Document Number: P2660R0
Date: 2022-10-14
Revises:
Reply to: Brian Bi
Bloomberg
bbi10@bloomberg.net

Working Draft, Extensions to C++ for Contracts

Note: this is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad formatting.

Contents

1 Scope [intro.scope]

1 This document describes extensions to the C++ Programming Language (Clause 2) that introduce *contracts*, which are programmatic descriptions of conditions that should hold at particular points of program execution. These extensions include new syntactic forms, as well as additions to the existing library facilities.

2 N4919, ISO/IEC DIS 14882, provides important context and specification for this document. This document is written as a set of changes against that specification. Instructions to modify or add paragraphs are written as explicit instructions. Modifications made directly to existing text from N4919 use underlining to represent added text and ~~strikethrough~~ to represent deleted text.

2 Normative references [intro.refs]

1 The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

(1.1) — N4919, *ISO/IEC DIS 14882*

3 Terms and definitions [intro.defs]

1 No terms and definitions are listed in this document. ISO and IEC maintain terminology databases for use in standardization at the following addresses:

(1.1) — ISO Online browsing platform: available at https://www.iso.org/obp

(1.2) — IEC Electropedia: available at http://www.electropedia.org

4 General principles [intro]

4.1 Implementation compliance [intro.compliance]

4.1.1 General [intro.compliance.general]

1 Conformance requirements for this document are those defined in N4919, 4.1, except that references to "this document" therein shall be taken as referring to the document that is the result of applying the editing instructions.

[*Note 1*: Conformance is defined in terms of the behavior of programs. — *end note*]

5 Lexical conventions [lex]

5.10 Identifiers [lex.name]

[1] In N4919 [lex.name], add the identifiers `assume`, `enforce`, `ignore`, and `observe` to the list of identifiers with special meaning in Table 5.

6 Expressions [expr]

6.7 Constant expressions [expr.const]

1 In N4919 [expr.const], apply these changes to paragraph 5:

— a non-constant library call (3.36); ~~or~~

— a `goto` statement (8.7.6)~~.~~; or

— a violation of a checked contract (9.4.2).

It is unspecified whether E is a core constant expression if E satisfies the constraints of a core constant expression, but evaluation of E would evaluate

— an operation that has undefined behavior as specified in Clause 17 through 33,

— an invocation of the `va_start` macro (17.15.2), ~~or~~

— a statement with an assumption (9.12.3) whose converted *conditional-expression*, if evaluated where the assumption appears, would not disqualify E from being a core constant expression and would not evaluate to `true`~~.~~

[*Note 4*: ... —*end note*]

, or

— a contract with the contract behavior `assume` (9.4.2.3) that would evaluate to `false`.

9 Declarations [dcl.dcl]

9.11 Attributes [dcl.attr]

9.11.1 Attribute syntax and semantics [dcl.attr.grammar]

1 In N4919 [dcl.attr.grammar], apply these changes to paragraph 1:

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

In N4919 [dcl.attr], after subsection [dcl.attr.depend], add a new subsection:

9.4.2 Contract attributes [dcl.attr.contract]

9.4.2.1 Syntax [dcl.attr.contract.syn]

1 Contract attributes are used to specify preconditions, postconditions, and assertions for functions.

contract-attribute-specifier:
`[ [ pre` *contract-behavior*$_{opt}$ `:` *conditional-expression* `] ]`
`[ [ post` *contract-behavior*$_{opt}$ *identifier*$_{opt}$ `:` *conditional-expression* `] ]`
`[ [ assert` *contract-behavior*$_{opt}$ `:` *conditional-expression* `] ]`

contract-behavior:
`assume`
`ignore`
`observe`
`enforce`

An ambiguity between a *contract-behavior* and an *identifier* is resolved in favor of *contract-behavior*.

2 A *contract-attribute-specifier* using `pre` is a *precondition*. It expresses a function's expectation on its arguments and/or the state of other objects using a predicate that is intended to hold upon entry into the function. The attribute may be applied to the function type of a function declaration.

3 A *contract-attribute-specifier* using `post` is a *postcondition*. It expresses a condition that a function should ensure for the return value and/or the state of objects using a predicate that is intended to hold upon exit from the function. The attribute may be applied to the function type of a function declaration. A postcondition may introduce an identifier to represent the glvalue result or the prvalue result object of the function. When the declared return type of a non-templated function contains a placeholder type, the optional *identifier* shall only be present in a definition.

[*Example 1*:

```
int f(char * c)
  [[post res: res > 0 && c != nullptr]];

int g(double * p)
  [[post res: res != 0 && p != nullptr && *p <= 0.0]];

auto h(int x)
  [[post res: true]];       // error: cannot name the return value
```

— *end example*]

4 A *contract-attribute-specifier* using `assert` is an *assertion*. It expresses a condition that is intended to be satisfied where it appears in a function body. The attribute may be applied to a null statement (8.3). An assertion is checked by evaluating its predicate as part of the evaluation of the null statement it applies to.

5 Preconditions, postconditions, and assertions are collectively called *contracts*. The *conditional-expression* in a contract is contextually converted to `bool` (7.3.1); the converted expression is called the *predicate* of the contract.

[*Note 1*: The predicate of a contract is potentially evaluated (6.3). — *end note*]

6 The only side effects of a predicate of a checked contract that are allowed in a *contract-attribute-specifier* are modifications of non-volatile objects whose lifetime began and ended within the evaluation of the predicate. An evaluation of a predicate that exits via an exception invokes the function `std::terminate` (14.6.2). The behavior of any other side effect is undefined.

[*Example 2*:

```
void push(int x, queue & q)
  [[pre enforce: !q.full()]]
  [[post enforce: !q.empty()]]
{
  /* ... */
  [[assert: q.is_valid()]];
  /* ... */
}

int min = -42;
constexpr int max = 42;

constexpr int g(int x)
  [[pre enforce: min <= x]]                // error
  [[pre enforce: x < max]]                 // OK
{
  /* ... */
  [[assert enforce: 2*x < max]];
  [[assert enforce: ++min > 0]];               // undefined behavior
  /* ... */
}
```

— *end example*]

9.4.2.2 Contract conditions [dcl.attr.contract.cond]

1 A *contract condition* is a precondition or a postcondition. The first declaration of a function shall specify all contract conditions (if any) of the function. Subsequent declarations shall either specify no contract conditions or the same list of contract conditions; no diagnostic is required if corresponding conditions will always evaluate to the same value. The list of contract conditions of a function shall be the same if the declarations of that function appear in different translation units; no diagnostic required. If a friend declaration is the first declaration of the function in a translation unit and has a contract condition, the declaration shall be a definition and shall be the only declaration of the function in the translation unit.

2 Two lists of contract conditions are the same if they consist of the same contract conditions in the same order. Two contract conditions are the same if their predicates are the same, and their *contract-behavior*s are the same or are both absent. Two predicates contained in *contract-attribute-specifier*s are the same if they would satisfy the one-definition rule (6.3) were they to appear in function definitions, except for renaming of parameters, return value identifiers (if any), and template parameters.

3 [*Note 1*: A function pointer cannot include contract conditions.

[*Example 1*:

```
typedef int (*fpt)() [[post r: r != 0]];      // error: contract condition not on a function declaration

int g(int x)
  [[pre: x >= 0]]
  [[post r: r > x]]
{
  return x+1;
}

int (*pf)(int) = g;                           // OK
int x = pf(5);                                // contract conditions of g are checked
```

— *end example*]

— *end note*]

4 The predicate of a contract condition has the same semantic restrictions as if it appeared as the first *expression-statement* in the body of the function it applies to, except that the return type of the function is

known in a contract condition appertaining to its definition, even if the return type contains a placeholder type.

5 A precondition is checked by evaluating its predicate immediately before starting evaluation of the function body.

[*Note 2*: The function body includes the *function-try-block* (Clause 14) and the *ctor-initializer* (11.9.3). — *end note*]

A postcondition is checked by evaluating its predicate immediately before returning control to the caller of the function.

[*Note 3*: The lifetime of local variables and temporaries has ended. Exiting via an exception or via `longjmp` (17.15.3) is not considered returning control to the caller of the function. — *end note*]

6 If a function has multiple preconditions, their evaluation (if any) will be performed in the order they appear lexically. If a function has multiple postconditions, their evaluation (if any) will be performed in the order they appear lexically.

[*Example 2*:

```
void f(int * p)
  [[pre: p != nullptr]]                    // #1
  [[post: *p == 1]]                        // #3
  [[pre: *p == 0]]                         // #2
{
  *p = 1;
}
```

— *end example*]

7 If a postcondition odr-uses (6.3) a non-reference parameter in its predicate and the function body makes direct or indirect modifications of the value of that parameter, the behavior is undefined.

[*Example 3*:

```
int f(int x)
  [[post enforce r: r == x]]
{
  return ++x;                    // undefined behavior
}

void g(int * p)
  [[post enforce: p != nullptr]]
{
  *p = 42;                       // OK, p is not modified
}

int h(int x)
  [[post enforce r: r == x]]
{
  potentially_modify(x);         // undefined behavior if x is modified
  return x;
}
```

— *end example*]

9.4.2.3 Checking contracts [dcl.attr.contract.check]

1 The contract behavior of a *contract-attribute-specifier* is one of `ignore`, `assume`, `enforce`, and `observe` as specified by the *contract-behavior*. If the *contract-behavior* is absent, the implicit contract behavior of the translation unit is used. The translation of a program consisting of translation units where the implicit contract behavior is not the same in all translation units is conditionally-supported with implementation-defined semantics. There should be no programmatic way of setting, modifying, or querying the implicit contract behavior of a translation unit.

2 [*Note 1*: Multiple contract conditions may be applied to a function type with the same or different *contract-behavior*s.

[*Example 1*:

```
int z;

bool is_prime(int k);
```

```
void f(int x)
  [[pre: x > 0]]
  [[pre enforce: is_prime(x)]]
  [[post assume: z > 10]]
{
  /* ... */
}
```

— *end example*]

— *end note*]

3 The predicate of a contract with *contract-behavior* `ignore` or `assume` is an unevaluated operand (7.2). The predicate of a contract without *contract-behavior* where the implicit contract behavior is `ignore` or `assume` is not evaluated.

[*Note 2*: The predicate is potentially evaluated (6.3). — *end note*]

If the predicate of a contract with the contract behavior `assume` would evaluate to `false`, the behavior is undefined.

4 The *violation handler* of a program is a function of type "`noexcept`$_{opt}$ function of (lvalue reference to `const std::experimental::contract_violation`) returning `void`". The violation handler is invoked when the predicate of a checked contract evaluates to `false` (called a *contract violation*). There should be no programmatic way of setting or modifying the violation handler. It is implementation-defined how the violation handler is established for a program and how the `std::experimental::contract_violation` (17.14.2) argument value is set, except as specified below. Implementations are encouraged to provide a default violation handler that outputs the contents of the `std::experimental::contract_violation` object and then returns normally. If a precondition is violated, the source location of the violation is implementation-defined.

[*Note 3*: Implementations are encouraged but not required to report the caller site. — *end note*]

If a postcondition is violated, the source location of the violation is the source location of the function definition. If an assertion is violated, the source location of the violation is the source location of the statement to which the assertion is applied.

5 If a violation handler exits by throwing an exception and a contract is violated on a call to a function with a non-throwing exception specification, then the behavior is as if the exception escaped the function body.

[*Note 4*: The function `std::terminate` is invoked (14.6.2). — *end note*]

[*Example 2*:

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

void g() {
  f(0);                                   // std::terminate() if violation handler throws
  /* ... */
}
```

— *end example*]

6 A checked contract is a contract with the contract behavior `observe` or `enforce`. If the contract behavior of a violated contract is `enforce` and execution of the violation handler does not exit via an exception, execution is terminated by invoking the function `std::terminate` (14.6.2).

[*Note 5*: A contract behavior of `enforce` is for detecting and ending a program as soon as a bug has been found. A contract behavior of `observe` provides the opportunity to instrument a contract into a pre-existing code base and fix errors before enforcing the check. — *end note*]

[*Example 3*:

```
void f(int x) [[pre observe: x > 0]];
void g(int x) [[pre enforce: x > 0]];
void h() {
  f(0);          // Violation handler invoked.
  g(0);          // Violation handler invoked then std::terminate() after handler.
  /* ... */
}
```

— *end example*]

11 Classes [class]

11.4 Class members [class.mem]

11.4.1 General [class.mem.general]

1 In N4919 [class.mem.general], apply these changes to paragraph 7:

A *complete-class context* of a class is a

— function body (9.5.1),

— default argument (9.3.4.7),

— default template argument (13.2),

— *noexcept-specifier* (14.5), ~~or~~

— default member initializer, or

— contract condition (9.4.2)

within the *member-specification* of the class.

[*Note 4*: ... — *end note*]

11.7 Derived classes [class.derived]

11.7.3 Virtual functions [class.virtual]

1 In N4919, add a paragraph at the end of [class.virtual]:

If an overriding function specifies contract conditions (9.4.2), it shall specify the same list of contract conditions as its overridden functions; no diagnostic is required if corresponding conditions will always evaluate to the same value. Otherwise, it is considered to have the list of contract conditions from one of its overridden functions; the names in the contract conditions are bound, and the semantic constraints are checked, at the point where the contract conditions appear. Given a virtual function `f` with a contract condition that odr-uses `*this` (6.3), the class of which `f` is a direct member shall be be an unambiguous and accessible base class of any class in which `f` is overridden. If a function overrides more than one function, all of the overridden functions shall have the same list of contract conditions (9.4.2); no diagnostic is required if corresponding conditions will always evaluate to the same value.

[*Example 11*:

```
struct A {
  virtual void g() [[pre: x == 0]];
  int x = 42;
};

int x = 42;
struct B {
  virtual void g() [[pre: x == 0]];
}

struct C : A, B {
  virtual void g();              // error: preconditions of overridden functions are not the same
};
```

— *end example*]

13 Templates [temp]

13.8 Name resolution [temp.res]

13.8.3 Dependent names [temp.dep]

13.8.3.3 Type-dependent expressions [temp.dep.expr]

1 In N4919 [temp.dep.expr], apply these changes to paragraph 3:

— a *conversion-function-id* that specifies a dependent type, ~~or~~

— the *identifier* introduced in a postcondition (9.4.2) to represent the result of a templated function whose declared return type contains a placeholder type, or

— dependent

14 Exception handling [except]

14.6 Special functions [except.special]

14.6.2 The `std::terminate` function [except.terminate]

1 In N4919 [except.terminate], apply these changes to paragraph 1:

— when a call to a `wait()`, `wait_until()`, or `wait_for()` function on a condition variable (33.7.4, 33.7.5) fails to meet a postcondition~~.~~, or

— when evaluation of the predicate of a contract (9.4.2) exits via an exception, or

— when the violation handler invoked for a failed contract condition check (9.4.2) on a `noexcept` function exits via an exception, or

— when the violation handler has completed after a failed contract check with the `enforce` contract behavior.

15 Preprocessing directives [cpp]

15.2 Conditional inclusion [cpp.cond]

1 In N4919 [cpp.cond], add the attributes `assert`, `post`, and `pre` to Table 22 with the value `202306L`.

15.11 Predefined macro names [cpp.predefined]

In N4919 [cpp.predefined], add the macro name `__cpp_contracts` to Table 23 with the value `202306L`.

16 Library introduction [library]

16.3 Method of description [description]

16.3.2 Structure of each clause [structure]

16.3.2.4 Detailed specifications [structure.specifications]

1 In N4919 [structure.specifications], apply these changes to paragraph 3, bullet 3:

> *Preconditions*: the conditions that the function assumes to hold whenever it is called; violation of any preconditions results in undefined behavior.
>
> [*Example 3*: An implementation might express such conditions via an attribute such as `[[pre]]` (9.4.2). However, some conditions might not lend themselves to expression via code. — *end example*]

16.4 Library-wide requirements [requirements]

16.4.2 Library contents and organization [organization]

16.4.2.3 Headers [headers]

1 In N4919 [headers], add the entry `<experimental/contract>` to Table 25.

16.4.5 Constraints on programs [constraints]

16.4.5.3 Reserved names [reserved.names]

16.4.5.3.3 Macro names [macro.names]

1 In N4919 [macro.names], apply these changes to paragraph 2:

> A translation unit shall not `#define` or `#undef` names lexically identical to keywords, to the identifiers listed in Table 5, ~~or~~ to the *attribute-token*s described in 9.11, or to the identifiers `assert`, `post`, or `pre`, except that the names `assert`, `likely`, and `unlikely` may be defined as function-like macros (15.6).

17 Language support library [support]

17.1 General [support.general]

1 In N4919 [support.general], apply these changes to paragraph 2:

> The following subclauses describe common type definitions used throughout the library, characteristics of the predefined types, functions supporting start and termination of a C++ program, support for dynamic memory management, support for dynamic type identification, support for exception processing, support for initializer lists, support for contract violation handling, and other runtime support, as summarized in Table 39.

Also add a row to Table 39, after subclause "Coroutines":

Table 1: Language support library summary [tab:support.summary]

Subclause		Header
17.14	Contract violation handling	`<experimental/contract>`

and increment the section number of all following rows.

17.3 Implementation properties [support.limits]

17.3.2 Header `<version>` synopsis [version.syn]

In N4919 [version.syn], apply these changes to paragraph 2:

```
#define __cpp_lib_containers_ranges                 202202L
  // also in <vector>, <list>, <forward_list>, <map>, <set>, <unordered_map>, <unordered_set>,
  // <deque>, <queue>, <stack>, <string>
#define __cpp_lib_contracts                         202306L // also in <experimental/contract>
#define __cpp_lib_coroutine                         201902L // also in <coroutine>
```

In N4919 [support], after section [support.coroutine], add a new section:

17.14 Contract violation handling [support.contract]

17.14.1 Header `<experimental/contract>` synopsis [contract.syn]

The header `<experimental/contract>` defines a type for reporting information about contract violations generated by the implementation.

```
namespace std {
  class contract_violation;
}
```

17.14.2 Class `contract_violation` [support.contract.cviol]

```
namespace std {
namespace experimental {
  class contract_violation {
  public:
    uint_least32_t line_number() const noexcept;
    string_view file_name() const noexcept;
    string_view function_name() const noexcept;
    string_view comment() const noexcept;
  };
}
}
```

1 The class `contract_violation` describes information about a contract violation generated by the implementation.

```
uint_least32_t line_number() const noexcept;
```

2 *Returns*: The source code location where the contract violation happened (9.4.2). If the location is unknown, an implementation may return `0`.

```
string_view file_name() const noexcept;
```

3 *Returns*: The source file name where the contract violation happened (9.4.2). If the file name is unknown, an implementation may return `string_view{}`.

```
string_view function_name() const noexcept;
```

4 *Returns*: The name of the function where the contract violation happened (9.4.2). If the function name is unknown, an implementation may return `string_view{}`.

```
string_view comment() const noexcept;
```

5 *Returns*: Implementation-defined text describing the predicate of the violated contract.

Annex A (informative) Grammar summary [gram]

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

contract-attribute-specifier:
    `[ [ pre` *contract-behavior*$_{opt}$ `:` *conditional-expression* `] ]`
    `[ [ post` *contract-behavior*$_{opt}$ *identifier*$_{opt}$ `:` *conditional-expression* `] ]`
    `[ [ assert` *contract-behavior*$_{opt}$ `:` *conditional-expression* `] ]`

contract-behavior:
    `assume`
    `ignore`
    `observe`
    `enforce`

Annex C (informative) Compatibility [diff]

C.1 Extensions to C++ for Contracts and N4919 [diff.cpp23]

C.1.1 General [diff.cpp23.general]

1 Subclause C.1 lists the differences between Extensions to C++ for Contracts and N4919 (ISO/IEC DIS 14882).

C.1.2 Clause 5: lexical conventions [diff.cpp23.lex]

1 **Affected subclause:** 5.10
Change: New identifiers with special meaning.
Rationale: New core language functionality that may be used by the C++ standard library.
Effect on original feature: The identifiers `assume`, `enforce`, `ignore`, and `observe` have new special meanings in this Technical Specification. Valid C++ 2023 code that `#define`s or `#undef`s a macro name lexically identical to one of these identifiers may be invalid in this extension of C++ (16.4.5.3.3).

C.1.3 Clause 16: library introduction [diff.cpp23.library]

1 **Affected subclause:** 16.4.2.3
Change: New header.
Rationale: New functionality.
Effect on original feature: The C++ header `<experimental/contract>` (17.14) is new. Valid C++ 2023 code that `#include`s a header with this name may be invalid in this extension of C++.

2 **Affected subclause:** 16.4.5.3.3
Change: New reserved names.
Rationale: New core language functionality that may be used by the C++ standard library.
Effect on original feature: The reserved names `post` and `pre` are new. Valid C++ 2023 code that `#define`s or `#undef`s a macro name lexically identical to one of these identifiers may be invalid in this extension of C++.

Cross references

Each clause and subclause label is listed below along with the corresponding clause or subclause number and page number, in alphabetical order by label.

Index

Index of grammar productions

The first bold page number for each entry is the page in the general text where the grammar production is defined. The second bold page number is the corresponding page in the Grammar summary (Annex A). Other page numbers refer to pages where the grammar production is mentioned in the general text.

Index of library headers

The bold page number for each entry refers to the page where the synopsis of the header is shown.

Index of library names

Index of implementation-defined behavior

The entries in this index are rough descriptions; exact specifications are at the indicated page in the general text.