completely different design requirements. Further, attributes are intended to be used sparingly, while we expect that Contracts will be used widely, and in particular in many public-facing APIs, which calls for a more lightweight, natural-looking syntax. Finally, preconditions and postconditions are an integral part of a function declaration and therefore should have a syntax consistent with other parts of such a declaration, such as `requires`, `noexcept`, etc., all of which are introduced via a proper C++ keyword rather than delimiter tokens.

Technically speaking, the attribute-like syntax for contract checks does not fully conform to the grammar for attributes (due to the presence of the single colon which is not allowed for attributes), but it is syntactically very close, with a significant overlap:

```
[[ pre(foo(x)) ]]  // attribute
[[ pre: foo(x) ]]  // contract
[[ pre:: foo(x) ]] // attribute
```

As a result, we expect that with this syntax, most users will perceive contract checks as a kind of attribute. However, contracts are not attributes and do not behave as such. Attributes are ignorable (see [P2552R3]) while contracts are not (except in a program with no contract violations or if the contract semantic has been explicitly set to *ignore* by the user). A contract check may even create an entirely new code path out of a function — for assertions even out of the middle of a function body — e.g. via a throwing violation handler. This is functionality that standard attributes were never meant to allow. There are many other differences between contracts and standard attributes; see [P2487R1] for an analysis. It is particularly instructive to refer to the original paper [N2761] that introduced attributes to the C++ language, specifically section 7 "Guidance on when to use/reuse a keyword and when to use an attribute": Contracts satisfy almost none of the properties listed for a feature that should be an attribute, and almost all of the properties listed for a feature that should be a keyword[1].

Re-using the syntactic position of attributes for contract checks is problematic. Attributes always syntactically appertain to some other entity such as a type, a variable, or a statement. For

[1]The only listed property for a feature that should be a keyword that Contracts do *not* satisfy is interaction with the type system; however, [P2935R3] suggests a possible future extension where they do, namely that contract checks could be placed on function types, enabling e.g. `std::function` to only accept functions with a certain contract.

preconditions and postconditions, attribute-like syntax uses the position of attributes appertaining to a function type, which means they need to go before any trailing return type, before virtual specifiers such as `override` and `final`, and before the `requires` clause. This goes against the natural reading order of a function declaration and requires delayed parsing of postconditions. For assertions, attribute-like syntax uses the position of attributes appertaining to a null statement, which means they cannot be used freely as an expression and cannot serve as a complete drop-in replacement for the existing `assert` macro.

[P2935R3] also considers an alternative position for attribute-like preconditions and postconditions at the end of a function declaration, and for attribute-like assertions to be expressions rather than statements. But in both cases, we would need to place attribute-like entities at a novel syntactic position that is not currently supported for attributes, and that we do not have implementation experience with, thereby throwing away two major advantages of attribute-like syntax and creating several new problems (listed in [P2935R3] section 4.2).

The internal grammar of attribute-like syntax is also problematic. The basic syntactic structure is:

```
[[contract-kind *** :  predicate ]]
```

Everything else that is not the contract-kind (`pre`, `post`, `assert`) or the predicate — which includes the name for the return value and all possible post-MVP additions (labels, captures, structured bindings, `requires` clauses appertaining to the contract check, etc.) — has only one possible syntactic position: the one marked by `***` above. This means that the different parts of the contract check are not visually separated from each other, making more complex contracts hard to read and understand. It also gives rise to several syntactic ambiguities.

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:

```
int f(int x)
  [[ post foo: x > foo ]];    // Is foo a label or the return value?
```

One suggested workaround is to require extra parentheses around the return value:

```
int f(int x)
  [[ post (foo): x > foo]];    // foo is the return value
```

But that makes it syntactically ambiguous with an argument of the preceding label:

```
int f(int x)
  [[ post foo (bar): x > bar ]];    // is bar the return value or an argument of label foo?
```

As another example, there is an ambiguity between a capture and a structured binding for the return value (both desirable post-MVP extensions), as both use square brackets:

```
std::pair<int, int> f(int x, int y)
  [[ post [x, y]: x != y ]];    // is [x, y] a capture or a structured binding?
```

[P2935R3] suggests awkward workarounds: to allow only init-captures but not other types of captures, and to mandate an extra pair of parentheses around the structured binding.

Finally, we cannot have standard attributes appertaining to an attribute-like contract check 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 on alternative syntaxes

The first proposed alternative to attribute-like syntax for the Contracts MVP 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 check look very much like a lambda, even though contracts and lambdas are completely orthogonal language features.

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 checks. But it was the first proposal to use the basic syntactic structure that we also use in this paper:

contract-kind (*predicate*)

Along with this basic syntactic 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 syntactic 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 check and the syntax for captures. However, instead of using the problematic curly braces, we use parentheses around the predicate, following the basic syntactic 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 checks.

The author of the closure-based syntax has reviewed our proposal and decided to discontinue the closure-based syntax in favour of the natural syntax as it subsumes the ideas of the closure-based syntax and improves upon it. We are therefore left with attribute-like syntax and the natural syntax 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 checks without creating any confusion.

— A contract check 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 separate the different syntactic parts of a contract check. It should be possible to distinguish at a glance the contract kind, the predicate, the name for the return value, and (post MVP) the captures and labels. All of these should have their own distinct position in the syntax.

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$_{opt}$ *pre-or-post-condition-seq*$_{opt}$ *pure-specifier*$_{opt}$
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:
`contract_assert` *contract*

contract:
contract-condition *// can be expanded post-MVP, see section 6*

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

return-name:
identifier `:`

4.2 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.

There may be any number of preconditions or postconditions, in any order; preconditions and postconditions can be intermingled arbitrarily.

In a postcondition, a user-defined name for the return value of a function can be introduced via an identifier placed before the predicate, with a colon in between:

```
int f(int x)
  pre (x >= 0)
  post (r: r > x);    // r is the return value
```

Naming the return value in a postcondition is optional:

```
void clear()
  post (empty());    // OK
```

`pre` and `post` are contextual keywords. As we will see in section 5.1, this breaks no existing code and you can still use `pre` and `post` as an identifier everywhere this is well-formed today.

Preconditions and postconditions are positioned at the very end of a function declaration, immediately before the semicolon (or, if the declaration is a definition, the function body):

```
void f(int i) override final
  pre(i >= 0);
```

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

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

The only exception to this is the *pure-specifier* `= 0`, which comes *after* preconditions and postconditions to create consistency with `= default` and `= delete`, which are function bodies:

```
struct X {
  X()              pre(a) = default;
  X(const X&)      pre(b) = delete;
  virtual void f() pre(c) = 0;
};
```

Note that you can never execute a contract check on a deleted function, but you can do so on a defaulted function.

4.3 Assertions

Assertions use the same natural syntax of a keyword followed by a parenthesised predicate:

```
void f() {
  int i = get_i();
  contract_assert(i >= 0);
  use_i(i);
};
```

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, unlike `pre` and `post`, we need to claim a full keyword for the contract kind because assertions appear at block scope and could otherwise not be disambiguated from a function call[2]. This keyword cannot be `assert` due to the name clash with the existing `assert` macro from header `cassert` (see section 5.2).

After having carefully considered a large number of possible alternative keywords (see section 5.3 for a detailed discussion), we chose the alternative keyword `contract_assert`.

With this syntax, assertions are expressions, not statements. Consequently, 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(contract_assert(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 `assert` with `contract_assert`).

[2] Or at least, not without post-MVP syntactic additions such as captures that are not allowed on function calls.

5 Discussion

5.1 No parsing ambiguities with `pre` and `post`

It has been suggested that the natural syntax for preconditions and postconditions 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-expression*s 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 fine 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
```

5.2 The `assert` name clash

With the syntax we propose here, the best possible keyword for assertions would be `assert`. This is the keyword used in virtually all contracts-related proposals so far, including attribute-like syntax;

the keyword that most users would intuitively expect; and the keyword used in the vast majority of programming languages (Python, Rust, Scala, Swift, Kotlin, etc.) for this purpose. In short, the keyword `assert` is well-established practice and superior to any keyword that is not `assert`.

Unfortunately, it also creates a name clash with the existing `assert` macro from header `cassert`:

```
#include <cassert>

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

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. Let `assert` be the macro from header `cassert` when that header is included, and a contract assertion otherwise;
4. Use a keyword other than `assert` for contract assertions to avoid the name clash.

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

Option 2 looks very compelling. 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 is a significant problem with this approach: the behaviour of macro `assert` is is tied to whether `NDEBUG` is defined. To mimic the existing behaviour, we would have to change the default contract semantic[3] to always be *ignore* when `NDEBUG` is defined, and *enforce* otherwise. But this would not be enough: *ignore* would still parse and odr-use the predicate, even if it is not evaluated, whereas `assert` just macroes out all the tokens. Therefore, common programming patterns like the following would break if we switch `assert` from macro to contract:

```
#ifndef NDEBUG
  DebugThingy dbg;
#endif

void f() {
  assert(dbg.checkSomething());  // OK with macro, syntax error with contract if NDEBUG not defined
  // ...
}
```

[3]The status quo in the Contracts MVP is that the contract semantic of any contract check is implementation-defined; the recommended default contract semantic is *enforce* regardless of whether `NDEBUG` is defined, and this recommendation is not normative (see [P2877R0]).

On the one hand, this would lead to an unacceptably high amount of breakage in existing C++ code. On the other hand, changing contracts to mimic the existing behaviour of macro `assert` with `NDEBUG` defined is not possible because the choice of contract semantic (*enforce*, *observe* or *ignore*) is in general not known at compile time. It is therefore impossible to parse code differently, or take a different code path at compile time, depending on whether a contract is checked or ignored (see [P2877R0] and [P2834R1]).

The situation is complicated further by the fact that the meaning of the `assert` macro depends not only on whether `NDEBUG` is defined or not, but also on whether — and where — the `cassert` header is included or re-included after `NDEBUG` is defined, undefined, or redefined — all of which can happen at any arbitrary point in the code.

Option 3 has been suggested as another way out of this dilemma. Unlike Options 1 or 2, it would not break any existing code, as `assert` would not change its meaning if the `cassert` header is included. However, this approach is not viable either, for several reasons. First, it would render contract assertions unusable anywhere the `cassert` header is included, which would limit their use; second, it would require special rules as header `cassert` would effectively redefine a reserved keyword, which is undefined behaviour under the current rules; third, and most importantly, the same expression — `assert(x)` — would mean different things depending on whether the `cassert` header is included somewhere in the program, making it impossible to tell in general which one of the two possible meanings it has (consider, for example, writing code in a library header, not knowing whether someone might `#include <cassert>` before including that library header).

Considering all of the above, we consider Option 4 to be the only possible path forward. This direction was confirmed during the SG21 teleconference on 2023-09-21:

> **Poll:** If we adopt the syntax in P2961R0 for the Contracts MVP, we should use the keyword `assert` for contract assertions, replacing macro `assert`.
>
> | SF | F | N | A | SA |
> |---|---|---|---|---|
> | 0 | 2 | 3 | 6 | 5 |
>
> **Result:** Consensus against

Therefore, unless we settle for attribute-like syntax, we must find a viable alternative keyword that is not `assert`. Note that such a keyword may look weird and unfamiliar initially, but once it is standardised users tend to get used to it very quickly (see `co_yield`, `co_await`, etc.).

5.3 The search for a keyword alternative to `assert`

5.3.1 Design goals

A large number of alternative keywords have been suggested since the initial version of this paper was released. Before we pick one, we need to agree on a set of design goals that will guide our decision. Our design goals, ordered by priority, are:

1. The keyword should not break existing code.
2. The keyword should not be offensive or inappropriate.
3. The keyword should not be misleading or confusing in its meaning.
4. The keyword should fit in with the other parts of the Contracts MVP (`pre`, `post`, identifier for assertions in `std::contracts::contract_kind`).
5. The keyword should be sufficiently consistent with the rest of C++.

6. The keyword should be easy to remember.
7. The keyword should not be too long or cumbersome to type out.
8. The keyword should not be too difficult to read and pronounce.
9. The keyword should not look too much like a typo of a common English word.
10. The keyword should also work for the C language.

Note that these design goals are not absolute; they can be broken if there is a good engineering reason for it. Examples of existing keywords that go against some of these goals are: `requires`, claimed as a full keyword in C++20 that broke existing code; `alignas`, commonly treated as a typo for "aligns" by spell checkers; and `reinterpret_cast`, a long and ugly keyword. The latter is well-motivated: `reinterpret_cast` is an unsafe feature that should be used sparingly and therefore should be hard to type. Conversely, contracts exist to improve safety and correctness and their use should be encouraged. We therefore expect the new keyword to be typed frequently, so it should be easy to type.

5.3.2 Code search in existing C++ code

Having formulated our design goals, we can now evaluate how well our candidate keywords satisfy our first design goal: no code breakage. We ran all proposed candidate keywords through three different code search engines; the results are summarised in Table 1.

The first code search engine we consulted is codesearch.isocpp.org. It is based on the ACTCD19 dataset and was created specifically for studying existing practice of C++. The advantage of this engine is that it is token-based, that is, it finds usages of a given keyword as a user-declared identifier (which is exactly what we are after, as this is the usage that would break existing code) while excluding all other usages of the same keyword. The disadvantage of this engine is that the dataset is large, but not huge (877 MLoc — but at least we know the size of the dataset), predates C++20 (March 2019), and is somewhat skewed as it was taken from the source packages of the third party software package repository of a particular Linux distribution (Debian Sid).

The second code search engine we consulted is grep.app, which searches "over a half million public repositories on GitHub"; judging by the amount of matches, the total amount of available C++ code seems to be of the same order of magnitude as the ACTCD19 dataset. This engine is text-based, not token-based. We restricted the search to C++ and used the "Whole word" and "Code sensitive" options but could not figure out how to exclude usages of the keyword as a string literal and as a single word inside a comment. Due to the latter, there is a high number of false positives for words that are commonly used in English sentences, such as "must" and "check". However, the number of false positives is negligible for keywords that are not English words. In addition, this method also matches usages of the keyword as a header name in an include directive. This leads to a high number of false positives for `cassert` but is otherwise negligible.

The third code search engine we consulted is sourcegraph.com, which has the largest dataset of the three engines as it searches a much larger set of public GitHub repositories than grep.app (we do not know how large). This engine is also text-based but has a regex option. We restricted the search to C++ and used the regex `\b`*`keyword`*`\b` which only matches if the keyword is found as a whole word, case sensitive. Unfortunately, we hit a problem where the engine refuses to process more complicated regular expressions that could exclude more cases. Therefore, we have the same kinds of false positives as with grep.app.

We have not conducted any studies to evaluate usage of our candidate keywords in closed-source C++ code.

Keyword	ACTCD19	grep.app	Sourcegraph
`contractassert`	0	0	0
`mustexpr`	0	0	0
`dyn_assert`	0	0	0
`musthold`	0	0	0
`asrtexpr`	0	0	0
`stdassert`	0	0	1
`truexpr`	0	0	2
`co_assert`	0	0	7
`ccassert`	0	0	11
`contract_assert`	0	0	11
`std_assert`	0	0	11
`dyn_check`	0	0	18
`mustbetrue`	0	0	48
`assertexpr`	0	0	49
`assertion_check`	0	0	71
`cppassert`	0	1	103
`dynamic_assert`	0	20	631
`cca_assert`	0	0	0
`assrt`	2	5	1191
`runtime_assert`	5	68	1090
`_Assert`	8	43	3014
`xpct`	9	0	328
`assert_check`	20	13	667
`assert2`	40	40	3604
`cpp_assert`	59	31	2037
`affirm`	65	26	842
`__assert`	98	717	16256
`assess`	163	358	3814*
`insist`	174	422	18081*
`asrt`	256	28	1844
`cassert`	380	65348***	1024935*,***
`aver`	427	56	3310*
`posit`	617	74	9837*
`enforce`	1230	8347**	375985*,**
`audit`	1619	1268	161814*
`claim`	2800	21582**	784227*,**
`ass`	4145	602	14193*
`must`	4263	403923**	15899541*,**
`confirm`	4341	4716**	183121*,**
`assertion`	5433	16239**	903896*,**
`ensure`	8149	67176**	819402*,**
`chk`	11783	1329	211023*
`verify`	20727	33283**	1767686*,**
`expect`	43725	30710**	769879*,**
`check`	147315	228702**	5409830*,**

Table 1: Number of usages of candidate keywords as identifiers in existing open-source C++ code according to three different code search engines, sorted by matches in ACTCD19 (877 MLoC).
* The given number is a lower bound as the search engine hit a match limit.
** There is a significant number of false positives due to matches in code comments.
*** There is a significant number of false positives due to matches in include directives.

5.3.3 Choosing the best candidate

With the results in Table 1, we can narrow down our pool of candidates. While we certainly would not exclude an otherwise great keyword due to a handful of matches, we would like to avoid a keyword that has hundreds or even thousands of matches in existing open-source repositories. The quality of the matches also matters: matches in relatively unknown repositories have less significance than matches in highly popular, foundational C++ libraries such as Boost, Qt, Clang, or the Unreal Engine.

With that in mind, we can go through all candidate keywords from top to bottom and evaluate how well they satisfy our design goals formulated in 5.3.1.

The first keyword candidate, `contractassert`, is arguably harder to read than the spelling variant `contract_assert`.

The next candidate, `mustexpr` as well as other similar candidates not derived from the word "assert" such as `musthold`, `truexpr`, etc. are arguably not consistent with the existing community knowledge about Contracts. The terminology "assert" and the contract kind "assertion" are established terms of art and the keyword should show at least some connection to this terminology. Having a keyword not derived from "assert" also suffers from another problem: how should we name the corresponding enum value in `std::contracts::contract_kind`? Should we leave it at `assert`, thereby having a keyword completely inconsistent with the enum, or should we change it to match the keyword, thereby having an enum that is likewise inconsistent with established terminology? Note also that if we choose the latter, we cannot use the exact spelling of the keyword (a full keyword cannot be used as a name for an enum value), but need to come up with some alternative spelling of that keyword. None of these options seem particularly appealing.

Candidates such as `dyn_assert`, or any other keyword that contains variations of the words "dynamic" or "runtime", are misleading as a contract is not necessarily dynamic or checked at runtime (e.g. axiom contracts used for static analysis).

`asrtexpr`, `asrt`, and `assrt` are too hard to spell correctly as it is difficult to remember which letters need to be skipped. In addition, `asrt` and `assrt` might be interpreted as a typo and auto-corrected to "assert". Similar problems exist with `xpct` and `chk`.

`stdassert` and `std_assert` are somewhat misleading because the prefix `std` is usually used for features in the C++ Standard Library, whereas contracts are a core language feature.

`co_assert` is misleading as the prefix `co_` is used today exclusively for keywords that turn a function into a coroutine: `co_yield`, `co_await`, and `co_return`, but the keyword for contract assertions has nothing to do with coroutines.

`ccassert` is confusing as it is not clear at all with the `cc` stands for. We assume that it is intended as a clever portmanteau of "CCA" (contract-checking-annotation) and "assert", but this is a bit too clever for a standard C++ keyword. Besides, many users will not know what "CCA" stands for, which is why we also do not like `cca_assert`.

`contract_assert` and `assertexpr` do not seem to violate any of our design goals. Neither of them are perfect: `contract_assert` is a bit too long, but only two characters longer than the common keyword `static_assert` and therefore still seems viable; and `assertexpr` has the `expr` suffix which currently is only used for `constexpr`, a keyword used for declarations, which might be seen as a slight inconsistency. However, none of these properties are serious problems, and both keywords also have important advantages: `contract_assert` is extremely clear and descriptive, while `assertexpr` emphasises that a contract assertion is, in fact, an expression (and not a macro, a statement, or an attribute-like thing appertaining to a statement) and can be used everywhere an expression can be used. `assertexpr` is also nicely consistent with the corresponding grammar term *assert-expression* (see section 4.1). All things considered, we believe that `contract_assert` and `assertexpr` are

the two least bad options and therefore we would like to propose these two as our two preferred candidates.

Further down the list, there is `assertion_check`, which seems unnecessarily verbose; `_Assert` and `__assert`, which are reserved for implementations of the C++ Standard Library (and used in multiple such implementations); and various candidates that are very similar to the ones already rejected and suffer from the same problems.

At this point in the list, the number of existing usages as a user-defined identifier quickly starts to increase to an unacceptable level. We therefore did not include any further keyword candidates to the shortlist besides `contract_assert` and `assertexpr`.

The choice between these two candidates was made by poll at the SG21 telecon on 2023-10-26. The results were very clear:

> **Poll 1:** In case we choose to adopt the "natural syntax" for Contracts as proposed in P2961, we should use `assertexpr` as the keyword for assertions.
>
> | SF | F | N | A | SA |
> |---|---|---|---|---|
> | 1 | 3 | 8 | 7 | 0 |
>
> **Result:** No consensus
>
> **Poll 2:** In case we choose to adopt the "natural syntax" for Contracts as proposed in P2961, we should use `contract_assert` as the keyword for assertions.
>
> | SF | F | N | A | SA |
> |---|---|---|---|---|
> | 9 | 11 | 0 | 0 | 1 |
>
> **Result:** Consensus

The decisive argument was as follows. It was acknowledged that `assertexpr` and `contract_assert` are both equally good candidates if we look at C++ in isolation. However, if we look at the broader programming language community, where practically every other programming language uses `assert` for this purpose, `contract_assert` seems a lot more explainable than `assertexpr` because the `expr` suffix looks very unusual outside of the C++ context.

5.4 Implementation experience

A fully-functional experimental implementation of the natural syntax proposed here has been developed in GCC by Ville Voutilainen. It is currently publicly available on Compiler Explorer (https://godbolt.org) by selecting ""x86-64 gcc (contracts natural syntax)" from the compiler selection drop-down menu and passing the flags `-fcontracts -fcontracts-nonattr` to it. The first enables contracts in general, the second enables the natural syntax. A working test suite can be found at https://godbolt.org/z/qPKharqqv.

In addition, an implementation of the natural syntax proposed here has been developed in the cppfront compiler by Herb Sutter. Herb reported that this took him under an hour to convert his existing Contracts implementation from the attribute-like syntax to this paper's syntax, with some post-MVP extra features such as labels (aka "contract_group names"), including to update the uses of Contracts in the tests and the compiler itself to the new syntax. cppfront is also available on Compiler Explorer.

Neither of the two have reported significant challenges with implementing the natural syntax.

5.5 Viability for the C language

We consulted with SG22 (C/C++ Liaison Group) and requested their opinion on 1) whether there is interest in standardising Contracts for C in a way compatible with C++ and usable in code shared

between C and C++; 2) whether they would prefer the natural syntax presented in this paper or the attribute-like syntax [P2935R3] for this purpose; 3) whether there are C-specific technical concerns with either syntax. We received extensive feedback both on the SG22 reflector discussion and at a formal SG22 meeting that was convened for this purpose. In this section we summarise the feedback we received.

Regarding the first question, SG22 expressed interest in standardising Contracts for C in a way compatible with C++, and noted that it is possible they will end up standardising just a subset of the C++ facility.

Regarding the second question, SG22 expressed a preference for the natural syntax we propose in this paper over attribute-like syntax. The poll results were as follows:

> **Poll 1:** Would SG22 find it acceptable if SG21 adopts the attribute-like syntax as proposed in P2935 for Contracts in C++?

	SF	F	N	A	SA
WG14 members:	0	0	0	4	2
WG21 members:	0	2	2	6	0

> **Poll 2:** Would SG22 find it acceptable if SG21 adopts the natural syntax as proposed in P2961 for Contracts in C++?

	SF	F	N	A	SA
WG14 members:	3	1	2	0	0
WG21 members:	3	6	1	0	0

Regarding the third question, the only technical concern raised with the natural syntax is that C does not have conditional keywords and they would like to avoid introducing them. At the same time, `pre` and `post` are too frequent in existing C code to be claimed as full keywords. However, there is a straightforward workaround. We could define the keywords in C as `_Pre` and `_Post`, respectively, and then provide convenience macros `pre` and `post`, respectively, in a new C header `<stdcontracts.h>` that expand to the C keywords. The same technique has been successfully used for numerous earlier keywords such as `_Bool`, `_Static_assert`, `_Alignas`, `_Thread_local`, etc.

The SG22 discussion revealed that *backwards-compatibility*, i.e. the ability to add contract checks to existing code and to have this code still compile with an old compiler that does not know anything about Contracts, is a much more important concern in C than in C++[4]. The natural syntax provides such backwards-compatibility because it makes all contract checks syntactically compatible with function-like macros:

```
#if COMPILER_UNDERSTANDS_CONTRACTS
  #define pre(x) _Pre(x)
#else
  #define pre(x)
#endif
// same for post and assert
```

This property was stated as an important reason for supporting natural syntax over attribute-like syntax. However, the above technique will stop working if we introduce post-MVP extensions that add syntactic elements outside of the parentheses, such as captures and labels, which might also be relevant for C. Adding such elements inside the top-level parentheses would keep this technique working, especially if such elements themselves look like plain function invocations. Essentially, instead of

```
pre <labels> [captures] (predicate)
```

[4] According to the SG21 electronic poll on syntax requirements in [P2885R3], such backwards-compatibility was deemed *irrelevant* for Contracts in C++ by a majority of respondents.

we would have to do something along the lines of

```
pre (labels, captures, predicate)
```

It was further noted that many of the keywords starting with underscore + capital letter were later evolved to all lowercase keywords identical to C++ (`bool`, `static_assert`, etc.) which is desirable for Contracts too but is most likely not possible if we stick with the keywords `pre` and `post` given how common they are today as identifiers.

Regarding attribute-like syntax, an important difference between C and C++ is that the notion of ignorability of attributes is much stronger in C. In C++, attributes may be semantically optional but still need to be fully parsed, and syntax errors diagnosed (also known as the First Ignorability Rule for attributes, see [P2552R3]). On the other hand, in C, everything inside the `[[...]]` of an attribute may be treated as balanced token soup and skipped entirely. This is due to specific C compilers (outside of the three major ones) that choose to not implement attributes but wish to remain conforming. Using attribute-like syntax for Contracts in C suggests that contract checks may also be token-ignored in the same way; most people agreed that this is an undesirable property for contract checks.

Note that unlike the function-like macro approach, the allowance in C to token-ignore attributes does not actually provide backwards-compatibility of Contracts with older compilers in practice: all major C compilers treat attribute-like contract checks as syntax errors today because of the colon. Note further that regardless of which syntax we choose for Contracts, backwards-compatibility with older compilers can always be achieved by wrapping the contract checks themselves into macros rather than the keywords used for them; however, it was noted that from the C perspective it might be preferable to avoid this.

6 Post-MVP extensions

The natural syntax proposal 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 [P2885R3] or that have been brought up in discussion since the poll results were published.

6.1 Captures

The grammar of the natural syntax can be extended as follows to allow captures on contract checks:

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

assert-expression:
  `contract_assert` *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 natural syntax, a contract check 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 check 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 natural 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 natural syntax
```

Of course we could also choose to restrict ourselves to init-captures as [P2935R3] does.

6.2 Destructuring the return value

We can easily and naturally extend the natural syntax 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);
```

6.3 `requires`-clauses on contracts

The grammar of the natural syntax can be extended to allow a `requires` clause that appertains to an individual contract check. 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 check, 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 5.1. Note further that both options allow for `requires` clauses appertaining to individual contract checks 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);
```

6.4 Attributes appertaining to contracts

Although not covered in [P2885R3], it has been argued that any new syntactic construct in C++, including contract checks, 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 check.

Support for attributes appertaining to a contract check is easy to accommodate with natural 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 check:

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

In code:

```
void f(int 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:

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

In code:

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

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*.

6.5 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 the natural 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 check. 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 6.4), 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 check, 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 6.1), 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. The natural 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 check 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 the natural syntax proposal, we are not cutting off any other directions. The main difference to labels in attribute-like syntax is that in the natural 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, the natural syntax 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 natural syntax, labels are guaranteed to not clash with anything else because they are separated from all other parts of the contract check by their own delimiter tokens.

Note also that with natural syntax, we can support both standard attributes and non-attribute labels appertaining to the same contract check simultaneously, for example:

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

6.6 Class invariants

In principle, we can use the same natural syntax at class scope to express class invariants:

```
class sorted_vector {
  invariant(is_sorted());
  // members and member functions...
};
```

However, note that we cannot use a contextual keyword for the contract kind `invariant` as the grammar at class scope is already very crowded and there are multiple cases where this syntax is valid C++ code today:

```
struct invariant {
  invariant(int());    // Case 1: Constructor taking a function pointer
};

bool b = true;
struct X {
  invariant(b);    // Case 2: Member declarator with extra parentheses
};
```

We could attempt to fix the first case by introducing a new type of vexing parse for disambiguation, and the second case by banning the superfluous parentheses around a member declarator. A cleaner solution would be to claim a full keyword instead of a contextual one. However, the identifier `invariant` has 7379 matches in the ACTCD19 dataset which suggests that claiming it as a keyword would break too much existing code. If we want to introduce class invariants with this syntax, we would therefore have to get creative with the choice of keyword, similar to what we have to do for assertions (section 5.3). It has been suggested that we could even re-use the same keyword as for assertions, which would take on a different meaning at class scope to designate class invariants.

6.7 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, [P2885R3] and [P2935R3] mentioned the idea of integrating such procedural interfaces into a Contracts facility post-MVP.

With the natural syntax, 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 (...) {
    contract_assert(false);
  }
};
```

7 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 [P2935R3], side-by-side with the natural syntax proposed here, and discuss the different tradeoffs. Where appropriate, we mention different possible alternatives.

7.1 MVP functionality

7.1.1 Basic preconditions and postconditions

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

```
// P2935R3:

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

```
// This paper:

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

7.1.2 Assertion as a statement

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

```
// P2935R3:

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

```
// This paper:

void f() {
  int i = get_i();
  contract_assert(i > 0);
  use(i);
}
```

7.1.3 Assertion as an expression

The left-hand side of the code example below is taken directly from [P2935R3]. 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 natural syntax, using an assertion as an expression just works:

```
// P2935R3 (mentioned as possible extension):

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

```
// This paper:

struct S2 {
  int d_x;
  S2(int x)
    : d_x( contract_assert(x > 0), x )
  {}
};
```

7.1.4 Position inside more complex function declarations

The left-hand side of the code example below is taken directly from [P2935R3]. 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 [P2935R3]). 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 natural syntax, preconditions and postconditions are placed at the very end of a declaration, avoiding all of the above problems:

```
// P2935R3:

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 mentioned explicitly as an alternative in [P2935R3]. But this is not covered by existing standard attribute-grammar, requiring a novel grammar that we do not have implementation experience with, throwing away two major advantages of attribute-like syntax and creating several new problems (listed in [P2935R3] section 4.2).

7.1.5 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:

```
// P2935R3:

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

```
// This paper:

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

7.2 Post-MVP functionality

The side-by-side comparison in this chapter is speculative as none of the features in this section are currently being proposed for the C++ Standard.

7.2.1 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 7.2.2). [P2935R3] suggests to only allow init-captures and to surround structured bindings with an additional pair of parentheses for disambiguation. In natural syntax, neither is necessary as the two features have different syntactic positions that arise naturally from our proposed grammar:

```
// P2935R3:

// no support for non-init 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);
```

7.2.2 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 natural syntax, it is not:

// P2935R3:

```
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);
```

7.2.3 `requires` clause on the contract check

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

// P2935R3:

```
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);
```

7.2.4 `requires` clauses on both the contract and the function itself

In [P2935R3], if we wish to re-use the existing standard attribute syntax, the `requires` clause appertaining to the function itself comes after the contract check, whereas in natural syntax it comes before, making the declaration more readable.

// P2935R3, option 1: attribute position

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

// P2935R3, 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);
```

7.2.5 Attributes appertaining to contracts

In attribute-like syntax, a standard attribute appertaining to a contract check (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 (attributes appertaining to other attributes are not possible in existing C++ grammar). [P2935R3] clarifies that the only possible syntactic place for such attributes is — as you can perhaps guess by now — between the contract type and the colon.

On the other hand, in natural syntax, a standard attribute appertaining to a contract check looks as natural as a standard attribute appertaining to any other declaration:

```
// P2935R3:

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

```
// This paper, option 1:

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

// This paper, option 2:

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

Note that with the natural syntax, a standard attribute appertaining to a contract check is never ambiguous with a standard attribute appertaining to the function itself, the function type, or the trailing return type (see section 6.4).

7.2.6 Labels

As discussed in section 6.5, with the natural syntax we can choose to place labels either at the beginning or at the end of a contract check; 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 check:

```
// P2935R3:

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));
```

Attribute-like syntax suffers from parsing ambiguities between labels and the name for the return value that require awkward workarounds. In natural syntax, no such issues arise: the label, return value, and other parts of the contract all have their own unambiguous syntactic place:

```
// P2935R3:

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
```

```
// 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 ]];
  // using extra parens; is bar return value
  // or label argument?
```

```
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
```

7.2.7 Procedural interfaces

To spell procedural interfaces, [P2935R3] 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]:

```
// P2935R3:

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

```
// This paper:

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

8 Requirements from P2885

Our proposed natural syntax satisfies all relevant requirements for a Contracts syntax from [P2885R3]. Below we list all these requirements and discuss how the natural syntax satisfies them.

8.1 Basic requirements

8.1.1 Aesthetics [basic.aesthetic]

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

8.1.2 Brevity [basic.brief]

The natural syntax uses the least amount of tokens and characters possible.

8.1.3 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.

8.1.4 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.

8.1.5 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 checks 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.

8.2 Compatibility requirements

8.2.1 No breaking changes [compat.break]

As long as we use a keyword other than `assert` for assertions (see discussion in section 5.2), the natural syntax does not break or alter the meaning of any existing C++ code.

8.2.2 No macros [compat.macro]

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

8.2.3 Parsability [compat.parse]

To our best knowledge the syntax we propose does not introduce any parsing ambiguities; see detailed discussion in section 5.1. This has been confirmed by the experimental implementation of the natural syntax in GCC; see section 5.4.

8.2.4 Implementation experience [compat.impl]

Requirement satisfied; see section 5.4.

8.2.5 Backwards-compatibility [compat.back]

The natural syntax provides full backwards-compatibility with older compilers because it is compatible with wrapping contract checks into function-like macros, at least within the feature scope of the MVP (see discussion section 5.5). Attribute-like syntax does not provide such compatibility because even if attributes that are not recognised by an implementation are ignored, attribute-like contract checks are not attributes because of the colon, and are diagnosed as a syntax error by all major compilers today.

8.2.6 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 check 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.

8.2.7 C compatibility [compat.c]

We asked SG22 (C/C++ Liaison Group) to review this paper and confirm that the natural syntax is viable for C; the discussion is summarised in section 5.5.

8.3 Functional requirements

8.3.1 Predicate [func.pred]

Requirement satisfied.

8.3.2 Contract kind [func.kind]

With the natural syntax, `contract_assert` must be a full keyword, not a contextual keyword, otherwise `contract_assert(x)` would be ambiguous with a function call. We can therefore not use the same keyword for the enum value in `std::contracts::contract_kind` corresponding to assertions. The best option seems to be to simply keep using the identifier `assert` for this purpose.

8.3.3 Position and name lookup [func.pos]

Requirement satisfied.

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

Requirement satisfied.

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

Requirement satisfied.

8.3.6 Multiple pre/postconditions [func.multi]

Requirement satisfied.

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

Requirement satisfied.

8.3.8 Return value [func.retval]

Requirement satisfied.

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

According to the SG21 electronic poll in [P2885R3], 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.

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

Requirement satisfied.

8.4 Future evolution requirements

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

Requirement satisfied via captures.

8.4.2 Captures [future.captures]

The natural syntax can naturally be extended to support captures; see section 6.1 for discussion.

8.4.3 Structured binding return value [future.struct]

The natural syntax can naturally be extended to support destructuring the return value; see discussion in section 6.2 and comparison with attribute-like syntax in 7.2.2.

8.4.4 Contract reuse [future.reuse]

According to the SG21 electronic poll in [P2885R3], 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.

8.4.5 Meta-annotations [future.meta]

The natural syntax can naturally be extended to support labels and meta-annotations, offering the same syntactic freedom as attribute-like syntax; see section 6.5 for discussion.

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

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

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

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

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

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

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

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

8.4.10 Primary vs. secondary information [future.prim]

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

8.4.11 Invariants [future.invar]

The natural syntax can be extended to support invariants at class scope; see section 6.6 for discussion.

8.4.12 Procedural interfaces [future.interface]

The natural syntax can be naturally extended to support procedural interfaces as proposed in [P0465R0]; see discussion in section 6.7 and comparison with attribute-like syntax in 7.2.7.

8.4.13 `requires` clauses [future.requires]

The natural syntax can be naturally extended to support `requires` clauses on individual contract checks; see discussion in section 6.3.

8.4.14 Abbreviated syntax on parameter declarations [future.abbrev]

According to the SG21 electronic poll in [P2885R3], 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.

8.4.15 General extensibility [future.general]

As we have shown above, the natural syntax 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.

8.5 Additional requirements

8.5.1 Standard attributes appertaining to Contracts [app.attr]

Requirement satisfied; see section 6.4.

8.5.2 Forward-declaration of Contract labels [app.fwddecl]

There is nothing specific to the natural syntax that precludes this direction. For example, we could express forward-declared labels with `...` in the same syntactic place where we would normally place the labels themselves.

8.5.3 Contracts on function types [app.functype]

There is nothing specific to the natural syntax that precludes this direction. Although no proposals have been made yet demonstrating how this idea could work (regardless of syntax), we imagine it would be possible to have, for example, some kind of `using`-declaration naming a function type that can have contract checks attached to it. There might be some overlap with the idea of declaring reusable Contracts (see [future.reuse]).

Acknowledgements

We would like to thank Ville Voutilainen, Joshua Berne, Peter Brett, Gašper Ažman, and Andrzej Krzemieński for reviewing this paper and providing helpful feedback; Ran Regev, Jose Daniel García Sánchez, Barry Revzin, John Lakos, Aaron Ballman, Mark Gillard, and Alisdair Meredith for suggesting keyword alternatives to `assert`; Robert Seacord, Aaron Ballman, Niall Douglas, Martin Uecker, Jens Gustedt, JeanHeyd Meneide, and Joshua Cranmer for providing feedback on the viability of the natural syntax for the C language; Nina Ranns for convening an SG22 meeting for this purpose; Ville Voutilainen and Herb Sutter for their efforts with implementing the natural syntax in GCC and CppFront, respectively.

Bibliography

[N2761] Jens Maurer and Michael Wong. Towards support for attributes in C++ (Revision 6). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2761.pdf, 2008-09-18.

[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.

[P2552R3] Timur Doumler. On the ignorability of standard attributes. https://wg21.link/p2552r3, 2023-06-14.

[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.

[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.

[P2885R3] Timur Doumler, Gašper Ažman, Joshua Berne, Andrzej Krzemieński, Ville Voutilainen, and Tom Honermann. Requirements for a Contracts syntax. https://wg21.link/p2885r3, 2023-10-02.

[P2900R0] Joshua Berne, Timur Doumler, and Andrzej Krzemieński. Contracts for C++. https://wg21.link/p2900r0, 2023-09-27.

[P2935R3] Joshua Berne. An Attribute-Like Syntax for Contracts. https://wg21.link/p2935r3, 2023-10-04.