

# An Overview of Syntax Choices for Contracts

Document #: P3028R0  
Date: 2023-11-05  
Project: Programming Language C++  
Audience: SG21 (Contracts)  
Reply-to: Joshua Berne <[jberne4@bloomberg.net](mailto:jberne4@bloomberg.net)>  
Gašper Ažman <[gasper.azman@gmail.com](mailto:gasper.azman@gmail.com)>  
Rostislav Khlebnikov <[rkhlebnikov@bloomberg.net](mailto:rkhlebnikov@bloomberg.net)>  
Timur Doumler <[papers@timur.audio](mailto:papers@timur.audio)>

## Abstract

SG21 has two proposals for the syntax for Contracts ([[P2935R4](#)] and [[P2961R2](#)]) with multiple options. This paper attempts to identify the properties of each, so all participants can make informed decisions on the optimal syntax for Contracts in C++.

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Examples</b>	<b>2</b>
2.1	Basic Usage	3
2.2	Free Functions	4
2.3	Member Functions	7
2.4	Assertion Usage	9
2.5	Lambdas	10
2.6	Future Extensions	11
<b>3</b>	<b>Evaluation</b>	<b>15</b>
3.1	Requirements	15
3.2	Principles	20
<b>4</b>	<b>Conclusion</b>	<b>22</b>

# Revision History

Revision 0 (November 2023 WG21 meeting in Kona)

- Original version of the paper

## 1 Introduction

Following the plan laid out in [P2695R1], SG21 has been diligently reaching consensus on the fundamental behaviors to propose as part of the initial Contracts facility for C++ (see [P2900R1]), i.e., the Contracts MVP. Deciding on the syntax for expressing a contract-checking annotation (CCA) is a major task remaining before the discussion can move on to other groups and Contracts can be adopted into the draft C++ Standard.

SG21 has already gathered a number of requirements for a Contracts syntax that go above and beyond what a minimal product will require. Numerous participants hope to see this syntax choice be the foundation for a complete facility that meets the many varied needs of the C++ community. Those requirements have been gathered in [P2885R3].

Two papers have laid out options for this syntax, each of which makes a case for how they meet the various requirements captured in [P2885R3].

- [P2935R4] — “An Attribute-Like Syntax for Contracts” proposes the same syntax that was (for a time) in the draft C++20 Standard. In addition to faithfully reproducing that original syntax, three other alternatives are offered as options.
- [P2961R2] — “A natural syntax for Contracts” proposes a new syntax for CCAs that avoids the design space of attributes in favor of more closely resembling existing function-call-like C++ operators.

SG21’s plan (and hope) is to achieve consensus during the November 2023 in-person WG21 meeting in Kona. To aid in discussion, this paper will do two things.

1. In Section 2, we will show all proposed syntax alternatives with the same set of examples written in the corresponding syntax.
2. In Section 3, we will enumerate many of the decision-inspiring principles that might influence preference for one syntax over the other and provide what we hope and intend is an objective evaluation of how each individual syntax proposal satisfies the principles in question.

Our goal is that reading the alternatives and the principles together will assist participants in making the best choice for a syntax for Contracts for C++.

## 2 Examples

First, let’s look at the distinct alternative syntax proposals available to SG21 along with numerous examples to clarify the distinctions among the various choices. For each syntax, we present the same set of examples (even when the differences are minimal or irrelevant to that particular syntax) to facilitate comparing the various scenarios across the overall syntax choices. Predicates in the

examples are deliberately trivial (usually `true`) to avoid distracting from the facets of the syntax itself.

Any example (particularly of future potential extensions) marked with a comment noting its ambiguity will need either to have disambiguation rules applied or to be made ill-formed.

There are five distinct proposals for syntax to consider.

1. `Attrlike` — This syntax is proposed by [P2935R4] as Proposal 1-A, “C++20 Attribute-Like Syntax.” The C++20 attribute-like syntax uses `[[ ]]` to enclose the CCA in a form that resembles but does not grammatically qualify as an attribute. CCAs on functions are located where an attribute that would appertain to that function type would be located.
2. `Attrlike+Post` — This syntax is proposed by [P2935R4] as Proposal 1-B, “Post-Declaration Attribute-Like Syntax.” The post-declaration attribute-like syntax uses the same structure as the attribute-like syntax but places function CCAs at the end of function declarations instead of the attribute-related location specified by C++20 Contracts.
3. `Attrlike+Delim` — This syntax is proposed by [P2935R4] as Proposal 1-C, “Attribute-Like Syntax with Delimited Return-Value Specification.” This variation on the attribute-like syntax uses a colon to introduce the return-value identifier in postconditions.
4. `Attrlike+Post+Delim` — This syntax is proposed by [P2935R4] as Proposal 1-D, “Post-Declaration Attribute-Like Syntax with Delimited Return-Value Specification.” This variation of the attribute-like syntax makes both considered changes to the C++20 syntax, moving the location of function CCAs to the end of the declaration and adding the preceding colon to the return value identifier in postconditions.
5. `Natural` — This syntax is proposed in [P2961R2]. The natural syntax eschews the use of `[[ ]]` and instead introduces assertions with a new keyword and preconditions and postconditions with a context-sensitive keyword. Note that strong consensus was reached to use `contract_assert` as the keyword to introduce an assertion CCA and that it must be a *keyword*, not an *context-sensitive* keyword (an identifier with special meaning). Should no decision on the specific keyword to propose be reached soon, both decisions will be represented here as distinct syntax options.

## 2.1 Basic Usage

First, we will show the forms of the three basic *kinds* of CCAs on functions with no other particular language features in use.

### Precondition CCA

Attrlike	<code>void f() [[ pre : true ]];</code>
Attrlike+Post	
Attrlike+Delim	
Attrlike+Post+Delim	
Natural	<code>void f() pre( true );</code>

### Postcondition CCA

Attrlike Attrlike+Post Attrlike+Delim Attrlike+Post+Delim	<pre>void f() [[ post : true ]];</pre>
Natural	<pre>void f() post( true );</pre>

### Postcondition CCA with Return-Value Identifier

Attrlike Attrlike+Post	<pre>int f() [[ post r : true ]]; int f() [[ post (r) : true ]];</pre>
Attrlike+Delim Attrlike+Post+Delim	<pre>int f() [[ post : r : true ]];</pre>
Natural	<pre>int f() post( r : true );</pre>

### Assertion CCA

Attrlike Attrlike+Post Attrlike+Delim Attrlike+Post+Delim	<pre>void f() {     [[ assert : true ]]; }</pre>
Natural	<pre>void f() {     contract_assert( true ); }</pre>

## 2.2 Free Functions

CCAs on free functions with different annotations expose more of the differences between the syntax options available.

### Trailing Return Type

Attrlike Attrlike+Delim	<pre>auto f() [[ pre : true ]] -&gt; int;</pre>
Attrlike+Post Attrlike+Post+Delim	<pre>auto f() -&gt; int [[ pre : true ]];</pre>
Natural	<pre>auto f() -&gt; int pre( true );</pre>

### noexcept

Attrlike Attrlike+Post Attrlike+Delim Attrlike+Post+Delim	<pre>void f() noexcept [[ pre : true ]];</pre>
Natural	<pre>void f() noexcept pre( true );</pre>

## requires Clause

Attrlike Attrlike+Delim	<pre>template &lt;typename T&gt; int f() [[ pre : true ]] requires something&lt;T&gt;; template &lt;typename T&gt; auto f() [[ pre : true ]] -&gt; int requires something&lt;T&gt;;</pre>
Attrlike+Post Attrlike+Post+Delim	<pre>template &lt;typename T&gt; int f() requires something&lt;T&gt; [[ pre : true ]]; template &lt;typename T&gt; auto f() -&gt; int requires something&lt;T&gt; [[ pre : true ]];</pre>
Natural	<pre>template &lt;typename T&gt; int f() requires something&lt;T&gt; pre( true ); template &lt;typename T&gt; auto f() -&gt; int requires something&lt;T&gt; pre( true );</pre>

## Function Returning Pointer to Array

Attrlike Attrlike+Delim	<pre>int (*g(char i) [[ pre : true ]])[17];</pre>
Attrlike+Post Attrlike+Post+Delim	<pre>int (*g(char i))[17] [[ pre : true ]];</pre>
Natural	<pre>int (*g(char i))[17] pre( true );</pre>

## Deleted Function

Attrlike Attrlike+Post Attrlike+Delim Attrlike+Post+Delim	<pre>void f() [[ pre : true ]] = delete;</pre>
Natural	<pre>void f() pre( true ) = delete;</pre>

## Attributes

Attrlike Attrlike+Delim	<pre>void [[ return_type_attribute ]] f()   [[ pre : 1    true ]]   [[ function_type_attribute1 ]]   [[ pre : 2    true ]]   [[ function_type_attribute2 ]];</pre>
Attrlike+Post Attrlike+Post+Delim	<pre>void [[ return_type_attribute1 ]] f()   [[ function_type_attribute1 ]]   [[ function_type_attribute2 ]]   [[ pre : 1    true ]]   [[ pre : 2    true ]];</pre>
Natural	<pre>void [[ return_type_attribute1 ]] f()   [[ function_type_attribute1 ]]   [[ function_type_attribute2 ]] pre( 1    true ) pre( 2    true )</pre>

## Attributes with Trailing Return Type

Attrlike Attrlike+Delim	<pre>auto f()   [[ pre : 1    true ]]   [[ function_type_attribute1 ]]   [[ pre : 2    true ]]   [[ function_type_attribute2 ]]; -&gt; int   [[ return_type_attribute ]]</pre>
Attrlike+Post Attrlike+Post+Delim	<pre>auto f()   [[ function_type_attribute1 ]]   [[ function_type_attribute2 ]] -&gt; int   [[ return_type_attribute1 ]]   [[ pre : 1    true ]]   [[ pre : 2    true ]];</pre>
Natural	<pre>auto f()   [[ function_type_attribute1 ]]   [[ function_type_attribute2 ]] -&gt; int   [[ return_type_attribute1 ]] pre( 1    true ) pre( 2    true )</pre>

## Postcondition Needing Return Type To Parse

Attrlike	<pre>auto f() [[ post r : r &gt; 0 ]] -&gt; int; // return type needed before seen</pre>
Attrlike+Post	<pre>auto f() -&gt; int [[ post r : r &gt; 0 ]];</pre>
Attrlike+Delim	<pre>auto f() [[ post : r : r &gt; 0 ]] -&gt; int; // return type needed before seen</pre>
Attrlike+Post+Delim	<pre>auto f() -&gt; int [[ post : r : r &gt; 0 ]];</pre>
Natural	<pre>auto f() -&gt; int post( r : r &gt; 0 );</pre>

## 2.3 Member Functions

Member functions introduce a number of additional qualifiers that interact with the placement of CCAs.

### Trailing Return Type

Attrlike Attrlike+Delim	<pre>struct S {     auto f() [[ pre : true ]] -&gt; int; };</pre>
Attrlike+Post Attrlike+Post+Delim	<pre>struct S {     auto f() -&gt; int [[ pre : true ]]; };</pre>
Natural	<pre>struct S {     auto f() -&gt; int pre( true ); };</pre>

### const and Reference Qualifier

Attrlike Attrlike+Post Attrlike+Delim Attrlike+Post+Delim	<pre>struct S {     void f() &amp; [[ pre : true ]];     void g() const&amp;&amp; [[ pre : true ]]; };</pre>
Natural	<pre>struct S {     void f() &amp; pre( true );     void g() const&amp;&amp; pre( true ); }</pre>

## Virtual Specifiers

Attrlike Attrlike+Delim	<pre>struct S {     virtual void f() [[ pre : true ]] override final; };</pre>
Attrlike+Post Attrlike+Post+Delim	<pre>struct S {     virtual void f() override final [[ pre : true ]]; };</pre>
Natural	<pre>struct S {     virtual void f() override final pre( true ); };</pre>

## Pure Specifier

Attrlike Attrlike+Post Attrlike+Delim Attrlike+Post+Delim	<pre>struct S {     virtual void f() [[ pre : true ]] = 0; };</pre>
Natural	<pre>struct S {     virtual void f() pre( true ) = 0; };</pre>

## Defaulted Function

Attrlike Attrlike+Post Attrlike+Delim Attrlike+Post+Delim	<pre>struct S {     bool operator=(const S&amp;) [[ pre : true ]] = default; };</pre>
Natural	<pre>struct S {     bool operator=(const S&amp;) pre( true ) = default; };</pre>

## Deleted Function

Attrlike Attrlike+Post Attrlike+Delim Attrlike+Post+Delim	<pre>struct S {     void f() [[ pre : true ]] = delete; };</pre>
Natural	<pre>struct S {     void f() pre( true ) = delete; };</pre>



## Everything But The Kitchen Sink

Attrlike	<pre>struct S {   template &lt;typename T&gt;   auto f() const&amp;&amp;     [[ pre : true ]]     [[ post r : true ]]   -&gt; int requires something&lt;T&gt;   { return 17; } }</pre>
Attrlike+Post	<pre>struct S {   template &lt;typename T&gt;   auto f() const&amp;&amp;   -&gt; int requires something&lt;T&gt;     [[ pre : true ]]     [[ post r : true ]]   { return 17; } }</pre>
Attrlike+Delim	<pre>struct S {   template &lt;typename T&gt;   auto f() const&amp;&amp;     [[ pre : true ]]     [[ post : r : true ]]   -&gt; int requires something&lt;T&gt;   { return 17; } }</pre>
Attrlike+Post+Delim	<pre>struct S {   template &lt;typename T&gt;   auto f() const&amp;&amp;   -&gt; int requires something&lt;T&gt;     [[ pre : true ]]     [[ post : r : true ]]   { return 17; } }</pre>
Natural	<pre>struct S {   template &lt;typename T&gt;   auto f() const&amp;&amp;   -&gt; int requires something&lt;T&gt;     pre( true )     post( r : true )   { return 17; } }</pre>

### 2.4 Assertion Usage

Assertions that are usable as expressions with a void type benefit from some concrete examples.

## Statement

Attrlike Attrlike+Post Attrlike+Delim Attrlike+Post+Delim	<pre>void f() {     [[ assert : true ]]; }</pre>
Natural	<pre>void f() {     contract_assert( true ); }</pre>

## In Return Statement

Attrlike Attrlike+Post Attrlike+Delim Attrlike+Post+Delim	<pre>void f() {     return [[ assert : true ]]; } int g() {     return [[ assert : true ]], 17; }</pre>
Natural	<pre>void f() {     return contract_assert( true ); } int g() {     return contract_assert( true ), 17; }</pre>

## In Member Initializer

Attrlike Attrlike+Post Attrlike+Delim Attrlike+Post+Delim	<pre>struct S {     int d_x;     S : d_x( [[ assert : true ]], 17 ) {} };</pre>
Natural	<pre>struct S {     int d_x;     S : d_x( contract_assert( true ), 17 ) {} };</pre>

## 2.5 Lambdas

Lambdas have a terse syntax within which CCAs must find a home.

## Lambda Expression

Attrlike Attrlike+Post Attrlike+Delim Attrlike+Post+Delim	<pre>auto x = [] (int a) [[ pre : true ]] {}; auto y = []          [[ pre : true ]] {};</pre>
Natural	<pre>auto x = [] (int a) pre( true ) {}; auto y = []          pre( true ) {};</pre>

## Lambda Expression with Trailing Return Type

Attrlike Attrlike+Delim	<pre>auto x = [] (int a) [[ pre : true ]] -&gt; int { return 17; }; auto y = []          [[ pre : true ]] -&gt; int { return 17; };</pre>
Attrlike+Post Attrlike+Post+Delim	<pre>auto x = [] (int a) -&gt; int [[ pre : true ]] { return 17; }; auto y = []          -&gt; int [[ pre : true ]] { return 17; };</pre>
Natural	<pre>auto x = [] (int a) -&gt; int pre( true ) { return 17; }; auto y = []          -&gt; int pre( true ) { return 17; };</pre>

## Lambda Expression with requires Clause

Attrlike Attrlike+Delim	<pre>auto x = [](auto a) [[ pre : true ]]     requires something&lt;decltype(a)&gt; {};</pre>
Attrlike+Post Attrlike+Post+Delim	<pre>auto x = [](auto a) requires something&lt;decltype(a)&gt;     [[ pre : true ]] {};</pre>
Natural	<pre>auto x = [](auto a) requires something&lt;decltype(a)&gt;     pre( true ) {};</pre>

## 2.6 Future Extensions

The choice of syntax has a significant impact on potential future features that build upon the Contracts MVP.

## Attribute Appertaining to CCA

Attrlike Attrlike+Post	<pre> void f()   [[ pre  [[ clang::weeble ]]   : true ]]   [[ post [[ gcc::wibble ]]     : true ]]   [[ post [[ msvc::wobble ]]   r : true ]] {   [[ assert [[ icc::falldown ]] : true ]];   return x; } </pre>
Attrlike+Delim	<pre> void f()   [[ pre  [[ clang::weeble ]]   : true ]]   [[ post [[ gcc::wibble ]]     : true ]]   [[ post [[ msvc::wobble ]]   : r : true ]] {   [[ assert [[ icc::falldown ]] : true ]];   return x; } </pre>
Attrlike+Post+Delim	<pre> void f()   [[ pre  [[ clang::weeble ]]   : true ]]   [[ post [[ gcc::wibble ]]     : true ]]   [[ post [[ msvc::wobble ]]   : r : true ]] {   [[ assert [[ icc::falldown ]] : true ]];   return x; } </pre>
Natural	<pre> void f()   pre( true )      [[ clang::weeble ]]   post( true )     [[ gcc::wibble ]]   post( r : true ) [[ msvc::wobble ]] {   contract_assert( true ) [[ icc::falldown ]];   return x; } </pre>

## Structured Binding Return Value

Attrlike Attrlike+Post	<pre> std::tuple&lt;int,int&gt; f()   [[ post [a,b] : true ]]; </pre>
Attrlike+Delim Attrlike+Post+Delim	<pre> std::tuple&lt;int,int&gt; f()   [[ post : [a,b] : true ]]; </pre>
Natural	<pre> std::tuple&lt;int,int&gt; f()   post( [a,b] : true ); </pre>

### Capture Values for Postcondition

Attrlike Attrlike+Post	<pre>void f(int a)   [[ post [a] : true ]]    // ambiguous with structured binding   [[ post [a=a] : true ]]; // ok</pre>
Attrlike+Delim Attrlike+Post+Delim	<pre>void f(int a)   [[ post [a] : true ]]   [[ post [a=a] : true ]];</pre>
Natural	<pre>void f(int a)   post [a] ( true )   post [a=a] ( true );</pre>

### Structured Binding and Captures

Attrlike Attrlike+Post	<pre>std::tuple&lt;int,int&gt; f(int a)   [[ post [a] [x,y] : true ]]    // visually ambiguous   [[ post [a=a] [x,y] : true ]];</pre>
Attrlike+Delim Attrlike+Post+Delim	<pre>std::tuple&lt;int,int&gt; f(int a)   [[ post [a] : [x,y] : true ]]   [[ post [a=a] : [x,y] : true ]];</pre>
Natural	<pre>std::tuple&lt;int,int&gt; f(int a)   post [a] ( [x,y] : true )   post [a=a] ( [x,y] : true );</pre>

### Labels

Attrlike Attrlike+Post Attrlike+Delim Attrlike+Post+Delim	<pre>void f()   [[ pre audit : true ]]   [[ post audit : true ]]   {     [[ assert audit : true ]];   }</pre>
Natural	<pre>void f()   pre( true ) [audit]   post( true ) [audit]   {     contract_assert( true ) [audit];   }</pre>

## Label Conflicting with Return-Value Name

Attrlike Attrlike+Post	int f() [[ post audit (audit) : audit > 0 ]];
Attrlike+Delim Attrlike+Post+Delim	int f() [[ post audit : audit : audit > 0 ]];
Natural	int f() post( audit : audit > 0 ) [audit];

## Parameterized Labels with <>

Attrlike Attrlike+Post Attrlike+Delim Attrlike+Post+Delim	void f() [[ pre label<a> : true ]];
Natural	void f() pre( true ) [label<a>];

## Parameterized Labels with ()

Attrlike Attrlike+Post	void f() [[ pre label(a) : true ]] [[ post label(a) : true ]] // <i>ambiguous with return value name</i> { [[ assert label(a) : true ]]; }
Attrlike+Delim Attrlike+Post+Delim	void f() [[ pre label(a) : true ]] [[ post label(a) : true ]] { [[ assert label(a) : true ]]; }
Natural	void f() pre( true ) [label(a)] post( true ) [label(a)] { contract_assert( true ) [label(a)]; }

## requires Clause on CCA

Attrlike Attrlike+Post Attrlike+Delim Attrlike+Post+Delim	<pre>template &lt;typename T&gt;   [[ pre requires something&lt;T&gt; : true ]];</pre>
Natural	<pre>template &lt;typename T&gt;   pre requires something&lt;T&gt; ( true );</pre>





## Control Object Parameterizing CCA

Attrlike Attrlike+Post Attrlike+Delim Attrlike+Post+Delim	<pre>void f() [[ pre&lt;control()&gt; : true ]];</pre>
Natural	<pre>void f() pre&lt;control()&gt;( true );</pre>

## 3 Evaluation

Both syntax papers, [P2935R4] and [P2961R2], strive to explain how they satisfy the requirements for a Contracts syntax that were gathered in [P2885R3]. In addition, considering fundamental principles might shine additional light on the distinct qualities of each syntax proposal.

When we can objectively measure how well the syntax choices meet an individual concern, we will use the following notation.

-  — The syntax satisfies the concern completely.
-  — The syntax largely satisfies the concern with some seemingly acceptable caveats.
-  — The syntax satisfies part of the concern but overall fails to meet it.
-  — The syntax fails to meet the specified concern.

Note that we do not intend to suggest that whichever syntax has the most green or least red must be the syntax we choose. Each member of SG21 must determine which concerns they consider to be important for this decision and choose accordingly; this analysis is not intended as prescriptive and aims to aid in having a common understanding of the relationship between the syntax choices and the concerns.

### 3.1 Requirements

- [basic.aesthetic]<sup>1</sup> — The aesthetic appeal of a syntax decision is almost completely subjective, so we can provide no objective measure to help guide the syntax decision beyond the reader’s own conclusion about the appeal of any particular syntax decision.

<sup>1</sup>See [P2885R3], Section 4.1, “Aesthetics” [basic.aesthetic].

- [basic.brief]<sup>2</sup> — The number of characters and tokens needed for each syntax can be measured to give an idea of the weight of the overhead involved in any of the syntax choices.

Characters of Overhead					
	Attrlike	Attrlike +Post	Attrlike +Delim	Attrlike +Post +Delim	Natural
Precondition	8	8	8	8	5
Postcondition	9	9–10	9	9–10	6–7
Assertion	11	11	11	11	17

Tokens of Overhead					
	Attrlike	Attrlike +Post	Attrlike +Delim	Attrlike +Post +Delim	Natural
Precondition	6	6	6	6	3
Postcondition	6	6–7	6	6–7	3–4
Assertion	6	6	6	6	3

Beyond the tables above, the judgment as to whether these quantities of characters and tokens warrant considering either syntax brief or burdensome is subjective and left to the reader.

- [basic.teach]<sup>3</sup> —
- [basic.practice]<sup>4</sup> — Both proposals use the already commonplace choices of `pre` and `post` for precondition and postcondition CCAs, a decision that had strong consensus when originally proposed by [P1344R0].

On the other hand, the *natural* syntax is unable to use the identifier `assert`; this syntax needs the identifier it uses for assertion CCAs to be a language keyword, so it proposes the use of `contract_assert`. The use of `assert` as the keyword for assertions is common practice in many other languages, including, unfortunately, C. The inability with the *natural* syntax to use `assert` as the token in the Contracts syntax has raised concerns for many.

#### Concern 1: Use the Identifier `assert`

The syntax should use follow existing literature and industry practice by using the identifier `assert` for assertion CCAs.

Attrlike: ✓

Attrlike+Post: ✓

Attrlike+Delim: ✓

Attrlike+Post+Delim: ✓

Natural: ✗

<sup>2</sup>See [P2885R3], Section 4.2, “Brevity” [basic.brief].

<sup>3</sup>See [P2885R3], Section 4.3, “Teachability” [basic.teach].

<sup>4</sup>See [P2885R3], Section 4.4, “Consistency with existing practice” [basic.practice].



- `[basic.cpp]`<sup>5</sup> — Both syntax choices are using syntactic constructs that strongly resemble existing C++ constructs. The attribute-like syntax takes the form of C++ attributes in most readers’ opinions. The natural syntax looks like a function invocation or named operator.

The attribute-like syntax might not grammatically be an attribute but does follow the normal attribute appertainment rules (as obscure as they might be) and otherwise builds on an existing language construct. When placed in the postcondition location, those obscure rules are, however, ignored.

#### Concern 2: Consistent Use of Language Constructs

The syntax should make consistent use of existing language constructs.

Attrlike: ✓

Attrlike+Post: ✗

Attrlike+Delim: ✓

Attrlike+Post+Delim: ✗

Natural: ✓

- `[compat.break]`<sup>6</sup> — The attribute-like syntax proposals will have no impact on existing code. The natural syntax comes with a minor concern that an assertion might conflict with an existing function invocation or that the introduction of a new keyword for assertions might break existing code that uses that keyword as an identifier. Initial research has indicated that neither is a major concern.

#### Concern 3: No Breaking Changes

Do not alter the meaning or break existing C++ code.

Attrlike: ✓

Attrlike+Post: ✓

Attrlike+Delim: ✓

Attrlike+Post+Delim: ✓

Natural: ✓

- `[compat.macro]`<sup>7</sup> — None of the syntax choices depend on macros for their use.
- `[compat.parse]`<sup>8</sup> — All the syntax choices appear parseable.
- `[compat.impl]`<sup>9</sup> — None of the syntax proposals have had any implementation concerns raised. Publicly available patches to the major open-source compilers will likely be available by the time the SG21 MVP is ready to present to other study groups, regardless of which syntax decision is made.

GCC 13 with the `-fcontracts` option implements the C++20 attribute-like syntax excluding the optional `()`s around return value identifiers. The other attribute-like syntaxes are largely

<sup>5</sup>See [P2885R3], Section 4.5, “Consistency with the rest of the C++ language” [`basic.cpp`].

<sup>6</sup>See [P2885R3], Section 5.1, “No breaking changes” [`compat.break`].

<sup>7</sup>See [P2885R3], Section 5.2, “No macros” [`compat.macro`].

<sup>8</sup>See [P2885R3], Section 5.3, “Parsability” [`compat.parse`].

<sup>9</sup>See [P2885R3], Section 5.4, “Implementation experience” [`compat.impl`].

similar in terms of parsing the CCA itself, but we do not have implementation experience with those syntax choices in their entirety.

There is also a GCC branch developed by Ville Voutilainen available under the name “x86-64 gcc (contracts natural syntax)” at <http://godbolt.org> which implements the precondition and postcondition part of the natural syntax for contracts. It does not implement (as of this writing) the assertions as expressions, but neither does any other current implementation.

#### Concern 4: Implementation Experience

The specified syntax will have implementation experience in a modern C++ compiler.

Attrlike: ✓

Attrlike+Post: ✗

Attrlike+Delim: ✗

Attrlike+Post+Delim: ✗

Natural: ✓

- `[compat.back]`<sup>10</sup> — The attribute-like syntaxes all propose a possible alternative for allowing precondition and postcondition CCAs to fully resemble an attribute by making use of optional parentheses around everything except the *kind*. None, however, actually propose this choice, and deploying the choice would come with other issues due to GCC 13 compilers already treating attribute-like constructs with `pre` and `post` as ill-formed if they do not meet the C++20 Contracts syntax, even without a top-level colon in the tokens inside the attribute brackets.

The natural syntax could be elided from code built on older compilers through the use of function-like macros that expand to nothing yet would likely encounter problems with macros using the relatively common identifiers of `pre` and `post`.

Therefore, none of the syntax choices seem to directly offer a complete route for supporting backward compatibility.

- `[compat.tools]`<sup>11</sup> — None of the syntax choices are any harder or easier to parse than the C++ declarations to which they would be attached.
- `[compat.c]`<sup>12</sup> — All the syntax choices could be added to the C grammar with the same ease with which they have been added to the C++ grammar.
- `[func.pred]`<sup>13</sup> — All the syntax choices allow for arbitrary C++ expressions as the CCA predicate.
- `[func.kind]`<sup>14</sup> — The natural syntax introduces a keyword for assertion CCAs, so the corresponding enumerator used in `std::contracts::contract_kind` will need to be distinct; otherwise, no syntax has noteworthy issues with this requirement.

<sup>10</sup>See [P2885R3], Section 5.5, “Backwards-compatibility” [`compat.back`].

<sup>11</sup>See [P2885R3], Section 5.6, “Toolability” [`compat.tools`].

<sup>12</sup>See [P2885R3], Section 5.7, “C compatibility” [`compat.c`].

<sup>13</sup>See [P2885R3], Section 6.1, “Predicate” [`func.pred`].

<sup>14</sup>See [P2885R3], Section 6.2, “Contract kind” [`func.kind`].

- `[func.pos]`<sup>15</sup> — The attribute-like syntaxes need to be parsed as tokens first and then fully processed after consuming the entire function declaration. The post-declaration attribute-like syntaxes and the natural syntax should all be parseable in terms of tokens that have already been seen.

#### Concern 5: Immediately Parseable

Function CCAs should not require delayed parsing.

Attrlike: ✗

Attrlike+Post: ✓

Attrlike+Delim: ✗

Attrlike+Post+Delim: ✓

Natural: ✓

- `[func.pos.prepost]`<sup>16</sup> — All the syntax choices place function CCAs after the function parameters.
- `[func.pos.assert]`<sup>17</sup> — Assertion CCAs as expressions is now part of the Contracts MVP, and all syntax proposals are compatible with this choice.
- `[func.multi]`<sup>18</sup> and `[func.mix]`<sup>19</sup> — All the syntax choices allow for any number of preconditions and postconditions to be interleaved on a single function.
- `[func.retval]`<sup>20</sup> and `[func.retval.userdef]`<sup>21</sup> — All the syntax proposals provide a location for optionally introducing an identifier for the return value of the function to be used within a postcondition CCA.

The attribute-like syntax without the additional delimiter for return-value identifiers does not provide a location, for naming the return value, that is clearly distinct and that will not conflict with future proposals that build on top of the initial Contracts MVP. In the original C++20 Contracts syntax, the return value needed to be actively disambiguated from the three potential labels that were available for contracts — `default`, `audit`, and the other one.<sup>22</sup>

<sup>15</sup>See [P2885R3], Section 6.3, “Position and name lookup” [`func.pos`].

<sup>16</sup>See [P2885R3], Section 6.4, “Pre/postconditions after parameters” [`func.pos.prepost`].

<sup>17</sup>See [P2885R3], Section 6.5, “Assertions anywhere an expression can go” [`func.pos.assert`].

<sup>18</sup>See [P2885R3], Section 6.6, “Multiple pre/postconditions” [`func.multi`].

<sup>19</sup>See [P2885R3], Section 6.7, “Mixed order of pre/postconditions” [`func.mix`].

<sup>20</sup>See [P2885R3], Section 6.8, “Return value” [`func.retval`].

<sup>21</sup>See [P2885R3], Section 6.10, “User-defined name for return value” [`func.retval.userdef`].

<sup>22</sup>See [P1672R0].

### Concern 6: Unambiguous Return-Value Identifier

The syntax shall provide an unambiguous location for introducing an identifier for the return value.

Attrlike: ✓

Attrlike+Post: ✓

Attrlike+Delim: ✓

Attrlike+Post+Delim: ✓

Natural: ✓

- `[func.retval.predef]`<sup>23</sup> — None of the syntax proposals provide a predefined name for the return value.
- `[app.functype]`<sup>24</sup> — All the syntax choices that place function CCAs at the end of the function declaration will have problems and ambiguities attempting to allow for function CCAs to be placed on function types — in particular when a function uses a trailing return type that is itself a pointer to a function type.

Only the attribute-like syntaxes in their original position leverage existing attribute locations to allow for unambiguously appertaining to either the function type of the function being declared or to a function type embedded within that declaration.

### Concern 7: Enable CCAs on Function Types

The syntax should allow for adding precondition and postcondition CCAs to function types with the same syntax as that used for function declarations.

Attrlike: ✓

Attrlike+Post: ✗

Attrlike+Delim: ✓

Attrlike+Post+Delim: ✗

Natural: ✗

The remaining concerns in [P2885R3] relate to specific future proposals, and we will cover those issues under the broader principles enumerated below.

## 3.2 Principles

A number of principles more fundamental than the requirements specified in [P2885R3] underly the overall design of new language features and, in particular, of Contracts for C++. Each of these will shine additional light on the distinctions between the available syntax proposals.

- Ambiguities in the parsing of the C++ language are always a source of user confusion, compiler complexity, and potentially broken specifications.

The C++20 attribute-like syntax overloaded the space between the *kind* and the colon both to contain meta-information about the CCA and to introduce the return value identifier. When

<sup>23</sup>See [P2885R3], Section 6.9, “Predefined name for return value” `[func.retval.predef]`.

<sup>24</sup>See [P2885R3], Appendix A.3, “Contracts on function types” `[app.functype]`.

ambiguous, the label (one of the three defined in the language at the time) was chosen in lieu of treating the identifier as a name for the return value. This need for disambiguation sorely restricts the ability to add additional tokens with meaning, including user-defined ones, to that same syntactic space. The natural syntax and the option to introduce the return value with a colon as a delimiter in the attribute-like syntax both alleviate this problem.

#### Concern 8: Enable Unambiguous Extensions of the Syntax

The syntax should clearly identify optional components so it will be unambiguous with future extensions that will themselves inherently be optional components of a CCA.

Attrlike: ✘

Attrlike+Post: ✘

Attrlike+Delim: ✔

Attrlike+Post+Delim: ✔

Natural: ✔

- We are currently aware of plans for a wide variety of potential features to build upon the Contracts MVP, such as those introduced in [P2755R0]. Assuming that is the full scope of what may be added to CCA specifications in the future would be pure hubris. Therefore, to maximize future evolutionary paths, we must minimize restrictions that would be placed on extensions to the accepted CCA syntax.

Each of the proposed syntax choices provides ample room for evolution, but some are subtly more limited than others. Some of these issues do not have currently known upcoming proposals.

- All the attribute-like syntax choices will have issues allowing a feature that is placed on assertion CCAs immediately after the *kind* and beginning with an open parenthesis due to that construct appearing to the preprocessor as a use of the `assert()` function-like macro from `<cassert>`.
- The attribute-like syntax proposals that do not introduce a delimiter before the return-value identifier introduce ambiguities when attempting to add any syntax between the *kind* and the return value that has trailing optional parentheses.
- The natural syntax lacks enclosing delimiters that clearly identify what is and is not part of a CCA. Due to this lack of clarity, careful analysis is needed when extending the syntax with features that might border arbitrary other syntax that might come before or after a CCA.

Other aspects of the syntax proposals would require possibly challenging disambiguation rules for known upcoming proposals.

- Two upcoming features — capturing values and using structured bindings for the return value — both use a syntax that is identified by a single set of `[]`s. Without an additional delimiter for the identifier, these can be disambiguated by requiring captures to have an initializer and recognizing that structured bindings are always a sequence of identifiers with no other tokens, but that is a subtle and sometimes surprising distinction.

#### Concern 9: Maximize Evolutionary Flexibility

The syntax should maximize the flexibility available for future additions.

Attrlike: ✘

Attrlike+Post: ✘

Attrlike+Delim: ✔

Attrlike+Post+Delim: ✔

Natural: ✔

- Contracts are, inherently, not intended to be evaluated as part of the essential behavior of a program. To many, this intention is clearly conveyed through the use of the attribute-like syntax, harkening to existing C++ attributes' known property of being *ignorable* when considering the intended behavior of a piece of code. The attribute-like syntax conveys this facet of Contracts; the natural syntax does not.

#### Concern 10: Indicate Nonessential Nature of Contracts

The syntax should convey that CCA predicates are not part of the essential behavior of a program.

Attrlike: ✔

Attrlike+Post: ✔

Attrlike+Delim: ✔

Attrlike+Post+Delim: ✔

Natural: ✘

- Overall guidance on language evolution or design of particular features is rarely something for which WG21 seeks or achieves consensus. For attributes, however, significant discussion about their intent and design culminated in [P2552R3].

While the attribute-like syntax does not meet the grammar of being an attribute, it does *appear* to be an attribute to most readers and is thus in conflict with the consensus in EWG that was reached for [P2552R3].

#### Concern 11: Follow EWG Guidance on Attributes

The syntax should meet design guidelines adopted by EWG which might appear to a reader to apply to that syntax.

Attrlike: ✘

Attrlike+Post: ✘

Attrlike+Delim: ✘

Attrlike+Post+Delim: ✘

Natural: ✔

## 4 Conclusion

With each of the requirements and principles considered in Section 3, we identified similarities and differences between the various syntax proposals that SG21 must consider. Those considerations

where all syntax proposals have equivalent responses are fundamentally nonviable mechanisms to guide the choice of syntax.

In the table below, we therefore gather those data points where the syntax choices *do* differ along with a brief summary of each syntax’s result for that concern.

Concern	Attrlike	Attrlike +Post	Attrlike +Delim	Attrlike +Post +Delim	Natural
1 Use the Identifier <code>assert</code>	✓	✓	✓	✓	✗
2 Consistent Use of Language Constructs	✓	✗	✓	✗	✓
3 No Breaking Changes	✓	✓	✓	✓	✓
4 Implementation Experience	✓	✗	✗	✗	✓
5 Immediately Parseable	✗	✓	✗	✓	✓
6 Unambiguous Return-Value Identifier	✓	✓	✓	✓	✓
7 Enable CCAs on Function Types	✓	✗	✓	✗	✗
8 Enable Unambiguous Extensions of the Syntax	✗	✗	✓	✓	✓
9 Maximize Evolutionary Flexibility	✗	✗	✓	✓	✓
10 Indicates Nonessential Nature of Contracts	✓	✓	✓	✓	✗
11 Follow EWG Guidance on Attributes	✗	✗	✗	✗	✓

We sincerely hope that the above information helps participants in SG21 — and in WG21 as a whole — to select the optimal syntax for Contracts in C++.

## Acknowledgements

Thanks Tom Honermann, Lori Hughes, and Jens Maurer for careful reading and feedback on this paper.

## Bibliography

- [P1344R0] Nathan Myers, “Pre/Post vs. Enspects/Exsures”, 2019  
<http://wg21.link/P1344R0>
- [P1672R0] Joshua Berne, “"Axiom" is a False Friend”, 2019  
<http://wg21.link/P1672R0>
- [P2552R3] Timur Doumler, “On the ignorability of standard attributes”, 2023  
<http://wg21.link/P2552R3>
- [P2695R1] Timur Doumler and John Spicer, “A proposed plan for contracts in C++”, 2023  
<http://wg21.link/P2695R1>
- [P2755R0] Joshua Berne, Jake Fevold, and John Lakos, “A Bold Plan for a Complete Contracts Facility”, 2023  
<http://wg21.link/P2755R0>

- [P2885R3] Timur Doumler, Joshua Berne, Gašper Ažman, Andrzej Krzemieński, Ville Voutilainen, and Tom Honermann, “Requirements for a Contracts syntax”, 2023  
<http://wg21.link/P2885R3>
- [P2900R1] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, “Contracts for C++”, 2023  
<http://wg21.link/P2900R1>
- [P2935R4] Joshua Berne, “An Attribute-Like Syntax for Contracts”, 2023  
<http://wg21.link/P2935R4>
- [P2961R2] Timur Doumler and Jens Maurer, “A natural syntax for Contracts”, 2023  
<http://wg21.link/P2961R2>