Contracts for C++

Document #: P2900R11
Date: 2024-11-18
Project: Programming Language C++
Audience: EWG, LEWG

Reply-to: Joshua Berne <jberne4@bloomberg.net>
Timur Doumler <papers@timur.audio>
Andrzej Krzemieński <akrzemi1@gmail.com>
— with —
Gašper Ažman <gasper.azman@gmail.com>
Peter Bindels <dascandy@gmail.com>
Louis Dionne <ldionne@apple.com>
Tom Honermann <tom@honermann.net>
John Lakos <jlakos@bloomberg.net>
Lisa Lippincott <lisa.e.lippincott@gmail.com>
Jens Maurer <jens.maurer@gmx.net>
Ryan McDougall <mcdougall.ryan@gmail.com>
Jason Merrill <jason@redhat.com>
Oliver J. Rosten <oliver.rosten@gmail.com>
Iain Sandoe <iain@sandoe.co.uk>
Ville Voutilainen <ville.voutilainen@gmail.com>

Abstract

In this paper, we propose a Contracts facility for C++ that has been carefully considered by SG21 with a high bar set for level of consensus. The proposal includes syntax for specifying three kinds of contract assertions: precondition assertions, postcondition assertions, and assertion statements. In addition, we specify four evaluation semantics for these assertions — one non-checking semantic, *ignore*, and three checking semantics, *observe*, *enforce*, and *quick_enforce* — as well as the ability to specify a user-defined handler for contract violations. The features proposed in this paper allow C++ users to leverage contract assertions in their ecosystems in numerous ways.

Contents

Revision History

Revision 11 (Pre-Wrocław November 2024 Meeting SG21 Consensus)

- Added requirement that if a nonreference parameter is odr-used in `post` on a virtual function, the corresponding parameter on all declarations of all overriding functions must also be declared `const`
- Added requirement that a nonreference parameter that is odr-used in `post` must have an explicit `const` qualifier even if its type is dependent
- Added member functions `is_terminating` and `evaluation_exception` to class `contract_violation`
- Added missing values for *ignore* and *quick_enforce* to enum `evaluation_semantic`
- Added a new subsection, "Mixed Mode"
- Changed term *enforcing semantic* to *terminating semantic*
- Clarified intended behavior for objects passed and returned via registers
- Clarified that in a nontemplated function, the result name is late parsed, not dependent
- Clarified that trivial copies of the result object are in sequence with the evaluation of postcondition assertions
- Clarified the semantics of lambda expressions inside the redeclaration of a function-contract-assertion sequence
- Various minor clarifications and additional code examples

Revision 10 (October 2024 Mailing)

- Added support for `pre` and `post` on coroutines

Revision 9

- Made implicit `const` in contract predicates apply to all variables, rather than just those with automatic storage duration
- Added clarifications to the "Design Principles" section acknowledging the existence of programs that can detect the presence and semantics of a contract assertion
- Wording improvement: Added contract-assertion scopes to the list of scopes where lambdas may appear

Revision 8 (Post–St. Louis June 2024 Meeting EWG Feedback)

- Added support for `pre` and `post` on virtual functions
- Made contract assertions observable checkpoints
- Added discussion of pointers to member functions
- Added a clarification regarding constant evaluation of contract assertions

- Added discussion of concurrent invocation of contract-violation handlers
- Clarified that contract predicates are full expressions
- Numerous language and grammar edits

Revision 7 (Post-Tokyo March 2024 Meeting EWG and LEWG Feedback)

- Added the *quick_enforce* evaluation semantic
- Changed the *enforce* evaluation semantic from calling `std::abort()` back to terminating in an implementation-defined fashion, making it consistent with *quick_enforce*
- Added an implementation-defined upper bound to the number of repetitions of a contract-assertion evaluation; added a recommendation to provide an option to perform a specified number of repetitions, with no repetitions being the default
- Made it ill-formed for the predicate of a postcondition assertion to odr-use an array parameter
- Made it ill-formed to use `va_start` in a contract predicate
- Made underlying type of proposed enums unspecified rather than `int`
- Renamed enum `contract_kind` to `assertion_kind`
- Renamed enum `contract_semantic` to `evaluation_semantic`
- Renamed *checked* and *unchecked* evaluation semantics to *checking* and *non-checking*, respectively
- Added a new subsection, "Function Type Aliases"
- Added a new subsection, "Constructors and Destructors"
- Added a new subsection, "Differences Between Contract Assertions and the `assert` Macro"
- Expanded the "Design Principles" section
- Various minor clarifications and additional code examples
- Numerous language and grammatical edits

Revision 6 (Pre-Tokyo March 2024 Meeting, Forwarded to EWG and LEWG for Design Review)

- Allowed attributes in general and `[[maybe_unused]]` specifically to appertain to the result name
- Made ill-formed an *await-expression* or *yield-expression* appearing in the predicate of `contract_assert`
- Clarified that evaluating a predicate with side effects during constant evaluation might lead to an odr violation
- Expanded the "Design Principles" section
- Various minor clarifications and additional code examples

Revision 5 (February 2024 Mailing)

- Added proposed wording
- Made `contract_assert` a statement rather than an expression
- Made ill-formed `pre` and `post` on virtual functions
- Removed function `contract_violation::will_continue()`
- Removed enum value `detection_mode::evaluation_undefined_behavior`
- Introduced the distinct terms *function contract specifier* for the syntactic construct and *function contract assertion* for the entity it introduces
- Added rules for equivalence of two *function-contract-specifier-seq*s
- Allowed repeating the *function-contract-specifier-seq* on redeclarations
- Renamed *return name* to *result name*
- Added syntactic location for attributes appertaining to contract assertions
- Added a new subsection, “Function-Template Specializations”
- Added a new subsection, “Friend Declarations Inside Templates”
- Expanded the “Design Principles” section
- Various minor clarifications

Revision 4 (January 2024 Mailing)

- Added rules for constant evaluation of contract assertions
- Made header `<contracts>` freestanding
- Changed *enforce* from terminating in an implementation-defined fashion to calling `std::abort()`
- Clarified that side effects in checked predicates may be elided only if the evaluation returns normally
- Clarified that the memory for a `contract_violation` object is not allocated via `operator new` (similar to the memory for exception objects)
- Added a new subsection, “Design Principles”

Revision 3 (December 2023 Mailing)

- Made ill-formed `pre` and `post` on deleted functions
- Allowed `pre` and `post` on lambdas
- Added rule that contract assertions cannot trigger implicit lambda captures
- Added function `std::contracts::invoke_default_contract_violation_handler`
- Made local entities inside contract predicates implicitly `const`

- Clarified the semantics of the return name in `post`
- Added a new section, “Overview”
- Added a new subsection, “Recursive Contract Violations”

Revision 2 (Post November 2023 Kona Meeting SG21 Feedback)

- Adopted the “natural” syntax
- Made ill-formed `pre` and `post` on defaulted functions and on coroutines

Revision 1 (October 2023 Mailing)

- Added new subsections, “Contract Semantics” and “Throwing Violation Handlers”
- Added a synopsis of header `<contracts>`
- Various minor additions and clarifications

Revision 0 (Post-Varna June 2023 Meeting SG21 Feedback)

- Original version of the paper gathering the post-Varna SG21 consensus for the contents of the Contracts facility

1 Introduction

Behind the attempts to add a Contracts facility to C++ is a long and storied history. The next step for us, collectively, in that journey is for SG21 to produce a Contracts MVP (minimum viable product) as part of the plan set forth in [P2695R0]: This paper is that MVP.

This paper has three primary sections. "Overview" introduces the general concepts and the terminology that will be used throughout this paper and provides a view of the scope of the proposal. "Proposed Design" carefully, clearly, and precisely describes the design of the proposed Contracts facility. "Proposed Wording" contains the formal wording changes needed (relative to the current draft C++ Standard) to add Contracts to the C++ language. This paper is intended to contain enough information to clarify exactly what we intend for Contracts to do as well as the wording needed to match that information.

This paper is explicitly *not* a collection of motivations for using Contracts, instructions on how to use the facility, the history of how this design came to be, or an enumeration of alternative designs that have been considered. To avoid an excessively long paper, we have extracted all this information into a forthcoming companion paper, [P2899R0], "Contracts for C++ — Rationale." [P2899R0] will contain, for each subsection of the design section of this paper, a history — as complete as possible — for the decisions in that section. That paper will also describe our current implementation and deployment experience with the proposed Contracts facility and, importantly, contain citations to the *many* papers written by members of WG21 and SG21 that have contributed to making this proposal a complete thought.

2 Overview

We will begin by providing the general concepts and the terminology that will be used throughout this paper and, we hope, in many of the other papers discussing these topics. Then we will discuss the basic features and scope of the proposed Contracts facility.

For a summary of motivating use cases for Contracts and the history of Contracts in C++ and other programming languages, see [P2899R0], Section 2.

2.1 What Are Contracts?

A *contract* is a formal interface specification for a software component such as a function or a class. It is a set of conditions that expresses expectations about how the component interoperates with other components in a correct program and in accordance with a conceptual metaphor with the conditions and obligations of legal contracts.

A *contract violation* occurs when a condition that is part of a contract does not hold when the relevant program code is executed. A contract violation usually constitutes a bug in the code, which distinguishes it from an error. Errors are often recoverable at run time, whereas contract violations can usually be addressed only by fixing the bug in the code.

A *correct program* is one that will not violate any contracts under any circumstances. In general, we focus more on *correct behavior* where a particular program execution — with a particular set of inputs — violates no contracts applicable to any part of that evaluation.

Contracts are often specified in human language in software documentation, e.g., in the form of comments within the code or in a separate specification document; a contract specified this way is called a *plain-language contract*. For example, the C++ Standard defines plain-language contracts — preconditions, postconditions, and effects clauses — for the functions in the C++ Standard Library.

The various provisions of a plain-language contract fall into a variety of different categories.

- A *precondition* is a part of a function contract, and the responsibility for satisfying the precondition rests with the caller of the function. Typically, preconditions are requirements placed on the arguments passed to a function and/or on the global state of the program upon entry into a function.
- A *postcondition* is a part of a function contract, and the responsibility for satisfying the postcondition lies with the callee, i.e., the implementer of the function itself. Postconditions are generally conditions that will hold true regarding the return value of the function or the state of objects modified by the function when it completes execution normally.
- An *invariant* is a condition on the state of an object or a set of objects that is maintained over a certain amount of time. A *class invariant* is one kind of invariant and is a condition that a class type maintains throughout the lifetime of an object of that type between calls to its public member functions. Other invariants are often expected to hold on the entry or exit of functions or at specific points in control flow, and they are thus amenable to checking, using the same facilities that check preconditions and postconditions.

Some provisions of a plain-language contract can often be checked via an algorithm — one that either verifies compliance with that provision of the contract or identifies a violation of the contract. A *contract assertion* is a syntactic construct that specifies such an algorithm in code. When used correctly, contract assertions can significantly improve the safety and correctness of software.

A language feature that allows the programmer to specify such contract assertions is called a *Contracts facility*. Programming languages such as Eiffel and D have a Contracts facility; this paper proposes a Contracts facility for C++.

Note that not all parts of a contract can be specified via contract assertions, and of those that can, some cannot be checked at run time without violating the complexity guarantees of the function (e.g., the precondition of binary search that the input range is sorted), without additional instrumentation (e.g., a precondition that a pair of pointers denotes a valid range), or at all (e.g., a precondition that a passed-in function, if called, will return). Therefore, we do not expect that function contract assertions can, in general, specify the entire plain-language contract of a function; however, they should always specify a *subset* of the plain-language contract.

A corollary of this gap is that contract assertions, in general, cannot be used to verify compliance with the *entire* contract — i.e., to prove that the program is correct — but can only identify a subset of potential violations.

2.2 Proposed Features

The Contracts facility we propose will enable adding *contract assertions* to C++ code. We propose three kinds of contract assertions: *precondition assertions*, *postcondition assertions*, and *assertion statements*.

Precondition and postcondition assertions are placed on function declarations and collectively called *function contract assertions*. Assertion statements are placed inside function bodies. The following example contains all three kinds of contract assertions:

```
int f(const int x)
  pre (x != 1)               // a precondition assertion
  post(r : r != 2)           // a postcondition assertion; r names the result object of f
{
  contract_assert (x != 3);  // an assertion statement
  return x;
}
```

Each contract assertion has a *predicate*, which is a potentially evaluated expression that will be contextually converted to `bool` to identify a contract violation. When the predicate evaluates to `true`, no contract violation has been identified. When the predicate evaluates to `false` or when evaluation of the predicate exits via an exception, a contract violation has been identified. Other results that do not return control back up the stack through the evaluation of the contract assertion, such as terminating, entering an infinite loop, or invoking `longjmp`, happen as they would when evaluating any other C++ expression.

In the above code example, a contract violation will occur if `f` is called with a value of `1`, `2`, or `3`:

```
void g()
{
  f(0);  // no contract violation
  f(1);  // violates precondition assertion of f
  f(2);  // violates postcondition assertion of f
  f(3);  // violates assertion statement within f
  f(4);  // no contract violation
}
```

Each contract assertion has a *point of evaluation* based on its kind and syntactic position. Precondition assertions are evaluated immediately after function parameters are initialized and before entering the function body. Postcondition assertions are evaluated immediately after local variables in the function are destroyed when a function returns normally. Assertion statements are executed at the point in the function where control flow reaches them.

Each individual evaluation of a contract assertion is performed with a specific *evaluation semantic*. We propose four evaluation semantics (see Section 3.5.6): *ignore*, *observe*, *enforce*, and *quick_-enforce*. Which evaluation semantic is chosen for any particular evaluation of a contract assertion is implementation-defined.

The *ignore* semantic does nothing, whereas the *observe*, *enforce*, and *quick_enforce* semantics determine the value of the predicate to identify a contract violation. If a contract violation occurs, the *observe* and *enforce* semantic will invoke a function called the *contract-violation handler* (see Section 3.5.11). When this function returns normally, *observe* will continue program execution and *enforce* will terminate the program; *quick_enforce* will not call the contract-violation handler but will instead immediately terminate the program.

Users can install their own *user-defined contract-violation handler* at link time by defining their own function with the appropriate name and signature. This function takes one argument of type `const` reference to `std::contracts::contract_violation`. This type is defined in a new header, `<contracts>` (see Section 3.7). When the contract-violation handler is called, an object of this type is created by the implementation and passed in, providing access to some information about the contract violation that occurred, such as its source location and the used evaluation semantic.

Contract assertions can also be evaluated during constant evaluation (see Section 3.5.14), in which case the evaluation semantics behave slightly differently (notably, there is no compile-time contract-violation handler).

2.3 Features Not Proposed

To keep the scope of this MVP proposal *minimal* (while still viable), the following features are intentionally excluded from this proposal; we expect these features to be proposed as post-MVP extensions at a later time.

- The ability to specify precondition and postcondition assertions on pointers to functions, pointers to member functions, or type aliases for such types
- The ability to refer to the original values of parameters and other entities during the evaluation of the predicate of a postcondition
- The ability to assume that an unchecked contract predicate would evaluate to `true` and to allow the compiler to optimize based on that assumption, i.e., the *assume* semantic
- The ability to express the desired evaluation semantic directly on the contract assertion
- The ability to assign an assertion level to a contract assertion or, more generally, to specify in code properties of contracts and how they map to a contract semantic
- The ability to express postconditions of functions that do not exit normally, e.g., a postcondition that a function does or does not exit via an exception
- The ability to write a contract predicate that cannot be evaluated at run time, e.g., because it calls a function with no definition
- The ability to maintain state, for the purpose of evaluating contract assertions, that is populated and used beyond the scope of a single contract assertion, similar to what can be done with an `#ifndef NDEBUG` block when using `assert`.
- The ability to express invariants
- The ability to express procedural interfaces

Most of the above features were, in some shape or form, part of previous Contracts proposals; as a general rule, however, nothing in previous Contracts proposals should be assumed to be true about *this* proposal unless explicitly stated in this paper.

3 Proposed Design

3.1 Design Principles

The Contracts facility in this proposal has been guided by certain common principles that have helped clarify the optimal choices for how the facility should work and how it should integrate with the full breadth of the C++ language.

3.1.1 Contract Assertions Are Redundant Checks

The primary goal of this facility is to enable identifying, in code, when a program is *correct* or *incorrect*. To do so, making use of Contracts should be possible in ways that do not, just by being used, change whether a program is *correct* or *incorrect*. Time has made clear that this principle is, in fact, the foundation on which the rest of the design for Contracts is built.

Principle 1: Prime Directive

The presence or evaluation of a contract assertion in a program should not alter the correctness of that program (i.e., the property that evaluation of the program does not violate any provisions of its plain-language contract).

Closely related to the Prime Directive is another foundational principle of our proposed Contracts facility, the Redundancy Principle.

Principle 2: Redundancy Principle

In a correct program (i.e., one that does not violate any provisions of its plain-language contract), contract assertions are redundant; removal of any subset of the program's contract assertions should not alter the correctness of that program.

To maximize a programmer's ability to satisfy these foundational principles with regard to the mere presence or absence of contract assertions in their program, we can find the following three actionable secondary design principles.

Principle 3: Concepts Do Not See Contracts

The mere presence of a contract assertion on a function or in a block of code should not change the satisfiability of a concept, the result of overload resolution and SFINAE, the branch selected by `if constexpr`, or the value returned by the `noexcept` operator.

Principle 4: Zero Overhead

The presence of a contract assertion that is not actually checked — i.e., that is *ignored* — should not impact how a program behaves,[1] e.g., by triggering additional lambda captures that result in the inclusion of additional member variables to closure objects.

Principle 5: Independence from Chosen Semantic

Which evaluation semantic will be used for any given evaluation of a contract assertion and whether that evaluation semantic is a checking semantic should generally not be detectable at compile time; such detection might result in different programs being executed when contract checks are enabled.

When the above principles are violated, a contained program could be substantially changed by using Contracts in such a way. Therefore, we discourage this ability and have removed a number of places where, by adding even the simplest contract assertion, a program could, at compile time, implicitly be changed and thus made potentially incorrect.

Writing programs that violate the design principles 3–5 remains possible, and when written cleverly enough, such programs are not necessarily incorrect, but the general recommendation is to avoid employing such techniques.

3.1.2 Side Effects

Beyond the effect that the mere presence or absence of a contract assertion can have, we need to ensure, at run time, that another property can satisfy our foundational principles; i.e., the evaluation (or nonevaluation) of a contract-assertion predicate will, in and of itself, not change the correctness of a program.

We call a predicate whose presence or evaluation would change a program's correctness a *destructive predicate*, and we call any side effects that cause that change in correctness *destructive side effects*. With this definition, we can introduce our next principle, which is as foundational with respect to runtime evaluation as the Prime Directive is with respect to compile-time evaluation:

Principle 6: No Destructive Side Effects

Contract assertions whose predicates, when evaluated, could affect the correctness of the program should not be supported.

Note that a contract-assertion predicate whose evaluation does not have any side effects observable outside the cone of its evaluation is much more likely to satisfy this property but cannot guarantee it. Such a predicate might still violate complexity guarantees or other promises that are made in the plain-language contract and that are unknowable to the compiler, and thus the predicate might still be destructive under the above definition.

Further, such a restriction would be too limiting; e.g., for debugging, adding logging to a function called during the evaluation of a contract predicate might be useful, and such logging does not necessarily change the correctness of a program. Whether it does depends on the plain-language contract of that particular program (e.g., whether the program makes certain guarantees about what will be logged), which is unknowable to the compiler.

[1]Note that neither this proposal nor the C++ Standard itself can prescribe which or how many instructions are actually being emitted by the compiler; we can only reason about the behavior of the program within the C++ Abstract Machine.

To enable local reasoning about contract assertions and, more importantly, to enable global reasoning about how contract assertions are configured without needing to inspect each one, we should ensure the following important secondary principles that are related to the absence of destructive predicates.

Principle 7: Completeness of Contract Assertions

Each individual contract assertion encapsulates a complete check of a provision of the plain-language contract.

Principle 8: Independence of Contract Assertions

The result of evaluating a contract assertion should never affect the result of evaluating any other contract assertions.

Principle 9: Independence of Contract-Assertion Evaluations

The result of evaluating a contract assertion should never affect the result of subsequent evaluations of the same contract assertion.

A corollary of these principles is that contract-assertion predicates that, when evaluated, have side effects that maintain state affecting the correctness of the contract assertions themselves are destructive and therefore are not a correct use of the proposed Contracts facility.

Side effects in predicates are not ill-formed nor are they undefined behavior because the compiler cannot know which side effects are destructive; however, such side effects are not guaranteed to occur any particular number of times or at all and cannot be relied upon for correctness (see Section 3.5.10). We, therefore, do not, in this initial proposal, support contract assertions that, for example, increment a counter and then check whether the value of the counter is below a certain number or contract assertions in which one assertion sets a flag and another assertion unsets it. Note that existing macro-based facilities do nothing to prevent or dissuade their use with such predicates, which is one fundamental difference between this proposal and such facilities (see Section 3.6.7).

3.1.3 Contract Assertions and Plain-Language Contracts

Some additional principles involve defining our common understanding of the relationship between contract assertions and plain-language contracts.

Principle 10: Contract Assertions Check a Plain-Language Contract

The evaluation of a function contract assertion must be tied to the evaluation of the function to which the function contract assertion is attached so that the assertion will verify the plain-language contract (or some subset of the plain-language contract) of *that* function, not of some other function.

As an example of applying this principle, the function contract assertions attached to a virtual function (see Section 3.5.3) must not implicitly be applied to all overriding functions but rather should apply only when that function is the statically called function or when it is the final overrider selected by virtual dispatch.

Principle 11: Function Contract Assertions Serve Both Caller and Callee

A function contract assertion, much like a function declaration, is highly relevant to both the caller of and implementer of a function. In particular, as part of the agreement between callers and callees, two pairs of promises are made.

1. Callers promise to satisfy a function's preconditions, resulting in callees being able to rely upon those preconditions being true.
2. Callees (i.e., function implementers) promise to satisfy a function's postconditions when invoked properly, resulting in a caller's ability to rely upon those postconditions.

The answer to the commonly asked question of whether a function contract assertion is part of the interface of a function or of its implementation is, therefore, that it is part of *both*.

Principle 12: Contract Assertions Are Not Flow Control

While a contract assertion provides an algorithm to validate correctness, nothing about a contract assertion guarantees any particular runtime behavior associated with that syntactic construct.

Consequently, an unadorned contract assertion[2] might *enforce* the associated condition, terminating the program if it is violated, but might equally do nothing at all in another build, allowing violations to happen.

Importantly, this aspect of Contracts is why contract assertions must not be used for error handling and input validation: If a function has in-contract requirements to report certain events as errors, that handling must be done with standard C++ control statements that are not optional, never with contract assertions.[3]

3.1.4 Extensibility

The design of this proposal has been guided by two additional principles regarding how to address open design questions for which solutions are not yet agreed upon or known.

For any behavior that we define as part of a Contracts facility, certain rules must be followed in many cases. Those rules can be enforced in two primary ways: making violations ill-formed or making the behavior undefined when the rule is broken. For the specification and behavior of Contracts, we prioritize programs having well-defined behavior when using the new facility.

Principle 13: Explicitly Define All New Behavior

Contracts never explicitly introduce new undefined behavior when evaluating contract assertions.

Note that undefined behavior can still occur when the evaluation of a contract assertion encounters an expression for which the behavior is *already* undefined in C++ today (see Section 3.6.1).

[2]Future proposals might allow for more local control over the semantics with which a given contract assertion is evaluated, but such constraints on the possible semantic would always be a choice opted into via explicit annotations, not the default behavior of normal uses of the Contracts facility.

[3]See [P2053R1].

Principle 14: Choose Ill-Formed to Enable Flexible Evolution

When no clear consensus has become apparent regarding the proper solution to a problem that Contracts could address, the relevant constructs are left ill-formed.

This choice, rather than giving the relevant constructs unspecified or undefined behavior, enables conforming extensions to explore possible options, precluding none as an eventual solution that could be incorporated into the C++ Standard.

3.1.5 Compatibility

Finally, two more design principles ensure that adding Contracts to existing programs does not cause breakage in ways that could significantly hamper the adoption of Contracts in the field.

Principle 15: No Client-Side Language Break

For any existing function `f`, if function contract specifiers are added to `f` and the definition of `f` still compiles after this addition, then any existing, correct usage of `f` should continue to compile and work correctly.

Correct usages of a function include all uses that do not involve invoking the function such that the function's contract is violated, which is now partially checked due to the newly added contract assertions. These usages include any expressions that invoke the function, take its address, or assign that address to a pointer to function or pointer to member function as well as other functions that override the function.

Principle 16: No ABI Break

A conforming implementation should be able to guarantee that adding function contract specifiers to an existing function preserves ABI backward-compatibility.

3.2 Syntax

We propose three new syntactic constructs: precondition specifiers, postcondition specifiers, and assertion statements, spelled with `pre`, `post`, and `contract_assert`, respectively, and followed by the predicate in parentheses:

```
int f(const int x)
  pre (x != 1)               // precondition specifier
  post (r : r != 2)          // postcondition specifier; r names the result object of f
{
  contract_assert (x != 3);  // assertion statement
  return x;
}
```

The predicate is an expression contextually convertible to `bool`. The grammar requires the expression inside the parentheses to be a *conditional-expression*. This requirement guards against the common typo `a = b` (instead of `a == b`) by making the former ill-formed without an extra pair of parentheses around the expression.

3.2.1 Function Contract Specifiers

Precondition and postcondition specifiers are collectively called *function contract specifiers*. They may be applied to the declarator of a function (see Section 3.3.1 for which declarations) or of a lambda expression to introduce a function contract assertion[4] of the respective kind to the corresponding function. (For lambda expressions, the corresponding function is the call operator or operator template of the compiler-generated closure type.)

A precondition specifier is spelled with `pre` and introduces a precondition assertion to the corresponding function:

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

A postcondition specifier is introduced with `post` and introduces a postcondition assertion to the corresponding function:

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

A postcondition specifier may introduce a name to the *result object* of the function, called the *result name*, via a user-defined identifier preceding the predicate and separated from it by a colon:

```
int f(int i)
  post (r: r >= i);   // r refers to the result object of f.
```

The exact semantics of the result name are discussed in Section 3.4.3.

`pre` and `post` are contextual keywords. They are parsed as part of a function contract specifier only when they appear in the appropriate syntactic position. In all other contexts, they are parsed as identifiers. This property ensures that the introduction of `pre` and `post` does not break existing C++ code.

Function contract specifiers appear at the end of a function declarator, i.e., after trailing return types, after requires clauses, and immediately before the semicolon:

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

The only exception to this placement is the *pure-specifier* `= 0`, which can be interpreted as providing a definition for the function and thus appears after the function contract specifiers:

```
struct X {
  virtual void f(int i) pre (i > 0) = 0;
}
```

[4]The distinction between precondition and postcondition *specifiers* and precondition and postcondition *assertions* is analogous to the distinction between `noexcept` specifiers and exception specifications: The former refers to the syntactic construct, and the latter refers to the conceptual entity that is a property of a function. The distinction is important because a function that has function contract assertions might have multiple declarations, some of which might not have function contract specifiers (see Section 3.3.1 for details). Note that no such distinction is necessary for assertion statements.

Function contract specifiers on a definition appear in the corresponding location in the declaration part of the definition, immediately prior to the function body. Note that constructs such as `= default` and `= delete` are also function bodies.

For lambda expressions, function contract specifiers appear immediately prior to the lambda body:

```
int f() {
  auto f = [] (int i)
    pre (i > 0)
    { return ++i; };

  return f(42);
}
```

Any number of function contract specifiers, in any order, may be specified on a function declaration. Precondition specifiers do not have to precede postcondition specifiers but may be freely intermingled with them:

```
void f()
  pre (a)
  post (b)
  pre (c);  // OK
```

Evaluation of preconditions and postcondition assertions will still be done in their respective lexical order; see Section 3.5.2.

3.2.2 Assertion Statement

An *assertion statement* is a kind of contract assertion that may appear as a statement in the body of a function or lambda expression. An assertion statement is spelled with `contract_assert`, followed by the predicate in parentheses, followed by a semicolon:

```
void f() {
  int i = get_i();
  contract_assert(i != 0);
  // ...
}
```

Unlike `pre` and `post`, `contract_assert` is a full keyword, which is necessary for an assertion statement to be disambiguated from a function call. The keyword `contract_assert` is chosen instead of `assert` to avoid a clash with the existing `assert` macro from header `<cassert>`.

3.2.3 Attributes for Contract Assertions

All three kinds of contract assertions (`pre`, `post`, and `contract_assert`) permit attributes that appertain to the introduced contract assertion. We do not propose to add any such attributes to the C++ Standard itself, yet this permission can be useful for vendor-specific extensions to the functionality provided by this proposal. The syntactic location for such attributes specific to contract assertions is between the `pre`, `post`, or `contract_assert` and the predicate:

```
bool binary_search(Range r, const T& value)
  pre [[vendor::message("A nonsorted range has been provided")]] (is_sorted(r));

void f() {
  int i = get_i();
  contract_assert [[analyzer::prove_this]] (i > 0);
  // ...
}
```

In addition, attributes such as `[[likely]]` and `[[unlikely]]` that can appertain to other statements that involve some runtime evaluation can also appertain to `contract_assert`. The syntactic location for such attributes that appertain to the statement (rather than to the contract assertion it introduces) is before the statement:

```
void g(int x) {
  if (x >= 0) {
    [[likely]] contract_assert(x <= 100);      // OK, this branch is more likely.
    // ...
  }
  else {
    [[unlikely]] contract_assert(x >= -100);  // OK, this branch is less likely.
    // ...
  }
}
```

Finally, an attribute can also appertain to the result name optionally declared in a postcondition specifier:

```
int g()
  post (r [[maybe_unused]]: r > 0);
```

The attribute `[[maybe_unused]]` is explicitly allowed to appertain to the result name.

3.3 Syntactic Restrictions

3.3.1 Multiple Declarations

Any function declaration is a *first declaration* if no other declarations of the same function are reachable from that declaration; otherwise, it is a *redeclaration*. The sequence of function contract specifiers on a first declaration of a function introduces the corresponding function contract assertions that apply to that function.

It is ill-formed, no diagnostic required (IFNDR) if multiple first declarations for the same function are in different translation units that do not have the same sequence of function contract specifiers.

A redeclaration of a function shall have either no function contract specifiers or the same sequence of function contract specifiers as any first declaration reachable from it; otherwise, the program is ill-formed.

In effect, all places in which a function might be used or defined have a consistent and unambiguous view of that function's sequence of function contract specifiers.

Equivalence of function contract specifiers is determined as follows. Two sequences of function contract specifiers are considered to be the same if they consist of the same function contract specifiers in the same order. A function contract specifier *c1* on a function declaration *d1* is the same as a function contract specifier *c2* on a function declaration *d2* if their predicates *p1* and *p2* would satisfy the one-definition rule (odr) if placed in an imaginary function body on the declarations *d1* and *d2*, respectively, except the names of function parameters, names of template parameters, and the result name may be different.[5] (The entities found by name lookup will be the same.)

The rule above implies that the two lambda expressions that appear inside the predicates *p1* and *p2* are considered to be the same lambda expression if the odr considers the hypothetical function definitions containing those predicates to be the same:

```
void f() pre([]{ return true; }());
void f() pre([]{ return true; }());  // OK, redeclaration has same precondition specifier.
```

Note that this rule is in contrast with contexts other than function definitions and contract predicates, where each new instance of the lambda expression will generate a different closure type, for example:

```
template <typename T = decltype([]{ return true; })>
struct X {};

X x1;
X x2;
// x1 and x2 have different types.
```

3.3.2 Defaulted and Deleted Functions

For a declaration of a function defaulted on its first declaration to have precondition or postcondition specifiers is ill-formed:

```
struct X {
  X() pre (true) = default;     // error (pre on function defaulted on first declaration)
};

struct Y {
  Y() pre (true);
};

Y::Y() pre (true) = default;    // OK (not the first declaration; pre (true) can be omitted)
```

Further, for a declaration of an explicitly deleted function to have precondition or postcondition specifiers is ill-formed:

```
struct X {
  X() pre (true) = delete;   // error
};
```

[5] Note that the odr for function definitions does not allow for such exceptions: Multiple *definitions* of the same `inline` function in different translation units must be token-identical; different names for function parameters and template parameters are not allowed.

3.3.3 Await and Yield Expressions

If, in a coroutine, the predicate of a contract assertion contains an *await-expression* or *yield-expression* as a subexpression that is in the suspension context of that coroutine, the program is ill-formed:

```
std::generator<int> f() {
  contract_assert(((co_yield 1), true));  // error
}

stdex::task<void> g() {
  contract_assert((co_await query_database()) > 0);  // error
  // ...
}
```

An *await-expression* or *yield-expression* is allowed in the predicate of a contract assertion if it is not in the suspension context of that coroutine, e.g., because it appears inside an immediately invoked lambda that is not suspending the evaluation of the function or coroutine evaluating the predicate itself:

```
contract_assert(([]() -> std::generator<int> {
  co_yield 1;   // OK
}(), true));
```

3.3.4 Pointers to Functions and Pointers To Member Functions

Function contract specifiers may not be attached to a pointer to function or pointer to member function:

```
typedef int (*fpt)(int) post (r: r != 0);       // error

int f(int x) post (r: r != 0);
int (*fp)(int) post (r: r != 0) = f;            // error

int (X::*fptr)(int) post (r: r != 0) = &X::f;  // error
```

The contract assertions on a function have no impact on its type and thus no impact on the type of its address nor on what types of pointers to which the address of that function may be assigned:

```
int f(int x) post (r: r != 0);
int (*fp)(int) = f;  // OK
```

When a function *is* invoked through a pointer to function (e.g., when calling `f` through `fp` in the example above) or through a pointer to member function, the caller-facing function contract assertions of that invocation (see Section 3.5.1) are an empty set, but the callee-facing function contract assertions are still those of the invoked function and must be evaluated as normal. The same behavior applies to other kinds of indirect calls, such as through a pointer to member function or Standard Library function wrapper, such as `std::function`.

The consequence of this behavior is that, for such indirect calls, an implementation cannot, in general, check the precondition and postcondition assertions of the function at the call site since which function will end up being called is unknown. Such checks have to be performed either inside the function or in a thunk.

3.3.5 Function Type Aliases

Function contract specifiers may not be attached to a function type alias:

```
using ft = int(int) post (r: r != 0);  // error
```

However, function contract specifiers may be attached to a function declaration that uses a function type alias:

```
using ft = int(int);
ft f post (r: r != 0);  // OK
```

Note that such a function declaration does not introduce names for the parameters of the function and, therefore, does not provide a way to spell a contract predicate referring to these parameters.

3.3.6 Use of C Variadic-Function Parameters

If a contract predicate contains a use of the `va_start` macro as a subexpression, the program is ill-formed, no diagnostic required.

If we were to allow such use, we would have to require that any use of `va_start` within a contract-assertion predicate is matched by a use of `va_end` in the same predicate, and this cannot be checked statically. No diagnostic is required because, with current toolchain behaviors, this situation might not be diagnosable: On some implementations, `va_start` expands to a C++ expression along the lines of "address of previous argument plus one," losing the information that the `va_start` macro was used by the time the C++ front end receives the preprocessed stream of tokens.

The other macros involved in the processing of C variadic parameters — `va_arg` and `va_end` — do not need to be explicitly prohibited since they are useless without the use of `va_start`.

3.4 Semantics

3.4.1 Name Lookup and Access Control

For precondition assertions, name lookup in the predicate is performed as if the predicate came at the beginning of the body of the function or lambda expression although, once a name is resolved, the resulting expression will be `const` for entities declared outside the precondition assertion (see Section 3.4.2). This name lookup occurs as if a function body were specified on the declaration where the precondition specifier appears — i.e., using the parameter names on the declaration — instead of where the actual function definition appears (which might not even be visible) where a different declarator's parameter names would be in effect.

Access control is applied based on that behavior; i.e., the predicate may reference anything that might be referenced from within the body of the function or lambda expression. (A special rule, however, states that the program is ill-formed if such references trigger implicit lambda captures; see Section 3.4.8.) When the precondition assertion is part of a member function, protected and private data members of that function's type may be accessed. When a precondition assertion is part of a function that is a friend of a type, full access to that type is allowed.

For postcondition assertions, name lookup first considers its result name (see Section 3.4.3), if any, to be in a synthesized enclosing scope around the precondition assertion. For all other names, name lookup and access control is performed in the same fashion as for a precondition assertion.

For assertion statements, name lookup and access control occurs as if the predicate's expression were located in an expression statement at the location of the assertion statement.

3.4.2 Implicit `const`-ness

A contract check is supposed to observe, not change, the state of the program, exceptions such as logging notwithstanding. To prevent accidental bugs due to unintentional modifications of entities inside a contract predicate, any identifier used *within* a contract predicate that refers to a variable declared *outside* the predicate, including the function parameters and the result object (see Section 3.4.3), are `const` lvalues.

This behavior is very similar to how identifiers referring to members are implicitly `const` lvalues inside a `const` member function, except that this `const`-ness applies to all *id-expressions* denoting variables from outside the contract-assertion predicate, including those with automatic, static, and thread-local storage duration; those at block scope, class scope, and namespace scope; and structured bindings and the expressions `this` and `*this`, whether explicitly or implicitly used.

These `const` amendments are shallow (on the level of the lvalue only); attempting to invent so-called deep-`const` rules would likely make raw pointers and smart pointers behave differently, which is not desirable:

```
int global = 0;

void f(int x, int y, char *p, int& ref)
  pre((x = 0) == 0)             // error: assignment to const lvalue
  pre((*p = 5))                 // OK
  pre((ref = 5))                // error: assignment to const lvalue
  pre((global = 2))             // error: assignment to const lvalue
{
  int* gp = &global;
  contract_assert((gp = nullptr)); // error: assignment to const lvalue
  contract_assert((*gp = 6));      // OK
}
```

Class members declared mutable can be modified as before. Expressions that are not lexically part of the contract condition are not changed. The result of `decltype(x)` is not changed and still produces the declared type of the entity denoted by `x` (which might not be `const`). However, `decltype((x))` yields `const T&` where `T` is the type of the expression `x`.

Modifications of variables and parameters declared outside a contract predicate from within that predicate are possible — although discouraged — via applying a `const_cast`, but modifications of `const` objects continue to be undefined behavior as elsewhere in C++. This includes parameters required to be declared `const` because they are used in a postcondition (see Section 3.4.4):

```
int g(int i, const int j)
  pre(++const_cast<int&>(i))     // OK (but discouraged)
  pre(++const_cast<int&>(j))     // undefined behavior
  post(++const_cast<int&>(i))    // OK (but discouraged)
  post(++const_cast<int&>(j))    // undefined behavior
{
  int k = 0;
  const int l = 1;
  contract_assert(++const_cast<int&>(k));  // OK (but discouraged)
  contract_assert(++const_cast<int&>(l));  // undefined behavior
}
```

Overload resolution results (and thus semantics) may change if a predicate is hoisted into or out of a contract predicate:

```
struct X {};
bool p(X&) { return true; }
bool p(const X&) { return false; }

void my_assert(bool b) { if (!b) std::terminate(); }

void f(X x1)
  pre(p(x1))          // fails
{
  my_assert(p(x1));  // passes

  X x2;
  contract_assert(p(x2)); // fails
  my_assert(p(x2));       // passes
}
```

However, such an overload set that yields different results depending on the `const`-ness of the parameter is, arguably, in itself a bug.

When a lambda inside a contract predicate captures a nonfunction entity by copy, the type of the implicitly declared data member is `T`, but (as usual) naming such a data member inside the body of the lambda yields a `const` lvalue unless the lambda is declared `mutable`. When the lambda captures such an entity by reference, the type of an expression naming the reference is `const T`. When the lambda captures `this` of type pointer to `T`, the type of the implicitly declared data member is pointer to `const T`:

```
void f(int x)
  pre([x] { return x = 2; }())              // error: x is const
  pre([x] mutable { return x = 2; }())      // OK, modifies the copy of the parameter
  pre([&x] { return x = 2; }())             // error: ill-formed assignment to const lvalue
  pre([&x] mutable { return x = 2; }());    // error: ill-formed assignment to const lvalue

struct S {
  int dm;
  void mf() // not const
    pre([this]{ dm = 1; }())                // error: ill-formed assignment to const lvalue
    pre([this] () mutable { dm = 1; }())    // error: ill-formed assignment to const lvalue
    pre([*this]{ dm = 1; }())              // error: ill−formed assignment to const lvalue
    pre([*this] () mutable { dm = 1; }()) // OK, modifies a copy of *this
  {}
};
```

When a lambda inside a contract predicate refers to an externally declared entity that is not captured, such as a global or static variable, that entity is implicitly `const`, including in nested lambdas; entities declared inside such a lambda are unaffected:

```
void f() {
  static int i = 0;
  contract_assert([]{
    ++i;    // error: modifying const lvalue
    int j;
    [&j]{
      int k;
      ++i; // error: modifying const lvalue
      ++j; // OK
      ++k; // OK
    }();
    return true;
  }());
}
```

3.4.3 Postconditions: Referring to the Result Object

A postcondition specifier may optionally specify a *result name*, introducing a name that refers to the result object of the function. This functionality is conceptually similar to how the identifiers in a structured binding are not references but merely names referring to the elements of the unnamed structured-binding object. As with a variable declared within the body of a function or lambda expression, the introduced name cannot shadow function-parameter names. Note that this introduced name is visible only in the predicate to which it applies and does not introduce a new name into the scope of the function.

For a function `f` with the return type `T`, the result name is an lvalue of type `const T`, `decltype(r)` is `T`, and `decltype((r))` is `const T&`. This behavior is a consequence of the implicit `const`-ness of identifiers in contract predicates (see Section 3.4.2).

Although strongly discouraged, modifications of the return value in the postcondition-assertion predicate are possible via applying a `const_cast`. Note that even if the object is declared `const` at the call site or the function's return type is `const`-qualified, such modifications are not undefined behavior because, at the point where the postcondition is checked, initialization of the result object has not yet completed, and therefore, `const` semantics do not apply to it:

```
struct S {
  S();
  S(const S&) = delete; // noncopyable, nonmovable
  int i = 0;
  bool foo() const;
};
```

```
const S f()
  post(r: const_cast<S&>(r).i = 1)  // OK (but discouraged)
{
  return S{};
}

const S y = f();      // well-defined behavior
bool b = f().foo();   // well-defined behavior
```

Clarifying the relevant existing wording to make this intent clearer might be useful; such a clarification is being proposed in [CWG2841].

If a postcondition names the return value on a nontemplated function with a deduced return type, that postcondition must be attached to the declaration that is also the definition (and thus there can be no earlier declaration):

```
auto f1() post (r : r > 0);  // error: type of r is not readily available.

auto f2() post (r : r > 0)   // OK, type of r is deduced below.
{ return 5; }

template <typename T>
auto f3() post (r : r > 0);  // OK, postcondition instantiated with template

auto f4() post (true);       // OK, return value not named
```

A type or name dependent upon the deduced return type may appear in the postcondition predicate expression. If the function is a nontemplated function, such a type or name is not treated as dependent, which means that `template` and `typename` disambiguators are not required:

```
struct A {
  template <int N> bool f() const;
};

auto g()
  post (v: v.f<6>())  // OK, v.template f<6> not required
{
  return A{};
}

template <typename> struct X { enum { Nested }; };
template <> struct X<int>    { struct Nested {}; };

auto h()
  post (r: sizeof(X<decltype(ret)>::Nested) // OK, typename X<... not required
{
  return 42;
}
```

Delaying the parsing of the postcondition until the return type is known is a possible implementation strategy for the above semantics.

3.4.4 Postconditions: Referring to Parameters

If a function parameter is odr-used by a postcondition-assertion predicate, that function parameter must have reference type or be `const`. That function parameter must be declared `const` on all declarations of the function (even though top-level `const`-qualification of function parameters is discarded in other cases), including the declaration that is part of the definition:

```
void f(int i) post ( i != 0 );        // error: i must be const.

void g(const int i) post ( i != 0 );
void g(int i) {}                      // error: missing const for i in definition

void h(const int i) post (i != 0);
void h(const int i) {}
void h(int i);                        // error: missing const for i in redeclaration
```

Without this rule, reasoning about postcondition predicates on a function declaration would be impossible without also inspecting the definition because the parameter value might have been modified there. Consider:

```
double clamp(double min, double max, double value)
  post( r : (value < min && r == min)
         || (value > max && r == max)
         || (r == value) );
```

The postcondition is clearly intended to validate that `value` is clamped to be within the range `[min,max]`. The following, however, would be an implementation of `clamp` that would both fail to violate the postcondition *and* fail to be remotely useful:

```
double clamp(double min, double max, double value) {
  min = max = value = 0.0;
  return 0.0;
}
```

Such modifications of parameters can also happen *implicitly* rather than explicitly. For example, returning a parameter object by value could break a postcondition check because it would observe a moved-from value:

```
std::string g(std::string p)
  post (r: starts_with(p))
{
  return p;
}
```

Requiring that parameters be `const` if a postcondition predicate refers to them avoids such extreme failures and subtle variations on this theme by making modification of the parameters in the definition impossible.

If the function in question is a virtual function, then the parameter object could also be modified in the body of an overriding function. To prevent this case, the corresponding parameter in every declaration of every overriding function needs to be declared `const` as well, even if such an overriding function does not itself have a postcondition assertion:

```
struct Base {
  virtual std::string g(const std::string p)  // OK, p declared const
    post (r: starts_with(p));
};

struct Derived : Base {
  std::string g(std::string p) override;      // error: p must be declared const here
};
```

In a template, if the parameter has a dependent type, every declaration of the parameter is still required to have an explicit `const` qualifier; that is, `const` being part of the dependent type is insufficient:

```
template <typename T>
void f(T t) post(t > 0);  // error (even if instantiated only as f<const int> or not at all)
```

Further, when a function is defined to be a coroutine, its parameters may be modified even if they are declared `const` by the user on all declarations of the function. The reason is that the coroutine initializes copies of the parameters in the coroutine frame with modifiable xvalues referring to the original parameters, ignoring any `const` qualification on those original parameters,[6] which means that such parameters may be moved from. Therefore, if a function parameter is odr-used by a postcondition-assertion predicate and that function is defined to be a coroutine, the program is ill-formed, even if that parameter is declared `const` on all declarations written by the user.[7]

Effectively, a coroutine behaves as if its function parameters had their `const` qualification removed from the defining declaration, except that

- they retain their original `const` qualification when passed to the allocating function
- they retain their original `const` qualification when the types of the parameter copies are determined

Finally, odr-using an array parameter by a postcondition-assertion predicate is ill-formed because such an array parameter will decay to a pointer, and making this resulting pointer `const` to prevent such cases is not possible:

```
void f(const int a[])  post (a[0] == 5)  // error
{
  static int x[1];
  a = x;
  x[0] = 5;  // ...otherwise, we could do this to satisfy the postcondition above
}
```

Note that this restriction applies only to array parameters, not references to arrays:

```
void f(const int (&a)[N]) post (a[0] == 5);  // OK
```

[6]See [dcl.fct.def.coroutine]/13.

[7]Note that, in this case, the error would be on the *definition* of the function that makes it a coroutine, not on the *declaration* of that function, in the same way that dropping `const` on a nonreference parameter in the definition of a function would result in an error on that definition; the proposed rule thus does not compromise the property that the coroutineness of a function is an implementation detail.

3.4.5 Objects Passed and Returned in Registers

For efficiency, the major ABIs used for implementations of C++ allow objects to be passed to a function and returned from a function via registers when the type of the object satisfies certain requirements: The type of the object must have at least one eligible copy or move constructor, each such constructor must be trivial, and its destructor must be either trivial or deleted. The C++ Standard enables passing via registers for types that meet these requirements in a very specific manner: Implementations are permitted to create a temporary object and then use that temporary to initialize (through a trivial copy operation) the final parameter or result object.[8]

To make preconditions and postconditions checkable on both the caller side and callee side (see Section 3.5.1), they are allowed to observe either the temporary copy or the final object.

For a nonreference parameter in a precondition and the return object in a postcondition, access to this temporary happens before the final object is initialized, and access to the final object happens after it is initialized from the temporary. Though the objects observed in the predicate might have different addresses, the state of the object will still be easily reasoned about as a single object:

```
class X { /* ... */ };

X f(const X* ptr) post(r: &r == ptr) {
  return X{};
}

int main() {
  X x = f(&x);
}
```

If `X` is *not* a type eligible to be passed via registers, the postcondition check in `f` will pass because `r` must denote the return object `x` in `main()`. If, however, `X` *is* a type eligible to be passed via registers, the postcondition check might fail[9] because `r` may instead denote a temporary object. Importantly, in both cases, the object will have the same value.

Similarly, we can do the same thing with function parameters:

```
X* ptr;

void f(const X x) post (ptr == &x) {
  ptr = &x;
}
```

If and only if `X` *is* a type eligible to be passed via registers, the postcondition check in `f` might fail, but again, in both cases, the object will have the same value.

However, for a nonreference parameter in a postcondition, the situation is more complex because the temporary copy is made when the function is called, the postcondition assertion is checked when the function returns, and arbitrary code might be executed in between. The parameter needs to be declared `const`, but it might have `mutable` members that can be modified. Therefore, the state of

[8]See [class.temporary]/3.

[9]In practice, whether this check fails will depend on both optimization levels and whether `f` is inlined into `main()`.

the parameter object that the function body observes can diverge from the state of the parameter object that the postcondition assertion observes since the latter is an older copy of the object:

```
class RandomInteger {
  mutable bool _computed = false;
  mutable int _value;
public:
  int value() const {
    if (!_computed) {
      _value = rand();
      _computed = true;
    }
    return _value;
  }
};

int f(const RandomInteger i) post(r: r & i.value() == 0) {
  return ~i.value();
}
```

Because `RandomInteger` is trivially copyable and trivially destructible, it may be passed in registers. Therefore, the postcondition check might see a different value being returned from `i.value()` than the function body does, and the postcondition check might fail.

While such failures might seem very surprising if the user is unfamiliar with the rules for objects passed and returned via registers, we consider them to be rare, diagnosable with a compiler warning, and easily avoided by using well-designed parameter types. More importantly, the intent of requiring that such parameters be `const` is to guarantee that the value of the parameter is one that the caller can reason about, and a temporary that was never directly seen by the function body is certainly such a value.

3.4.6 Not Part of the Immediate Context

The predicate of a function contract assertion, while lexically part of a function declaration, is not considered part of the immediate context:

```
template <std::regular T>
void f(T v, T u)
  pre ( v < u ); // not part of std::regular

template <typename T>
constexpr bool has_f =
  std::regular<T> &&
  requires(T v, T u) { f(v, u); };

static_assert( has_f<std::string>);          // OK, has_f returns true.
static_assert(!has_f<std::complex<float>>); // error: has_f causes hard instantiation error.
```

As a consequence, contract assertions are able to expand the requirements of a function template in the same way other parts of the function declaration can, causing a program to be irrecoverably ill-formed (i.e., not subject to SFINAE) if those requirements are not met for a given set of function-template arguments.

3.4.7 Function-Template Specializations

The function contract assertions of an explicit specialization of a function template are independent of the function contract assertions of the primary template:

```
bool a = true;
bool b = false;

template <typename T>
void f() pre(a) {}

template<>
void f<int>() pre(b) {}  // OK, precondition assertion different from that of primary template

template<>
void f<bool>() {}        // OK, no precondition assertion
```

3.4.8 No Implicit Lambda Captures

For lambdas with default captures, contract assertions that are part of the lambda need to be prevented from triggering implicit lambda captures that would otherwise not be triggered. If we allowed such captures, adding a contract assertion to an existing program could change the observable properties of the closure type or cause additional copies or destructions to be performed, violating Design Principle 4 (Zero Overhead). Therefore, if all potential references to a local entity implicitly captured by a lambda occur only within contract assertions attached to that lambda (precondition or postcondition specifiers on its declarator or assertion statements inside its body), the program is ill-formed:

```
static int i = 0;

void test() {
  auto f1 = [=] pre(i > 0) {  // OK, no local entities are captured.
  };

  int i = 1;

  auto f2 = [=] pre(i > 0) {  // error: cannot implicitly capture i here
  };

  auto f3 = [i] pre(i > 0) {  // OK, i is captured explicitly.
  };

  auto f4 = [=] {
    contract_assert(i > 0);   // error: cannot implicitly capture i here
  };

  auto f5 = [=] {
    contract_assert(i > 0);   // OK, i is referenced elsewhere.
```

```
        (void)i;
    };

    auto f6 = [=] pre([]{
        bool x = true;
        return [=]{ return x; }();   // OK, x is captured implicitly.
      }()) {};
}
```

3.5 Evaluation

3.5.1 Caller-Facing and Callee-Facing Function Contract Assertions

On any function invocation, two sets of function contract assertions are going to be evaluated: those facing the caller and those facing the callee. Before the function is invoked, caller-facing preconditions are checked, followed by callee-facing preconditions. When a function returns normally, the contract assertions are evaluated in reverse order, i.e., callee-facing postconditions followed by caller-facing postconditions. This evaluation sequence is illustrated in Figure 1.

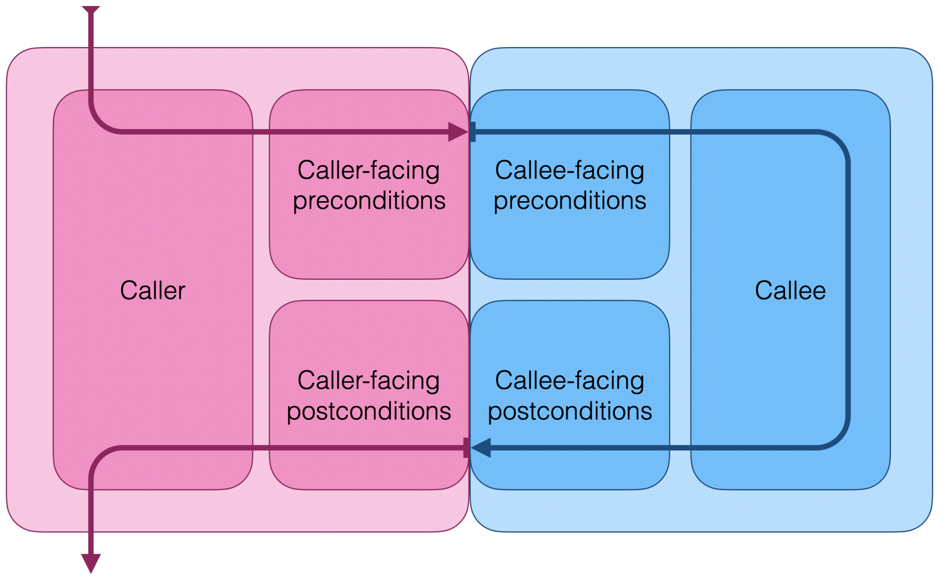
Caller
Caller-facing preconditions
Caller-facing postconditions
Callee-facing preconditions
Callee-facing postconditions
Callee

Figure 1: Evaluation sequence of caller-facing and callee-facing function contract assertions

The callee-facing function contract assertions of a function call are *always* the function contract assertions attached to the function whose body is being evaluated.

The caller-facing function contract assertions of a function call, on the other hand, are determined based on how the function is invoked.

- For a direct invocation of a function by name, the caller-facing contract assertions are those of the function being invoked. Note that for any such direct invocation, the caller-facing contract assertions will be the same as the callee-facing function contract assertions.

- When invoking a function through a pointer to function or a pointer to member function, the caller-facing function contract assertions are those of the pointer — i.e., an empty set of contract assertions since this proposal does not allow placing function contract assertions directly on pointers (which might be changed by a future proposal).
- When doing virtual dispatch through a virtual function by name (i.e., when the function name is a class member access expression implicitly or explicitly), the caller-facing function contract assertions are those of the statically invoked function, i.e., the virtual function on the static type of the left operand of the class member access expression.

The possible sets of caller-facing and callee-facing contract assertions are summarized in Table 1.

Invocation	**Caller-facing `pre`/`post` assertions**	**Callee-facing `pre`/`post` assertions**
Direct function call (including qualified call of a virtual function)	Those of the function	Those of the function
Call through a pointer to function or pointer to member function	Those of the pointer, i.e., none	Those of the function pointed to
Virtual function call	Those of the statically called function	Those of the final overrider
Virtual function call through a pointer to member function	Those of the pointer, i.e., none	Those of the final overrider

Table 1: Caller-facing and callee-facing function contract assertions for different ways of invoking a function

Future proposals to allow contract assertions on pointers to functions and pointers to member functions — or alternate types that behave like such pointers but allow for the addition of function contract assertions — might introduce more possibilities for distinct nonempty caller-facing sets of function contract assertions.

3.5.2 Point of Evaluation

All precondition assertions of a function invocation — first caller-facing and then callee-facing — are evaluated after the initialization of function parameters and before the evaluation of the function body begins.

All postcondition assertions of a function invocation — first callee-facing and then caller-facing — are evaluated after the return value has been initialized and after the destruction of any local variables in scopes exited by the return statement[10] but prior to the destruction of function parameters. Note the potential subtleties of what this specification means for constructors and destructors (see Section 3.6.2) and for coroutines (see Section 3.5.4).

[10]Note that in the current C++ Standard, the term "return statement" does not necessarily mean a literal return statement that the user wrote but includes other situations in which a function returns control to the caller, in particular when flowing off the end of a `void` function, a constructor, or a destructor.

Multiple precondition or postcondition assertions within the caller-facing and callee-facing sequences of function contract assertions are evaluated in the order in which they are declared. If the return type of a function is eligible to be passed in registers, the compiler is allowed to make extra trivial copies of the return object, and the postcondition assertions may then refer to those copies; however, those copies need to be done in sequence with the evaluation of the postcondition assertions. Therefore, if the program is built such that every contract assertion will be evaluated with a checked semantic exactly once, then in the following example,[11] both postcondition assertions must evaluate to `true`, regardless of whether `r` refers to the same object:

```
int f()
  post(r : ++const_cast<int&>(r) == 1)
  post(r : ++const_cast<int&>(r) == 2)
{
  return 0;
}
```

An assertion statement will be executed at the point where control flow reaches the statement.

When the predicate expression of a contract assertion is evaluated, it is contextually converted to `bool`. The predicate expression is a full expression; therefore, any temporaries created during the predicate's evaluation are destroyed when that evaluation is complete.

Precondition assertions, postcondition assertions, and assertion statements are, therefore, distinguished from one another by their points of evaluation, and (plain-language) preconditions and postconditions are distinguished by who — the caller or the callee — is responsible for ensuring that they are true (see Section 2.1). In most but not all cases, precondition and postcondition assertions are used to check preconditions and postconditions, respectively. In some cases, to check a (plain-language) precondition, we might use, at the beginning of a function body, an assertion statement (for example, to insulate the check from the caller if it is considered to be an implementation detail) or even a postcondition assertion (for example, because the precondition predicate can be evaluated algorithmically more efficiently after having evaluated the function body first).

3.5.3 Virtual Function Calls

The declaration of a virtual function can have precondition and postcondition specifiers. Following Design Principle 10 (Contract Assertions Check a Plain-Language Contract), these specifiers introduce the function contract assertions of *that* function. Function contract assertions are not inherited; i.e., the function contract assertions of an overriding function are independent of those of any overridden function. Note that contracts on virtual functions are handled differently in this proposal than in some other programming languages, such as D and Eiffel.

When a virtual function call happens, which function contract assertions are evaluated and in what order is determined by the rules for caller-facing and callee-facing function contract assertions described in Section 3.5.1.

[11] In general, no guarantee is made that both postconditions are evaluated with a checked semantic nor if they are, that both will be evaluated exactly once (see Section 3.5.9). Writing postcondition assertions such as those in this example and expecting them to succeed is, therefore, not a correct use of the proposed Contracts facility; the example is used only to illustrate the point about materialized temporaries.

The caller-facing function contract assertions are those of the statically called function, and the callee-facing function contract assertions are those of the final overrider.

When a virtual function call happens through a pointer to member function, the caller-facing function contract assertions are those of that pointer to member function, i.e., *none* in this proposal. The callee-facing function contract assertions in that case, however, are still those of the final overrider.

Let us illustrate the above specification with some code examples. Consider a base class, `FancyOperation`, with a pure virtual function, `apply`, whose contract consists of a precondition that the value passed in is non-negative and a postcondition that the return value is between 0 and the value passed in:

```
struct FancyOperation {
  virtual int apply(const int x)
    pre ( x >= 0 )
    post ( r: r >= 0 )
    post ( r: r <= x ) = 0;
};
```

Now we add a derived class, `Identity`, with an override of `apply` that implements the identity function. This override has a different contract; it has no preconditions, and its postcondition is that the output is identical to the input:

```
struct Identity : FancyOperation {
  int apply(const int x) override
    post (r: r == x);
};
```

Now, if we perform a virtual function call where the statically called function is `FancyOperation::apply` but the dynamic type of the object is `Identity`, both sets of function contract assertions will be evaluated:

```
void test() {
  Identity identity;
  FancyOperation& fancyOp = identity;
  int i1 = fancyop.apply(-1); // error: FancyOperation precondition violated
  int i2 = fancyop.apply(1);  // OK
}
```

However, if we perform the call directly on an object of type `Identity` (or a pointer or reference to `Identity`), only the function contract assertions of `Identity::apply` will be evaluated:

```
void test() {
  Identity identity;
  int i = identity(-1);  // OK; i == -1
}
```

In the above code, each function contract assertion of `Identity::apply` are evaluated twice (since it is a virtual function call and `Identity::apply` is both the statically called function and the overrider), but the compiler may elide the duplicate evaluations (see Section 3.5.10).

If we call `Identity::apply` through a pointer to member function, only the function contract assertions of that function will be called, even if the pointer to member function points to a member function of the base class `FancyOperation`:

```
void test() {
  int (FancyOperation::*applyPtr)(int) = &FancyOperation::apply;

  Identity identity;
  FancyOperation& fancyOp = identity;

  int i = (fancyOp.*applyPtr)(-1); // OK; i == -1
```

Function contract assertions of overrides in intermediate classes are not considered; for example, if we inherit `MoreFancyOperation` from `FancyOperation`, then inherit `Identity` from `MoreFancyOperation`, then create an object of dynamic type `Identity` and call `apply` through a reference to `FancyOperation`, the function contract assertions of `FancyOperation::apply` and `Identity::apply` will be evaluated, but the function contract assertions of `MoreFancyOperation` will not.

When using multiple inheritance, function contract assertions in sibling classes are also not considered:

```
struct FancyOperation {
  virtual int apply(const int x)
    pre ( x >= 0 )
    post ( r: r >= 0 )
    post ( r: r <= x ) = 0;
};

struct AntiFancyOperation {
  virtual int apply(const int x)
    pre ( x <= 0 )
    post ( r : r <= 0 )
    post ( r : r >= x ) = 0;
}

struct Identity : FancyOperation, AntiFancyOperation {
  int apply(const int x) override
    post ( r: r == x );
}

void test() {
    Identity identity;

    FancyOperation& op = identity;
    int i1 = op.apply(1);  // OK
    int i2 = op.apply(-1); // error: precondition violation in FancyOperation::apply

    AntiFancyOperation& antiOp = identity;
    int i3 = op.apply(1);  // error: precondition violation in AntiFancyOperation::apply
    int i4 = op.apply(-1); // OK
}
```

3.5.4 Coroutines

Coroutines are allowed to have precondition assertions, postcondition assertions, and assertion statements. We already discussed some syntactic rules (see Section 3.3.3) and rules for odr-using function parameters in postcondition assertions (see Section 3.4.4) that are relevant for coroutines. In this section, we cover how the point of evaluation of precondition and postcondition assertions (see Section 3.5.2) applies to coroutines.

At first glance, we see no obvious explanation for how precondition and postcondition assertions should work for a coroutine that can be suspended and resumed and thus has more than one entry and exit point; however, upon closer inspection, only one method can possibly work.

The key observation is that the fundamental design principle of coroutines in C++ states that the coroutineness of a function is an implementation detail; we cannot tell from a function declaration or from a function call whether the callee is a coroutine or a noncoroutine function.

When we call a coroutine, the function that is effectively called is *not* the coroutine body that the user wrote, but a function synthesized by the compiler, sometimes called the *ramp function*.[12] The object returned by this ramp function is not the ultimate result of the operation (such as the values `co_yield`-ed or `co-return`-ed by the coroutine), but a result object that can be used to advance the state of the coroutine, such as a generator, a task, or an awaitable. Since the precondition and postcondition assertions apply to the declaration of a function and do not see its coroutineness, these assertions must apply to the ramp function, not to the user-provided coroutine body.

To illustrate the design intent, consider the following function declaration:

```
// returns a generator for the integer sequence n, n + 1, n + 2...
generator<int> iota(int n);
```

One option to implement this function is as a coroutine:

```
generator<int> iota(int n) {
while (true)
  co_yield n++;
}
```

Another option is to implement this function as a noncoroutine function. We could manually initialize an object of type `generator<int>` and return it, using none of the C++ coroutine machinery. Such an initialization can be done with no observable change in behavior between the two implementations.

A third possible implementation is to implement `iota` as a noncoroutine function by wrapping a coroutine `iota_coro_impl`:

[12] First, the compiler transforms the user-provided body of the coroutine into a different function body, as specified in [dcl.fct.def.coroutine]/5. Note that this *replacement body* — again, as specified in [dcl.fct.def.coroutine]/5 — is itself also a coroutine which suspends the user-provided coroutine at its initial suspension point. The compiler effectively surrounds this replacement coroutine with yet another synthesized wrapper, the *ramp function*, which calls the replacement coroutine and returns the return object. The wording describing all this machinery is currently somewhat unclear. We should consider clarifying and simplifying that wording. Some first steps toward that goal are already realized in the wording proposed in this paper because the proposed wording helps us specify the behavior of `pre` and `post` on coroutines. However, further clarification of coroutines should probably be undertaken independent of Contracts.

```
generator<int> iota(int n) {
  return iota_coro_impl(n);
}
```

where `iota_coro_impl` is implemented like the first version of `iota` above.

The design intent is that function contract assertions should behave the same in each of the three cases above. The only necessary exception to this rule is that, because of how coroutines are specified today, if the predicate of a postcondition assertion odr-uses a nonreference parameter, such a function *cannot* be defined as a coroutine because that would mean the parameter would be moved-from before the postcondition assertion is evaluated, which would break the postcondition check (see Section 3.4.4).

Despite this restriction, postcondition assertions on coroutines are still useful to assert properties of the returned object:

```
awaitable<int> cancelable_session(int id)
  post (r: is_cancelable(r));
```

Precondition assertions are entirely unaffected by the coroutineness of a function, and their benefits are thus the same as for noncoroutine functions:

```
generator<int> sequence(int from, int to)
  pre (from <= to);
```

Precondition assertions are evaluated after the initialization of function parameters and before the evaluation of the function body begins (see Section 3.5.2). For a coroutine, this sequencing means that precondition assertions are evaluated *before* any of the coroutine-specific operations inside the ramp function, such as allocating storage for the coroutine state and creating copies of the function parameters. Creating these copies might perform a move; however, we can extrapolate that if a function parameter is odr-used in the precondition-assertion predicate, the expression refers to the original parameter objects, not the copies created.

Further, postcondition assertions are evaluated when the function returns control to the caller. For a coroutine, this point of evaluation means the (compiler-generated) return of the coroutine return object by the ramp function, not any of the suspension points inside the coroutine body.[13] Postcondition assertions are evaluated after the return value has been initialized and after the destruction of any local variables in scopes exited by the return statement. However, when a coroutine suspends, control flow returns without exiting any scopes.[14] Therefore, evaluation of the postcondition assertions of a coroutine is unsequenced with respect to the destruction of anything in the coroutine state — the promise object, copies of the parameter objects, or local variables in the coroutine body.

[13]Note that the return to the caller *can* correspond to the first suspension of a coroutine, but all other suspension points return to an alternate resumer (i.e., not to the caller of the ramp function). Note further that the return need not correspond to a suspension in general; the coroutine could run synchronously to completion or be destroyed in response to some interaction before reaching any active suspension.

[14]See [expr.await]/5.1.3.

In other words, the postcondition assertions of a coroutine may be evaluated before the coroutine body completes or even after the entire coroutine state has been deallocated.[15] This behavior is a direct consequence of the design and specification of coroutines in C++; it is not a special rule introduced by Contracts.

3.5.5 Observable Checkpoints

[P1494R3] introduces the notion of *observable checkpoints* to the C++ language. This proposal builds on top of [P1494R3] and introduces the following events in the execution of a program as such observable checkpoints:

- The beginning of evaluation of a contract predicate when evaluating a contract assertion
- The contract-violation handler returning normally when it is invoked from a contract assertion having the *observe* semantic

This specification mitigates certain cases of unwanted interaction between contract assertions and undefined behavior. One such case is undefined behavior that occurs when evaluating the predicate of a contract assertion:

```
int i = 0;
void f(int *p) {
  if (p != nullptr) // #1
    ++i;

  contract_assert( *p >= 0 ); // undefined behavior if p == nullptr
}
```

Making the beginning of the evaluation of the `contract_assert` an observable checkpoint prevents time-travel optimizations that could alter the behavior of earlier evaluations, such as eliding the check at `#1`, which could otherwise result in observing the increment of `i` even when `f(nullptr)` is invoked and the contract assertion is evaluated with the *enforce* semantic.

Another case is undefined behavior that occurs *after* the evaluation of a contract assertion:

```
void g(int *p)
{
  contract_assert (p != nullptr); // #2
  ++(*p); // undefined behavior if p == nullptr
}
```

If the contract semantic at `#2` is *observe*, execution will continue into the line of code where the undefined behavior occurs, even if the check at `#2` fails. Making the return of the contract-violation handler an observable checkpoint prevents time-travel optimizations that could otherwise elide the contract check at `#2` itself.

Note that this specification does not eliminate all possible cases where a contract check could be elided due to undefined behavior (see Section 3.6.1).

[15]The coroutine state might have already been deallocated by the time the postcondition assertion is evaluated because this deallocation can happen after the initial suspend and before control is returned to the caller.

3.5.6 Evaluation Semantics: *ignore*, *observe*, *enforce*, *quick_enforce*

Contract assertions are evaluated during constant evaluation as well as at run time. Each evaluation of a contract assertion is performed with a specific *evaluation semantic*, which might or might not evaluate the predicate.

We propose four evaluation semantics: *ignore*, *observe*, *enforce*, and *quick_enforce*. An implementation may provide additional evaluation semantics, with implementation-defined behavior, as a vendor extension. The four proposed evaluation semantics have the following characteristics:

- The *observe*, *enforce*, and *quick_enforce* semantics are collectively called *checking semantics*; evaluating a contract assertion with a checking semantic is also called performing a *contract check*. A contract check determines the value of the predicate to identify contract violations.
- The *ignore* semantic is not a checking semantic; evaluating a contract assertion with the *ignore* semantic has no effect. Note that a predicate is still parsed and is a *potentially evaluated* expression; i.e., a predicate odr-uses entities that it references. Therefore, a predicate must always be a well-formed, evaluable expression. See Section 3.6.7 for how this behavior differs from that of an `assert` macro that is disabled by defining `NDEBUG`.

A contract check might result in a contract violation being identified; see Section 3.5.8 for a description of how a contract check is performed.

If no contract violation is identified, program execution will continue from the point of evaluation of the contract assertion.

If a contract violation is identified at *run time*, the behavior is as follows.

- The *observe* semantic will invoke the contract-violation handler; if the contract-violation handler returns normally, program execution will continue from the point of evaluation of the contract assertion.
- The *enforce* semantic will invoke the contract-violation handler; if the contract-violation handler returns normally, the program is terminated in an implementation-defined fashion.
- The *quick_enforce* semantic will not invoke the contract-violation handler but will instead immediately terminate the program in an implementation-defined fashion.

The *enforce* and *quick_enforce* semantics are collectively called *terminating semantics* because when a contract violation occurs, they will prevent program execution from continuing into the code following the violated contract assertion by terminating the program.

Note that the fashion of termination can be different for different contract-assertion evaluations in the same program. For example, it is a conforming implementation of the *quick_enforce* semantic to call `__builtin_trap()` when the predicate evaluates to `false` but to call `std::terminate` when evaluation of the predicate exits via an exception.

If a contract violation is identified at *compile time* (during constant evaluation), the behavior is as follows.

- The *observe* semantic will issue a diagnostic (a warning).
- The *enforce* and *quick_enforce* semantics will render the program ill-formed.

See Section 3.5.14 for more details on constant evaluation of contract assertions.

3.5.7 Selection of Semantics

The semantic a contract assertion will have when evaluated is implementation-defined. The selection of semantic (*ignore*, *observe*, *enforce*, or *quick_enforce*) may happen at compile time, link time, load time, or run time. In practice, the choice of semantic will most likely be controlled by a command-line option to the compiler, although platforms might provide other avenues for selecting a semantic, and this proposal does not mandate the exact forms and flexibility of this selection.

Different contract assertions can have different semantics, even in the same function. The same contract assertion may even have different semantics for different evaluations. Chains of consecutive evaluations of contract assertions may have individual contract assertions repeated any number of times (with certain restrictions and limitations; see Section 3.5.9) and may involve evaluating the same contract assertion with different evaluation semantics.

The semantic a contract assertion will have when evaluated should, in general, not be identifiable through any reflective functionality of the C++ language. Branching at compile time based on whether a contract assertion will be checked or unchecked or on which concrete semantic it will have when evaluated is, therefore, not recommended, and the design proposed here does not include any features that facilitate such branching (although it remains possible to construct programs that achieve this effect). This is another important difference between contract assertions and the `assert` macro (see Section 3.6.7; our proposal does not contain any equivalent for `NDEBUG`).

We expect that implementations will provide appropriate compiler flags to choose the evaluation semantics assigned to contract assertions and that these flags can vary across translation units. Whether the contract assertion semantic choice for runtime evaluation can be delayed until link or run time is also, similarly, likely to be controlled through additional compiler flags.

We recommend that an implementation provide modes to set all contract assertions to have, at translation time, the *enforce* or the *ignore* semantic for runtime evaluation.

We recommend that a contract assertion will have the *enforce* semantic at run time when nothing else has been specified by a user. Compiler flags like `-DNDEBUG`, `-O3`, or similar are understood to perhaps be considered to be "doing something" to indicate a desire to prefer speed over correctness, and these flags are certainly conforming decisions. The ideal practice, however, is to make sure that the beginner student, when first compiling software in C++, does not need to understand Contracts to benefit from the aid that Contracts will provide by notifying that student of their own mistakes.

A compiler may offer separate compiler flags for selecting an evaluation semantic for constant evaluation, e.g., if the user wishes to ignore contracts at compile time to minimize compile times but still perform contract checks at run time. A reasonable default configuration for an optimized *Release* build might be to *enforce* contract assertions at compile time but to *ignore* them at run time (to maximize runtime performance with C++'s usual disregard for moderate increases in compile time).

3.5.8 Checking the Contract Predicate

When a contract assertion is being evaluated with a checking semantic, a *contract check* is performed to determine the result of evaluating the contract predicate.

If the result of the predicate can be determined, two possible results appear.

1. The predicate evaluates to `true`.
2. The predicate evaluates to `false`.

If the predicate evaluates to `true`, no contract violation has been identified. Execution will continue normally after the point of evaluation of the contract assertion.

If the predicate evaluates to `false`, a contract violation has been identified. The contract-violation handling process will be invoked; if the contract violation occurs at run time, the contract-violation handler will be called with the value `predicate_false` for `detection_mode` (see Section 3.5.12).

If evaluation of the predicate does not produce a value, two more possible outcomes of the contract check appear.

3. Control remains in the purview of the contract-checking process. This occurs when
   - evaluation of the predicate exits via an exception
   - evaluation of the predicate happens as part of constant evaluation, i.e., at compile time, and the predicate is not a core constant expression, i.e., cannot be evaluated at compile time (see Section 3.5.14).
4. Control never returns to the purview of the contract-checking process. This occurs when
   - evaluation of the predicate enters an infinite loop or suspends the thread indefinitely
   - evaluation of the predicate results in a call to `longjmp`
   - evaluation of the predicate results in program termination

In this paper, we made the decision to refer to case 3 as a form of contract violation,[16] and the contract-checking process will treat it as such. When this happens because an exception is thrown at run time, the contract-violation handler will be called with the value `evaluation_exception` for `detection_mode`; the exception itself is provided to the handler via the member function `evaluation_exception()` of the passed-in `contract_violation` object (see Section 3.5.12).

In case 4, any effects of the incomplete evaluation of the predicate, such as a call to `longjmp` or program termination, happen according to the normal rules of the C++ language.

[16]This situation possibly occurs when the actual plain-language contract has not been violated, such as when evaluation of the predicate hits a resource limit that the actual function invocation will not hit. In such situations, we still treat this scenario as a runtime contract violation and defer to the contract-violation handler to make a determination as to what the proper next course of action will be.

3.5.9 Consecutive and Repeated Evaluations

A vacuous operation is one that should not, a priori, be able to alter the state of a program that a contract could observe, and thus such an operation could not induce a contract violation. Examples of such vacuous operations include

- doing nothing, such as an empty statement
- performing trivial initialization, including trivial constructors and value-initializing scalar objects
- initializing reference variables
- transferring control via function invocation or a return statement, though note the corresponding function parameter and resulting value initialization might not be vacuous

Note that this list does not include destruction — even trivial destruction — since any contract assertion that would reference an object that is destroyed would no longer be meaningful to evaluate after the referenced object's lifetime has ended.

Two contract assertions shall be considered *consecutive* when they are separated only by vacuous operations. A *contract-assertion sequence* is a sequence of consecutive contract assertions. These will naturally include checking

- all caller-facing and callee-facing precondition assertions on a single function when invoking that function
- all caller-facing and callee-facing postcondition assertions on a single function when that function returns normally
- consecutive assertion statements
- the precondition assertions of a function and any assertion statements that are at the beginning of the body of that function
- the precondition assertions of a function `f1` and the precondition assertions of the first function `f2` invoked by `f1` when all statements preceding the invocation of `f2` and preparing the arguments to the invoked function `f2` involve only vacuous operations
- if a function `f1` has no function parameters and returns void, the postcondition assertions of `f1` and the precondition assertions of the next function `f2` invoked immediately after `f1` returns when the preparation of the arguments of `f2` involves only vacuous operations

At any point within a contract-assertion sequence, any previously evaluated contract assertions may be evaluated again, with the same or a different evaluation semantic,[17] up to an implementation-defined number of times. If the same contract assertion is evaluated multiple times in a contract-assertion sequence (as part of the same function invocation), the evaluation is not implicitly skippable, but an implementation is free to document that such repeated evaluations[18] will be *ignored*.

[17]Note that an equivalent formulation is that the entire sequence of contract assertions already evaluated up to a point may be repeated with an arbitrary subset of those contract assertions evaluated with the *ignore* semantic.

[18]Such a repeated evaluation might happen when invoking a virtual function on an object whose dynamic type is statically known and that does not have a separate override of that function. In such cases, the function contract assertions of that function will be evaluated twice.

As a recommended practice, an implementation should provide an option to perform a specified number of repeated evaluations for contract assertions. By default, no additional repetitions should be performed, i.e., each contract assertion should be evaluated exactly once.

In practice, the above rules mean that the preconditions and postconditions of a function may be evaluated, as a group, any number of times. Evaluations still, however, occur in sequence, and thus later contract assertions will never be evaluated until after earlier ones are evaluated. For example, consider this function:

```
void f(int *p)
  pre( p != nullptr )  // precondition 1
  pre( *p > 0 );       // precondition 2
```

An invocation of `f` will always evaluate precondition 1 first. After that, precondition 1 may be repeated any time later during the sequence. Precondition 2 will always be evaluated after precondition 1 has been evaluated at least once, and after that, 2 may be evaluated again as well. On many platforms, the simplest sequence 1 – 2 will be evaluated, with each precondition being evaluated exactly once and in order. In other situations, such as when function contract assertion evaluations are emitted at both the call site and within the function body, the sequence 1 – 2 – 1 – 2 will be evaluated. Beyond those most common cases, the following sequences of evaluation are conforming:

1 – 1 – 2
1 – 2 – 2
1 – 2 – 2 – 1, ...

and the following are not:

2 – 1,
2 – 2,
1,
1 – 1, ...

Repeated evaluations may also be done with different semantics, allowing a compiler to emit checks of related contracts (such as a precondition and a postcondition that relate to the same data) adjacent to one another, possibly resulting in the ability to elide one or both when they can be statically proven to hold.

Note the distinction between evaluating a contract assertion and evaluating its predicate. Evaluating a contract assertion with the *ignore* semantic also counts as an evaluation of the contract assertion, even though its predicate is not evaluated.

3.5.10 Predicate Side Effects

The predicate of a contract assertion is an expression that, when evaluated, follows the normal C++ rules for expression evaluation. The contract-assertion predicate is, therefore, allowed to have observable side effects, such as logging.

If the compiler can prove that evaluation of the predicate would result in the values `true` or `false` (i.e., it cannot throw an exception, cause a call to `longjmp`, or trigger program termination), the compiler is allowed to elide all the side effects of evaluating the predicate. In other words, the compiler may generate a side-effect-free expression that provably produces the same result as the predicate and may evaluate that expression instead of the predicate. By evaluating this replacement expression, the compiler effectively elides the evaluation of the entire predicate, resulting in no side effects of the predicate occurring. This ability to replace an expression that has side effects with one that has none applies only to the entire predicate; i.e., either all or none of the side effects of the predicate expression will be observed. The compiler also may not introduce new side effects.

As with many other allowed program transformations, this replacement of the predicate with a side-effect-free expression must be equivalent for only evaluations with well-defined behavior. In other words, the replacement predicate might have undefined behavior when the actual predicate would.

If the compiler cannot prove that evaluation of the predicate will not exit via an exception, then the compiler is not allowed to elide the evaluation of the predicate because the thrown exception must be available in the contract-violation handler (see Section 3.5.12).

Likewise, if the compiler cannot prove that evaluation of the predicate will not call `longjmp` or cause program termination, then the compiler is not allowed to elide the evaluation of the predicate because, during predicate evaluation, such calls are guaranteed to happen as normal.

Further, as described in Section 3.5.9, contract predicates may be evaluated repeatedly within a chain, even a chain of a single contract assertion. Therefore, in general, observable side effects of the predicate evaluation may happen zero, one, or many times:

```
int i = 0;
void f() pre ((++i, true));
void g() {
  f();  // i may be 0, 1, 17, etc.
}
```

If the chosen semantic for these preconditions is *observe* and the contract-violation handler returns normally on each violation, multiple violations might result:

```
int i = 0;
void f() pre ((++i, false));
void g() {
  f();  // i may be any value; the contract−violation handler
        // will be invoked at most that number of times.
}
```

In other cases, if the compiler cannot prove that `true` and `false` are the only results possible, it cannot check the contract assertion without evaluating the contract predicate. In such cases, observable side effects of the predicate evaluation must happen at least once but may happen many times:

```
int i = 0;
void f() pre ((++i, throw true));
void g() {
  f();  // i may be 1, 2, 17, etc. The same number of contract violations
        // will be reported to the contract−violation handler.
}
```

Since we cannot rely on the side effects of predicate evaluation happening any particular number of times or at all, the use of contract predicates with side effects is generally discouraged. Note that if the predicate is a side-effect-free expression, neither elision nor repetition of evaluating the predicate is observable, and a contract check that does not result in a violation is, therefore, as-if-equivalent to evaluating the predicate once.

3.5.11 The Contract-Violation Handler

The contract-violation handler is a function named `::handle_contract_violation` that is attached to the global module and has C++ language linkage. This function will be invoked when a contract violation is identified at run time.

This function

- shall take a single argument of type `const std::contracts::contract_violation&`
- shall return `void`
- may be `noexcept`

The implementation shall provide a definition of this function, which is called the *default contract-violation handler* and has implementation-defined effects. The recommended practice is that the default contract-violation handler will output diagnostic information describing the pertinent properties of the provided `std::contracts::contract_violation` object. Whether the default contract-violation handler itself is `noexcept` is implementation-defined, though the recommended implementation certainly could be.

The Standard Library provides no user-accessible declaration of the default contract-violation handler, and users have no way to call it directly. No implicit declaration of this function occurs in any translation unit, even though the function might be directly or indirectly invoked from the evaluation of any contract assertion. If a declaration were available, it would not be easily called from outside a contract-violation handler because users have no way to create `contract_violation` objects. Such a declaration would also prevent users from choosing properties of their own replacement function, such as whether it is `noexcept` or `[[noreturn]]` or whether it has its own preconditions and postconditions.

Whether `::handle_contract_violation` is replaceable is implementation-defined. When it is replaceable, that replacement is done in the same way it would be done for the global `operator new` and `operator delete`, i.e., by defining a function that has the correct signature (function name and argument types), has the correct return type, and satisfies the requirements listed above. Such a function is called a *user-defined contract-violation handler*.

A user-provided contract-violation handler may have any exception specification; i.e., it is free to be `noexcept(true)` or `noexcept(false)`. Enabling this flexibility is a primary motivation for not providing any declaration of `::handle_contract_violation` in the Standard Library; whether that declaration was `noexcept` would force that decision on user-provided contract-violation handlers, like it does for the global `operator new` and `operator delete`, which have declarations that are `noexcept` provided in the Standard Library.

On platforms where there is no support for a user-defined contract-violation handler, providing a function with the signature and return type needed to attempt to replace the default contract-violation handler is ill-formed, no diagnostic required. Platforms can, therefore, issue a diagnostic informing a user that their attempt to replace the contract-violation handler will fail on their chosen platform. At the same time, not requiring such a diagnostic allows use cases like compiling a translation unit on a platform that supports user-defined contract-violation handlers but linking it on a platform that does not, without forcing changes to the linker to detect the presence of a user-defined contract-violation handler that will not be used.

3.5.12 The Contract-Violation Handling Process

When a contract violation (see Section 3.5.8) is identified at run time, the contract-violation handling process will be invoked. An object of type `std::contracts::contract_violation` will be produced and passed to the violation handler. This object provides information about the contract violation that has occurred via a set of property functions, such as `location` (returning a `source_location` associated with the contract violation), `comment` (returning a string with a textual representation of the contract predicate), `assertion_kind` (the kind of contract assertion — `pre`, `post`, or `contract_assert`), and `semantic` (the evaluation semantic of the contract assertion that caused the contract violation). This API is described in more detail in Section 3.7.

The manner in which this `contract_violation` object is produced is unspecified other than that the memory for it is not allocated via `operator new` (similar to the memory for exception objects). This object might already exist in read-only memory, or it might be populated at run time on the stack. The lifetime of this object will continue at least through the point at which the violation handler completes execution. The same lifetime guarantee applies to any objects accessible through the `contract_violation` object's interface, such as the string returned by the `comment` property.

Further, if the contract violation was caused by the evaluation of the predicate exiting via an exception, the contract-violation handler is invoked as if from within a handler for that exception generated by the implementation. Inside the contract-violation handler, that exception is available via the member function `evaluation_exception()` of the passed-in `contract_violation` object.[19] Since the exception is considered to be handled by the contract-violation handler, it will not be rethrown automatically when the contract-violation handler returns, but the user can do so manually using `std::rethrow_exception`.

For expository purposes, assume that we can represent the process with some magic compiler intrinsics.

- `std::contracts::evaluation_semantic __current_semantic()` — Return the semantic with which to evaluate the current contract assertion. This intrinsic is `constexpr`; i.e., it may be called either during constant evaluation (see Section 3.5.14) or at run time. The result may

[19]Since the exception thrown during predicate evaluation is the currently handled exception when the contract-violation handler is called, it may also be accessed within the contract-violation handler by calling `std::current_exception()`. However, using `std::current_exception()` for this purpose can be error prone because unlike `evaluation_exception()`, which will only return a nonempty `std::exception_ptr` if the exception was thrown during predicate evaluation, `std::current_exception()` simply returns a pointer to the currently handled exception and will thus return a nonempty value even if the predicate did *not* throw if the enclosing code is handling an unrelated exception (i.e., the contract violation occurred inside a `catch` clause).

be a compile-time value (e.g., controlled by a compiler flag or a platform-specific annotation on the contract assertion) or, for a contract evaluation at run time, may even be a value determined at run time based on what the platform provides.

- `__check_predicate(`*X*`)` — Determine the result of the predicate *X* at run time either by returning `true` or `false` if the result does not need evaluation of *X* or by evaluating *X* (and thus potentially also invoking `longjmp`, terminating execution, or letting an exception escape the invocation of this intrinsic).

- `__handle_contract_violation(evaluation_semantic, detection_mode)` — Handle a runtime contract violation of the current contract. This intrinsic will produce a `contract_violation` object populated with the appropriate location and comment for the current contract, along with the specified semantic and detection mode. The lifetime of the produced `contract_violation` object and all its properties must last through the invocation of the contract-violation handler.

Building from these intrinsics, the evaluation of a contract assertion is notionally equivalent to the following exposition-only pseudocode:

```
evaluation_semantic _semantic = __current_semantic();
if (evaluation_semantic::ignore == _semantic) {
  // Do nothing.
}
else if (evaluation_semantic::observe == _semantic
      || evaluation_semantic::enforce == _semantic
      || evaluation_semantic::quick_enforce == _semantic)
{
  // checking semantic

  if consteval {
    // See Section 3.5.14.
  }
  else {
    // exposition-only variables for control flow
    bool _violation;          // Violation handler should be invoked.
    bool _handled = false;    // Violation handler has been invoked.

    // Check the predicate and invoke the violation handler if needed.
    try {
      _violation = __check_predicate(X);
    }
    catch (...) {
      if (evaluation_semantic::quick_enforce == _semantic) {
        std::terminate();   // implementation-defined program termination
      } else {
        // Handle the violation within the exception handler.
        _violation = true;
        __handle_contract_violation(_semantic,
                                    detection_mode::evaluation_exception);
        _handled = true;
      }
    }
    if (_violation && evaluation_semantic::quick_enforce == _semantic) {
      __builtin_trap();  // implementation−defined program termination
    }
    if (_violation && !_handled) {
      __handle_contract_violation(_semantic,
                                  detection_mode::predicate_false);
    }

    if (_violation && evaluation_semantic::enforce == _semantic) {
      abort();  // implementation−defined program termination
    }
  }
}
else {
  // implementation−defined _semantic
}
```

If the semantic is known at compile time to be *ignore*, the above is functionally equivalent to `sizeof( (X) ? true : false );` — i.e., the expression X is still parsed and odr-used, but it is used only on discarded branches.

The invocation of the contract-violation handler when an exception is thrown by the evaluation of the contract assertion's predicate must be done within the compiler-generated `catch` block for that exception. The invocation when *no* exception is thrown must be done *outside* the compiler-generated `try` block that would catch that exception. This behavior could be accomplished in many ways; the exposition-only boolean variables above are just one possible solution.

3.5.13 Mixed Mode

One important takeaway from having the semantic of evaluation being effectively unspecified until run time is that, unlike a macro-based solution, a contract assertion's definition — and thus the definition of the function to which it applies — is the same even though individual evaluations of that contract assertion might have different evaluation semantics. This feature means that an implementation that supports mixing translation units where contract assertions are configured to have different evaluation semantics is not, in and of itself, an odr violation.[20]

The possibility to have a well-formed program in which the same function was compiled with different evaluation semantics in different translation units (colloquially called "mixed mode") raises the question of which evaluation semantic will apply when that function is `inline` but is not actually inlined by the compiler and is then invoked. The answer is simply that we will get one of the evaluation semantics with which we compiled.

For use cases where users require strong guarantees about the evaluation semantics that will apply to inline functions, compiler vendors can add the appropriate information about the evaluation semantic as an ABI extension so that link-time scripts can select a preferred inline definition of the function based on the configuration of those definitions. We expect vendors to provide a default that

[20]Different behaviors might be observed for the same function compiled in different translation units; such behavioral differences, however, are a product of the same sequence of tokens even when they result in different generated instructions. This situation is similar to multiple versions of the same inline function being optimized differently in different TUs, which is also not an odr violation.

selects the most conservative of available definitions as well as options that allows users to define the required evaluation semantic ordering themselves. As an alternative approach, the compiler can add a hook for every contract check and then give users the option to select the desired evaluation semantic at load time or at run time.

If such deterministic selection of the evaluation semantic in "mixed mode" is not required or is desired but not possible (for example, because a user cannot afford to upgrade their linker and recompile their program), the remaining option is that the linker can simply choose either semantic. Such an implementation would be compatible with both Principle 4 (Zero Overhead) and Principle 16 (No ABI Break). In practice, this solution will often be good enough. The only failure mode of such an implementation is that a contract check that was expected does not happen. For most use cases, this failure mode will be much better than undefined behavior, IFNDR, or requiring linker upgrades before we can use Contracts at all.

3.5.14 Compile-Time Evaluation

Contract assertions may be evaluated during constant evaluation (at compile time). During constant evaluation, the four possible evaluation semantics have the following meanings.

- *ignore* — Nothing happens during constant evaluation; the contract expression must still be a valid expression that might odr-use other entities.
- *observe* — Constant-evaluate the predicate; if a contract violation occurs, a diagnostic (warning) is emitted.
- *enforce* and *quick_enforce* — Constant-evaluate the predicate; if a contract violation occurs, the program is ill-formed.

Constant evaluation of the predicate can have one of three possible outcomes.

1. The result is `true`. — No contract violation.
2. The result is `false`. — Contract violation.
3. The predicate is not a core constant expression. — Contract violation.

To help satisfy Design Principle 3 (Concepts Do Not See Contracts), the mere presence of a contract assertion should not alter whether containing expressions are or are not eligible to be constant expressions, particularly because it is possible to SFINAE on whether an expression is a core constant expression. Therefore, evaluating a contract assertion in and of itself never makes an expression ineligible to be a core constant expression, although its predicate being ineligible to be evaluated will result in a contract violation.[21]

A special rule is applied to potentially constant variables that are not `constexpr`, such as variables with static or thread storage duration and non-`volatile` `const`-qualified variables of integral or enumeration type. Such variables may be constant-initialized (at compile time) or dynamically initialized (at run time) depending on whether the initializer is a core constant expression:

[21]This situation is conceptually somewhat similar to evaluation of the predicate exiting with an exception and possibly occurs when the actual plain-language contract has not been violated, but we cannot tell because we cannot evaluate the contract predicate. We still treat this case as a compile-time contract violation.

```
int compute_at_runtime(int n);  // not constexpr

constexpr int compute(int n) {
  return n == 0 ? 42: compute_at_runtime(n);
}

void f() {
  const int i = compute(0); // constant initialization
  const int j = compute(1); // dynamic initialization
}
```

In such cases, the compiler first determines whether the initializer is a core constant expression by performing trial evaluation[22] with all contract assertions *ignored*. (Therefore, contract assertions cannot trigger a contract violation during trial evaluation or otherwise influence the determination performed by the trial evaluation.) If and only if this trial evaluation determines that the expression is a core constant expression, then the variable is constant-initialized and its initializer is now a manifestly constant-evaluated context.

For any manifestly constant-evaluated context (including the initialization of `constexpr` variables, template parameters, array bounds, and variables where trial evaluation has determined that the variable is constant-initialized), the expression is then evaluated *with* the contract assertions having the semantics *ignore*, *observe*, *enforce*, or *quick_enforce* chosen in an implementation-defined manner. This evaluation behaves normally with regard to possible contract violations.

This rule is again derived from Design Principle 4 (Zero Overhead). In the example above, adding a contract assertion to `compute` (i.e., when called with `0`) must not silently flip the initialization of `i` from constant to dynamic, thereby changing the semantics of the program. By the same token, if `compute` is already *not* a core constant expression and is evaluated at run time (i.e., when called with a value other than `0`), a contract assertion must not lead to it instead being evaluated at compile time and causing a compile-time contract violation. This rule avoids aggressive enforcement of contract checks at compile time for functions that would otherwise be evaluated at run time (at which point the contract check might succeed). Consider adding the following precondition assertion:

```
constexpr int compute(int n)
  pre (n == 0 || !std::is_constant_evaluated())  // passes for both i and j
{
  return n == 0 ? 42: compute_at_runtime(n);
}

void f() {
  const int i = compute(0); // constant initialization
  const int j = compute(1); // dynamic initialization
}
```

The above precondition check would fail for `j` if it were evaluated at compile time. However, `compute` is not evaluated at compile time for `j` because trial evaluation (which does not consider contract annotations) determines that `compute(1)` is not a core constant expression (due to the

[22]Trial evaluation is performed notionally (as specified in [expr.const]). In practice, an implementation is allowed to perform the constant evaluation of the initializer in one step as long as the result is the same.

call to `compute_at_runtime`), and `j` will, therefore, be initialized at run time, at which point the precondition passes. The above program, therefore, contains no contract violations.

The program is ill-formed if trial evaluation (with all contract assertions *ignored*) determines that the initializer is a core constant expression, the variable is constant-initialized with all contract assertions checked in a manifestly constant-evaluated context, and any such constant-evaluated predicate then causes the initializer to no longer be a core constant expression:

```
constexpr int foo(int i) {
    return i == 0  ? 0 : throw 0; // error: not a core constant expression
}

constexpr int bar(int * p)
    pre((*p = 1)) {
    return foo(*p);
}

constexpr int baz(int i) {
    return bar(&i);
}

static int x = baz(0);   // constant initialization
```

The rules regarding elision and duplication of side effects described in Section 3.5.10 apply equally during constant evaluation:

```
constexpr int f(int i)
  pre ((++const_cast<int&>(i), true)) {
  return i;
}

inline std::size_t g() {
  int a[f(0)];
  return a.size();      // may be 0, 1, 17, etc.
}
```

In the above example, different translation units might have different declarations for the array `a`, resulting in multiple distinct definitions — an odr violation — for the function `g`. Considering that such odr violations happen only when function contract assertions are already unwisely jumping through `const_cast` hoops to modify function parameters, this is a recognized but insignificant concern. Note further that even without the possibility to elide or duplicate side effects, the odr violation would still occur because the type of `a` would still depend on whether the contract assertion would be evaluated with a checking or non-checking evaluation semantic when determining the size of the array `a`.

Finally, note that none of these rules apply when constant evaluation is not semantically possible, even if a compiler might fold such evaluations into constants under the *as-if* rule:

```
constexpr int f(int x) pre( x > 0 );
void g()
{
  int bad = f(0);   // always initialized at run time, violates the precondition
}
```

3.6 Noteworthy Design Consequences

3.6.1 Undefined Behavior During Contract Checking

Following Design Principle 13 (Explicitly Define All New Behavior), the design of this proposal has deliberately not introduced any new explicitly undefined behavior into the C++ language and, we hope, does not introduce any other undefined behavior through new holes in the specification. At the same time, contract assertions are not immune to undefined behavior that arises due to existing C++ rules.

Though making contract assertions observable checkpoints (see Section 3.5.5) mitigates certain unwanted time-travel optimizations and elisions of observed contract assertions due to undefined behavior *outside* of contract assertions, no special protection is offered against undefined behavior *inside* a contract-assertion check since contract predicates are normal C++ expressions and, therefore, follow the normal rules for C++ expressions when evaluated. In other words, if a contract assertion is evaluated with a checking semantic and the resulting predicate evaluation has undefined behavior, then the evaluation of the contract assertion itself has undefined behavior. Consider:

```
int f(int a) { return a + 100; }
int g(int a) pre (f(a) > a);
```

In this program, the compiler is allowed to assume that the signed integer addition inside `f` will never overflow (because this would be undefined behavior) and replace the precondition assertion of `g` with `pre(true)`, or in other words, elide the precondition assertion entirely, even if the evaluation semantic is *enforce* or *quick_enforce*.

3.6.2 Constructors and Destructors

Constructors and destructors both follow the same rules as those for regular function invocations such that precondition and postcondition assertions are evaluated as control transfers in and out of the constructor or destructor. Clarity about what this means is important.

Two cases are worth calling out because they provide a place where user-provided code will be evaluated where none was explicitly possible before.

1. The precondition assertions of a constructor are evaluated before the complete function body, which includes the function-try block and member initializer list.
2. The postcondition assertions of a destructor are evaluated before returning to the caller and thus occur after the destruction of all members and base classes.

During the above situations, members, bases, and the object itself are not within their lifetimes; accessing any of these or doing anything that depends on the dynamic type of these objects (such as `dynamic_cast`, `typeid`, invoking a virtual member function, or accessing a member of a virtual base class) will, therefore, have undefined behavior. The value of `this` as a location for the storage of the object about to be constructed or already destroyed is still, however, quite useful for many contract assertions.

For the remaining function contract assertions of constructors and destructors (postconditions of constructors and preconditions of destructors), the dynamic type of `this` is not known. When evaluating these function contract assertions, the same rules for the dynamic type apparent during the constructor or destructor body apply to the function contract assertion, namely that the type will be that of the constructor's or destructor's class, not the class of the complete object:

```
struct B { virtual ~B(); }  // polymorphic base

template <typename Base>
struct D : public Base {};  // generic derived class

struct C : public B {
  C()
    post( typeid(*this) == typeid(C) )            // Type is always C here.
    post( dynamic_cast<C*   >(this) == this )     // This dynamic_cast works.
    post( dynamic_cast<D<C>*>(this) == nullptr ); // never derived class here

  ~C()
    pre( typeid(*this) == typeid(C) )             // same as above
    pre( dynamic_cast<C*   >(this) == this )
    pre( dynamic_cast<D<C>*>(this) == nullptr );
};
```

3.6.3 Friend Declarations Inside Templates

As described in Section 3.3.1, if a function has function contract assertions, then the function contract specifiers introducing these assertions need to be placed on every first declaration (i.e., every declaration from which no other declaration is reachable) but can be omitted on redeclarations. However, in certain situations, reasoning about which declarations are first declarations and which are redeclarations can be difficult because the notion of first declaration is defined via reachability and has nothing to do with which declaration appears lexically first in a given translation unit. One particularly interesting case are friend declarations inside templates.

According to the existing language rules for templates, a friend declaration of a function inside a template becomes reachable only from the point at which the template is instantiated. Consider a program that has multiple templates declaring the same function as a friend and a separate declaration of that function, all located in different headers:

```
// x.h
template <typename T>
struct X {
  friend void f() pre (x);  // 1
};

// y.h
template <typename T>
struct Y {
  friend void f() pre (x);  // 2
};

// f.h
void f() pre (x);  // 3
```

Now consider an implementation file that makes use of these headers:

```
#include <x.h>
#include <y.h>
int g() {
  Y<int>   y1;  // 4
  Y<long>  y2;  // 5
  X<int>   x;   // 6
}
#include <f.h>
```

A number of things worth noting happen here.

- At 4, the definition of `Y<int>` is instantiated, and the friend declaration located at 2 is instantiated as part of that friend declaration. Since no other definition of `f` is reachable at this point, 2 is a first declaration for `f`.
- At 5, the definition of `Y<long>` is instantiated, and the friend declaration located at 2 is instantiated again, this time as a redeclaration of `f`. Since `f` has a precondition specifier, that specifier is compared to the previous declaration of `f`, and we determine that the specifiers match. These are, after all, from the same line of code.
- At 6, the definition of `X<int>` is instantiated, and the friend declaration located at 1 is instantiated. This is a redeclaration since the two declarations instantiated from 2 are both reachable.
- At 3, included after the definition of `g`, we finally have a namespace-scope declaration of `f` with three reachable declarations of `f` appearing prior to it in our translation unit, and thus they must match.

Another translation unit might instantiate `X` and `Y` in different orders, resulting in 1 potentially being a first declaration. Including `<f.h>` prior to `<x.h>` and `<y.h>` will result in the declaration at 3 always being the first declaration. Thus, the small change of adding `#include <f.h>` to the start of `x.h` and `y.h` will result in 3 always being the first declaration across all translation units.

If the precondition specifier is omitted from any declaration of `f` that might be a first declaration in some translation unit, then the program will be ill-formed (unless the precondition specifier is removed from *all* declarations of `f`). If that same translation unit includes a declaration with the precondition specifier later, a diagnostic is required; otherwise, it is not.

To avoid cases that make compiling correctly in all contexts challenging for a template, always doing one of the following is recommended when using a friend declaration of a function with function contract assertions inside a template.

- Befriend functions that have reachable declarations, such that the friend declaration will always be a redeclaration.
- Duplicate the function contract specifiers on each friend declaration.
- Make the function a hidden friend; i.e., the friend declaration is the only declaration of the function and is also a definition.

3.6.4 Recursive Contract Violations

No dispensation is provided to disable contract checking during the evaluation of a contract assertion's predicate or the evaluation of the contract-violation handler; in both cases, contract checks behave as usual. Therefore, if a contract-violation handler calls a function containing a contract assertion that is violated and this contract assertion is evaluated with a checking semantic, the contract-violation handler will be called recursively.

A user-defined contract-violation handler is responsible for handling recursive violations explicitly if the user wishes to avoid overflowing the call stack. Identifying and preventing such recursion would require the overhead of a thread-local variable, so we do not impose such additional complexity on all users of the Contracts facility. A user-defined contract-violation handler could, however, prevent such recursion:

```
void handle_contract_violation(const contract_violation& violation)
{
  thread_local bool handling = false;
  if (handling) {
    // violation encountered recursively.
    std::abort();
  }
  handling = true;

  // ... Do what needs to be done on a violation.

  handling = false;
}
```

3.6.5 Concurrent Contract Violations

The violation-handling process does nothing to prevent the contract-violation handler from being invoked multiple times concurrently. The default contract-violation handler must be safe in such scenarios, and any user-provided contract-violation handler is responsible for being similarly safe when invoked concurrently.

Note that even when contract assertions are enforced, one thread can possibly encounter a prior contract violation while another thread is actively executing the contract-violation handler and has not yet reached termination.

Just as with preventing recursion, preventing concurrent invocation would require potentially significant overhead in the contract-violation handling process and should not be imposed on all programs.

3.6.6 Throwing Violation Handlers

No restrictions are placed on what a user-defined contract-violation handler is allowed to do. In particular, a user-defined contract-violation handler is allowed to exit other than by returning, e.g., terminating, calling `longjmp`, and so on. In all cases, evaluation happens as described above. The same applies to the case in which a user-defined contract-violation handler that is not `noexcept` throws an exception:

```
void handle_contract_violation(const std::contracts::contract_violation& v) {
  throw my_contract_violation_exception(v);
}
```

Such an exception will escape the contract-violation handler and unwind the stack as usual until it is caught or control flow reaches a `noexcept` boundary. Such a contract-violation handler, therefore, bypasses the termination of the program that would occur when the contract-violation handler returns from a contract-assertion evaluation with the *enforce* semantic.

For contract violations inside function contract assertions, the contract-violation handler is treated as if the exception had been thrown inside the function body. Therefore, if the function in question is `noexcept`, a user-defined contract-violation handler that throws an exception from a precondition or postcondition check results in `std::terminate` being called, regardless of whether the semantic is *enforce* or *observe*.

3.6.7 Differences Between Contract Assertions and the `assert` Macro

Contract assertions are not designed as a drop-in replacement for the `assert` macro or similar assertion macros. Apart from the obvious difference that `pre` and `post` are part of a function declaration, which is not possible with a macro, even `contract_assert` behaves differently from `assert` in numerous ways.

First, macro `assert` can be used as an expression:

```
const int j = (assert(i > 0), i);
```

On the other hand, `contract_assert` is a statement. A possible workaround is to wrap `contract_assert` into an immediately invoked lambda, which makes it usable in places that require an expression (see Section 3.2.2):

```
const int j = ([i]{ contract_assert(i > 0); }(), i);
```

or, perhaps more idiomatically,

```
const int j = [i]{ contract_assert(i > 0); return i; }();
```

In other cases, such usages of assertions are better expressed with a precondition assertion. For example, an assertion subexpression in the member initializer list of a constructor can be better expressed with a precondition assertion on that constructor.

Second, entities declared outside a contract assertion are implicitly `const` (see Section 3.4.2) when referenced by name to discourage contract predicates that have observable side effects. One consequence is that predicates that attempt to modify a variable will compile in an `assert` macro but not in a contract assertion. Further, due to the implicit `const`, the predicate in a contract assertion can yield different overload resolution results (and thus semantics) from the predicate in a `assert` macro. A possible workaround for both issues is to use `const_cast`.

Third, in a disabled `assert` macro (when `NDEBUG` is defined), all tokens are simply removed by the processor. On the other hand, contract assertions having the *ignore* semantic do not evaluate any code, yet the predicate expression is still parsed and the entities inside are odr-used (see

Section 3.5.6). Therefore, in a contract assertion, the predicate always needs to be a well-formed, evaluable expression, even if checks are disabled. The primary benefit of this behavior is that the code within the contract assertion cannot become uncompilable at any time — a common problem with macro-based assertion facilities that can lead to libraries in which too much technical debt prevents any attempt to re-enable assertions after a period of unuse. In addition, treating the predicate consistently, independent of the semantic with which it is evaluated, helps to ensure that we do not need to treat distinct choices of semantics as an odr violation.

Fourth, with macro `assert`, entities can be declared, using an `#ifndef NDEBUG` block, such that they will exist only when checks are enabled:

```
#ifndef NDEBUG
  DebugThingy myDebugThingy;
#endif
  // ...
  assert(myDebugThingy.ok());
```

On the other hand, the Contracts facility (currently) provides no mechanism to introduce declarations of variables or other code that is conditional on whether contract checks are enabled or on whether a particular contract assertion will be checked. Following Design Principle 5, programmatically detecting the evaluation semantic of any contract assertion is discouraged; therefore, no explicit facility for such detection is provided (although constructing programs that achieve this effect remains possible). We thus minimize the likelihood that a contract assertion will end up modifying the compile-time semantics of the program it is supposed to observe. That said, we do expect a future extension proposal to offer an alternative mechanism for providing code that supports the evaluation of contract assertions in a similar fashion to blocks guarded by the preprocessor in current usages of `assert` while being compatible with the above design principle.

Fifth, the predicate in an `assert` macro is evaluated either zero times (when `NDEBUG` is defined) or exactly once (when it is not). On the other hand, contract assertions do not provide such a guarantee: Checked predicates might be evaluated any number of times (see Sections 3.5.9 and 3.5.10). Therefore, depending on the side effects within a contract assertion happening exactly once when the contract assertion is checked is not a correct use of the proposed Contracts facility.

Consider how we might use an `assert` macro to both increment a counter and check that it is within some range, like in the following example (a paraphrased code snippet from Clang):

```
#ifndef NDEBUG
  unsigned nIter = 0;
#endif
  while (keepIterating()) {
    assert(++nIter < 6);  // A bug occurs if we end up iterating more than 6 times.
    // ...
  }
```

The above example would not compile, for several reasons, with the facility proposed here: As mentioned above, we provide no mechanism to conditionally control the declaration of variables such as `nIter` based on whether a particular contract assertion will be evaluated, and in addition, an attempt to modify the counter in a `contract_assert` would require a `const_cast` to perform the modification. But more importantly, attempting to perform a side effect in a contract-assertion evaluation that is depended on in subsequent evaluations is ill-advised since whether or how many times such a side effect might occur is not guaranteed. Instead, the appropriate transformation is to move the maintenance of values upon which the assertion depends to outside the assertion itself such that the predicate of the assertion becomes free of side effects:

```
unsigned nIter = 0;
while (keepIterating()) {
  ++nIter;
  assert(nIter < 6);  // A bug occurs if we end up iterating more than 6 times.
  // ...
}
```

If needed, backward-compatibility with the behavior of the `assert` macro can be achieved for such cases via an alternate macro that evaluates the expression outside the contract assertion and has the same relationship to `NDEBUG` as the existing `assert` macro, while still consistently tying into the Contracts facility proposed here:

```
#ifndef NDEBUG
 #define MY_ASSERT(X) [](const bool b){ contract_assert(b); }(X)
#else
 #define MY_ASSERT(X) static_cast<void>(0)
#endif
```

The trade-off of the above macro is that information about the predicate expression `X` will not be propagated to the contract-violation handler, although an implementation providing extra platform-specific mechanisms to achieve the same behavior with better diagnostics does seem feasible.

3.7 Standard Library API

3.7.1 The `<contracts>` Header

A new header, `<contracts>`, is added to the C++ Standard Library. The facilities provided in this header are all freestanding. They have a specific intended usage audience: those writing user-defined contract-violation handlers and, in future extensions, other functionality for customizing the behavior of the Contracts facility in C++. Because these uses are intended to be infrequent, everything in this header is declared in namespace `std::contracts` rather than namespace `std`. In particular, including the `<contracts>` header is unnecessary for writing contract assertions.

The `<contracts>` header provides the following types and functions:

```
// all freestanding
namespace std::contracts {

  enum class assertion_kind : unspecified {
    pre = 1,
    post = 2,
    assert = 3
    /* to be extended with implementation-defined values and by future extensions */
    /* Implementation-defined values should have a minimum value of 1000. */
  };
```

```
    enum class evaluation_semantic : unspecified {
      ignore = 1,
      observe = 2,
      enforce = 3,
      quick_enforce = 4,
      // assume = 5        // expected as a future extension
      /* to be extended with implementation-defined values and by future extensions */
      /* Implementation-defined values should have a minimum value of 1000. */
    };

    enum class detection_mode : unspecified {
      predicate_false = 1,
      evaluation_exception = 2,
      /* to be extended with implementation-defined values and by future extensions */
      /* Implementation-defined values should have a minimum value of 1000. */
    };

    class contract_violation {
      // no user-accessible constructor; cannot be copied, moved, or assigned to
    public:
      const char* comment() const noexcept;
      std::contracts::detection_mode detection_mode() const noexcept;
      std::exception_ptr evaluation_exception() const noexcept;
      bool is_terminating() const noexcept;
      assertion_kind kind() const noexcept;
      source_location location() const noexcept;
      evaluation_semantic semantic() const noexcept;
    };

    void invoke_default_contract_violation_handler(const contract_violation&);

  }
```

3.7.2 Enumerations

Each enumeration used for values of the `contract_violation` object's properties is defined in the `<contracts>` header. All use `enum class`. The underlying type is unspecified but must be large enough to hold all possible values, including any implementation-defined extension values.

Fixed values for each enumerator are standardized to allow for portability, particularly for those logging these values without the step of converting them to human-readable enumerator names.

The following enumerations are provided.

- `enum class assertion_kind :` *`unspecified`* — Identifies one of the three potential kinds of contract assertion, with implementation-defined alternatives a possibility when the contract-violation handler is invoked outside the purview of a contract assertion with one of those kinds:

  - `pre` — A precondition assertion
  - `post` — A postcondition assertion
  - `assert` — An assertion statement

  Implementation-defined values indicate other kinds of contract assertions that may be available as a vendor extension.

  Note that the enumerators `pre` and `post` match the contextual keyword that introduces the respective contract-assertion kind; however, assertions use `assert` for the enumerator but `contract_assert` for the keyword as the latter needs to be a full keyword and, therefore, cannot be used as an enumerator name. Though the `assert` enumerator might appear to be in conflict with the function-like macro of the same name defined in `<cassert>`, no issues will arise in practice since the enumerator will not be used immediately prior to an opening parenthesis and, therefore, will not be expanded as the function-like macro. Using `precondition` and `postcondition` has been explicitly avoided because those terms refer to conditions based on responsibility (inside and outside of the function; see Section 2.1) and not those based on points in time of checking.

- `enum class evaluation_semantic :` ***`unspecified`*** — A reification of the evaluation semantic that can be chosen for the evaluation of a contract assertion:
  - `ignore` — the *ignore* semantic
  - `observe` — the *observe* semantic
  - `enforce` — the *enforce* semantic
  - `quick_enforce` — the *quick_enforce* semantic

  Implementation-defined values indicate other evaluation semantics that may be available as a vendor extension.

  Note that the enumeration `evaluation_semantic` provides enumerators for all four proposed evaluation semantics, even though only *observe* and *enforce* can result in the invocation of the contract-violation handler and, therefore, only the `observe` and `enforce` enumerators can occur inside that handler. The reason is that `evaluation_semantic` is provided for logging and as a vocabulary type to denote evaluation semantics in other contexts, such as vendor-specific attributes on contract assertions.

- `enum class detection_mode :` ***`unspecified`*** — An enumeration to identify the various mechanisms via which a contract violation might be identified and the contract-violation handling process might be invoked at run time:
  - `predicate_false` — To indicate that the predicate either was evaluated and produced a value of `false` or the predicate would have produced a value of `false` if it were evaluated
  - `evaluation_exception` — To indicate that the predicate was evaluated and that evaluation exited via an exception

  Implementation-defined values indicate an alternate method provided by the implementation in which a contract violation was identified.

For all the above enumerations, any implementation-defined enumerators should have a minimum value of `1000` and a name that is an identifier reserved for the implementation (starting with double underscore or underscore followed by a capital letter) to avoid possible name clashes with enumerators newly introduced in a future Standard.

3.7.3 The Class `std::contracts::contract_violation`

The `contract_violation` object is provided to the `handle_contract_violation` function when a contract violation has occurred at run time. This object cannot be constructed, copied, moved, or mutated by the user. Whether the object is polymorphic is implementation-defined; if it is polymorphic, the primary purpose in being so is to allow for the use of `dynamic_cast` to identify whether the provided object is an instance of an implementation-defined subclass of `std::contracts::contract_violation`.

The various properties of a `contract_violation` object are all accessed by `const`, non-`virtual` member functions (not as named member variables) to maximize implementation freedom.

Each contract-violation object has the following properties.

- `const char* comment() const noexcept` — The value returned should be a null-terminated multi-byte string (NTMBS) in the ordinary literal encoding; it is otherwise unspecified. We recommend that this value contain a textual representation of the predicate of the contract assertion that has been violated. Providing the empty string, a pretty-printed, truncated or otherwise modified version of the predicate, or some other message intended to identify the contract assertion for the purpose of aiding in diagnosing the bug are all conforming implementations. A conforming implementation may also allow users to select a mode where an empty string is returned, in which case we could assume that this information is not present in generated object files and executables.

- `std::contracts::detection_mode detection_mode() const noexcept` — The method by which a violation of the contract assertion was identified.

- `std::exception_ptr evaluation_exception() const noexcept` — If the contract violation occurred because predicate evaluation exited via an exception, the value returned is a pointer to that exception; otherwise, the value returned is an empty `std::exception_ptr`.

- `bool is_terminating() const noexcept` — `true` if the current evaluation semantic is a terminating semantic, i.e., if the contract-violation handling mechanism will attempt to terminate the program after the contract-violation handler has returned; `false`, otherwise. Note that `evaluation_semantic()` cannot be portably used to determine whether the current evaluation semantic is a terminating semantic as it returns an open-ended enum.

- `assertion_kind kind() const noexcept` — The kind of the contract assertion that has been violated.

- `std::source_location location() const noexcept` — The value returned is unspecified. That the value be the source location of the caller of a function when a precondition is violated is recommended. For other contract assertion kinds or when the location of the caller is not used, we recommend that the source location of the contract assertion itself is used. Returning a default-constructed `source_location` or some other value are all conforming implementations.

A conforming implementation may also allow users to select a mode based on whether a meaningful value or a default-constructed value is returned.

- `evaluation_semantic semantic() const noexcept` — The semantic with which the violated contract assertion was being evaluated.

3.7.4 The Function `invoke_default_contract_violation_handler`

The Standard Library provides a function, `invoke_default_contract_violation_handler`, which has behavior matching that of the default contract-violation handler. This function is useful if the user wishes to fall back to the default contract-violation handler after having performed some custom action (such as additional logging).

`invoke_default_contract_violation_handler` takes a single argument of type lvalue reference to `const contract_violation`. Since such an object cannot be constructed or copied by the user and is provided only by the implementation during contract-violation handling, this function can be called only during the execution of a user-defined contract-violation handler.

`invoke_default_contract_violation_handler` is not specified to be `noexcept`. However, just like with all other functions in the Standard Library that are known to never throw an exception, a conforming implementation is free to add `noexcept` to this function if it is known that, on this implementation, the default contract-violation handler will never throw an exception.

3.7.5 Standard Library Contracts

We do not propose any changes to the specification of existing Standard Library facilities to mandate the use of the Contracts facility (e.g., to check the preconditions and postconditions specified for Standard Library functions), but such use should be permitted. Given that a violation of a precondition when using a Standard Library function is undefined behavior, Standard Library implementations are already free to choose to use the Contracts facility themselves as soon as it is available.

Note that Standard Library implementers and compiler implementers must work together to make use of contract assertions on Standard Library functions. Currently, compilers, as part of the platform defined by the C++ Standard, take advantage of knowledge that certain Standard Library invocations are undefined behavior. Such optimizations must be skipped to meaningfully evaluate a contract assertion when that same contract has been violated. This agreement between library implementers and compiler vendors is needed because, as far as the Standard is concerned, they are the same entity and provide a single interface to users.

4 Proposed Wording

The wording below serves to formally specify the design described in Section 3. In the case of divergence or contradiction between the design description in Section 3 and the wording, the design intent is determined by the design description in Section 3.

The proposed changes are relative to the C++26 working draft [N4981] and [P1494R3].

Modify [intro.compliance], paragraph 2:

- [...]
- Otherwise, if a program contains
  - a violation of any diagnosable rule,
  - a preprocessing translation unit with a `#warning` preprocessing directive ([cpp.error]), ~~or~~
  - an occurrence of a construct described in this document as "conditionally-supported" when the implementation does not support that construct, or
  - a contract assertion ([basic.contract.eval]) evaluated with a checking semantic in a manifestly constant-evaluated context resulting in a contract violation,

  a conforming implementation shall issue at least one diagnostic message.

  [*Note:* During template argument deduction and substitution, certain constructs that in other contexts require a diagnostic are treated differently; see [temp.deduct]. — *end note*]

  Furthermore, a conforming implementation shall not accept

- a preprocessing translation unit containing a `#error` preprocessing directive ([cpp.error]), ~~or~~
- a translation unit with a `static_assert`-declaration that fails ([dcl.pre]), or
- a contract assertion ([basic.contract.eval]) evaluated with the enforce or quick_enforce semantic in a manifestly constant-evaluated context resulting in a contract violation.

Modify the [intro.abstract] paragraph introduced by [P1494R3] before paragraph 5:

> Certain events in the execution of a program are termed *observable checkpoints*. Program termination is one such. [*Note:* A call to `std::observable` ([support.start.term]) is also an observable checkpoint as are certain parts of the evaluation of contract assertions ([basic.contract]). — *end note*]

Modify [lex.name], Table 4: Identifiers with special meaning:

`[...]`
`override`
`post`
`pre`

Modify [lex.key], Table 5: Keywords:

`[...]`
`continue`
`contract_assert`
`co_await`
`[...]`

Modify [basic.pre], paragraph 5:

Every name is introduced by a declaration, which is a

- [...]
- *exception-declaration* ([except.pre]),~~ or~~
- implicit declaration of an injected-class-name ([class.pre]), or
- *result-name-introducer* in a postcondition assertion ([dcl.contract.res]).

Modify [basic.def], paragraph 1:

A declaration may (re)introduce one or more names and/or entities into a translation unit. If so, the declaration specifies the interpretation and semantic properties of these names. A declaration of an entity or *typedef-name* X is a redeclaration of X if another declaration of X is reachable from it ([module.reach]); otherwise, it is a first declaration.

Modify [basic.def], paragraph 2:

Each entity declared by a declaration is also defined by that declaration unless

- [...]
- It is a *static_assert-declaration* ([dcl.pre]),
- It is a *result-name-introducer* ([dcl.contract.res]),
- It is an *attribute-declaration* ([dcl.pre]),
- [...]

Modify [basic.def.odr], paragraph 10:

A local entity ([basic.pre]) is *odr-usable* in a scope ([basic.scope.scope]) if

- either the local entity is not `*this` or an enclosing class or non-lambda function parameter scope exists and, if the innermost such scope is a function parameter scope, it corresponds to a non-static member function, and
- for each intervening scope ([basic.scope.scope]) between the point at which the entity is introduced and the scope (where `*this` is considered to be introduced within the innermost enclosing class or non-lambda function definition scope), either:
  - the intervening scope is a block scope, or
  - the intervening scope is a contract-assertion scope ([basic.scope.contract]), or

— the intervening scope is the function parameter scope of a *lambda-expression*, or

— the intervening scope is the lambda scope of a *lambda-expression* that has a *simple-capture* naming the entity or has a *capture-default*, and the block scope of the *lambda-expression* is also an intervening scope.

Modify [basic.scope], paragraph 1:

The declarations in a program appear in a number of *scopes* that are in general discontiguous. The *global* scope contains the entire program; every other scope S is introduced by a declaration, parameter-declaration-clause, statement, ~~or~~ handler, or contract assertion (as described in the following subclauses of [basic.scope]) appearing in another scope which thereby contains S. An *enclosing scope* at a program point is any scope that contains it; the smallest such scope is said to be the *immediate scope* at that point. A scope *intervenes* between a program point P and a scope S (that does not contain P) if it is or contains S but does not contain P.

Add a new paragraph after [basic.scope.decl], paragraph 13:

The locus of the *result-name-introducer* in a postcondition assertion ([dcl.contract.res]) is immediately after it.

Add a new section after [basic.scope.temp]:

Contract-assertion scope **[basic.scope.contract]**

Each contract assertion ([basic.contract]) introduces a *contract-assertion scope* that includes its *conditional-expression*.

If a *result-name-introducer* ([dcl.contract.res]) potentially conflicts with a declaration whose target scope is the parameter scope or, if associated with a *lambda-declarator*, the nearest enclosing lambda scope of the contract assertion, the program is ill-formed.

Modify [basic.stc.dynamic.general], paragraph 2:

The library provides default definitions for the global allocation and deallocation functions. Some global allocation and deallocation functions ([new.delete]) are replaceable ~~([new.delete])~~([dcl.fct.def.replace]); these are attached to the global module ([module.unit]). ~~A C++ program shall provide at most one definition of a replaceable allocation or deallocation function. Any such function definition replaces the default version provided in the library ([replacement.functions]).~~ The following allocation and deallocation functions ([support.dynamic]) are implicitly declared in global scope in each translation unit of a program.

Modify [basic.stc.dynamic.allocation], paragraph 5:

A global allocation function is only called as the result of a new expression ([expr.new]), or called directly using the function call syntax ([expr.call]), or called indirectly to allocate storage for a coroutine state ([dcl.fct.def.coroutine]), or called indirectly through calls to the functions in the C++ standard library.

[*Note:* In particular, a global allocation function is not called to allocate storage for objects with static storage duration ([basic.stc.static]), for objects or references with thread storage duration ([basic.stc.thread]), for objects of type `std::type_info` ([expr.typeid]), for an object of type `std::contracts::contract_violation` when a contract violation occurs ([basic.contract.eval]), or for an exception object ([except.throw]). — *end note*]

Modify [intro.execution], paragraph 3:

The *immediate subexpressions* of an expression E are

— the constituent expressions of E's operands ([expr.prop]),

— any function call that E implicitly invokes,

— if E is a *lambda-expression* ([expr.prim.lambda]), the initialization of the entities captured by copy and the constituent expressions of the *initializer* of the *init-capture*s,

— if E is a function call ([expr.call]) or implicitly invokes a function, the constituent expressions of each default argument ([dcl.fct.default]) used in the call and the predicates of any contract assertions in the function contract assertions of that function call ([basic.contract]), or

— if E creates an aggregate object ([dcl.init.aggr]), the constituent expressions of each default member initializer ([class.mem]) used in the initialization.

Modify [intro.execution], paragraph 5:

A *full-expression* is

— an unevaluated operand ([expr.context]),

— a *constant-expression* ([expr.const]),

— an immediate invocation ([expr.const]),

— an *init-declarator* ([dcl.decl]) or a *mem-initializer* ([class.base.init]), including the constituent expressions of the initializer,

— an invocation of a destructor generated at the end of the lifetime of an object other than a temporary object ([class.temporary]) whose lifetime has not been extended, ~~or~~

— the predicate of a contract assertion ([basic.contract]), or

— an expression that is not a subexpression of another expression and that is not otherwise part of a full-expression.

...

Modify [intro.execution], paragraph 11, and split into multiple paragraphs as follows:

[11] When invoking a function `f` (whether or not the function is inline), every argument expression and the postfix expression designating `f` ~~the called function~~ are sequenced before every precondition assertion of the function call ([expr.call]), which in turn are sequenced before every expression or statement in the body of `f`, which in turn are

sequenced before every postcondition assertion of the function call.~~the called function. For each function invocation or evaluation of an *await-expression* F, each evaluation that does not occur within F but is evaluated on the same thread and as part of the same signal handler (if any) is either sequenced before all evaluations that occur within F or sequenced after all evaluations that occur within F; if F invokes or resumes a coroutine ([expr.await]), only evaluations subsequent to the previous suspension (if any) and prior to the next suspension (if any) are considered to occur within F.~~

Several contexts in C++ cause evaluation of a function call, even though no corresponding function call syntax appears in the translation unit.

[*Example:* Evaluation of a *new-expression* invokes one or more allocation and constructor functions; see [expr.new]. For another example, invocation of a conversion function ([class.conv.fct]) can arise in contexts in which no function call syntax appears. — *end example*]

The sequencing constraints on the execution of the called function (as described above) are features of the function calls as evaluated, regardless of the syntax of the expression that calls the function.

[12] For each function invocation or evaluation of an *await-expression* F, each evaluation that does not occur within F but is evaluated on the same thread and as part of the same signal handler (if any) is either sequenced before all evaluations that occur within F or sequenced after all evaluations that occur within F; if F invokes or resumes a coroutine ([expr.await]), only evaluations subsequent to the previous suspension (if any) and prior to the next suspension (if any) are considered to occur within F.

Add a new subclause after [basic.exec]:

Contract assertions [basic.contract]

General [basic.contract.general]

Contract assertions allow the programmer to specify states of the program that are considered incorrect at certain points in the program execution. Contract assertions are introduced by *precondition-specifier*s, *postcondition-specifier*s ([dcl.contract.func]), and *assertion-statement*s ([stmt.contract.assert]).

The *conditional-expression* of a *precondition-specifier*, *postcondition-specifier*, or *assertion-statement* is contextually converted to `bool` ([conv.general]); the converted expression is called the *predicate* of the corresponding contract assertion.

An invocation of the macro `va_start` ([cstdarg.syn]) shall not be a subexpression of the predicate of a contract assertion, no diagnostic required.

[*Note:* Within the predicate of a contract assertion, *id-expression*s referring to variables with automatic storage duration are `const` ([expr.prim.id.unqual]), `this` is a pointer to `const` ([expr.prim.this]), and the result object can be named if a *result-name-introducer* ([dcl.contract.res]) has been specified. — *end note*]

Evaluation [basic.contract.eval]

A contract assertion may be evaluated using one of the following four *evaluation semantics*: *ignore*, *observe*, *enforce*, or *quick_enforce*. Observe, enforce, and quick_enforce are *checking semantics*; enforce and quick_enforce are *terminating semantics*.

Which evaluation semantic is used for any given evaluation of a contract assertion is implementation-defined. [*Note:* Different evaluations of the same contract assertion might use different evaluation semantics. This includes evaluations of contract assertions during constant evaluation. — *end note*]

Recommended practice: An implementation should provide the option to translate a program such that all contract-assertion evaluations have the ignore semantic as well as the option to translate a program such that all contract-assertion evaluations have the enforce semantic. By default, contract-assertion evaluations should have the enforce semantic.

The evaluation of a contract assertion with the ignore semantic has no effect. [*Note:* The predicate is potentially evaluated ([basic.def.odr]) but not evaluated. — *end note*]

The evaluation of a contact assertion with a checking semantic determines the value of the predicate. If the value B of the predicate can be determined without evaluating the predicate, that value may be used; otherwise, the predicate is evaluated and B is the result of that evaluation. [*Note:* To determine whether a predicate would evaluate to `true` or `false`, an alternative evaluation that produces the same value as the predicate but has no side effects might be evaluated instead of the predicate, resulting in the side effects of the predicate not occurring. — *end note*] There is an observable checkpoint ([intro.abstract]) C that happens before A such that any other operation O that happens before A also happens before C.

If B is false or if the evaluation of the predicate exits via an exception or is performed in a context that is manifestly constant-evaluated ([expr.const]) and the predicate is not a core constant expression, a contract violation occurs. [*Note:* If B is true, no contract violation occurs and control flow continues normally after the point of evaluation of the contract assertion. If the evaluation of the predicate does not produce a value and no contract violation occurs, e.g., because the evaluation of the predicate calls `longjmp` ([cset.jmp.syn]) or causes program termination, this evaluation is performed as usual. — *end note*]

If a contract violation occurs in a context that is manifestly constant-evaluated ([expr.const]), a diagnostic is produced; if the evaluation semantic is enforce or quick_enforce, the program is ill-formed.

[*Note:* Different evaluation semantics chosen for the same contract assertion in different translation units may result in violations of the one definition rule ([basic.def.odr]) when a contract assertion has side effects that alter the value produced by a constant expression. — *end note*] [*Example:*

```
constexpr int f(int i)
{
    contract_assert((++const_cast<int&>(i), true));
    return i;
  }
  inline void g()
  {
    int a[f(1)];  // size dependent on the evaluation semantic of contract_assert above
  }
```

— *end example*]

If a contract violation occurs in a context that is not manifestly constant-evaluated and the evaluation semantic is quick_enforce, the program is immediately terminated in an implementation-defined fashion.

If a contract violation occurs in a context that is not manifestly constant-evaluated and the evaluation semantic is enforce or observe, an object v of type `std::contracts::contract_violation` ([support.contracts.violation]) containing information about the contract violation is created in an unspecified manner, and the contract-violation handler (see below) is invoked with v as its only argument. Storage for v is allocated in an unspecified manner except as noted in [basic.stc.dynamic.allocation]. The destruction of v is sequenced after the corresponding contract-violation handler exits. If the contract violation occurred because the evaluation of the predicate exited via an exception, the contract-violation handler is invoked while that exception is the currently handled exception ([except.handle]). [*Note:* This allows the exception to be inspected within the contract-violation handler ([basic.contract.handler]). — *end note*]

If the contract-violation handler returns normally and the evaluation semantic is enforce, the program is terminated in an implementation-defined fashion.

If the contract-violation handler returns normally and the evaluation semantic is observe, control flow continues normally after the point of evaluation of the contract assertion. There is an observable checkpoint ([intro.abstract]) C that happens after the contract-violation handler returns normally such that any other operation O that happens after the contract-violation handler returns also happens after C.

[*Note:* The terminating semantics terminate the program if execution would otherwise continue normally past a contract violation: the enforce semantic provides the opportunity to log information about the contract violation before exiting the program, and the quick_enforce semantic is intended to terminate the program as soon as possible as well as to minimize the impact of contract checks on the generated code size. Conversely, the observe semantic provides the opportunity to log information about the contract violation without having to terminate the program. — *end note*]

If a contract-violation handler invoked from the evaluation of a function contract assertion exits via an exception, the behavior is as if the function body exits via that same exception. [*Note:* A *function-try-block* ([except.pre]) is part of the function body and thus does not have an opportunity to catch the exception. — *end note*] [*Note:* If the contract-violation handler exits via an exception on a call to a function with a non-throwing exception specification, the function `std::terminate()` is invoked ([except.terminate]). — *end note*]

If a contract-violation handler invoked from an assertion-statement ([stmt.contract.assert]) exits via an exception, the exception propagates from the execution of that statement.

The evaluations of two contract assertions A_1 and A_2 are *consecutive* when the only operations sequenced after A_1 and sequenced before A_2 are

— trivial initialization, construction, and destruction of objects,

— initialization of references,

— transfer of control via function invocation or a return statement.

[*Note:* This list contains vacuous operations whose evaluation will not invalidate the conditions that might be asserted by a contract assertion when performing a mix of returning from and invoking a series of functions. *— end note*]

A *contract-assertion sequence* is a sequence of contract assertions that are consecutive. At any point within a contract-assertion sequence, any previously evaluated contract assertion may be evaluated again with the same or a different evaluation semantic. Such repeated evaluations of a contract assertion may happen up to an implementation-defined number of times. [*Note:* For example, all function contract assertions might be evaluated twice for a single function invocation, once in the caller's translation unit as part of the invoking expression and once in the callee's translation unit as part of the function definition. This allowance also extends to evaluations of contract assertions during constant evaluation. *— end note*]

Recommended practice: An implementation should provide an option to perform a specified number of repeated evaluations for contract assertions. By default, no repeated evaluations should be performed.

Contract-violation handler **[basic.contract.handler]**

The *contract-violation handler* of a program is a function named `::handle_contract_violation` that is attached to the global module. The contract-violation handler shall take a single argument of type lvalue reference to `const std::contracts::contract_violation` and shall return `void`. The contract-violation handler may be `noexcept`. The implementation shall provide a definition of the contract-violation handler, called the *default contract-violation handler*. [*Note:* No declaration for the default contract-violation handler is provided by any standard library header. *— end note*]

Recommended practice: The default contract-violation handler should produce diagnostic output that suitably formats the most relevant contents of the `std::contracts::contract_violation` object, rate-limited for potentially repeated violations of observed contract assertions, and then return normally.

Whether the default contract-violation handler is replaceable ([dcl.fct.def.replace]) is implementation-defined. [*Note:* A program providing a definition for `::handle_contract_violation` when it is not replaceable will result in multiple definitions of the contract-violation handler and is thus ill-formed, no diagnostic required. *— end note*]

Add a new paragraph after [expr.prim.this], paragraph 2:

If the expression `this` appears within the *conditional-expression* of a contract assertion ([basic.contract.general]) (including as the result of the implicit transformation in the body of a non-static member function and including in the bodies of nested *lambda-expression*s), `const` is combined with the *cv-qualifier-seq* used to generate the resulting type (see below).

Modify [expr.prim.id.unqual], paragraph 3, and split into multiple paragraphs as follows:

[3] The result is the entity denoted by the *unqualified-id* ([basic.lookup.unqual]).

[4] If the *unqualified-id* appears in a *lambda-expression* at program point P and the entity is a local entity ([basic.pre]) or a variable declared by an *init-capture* ([expr.prim.lambda.capture]), then let S be the *compound-statement* of the innermost enclosing *lambda-expression* of P. If naming the entity from outside of an unevaluated operand within S would refer to an entity captured by copy in some intervening *lambda-expression*, then let E be the innermost such lambda-expression.

— If there is such a lambda-expression and if P is in E's function parameter scope but not its *parameter-declaration-clause*, then the type of the expression is the type of a class member access expression ([expr.ref]) naming the non-static data member that would be declared for such a capture in the object parameter ([dcl.fct]) of the function call operator of E. [*Note:* If E is not declared mutable, the type of such an identifier will typically be `const` qualified. — *end note*]

— Otherwise (if there is no such *lambda-expression* or if P either precedes E's function parameter scope or is in E's *parameter-declaration-clause*), the type of the expression is the type of the result.

[5] Otherwise, if the *unqualified-id* is the result name ([dcl.contract.res]) in a postcondition assertion attached to a function whose (possibly deduced, see [dcl.spec.auto]) return type is `T`, then the type of the expression is `const T`.

[6] Otherwise, if the *unqualified-id* appears in the predicate of a contract assertion C ([basic.contract]) and the entity is

— a variable declared outside of C of object type `T`, or

— a variable declared outside of C of type "reference to `T`", or

— a structured binding of type `T` whose corresponding variable is declared outside of C,

then the type of the expression is `const` T.

[*Example:*

```
int g = 0;
struct X { bool m(); };

void f(int i, int* p, int& r, X x, X* px)
  pre (++g)      // error: modifying const lvalue
  pre (++i)      // error: modifying const lvalue
  pre (++(*p))   // OK
  pre (++r)      // error: modifying const lvalue
  pre (x.m())    // error: calling non-const member function
  pre (px->m()) // OK
  pre ([&i]{
    ++g;         // error: modifying const lvalue
    ++i;         // error: modifying const lvalue
    int j;
    [&j]{
      int k;
      ++i; // error: modifying const lvalue
      ++j; // OK
      ++k; // OK
    }();
    return true;
  }());
```

— *end example*]

[7] [*Note:* If the entity is a template parameter object for a template parameter of type `T` ([temp.param]), the type of the expression is `const` T. — *end note*] [*Note:* The type will be adjusted as described in [expr.type] if it is *cv*-qualified or is a reference type. — *end note*]

[8] The expression is an xvalue if it is move-eligible (see below); an lvalue if the entity is a function, variable, structured binding ([dcl.struct.bind]), result name ([dcl.contract.res]), data member, or template parameter object; and a prvalue otherwise ([basic.lval]); it is a bit-field if the identifier designates a bit-field.

Modify [expr.prim.lambda.general], paragraph 1:

lambda-declarator :
  lambda-specifier-seq noexcept-specifier$_{opt}$ *attribute-specifier-seq*$_{opt}$
    trailing-return-type$_{opt}$ *function-contract-specifier-seq*$_{opt}$
  noexcept-specifier attribute-specifier-seq$_{opt}$ *trailing-return-type*$_{opt}$
    function-contract-specifier-seq$_{opt}$
  trailing-return-type$_{opt}$ *function-contract-specifier-seq*$_{opt}$
  (*parameter-declaration-clause*) *lambda-specifier-seq*$_{opt}$
    noexcept-specifier$_{opt}$ *attribute-specifier-seq*$_{opt}$ *trailing-return-type*$_{opt}$
    requires-clause$_{opt}$ *function-contract-specifier-seq*$_{opt}$

Modify [expr.prim.lambda.closure], paragraph 6:

[...] Any *noexcept-specifier* and *function-contract-specifier* ([dcl.contract.func]) specified on a *lambda-expression* applies to the corresponding function call operator or operator template. [...]

Add a new paragraph after [expr.prim.lambda.closure], paragraph 7:

If all potential references to a local entity implicitly captured by a *lambda-expression* L occur within the function contract assertions ([dcl.contract.func]) of the call operator or operator template of L or within assertion-statements ([stmt.contract.assert]) within the body of *L*, the program is ill-formed. [*Note:* This rule is intended to prevent situations in which adding a contract assertion to an existing C++ program could cause additional copies or destructions to be performed even if the contract assertion is never checked. — *end note*] [*Example:*

```
static int i = 0;

void test() {
  auto f1 = [=] pre(i > 0) {  // OK, no local entities are captured.
  };

  int i = 1;

  auto f2 = [=] pre(i > 0) {  // error: cannot implicitly capture i here
  };

  auto f3 = [i] pre(i > 0) {  // OK, i is captured explicitly.
  };

  auto f4 = [=] {
    contract_assert(i > 0);    // error: cannot implicitly capture i here
  };

  auto f5 = [=] {
    contract_assert(i > 0);    // OK, i is referenced elsewhere.
    (void)i;
  };

  auto f6 = [=] pre([]{
      bool x = true;
      return [=]{ return x; }();  // OK, x is captured implicitly.
    }()) {};

  bool y = true;
  auto f7 = [=] pre([=]{return y;}());  // error: outer capture of y is invalid.
}
```

— *end example*]

Modify [expr.prim.lambda.capture], paragraph 1: The body of a *lambda-expression* may refer to local entities of enclosing ~~block~~ scopes by capturing those entities, as described below.

Modify [expr.prim.lambda.capture], paragraph 3:

> A *lambda-expression* shall not have a *capture-default* or *simple-capture* in its *lambda-introducer* unless its innermost enclosing scope is a block scope ([basic.scope.block]) ~~or~~, it appears within a default member initializer and its innermost enclosing scope is the corresponding class scope ([basic.scope.class])~~.~~ , or it appears within a contract assertion and its innermost enclosing scope is the corresponding contract-assertion scope ([basic.scope.contract]).

Add new paragraphs to [expr.call] after paragraph 5:

> The callee-facing function contract assertions of a function call are those of the function that is being called.
>
> The caller-facing function contract assertions of a function call are determined as follows:
>
> — If the postfix expression is a class member access expression and the call is a virtual function call, the caller-facing function contract assertions are those of the statically chosen function.
>
> — Otherwise, there are no caller-facing function contract assertions.
>
> [*Note:* When a non-virtual call by name is performed, the caller-facing function contract assertions are effectively the same as the callee-facing function contract assertions of the invoked function and do not need to be (but might be) repeated. ([basic.contract.eval]) — *end note*]
>
> The precondition assertions of a function call are all precondition assertions in the caller-facing function contract assertions followed by all precondition assertions in the callee-facing function contract assertions. The postcondition assertions of a function call are all postcondition assertions of the callee-facing function contract assertions followed by all postcondition assertions of the caller-facing function contract assertions.

Modify [expr.call], paragraph 6:

> When a function is called, each parameter ([dcl.fct]) is initialized ([dcl.init], [class.copy.ctor]) with its corresponding argument, and each precondition assertion of the function call is evaluated. If the function is an explicit object member function and there is an implied object argument ([over.call.func]), the list of provided arguments is preceded by the implied object argument for the purposes of this correspondence. If there is no corresponding argument, the default argument for the parameter is used.

Modify [expr.call], paragraph 7:

> The *postfix-expression* is sequenced before each expression in the *expression-list* and any default argument. The initialization of a parameter, including every associated value computation and side effect, is indeterminately sequenced with respect to that of any other parameter. These evaluations are sequenced before the evaluation of the precondition assertions of the function call, which are evaluated in sequence ([dcl.contract.func]). If the implementation is permitted to introduce temporaries to hold any function parameters ([class.temporary]), evaluation of precondition assertions may happen before or happen after the initialization of the parameter objects from those temporaries.

Add a new paragraph after [expr.call], paragraph 8:

> When control is transferred back to this function call ([stmt.return], [expr.await]), all postcondition assertions of the function call are evaluated in sequence ([dcl.contract.func]). If the implementation is permitted to introduce a temporary to hold the result value ([class.temporary]), evaluation of postcondition assertions may happen before or happen

after the initialization of the result object from that temporary. [*Note:* These evaluations, in turn, are sequenced before the destruction of any function parameters. — *end note*]

Modify [expr.await], paragraph 2:

An *await-expression* shall appear only in a potentially-evaluated expression within the *compound-statement* of a *function-body* or *lambda-expression*, in either case outside of a *handler* ([except.pre]). In a *declaration-statement* or in the *simple-declaration* (if any) of an *init-statement*, an await-expression shall appear only in an *initializer* of that *declaration-statement* or *simple-declaration*. An *await-expression* shall not appear in a default argument ([dcl.fct.default]). An *await-expression* shall not appear in the initializer of a block variable with static or thread storage duration. In the predicate of a contract assertion ([basic.contract]), an await-expression shall not appear outside of a lambda-expression that is a subexpression of that predicate. A context within a function where an *await-expression* can appear is called a *suspension context* of the function.

Modify [expr.const], paragraph 2:

A variable or temporary object *o* is *constant-initialized* if

— either it has an initializer or its default-initialization results in some initialization being performed, and

— the full-expression of its initialization is a constant expression when interpreted as a constant-expression with all contract assertions having the ignore evaluation semantic ([basic.contract.eval]), except that if *o* is an object, that full-expression may also invoke constexpr constructors for *o* and its subobjects even if those objects are of non-literal class types. [*Note:* The initialization, when evaluated, might still evaluate contract assertions with other evaluation semantics, resulting in a diagnostic or ill-formed program if a contract violation occurs. — *end note*] [*Note:* Such a class can have a non-trivial destructor. Within this evaluation, `std::is_constant_evaluated()` ([meta.const.eval]) returns `true`. — *end note*]

Modify [expr.const], paragraph 19:

[*Example*:

```
[...]

template<class T>
constexpr int k(int) { // k<int> is not an immediate function because A(42) is a
  return A(42).y;      // constant expression and thus not immediate-escalating
}

constexpr int l(int c) pre(c >= 2) {
  return (c % 2 == 0) ? c / 0 : c;
}

const int i0 = l(0);  // dynamic initialization is contract violation or undefined behavior
const int i1 = l(1);  // static initialization to 1 or contract violation at compile time
const int i2 = l(2);  // dynamic initialization is undefined behavior
const int i3 = l(3);  // static initialization to 3
```

— *end example*]

Modify [expr.const], footnote 73:

Testing this condition can involve a trial evaluation of its initializer, with contract-assertion evaluations having the ignore evaluation semantic ([basic.contract.eval]), as described above.

Modify [stmt.pre], paragraph 1:

statement :
  attribute-specifier-seq$_{opt}$ *expression-statement*
  attribute-specifier-seq$_{opt}$ *compound-statement*
  attribute-specifier-seq$_{opt}$ *selection-statement*
  attribute-specifier-seq$_{opt}$ *iteration-statement*
  attribute-specifier-seq$_{opt}$ *jump-statement*
  attribute-specifier-seq$_{opt}$ *assertion-statement*
  declaration-statement
  attribute-specifier-seq$_{opt}$ *try-block*

Add a note after [stmt.return], paragraph 3:

[*Note:* Postcondition assertions of the function call ([expr.call]) are evaluated in sequence after the destruction of any local variables in scopes exited by the return statement and are, in turn, sequenced before the destruction of function parameters. — *end note*]

Modify [stmt.return], paragraph 5:

The copy-initialization of the result of the call is sequenced before the destruction of temporaries at the end of the full-expression established by the operand of the return statement, which, in turn, is sequenced before the destruction of local variables ([stmt.jump]) of the block enclosing the return statement. [*Note:* These operations, in turn, are sequenced before the destruction of local variables in each remaining enclosing block of the function, then the evaluation of postcondition assertions of the function call ([expr.call]), then the destruction of function parameters. — *end note*]

Add a new subclause after [stmt.jump]:

Assertion statement [stmt.contract.assert]

assertion-statement :
  `contract_assert` *attribute-specifier-seq*$_{opt}$ `(` *conditional-expression* `)` `;`

An *assertion-statement* introduces a contract assertion ([basic.contract]). The optional *attribute-specifier-seq* appertains to the introduced contract assertion. [*Note:* An *assertion-statement* allows the programmer to specify a state of the program that is considered incorrect when control flow reaches the assertion-statement. — *end note*]

Modify [dcl.decl.general], paragraph 1:

> *init-declarator :*
> *declarator initializer*~~$_{opt}$~~
> *declarator requires-clause*$_{opt}$ *function-contract-specifier-seq*$_{opt}$

Add a new paragraph after [dcl.decl.general], paragraph 4:

> The optional *function-contract-specifier-seq* ([dcl.contract.func]) in an *init-declarator* shall be present only if the *declarator* declares a function.

Add a new subclause after [dcl.decl]:

Function contract specifiers [dcl.contract]

General [dcl.contract.func]

> *function-contract-specifier-seq :*
> *function-contract-specifier function-contract-specifier function-contract-specifier-seq*
>
> *function-contract-specifier :*
> *precondition-specifier*
> *postcondition-specifier*
>
> *precondition-specifier :*
> `pre` *attribute-specifier-seq*$_{opt}$ `(` *conditional-expression* `)`
>
> *postcondition-specifier :*
> `post` *attribute-specifier-seq*$_{opt}$ `(` *result-name-introducer*$_{opt}$ *conditional-expression* `)`
>
> *result-name-introducer :*
> *attributed-identifier* `:`

A *function contract assertion* is a contract assertion ([basic.contract]) associated with a function. Each *function-contract-specifier* of a *function-contract-specifier-seq* (if any) of an unspecified first declaration of a function introduces a corresponding function contract assertion for that function. The optional *attribute-specifier-seq* following `pre` or `post` appertains to the introduced contract assertion. The optional *attribute-specifier-seq* of the *attributed-identifier* in a *result-name-introducer* appertains to the introduced result name (see below). [*Note:* The *function-contract-specifier-seq* of a *lambda-declarator* applies to the call operator or operator template of the corresponding closure type ([expr.prim.lambda.closure]). — *end note*]

A *precondition-specifier* introduces a *precondition assertion*, which is a function contract assertion. [*Note:* A precondition assertion allows the programmer to specify a state of the program that is considered incorrect when a function is invoked. — *end note*]

A *postcondition-specifier* introduces a *postcondition assertion*, which is a function contract assertion. [*Note:* A postcondition assertion allows the programmer to specify a state of the program that is considered incorrect when a function returns normally. It does not specify anything about a function that exits in another fashion, such as via an exception or via a call to `longjmp` ([cset.jmp.syn]). — *end note*]

A declaration E of a function `f` that is not a first declaration shall have either no *function-contract-specifier-seq* or the same *function-contract-specifier-seq* as any first declaration D reachable from E. If D and E are in different translation units, a diagnostic is required only if D is attached to a named module. If a declaration D_1 is a first declaration of `f` in one translation unit and a declaration D_2 is a first declaration of the same function `f` in another translation unit, D_1 and D_2 shall specify the same *function-contract-specifier-seq*, no diagnostic required.

A *function-contract-specifier-seq* *s1* is the same as a *function-contract-specifier-seq* *s2* if *s1* and *s2* consist of the same *function-contract-specifier*s in the same order. A *function-contract-specifier* *c1* on a function declaration *d1* is the same as a *function-contract-specifier* *c2* on a function declaration *d2* if their predicates ([basic.contract.general]), *p1* and *p2*, would satisfy the one-definition rule ([basic.def.odr]) if placed in function definitions on the declarations *d1* and *d2*, respectively, except for renaming of parameters, renaming of template parameters, and renaming of the result name ([dcl.contract.res]), if any.

[*Note:* As a result of the above, all uses and definitions of a function see the equivalent *function-contract-specifier-seq* for that function across all translation units. — *end note*]
[*Example:*

```
bool b1, b2;

void f() pre (b1) pre ([]{ return b2; }());
void f();                        // OK, function−contract specifiers omitted
void f() pre (b1) pre ([]{ return b2; }()); // OK, same by odr
void f() pre (b1);               // error: function−contract specifiers only partially repeated
void f() pre (b1) pre (b2);      // error: not same by odr
```

— *end example*]

A deleted function ([dcl.fct.def.delete]) or a function defaulted on its first declaration ([dcl.fct.def.default]) may not have a *function-contract-specifier-seq*.

If the implementation is permitted to introduce a temporary object to hold a parameter object value ([class.temporary]),

— an *id-expression* within the predicate of a precondition assertion will denote the temporary object if the evaluation of the assertion happens before the parameter object is initialized, and

— it is unspecified whether an *id-expression* within the predicate of a postcondition assertion will denote the temporary object or the parameter object

[*Note:* It follows that, for objects that can be passed in registers, a postcondition assertion might not see any modifications of mutable subobjects ([dcl.stc]) of the parameter object performed by the function or a function overriding it. — *end note*]

If the predicate of a postcondition assertion of a function f odr-uses ([basic.def.odr]) a non-reference parameter of f, all declarations of that parameter and the corresponding parameter on any functions that override f shall have a `const` qualifier and shall not have array or function type. [*Note:* This requirement applies even to declarations that do not specify the *postcondition-specifier*. Arrays and functions are still usable when declared with the equivalent pointer types ([dcl.fct]). — *end note*] [*Example:*

```
int f(const int i)
  post (r: r == i);

int g(int i)
  post (r: r == i);       // error: i is not declared const.

int f(int i)              // error: i is not declared const.
{
  return i;
}

int g(int i)              // error: i is not declared const.
{
  return i;
}

template <typename T>
void f(T t) post(t > 0); // error: t is not declared const.
```

— *end example*]

When an ordered set of function contract assertions S are *evaluated in sequence*, for any two function contract assertions X and Y in the set, the evaluation of X is sequenced before the evaluation of Y if X precedes Y in S.

[*Note:* The precondition assertions of a function are evaluated in sequence when the function is invoked ([intro.execution]); in a virtual function call ([expr.call]), the precondition assertions of the statically chosen function are evaluated first, followed by those of the final overrider. The postcondition assertions of a function are evaluated in sequence when a function returns normally ([stmt.return]); in a virtual function call, the postcondition assertions of the final overrider are evaluated first, followed by those of the statically chosen function. — *end note*]

[*Note:* The function contract assertions of a function are evaluated even when invoked indirectly, such as through virtual dispatch, a pointer to function, or a pointer to member function. A pointer to function, pointer to member function, or function type alias cannot have a *function-contract-specifier-seq* associated directly with it. — *end note*]

The function contract assertions of a function are considered to be needed when

— the function is odr-used ([basic.def.odr]) or, if it appears in an unevaluated operand, would be odr-used if the expression were potentially evaluated or

— its definition is instantiated.

The function contract assertions of a templated function are instantiated only when needed ([temp.inst]).

Referring to the result object [dcl.contract.res]

The *result-name-introducer* of a *postcondition-specifier* is a declaration. The *identifier* in the *result-name-introducer* is the *result name* of the corresponding postcondition assertion. The result name inhabits the contract-assertion scope ([basic.scope.contract]) and denotes the result object of the function. If a postcondition assertion has a result name and the return type of the function is `void`, the program is ill-formed. [*Note:* The result name when used as an *id-expression* is a `const` lvalue ([expr.prim.id.unqual]). — *end note*]

If the implementation is permitted to introduce a temporary object for the return value ([class.temporary]) and the postcondition assertion happens before the result object is initialized from that temporary ([expr.call]), the result name denotes that temporary object instead. [*Note:* Modifications to that temporary's value are still in sequence with the evaluation of the postcondition assertions and expected to be retained for the eventual result object. — *end note*] [*Example:*

```
int f()
  post(r : ++const_cast<int&>(r) == 1)
  post(r : ++const_cast<int&>(r) == 2) // The postcondition checks will succeed if
{                                       // both predicates are evaluated exactly once.
  return 0;
}

struct A {};  // trivially copyable and destructible

struct B {    // not trivially copyable and destructible
  B() {}
  B(const B&) {}
};

template <typename T>
T f(T* ptr)
  post(r: &r == ptr)
{
  return T{};
}

int main() {
  A a = f(&a);  // The postcondition check may fail.
  B b = f(&b);  // The postcondition check is guaranteed to succeed.
}
```

— *end example*]

When the declared return type of a non-templated function contains a placeholder type, a *postcondition-specifier* with a *result-name-introducer* shall be present only on a definition. [*Example:*

```
int f(int& p)
  post (p >= 0)      // OK
  post (r: r >= 0);  // OK
```

```
auto g(auto& p)
  post (p >= 0)       // OK
  post (r: r >= 0);   // OK

auto h(int& p)
  post (p >= 0)       // OK
  post (r: r >= 0);   // error: cannot name the return value

auto k(int& p)
  post (p >= 0)       // OK
  post (r: r >= 0)    // OK
{
  return p = 0;
}
```

— *end example*]

Modify [dcl.fct], paragraph 1:

In a declaration `T D` where `T` may be empty and `D` has the form

> `D1 (` *parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$
> *ref-qualifier*$_{opt}$ *noexcept-specifier*$_{opt}$ *attribute-specifier-seq*$_{opt}$
> *trailing-return-type*$_{opt}$ *function-contract-specifier-seq*$_{opt}$

a *derived-declarator-type-list* is determined as follows:

— If the *unqualified-id* of the *declarator-id* is a *conversion-function-id*, the *derived-declarator-type-list* is empty.

— Otherwise, the *derived-declarator-type-list* is as appears in the type "*derived-declarator-type-list* `T`" of the contained *declarator-id* in the declaration `T D1`.

The declared return type `U` of the function type is determined as follows:

— If the *trailing-return-type* is present, `T` shall be the single *type-specifier* `auto`, and `U` is the type specified by the *trailing-return-type*.

— Otherwise, if the declaration declares a conversion function, see [class.conv.fct].

— Otherwise, `U` is `T`.

The type of the *declarator-id* in `D` is "*derived-declarator-type-list* `noexcept`$_{opt}$ function of parameter-type-list *cv-qualifier-seq*$_{opt}$ *ref-qualifier*$_{opt}$ returning `U`", where

— the parameter-type-list is derived from the *parameter-declaration-clause* as described below and

— the optional `noexcept` is present if and only if the exception specification([except.spec]) is non-throwing.

The optional *attribute-specifier-seq* appertains to the function type.

Modify [dcl.fct.def.general], paragraph 1:

> *function-definition* :
> *attribute-specifier-seq*$_{opt}$ *decl-specifier-seq*$_{opt}$ *declarator virt-specifier-seq*$_{opt}$
> *function-contract-specifier-seq*$_{opt}$ *function-body*
> *attribute-specifier-seq*$_{opt}$ *decl-specifier-seq*$_{opt}$ *declarator requires-clause*
> *function-contract-specifier-seq*$_{opt}$ *function-body*

Modify [dcl.fct.def.coroutine], paragraph 5:

A coroutine behaves as if the top-level *cv*-qualifiers in all *parameter-declaration*s in the declarator of its *function-definition* were removed and its *function-body* were replaced by the following *replacement body*:

```
{
  promise-type promise promise-constructor-arguments ;
  [...]
```

Modify [dcl.fct.def.coroutine], paragraph 9:

An implementation may need to allocate additional storage for a coroutine. This storage is known as the *coroutine state* and is obtained by calling a non-array allocation function ([basic.stc.dynamic.allocation]) as part of the replacement body. The allocation function's name is looked up by searching for it in the scope of the promise type.

— If the search finds any declarations, overload resolution is performed on a function call created by assembling an argument list. The first argument is the amount of space requested, and is a prvalue of type `std::size_t`. The lvalues $p_1 \ldots p_n$ with their original *cv*-qualifiers are the successive arguments. If no viable function is found ([over.match.viable]), overload resolution is performed again on a function call created by passing just the amount of space required as a prvalue of type `std::size_t`.

Modify [dcl.fct.def.coroutine], paragraph 13:

When a coroutine is invoked~~, after initializing its parameters (*expr.call*)~~ at the beginning of the replacement body, a copy is created for each coroutine parameter. For a parameter whose original declaration was of type *cv* `T`:

— If `T` is a reference type, the copy is a reference of type *cv* `T` bound to the same object as the parameter,

— Otherwise, the copy is a variable of type *cv* `T` with automatic storage duration that is direct-initialized from an xvalue of type `T` referring to the parameter. [*Note:* An identifier in the *function-body* that names one of these parameters refers to the created copy, not the original parameter ([expr.prim.id.unqual]). — *end note*]

[*Note:* An original parameter object is never a const or volatile object ([basic.type.qualifier]). — *end note*]

Add new section after [dcl.fct.def.coroutine]:

Replaceable function definitions **[dcl.fct.def.replace]**

Certain functions for which a definition is supplied by the implementation are *replaceable*. A C++ program may provide a definition with the signature and return type of a replaceable function, called a *replacement function*. The replacement function is used instead of the default version supplied by the implementation. Such replacement occurs prior to program startup ([basic.def.odr], [basic.start]). The program's declarations

— shall not be specified as `inline`,

— shall be attached to the global module, and

— shall have C++ language linkage;

no diagnostic is required. [*Note:* The one-definition rule ([basic.def.odr]) applies to the definitions of a replaceable function provided by the program. The implementation-supplied function definition is an otherwise-unnamed function with no linkage. — *end note*] [*Note:* Some replaceable functions, such as those in header `<new>`, are also declared in a standard library header, and the function definition would be ill-formed without a compatible declaration; other replaceable functions, such as the contract-violation handler ([basic.contract.handler]) on implementations where it is replaceable, need only match the specified signature and return type. The exception specification ([except.spec]) is part of the declaration but not part of the signature. — *end note*]

Modify [dcl.attr.grammar], paragraph 1:

Attributes specify additional information for various source constructs such as types, variables, names, contract assertions, blocks, or translation units.

Modify [dcl.attr.unused], paragraph 2:

The attribute may be applied to the declaration of a class, *typedef-name*, variable (including a structured binding declaration), structured binding, result name, non-static data member, function, enumeration, or enumerator, or to an *identifier* label ([stmt.label]).

Modify [class.mem.general], paragraph 1:

member-declarator :
  declarator virt-specifier$_{opt}$ *function-contract-specifier-seq*$_{opt}$ *pure-specifier*$_{opt}$
  declarator requires-clause function-contract-specifier-seq$_{opt}$
  declarator brace-or-equals-initializer$_{opt}$
  identifier$_{opt}$ *attribute-specifier-seq*$_{opt}$: *brace-or-equals-initializer*$_{opt}$

Modify [class.mem.general], paragraph 8:

A complete class context of a class (template) is a

— function body ([dcl.fct.def.general]),

— default argument ([dcl.fct.default]),

— default template argument ([temp.param]),

— *noexcept-specifier* ([except.spec]),

— *function-contract-specifier* ([dcl.contract.func]), or

— default member initializer

within the *member-specification* of the class or class template.

Modify [class.access.general] paragraph 7:

[*Example:*

```
class A {
  typedef int I;      // private member
  I f();
  friend I g(I);
  static I x;
  template<int> struct Q;
  template<int> friend struct R;
protected:
    struct B { };
};

A::I A::f() pre(A::x > 0) { return 0; }
A::I g(A::I p = A::x) post(A::x <= 0);
A::I g(A::I p) { return 0; }
A::I A::x = 0;
template<A::I> struct A::Q { };
template<A::I> struct R { };

struct D: A::B, A { };
```

Here, all the uses of `A::I` are well-formed because `A::f`, `A::x`, and `A::Q` are members of class `A` and `g` and `R` are friends of class `A`. This implies, for example, that access checking on the first use of `A::I` must be deferred until it is determined that this use of `A::I` is as the return type of a member of class `A`. Similarly, the use of `A::B` as a *base-specifier* is well-formed because `D` is derived from `A`, so checking of *base-specifier*s must be deferred until the entire *base-specifier-list* has been seen. — *end example*]

Modify [class.base.init] paragraph 16:

Member functions (including virtual member functions, [class.virtual]) can be called for an object under construction. Similarly, an object under construction can be the operand of the `typeid` operator ([expr.typeid]) or of a `dynamic_cast` ([expr.dynamic.cast]). However, if these operations are performed in a *ctor-initializer* ~~(or in a function called directly or indirectly from a *ctor-initializer*)~~ before all the *mem-initializer*s for base classes have completed, during evaluation of a precondition assertion of a constructor or a postcondition assertion of a destructor ([dcl.contract.func]), or in a function called directly or indirectly from those contexts, the program has undefined behavior.

Modify [class.cdtor], paragraph 4:

> Member functions, including virtual functions ([class.virtual]), can be called during construction or destruction ([class.base.init]). When a virtual function is called directly or indirectly from a constructor or from a destructor, including during the construction or destruction of the class's non-static data members, or during the evaluation of a postcondition assertion of a constructor or a precondition assertion of a destructor ([dcl.contract.func]) and the object to which the call applies is the object (call it `x`) under construction or destruction, the function called is the final overrider in the constructor's or destructor's class and not one overriding it in a more-derived class. If the virtual function call uses an explicit class member access ([expr.ref]) and the object expression refers to the complete object of `x` or one of that object's base class subobjects but not `x` or one of its base class subobjects, the behavior is undefined.

Modify [temp.dep.expr], paragraph 3:

> An *id-expression* is type-dependent if it is a *template-id* that is not a concept-id and is dependent; or if its terminal name is
>
> — [...]
>
> — the identifier `__func__` ([dcl.fct.def.general]), where any enclosing function is a template, a member of a class template, or a generic lambda,
>
> — the result name ([dcl.contract.res]) of a postcondition assertion of a function whose return type is dependent,
>
> — a *conversion-function-id* that specifies a dependent type, or
>
> — [...]

Modify [temp.inst], paragraph 14:

> The *noexcept-specifier* ([except.spec]) and *function-contract-specifiers* ([dcl.contract.func]) of a function template are not instantiated along with the function declaration. The *noexcept-specifier* of a function template specialization is instantiated when the exception specification of that function is needed (see [except.spec]). The *function-contract-specifiers* of a function template specialization are instantiated when the function contract assertions of that function are needed (see [dcl.contract.func]). ~~of a function template specialization is not instantiated along with the function declaration; it is instantiated when needed ([except.spec]).~~ If such ~~an *noexcept-specifier*~~ a specifier is needed but has not yet been instantiated, the dependent names are looked up, the semantics constraints are checked, and the instantiation of any template used in the ~~*noexcept-specifier*~~ specifier is done as if it were being done as part of instantiating the declaration of the specialization at that point. [*Note:* Therefore, any errors that arise from instantiating these specifiers are not in the immediate context of the function declaration and can result in the program being ill-formed ([temp.deduct]). — *end note*]

Modify [temp.expl.spec], paragraph 14:

> Whether an explicit specialization of a function or variable template is inline, constexpr, constinit, or consteval is determined by the explicit specialization and is independent of those properties of the template. Similarly, attributes appearing in the declaration of a template have no effect on an explicit specialization of that template. [*Example:*
>
> ```
> [...]
> ```
>
> — *end example*] [*Note:* For an explicit specialization of a function template, the *function-contract-specifier-seq* ([dcl.contract.func]) of the explicit specialization is independent of that of the primary template. — *end note*]

Modify [temp.deduct.general], paragraph 7:

> [*Note:* The equivalent substitution in exception specifications and function contract assertions ([dcl.contract.func]) is done only when the *noexcept-specifier* or *function-contract-specifier*, respectively, is instantiated, at which point a program is ill-formed if the substitution results in an invalid type or expression. — *end note*]

Modify [except.spec], paragraph 13:

> An exception specification is considered to be *needed* when:
>
> — ~~in an expression, the function is selected by overload resolution ([over.match], [over.over]);~~
>
> — the function is odr-used ([basic.def.odr]) or, if it appears in an unevaluated operand, would be odr-used if the expression were potentially evaluated;
>
> — the exception specification is compared to that of another declaration (e.g., an explicit specialization or an overriding virtual function);
>
> — the function is defined; or
>
> — the exception specification is needed for a defaulted function that calls the function. [*Note:* A defaulted declaration does not require the exception specification of a base member function to be evaluated until the implicit exception specification of the derived function is needed, but an explicit *noexcept-specifier* needs the implicit exception specification to compare against. — *end note*]
>
> The exception specification of a defaulted function is evaluated as described above only when needed; similarly, the *noexcept-specifier* of a templated function ~~a specialization of a function template or member function of a class template~~ is instantiated only when needed.

Modify [except.terminate], paragraph 1:

> In some situations, exception handling is abandoned for less subtle error handling techniques.
>
> [*Note:* These situations are:
>
> — [...]

— when execution of a function registered with `std::atexit` or `std::at_quick_exit` exits via an exception ([support.start.term]), or

— when a contract-violation handler ([basic.contract.handler]) invoked from evaluating a function contract assertion on a function with a non-throwing exception specification exits via an exception, or

— [...]

— *end note*]

Modify [cpp.predefined], Table 22: Feature-test macros, with `XXXX` replaced by the appropriate value:

Macro name	Value
`[...]`	`[...]`
`__cpp_constinit`	`201907L`
`__cpp_contracts`	`20XXXXL`
`__cpp_decltype`	`200707L`
`[...]`	`[...]`

Modify [headers], Table 24: C++ library headers:

`[...]`
`<condition_variable>`
`<contracts>`
`<coroutine>`
`[...]`

Modify [structure.specifications], paragraph 3:

— ...

— Mandates: the conditions that, if not met, render the program ill-formed. [*Example:* An implementation can express such a condition via the *constant-expression* in a *static_assert-declaration* ([dcl.pre]). If the diagnostic is to be emitted only after the function has been selected by overload resolution, an implementation can express such a condition via a *constraint-expression* ([temp.constr.decl]) and also define the function as deleted. — *end example*]

— Preconditions: the conditions that the function assumes to hold whenever it is called; violation of any preconditions results in undefined behavior. [*Example:* An implementation can express such conditions via the use of a contract assertion such as a precondition assertion. — *end example*]

— Effects: the actions performed by the function.

— Synchronization: the synchronization operations ([intro.multithread]) applicable to the function.

— Postconditions: the conditions (sometimes termed observable results) established by the function. [*Example:* An implementation can express such conditions via the use of a contract assertion such as a postcondition assertion. — *end example*]

— Result: for a *typename-specifier*, a description of the named type; for an *expression*, a description of the type and value category of the expression; the expression is an lvalue if the type is an lvalue reference type, an xvalue if the type is an rvalue reference type, and a prvalue otherwise.

— ...

Modify [headers], Table 27: C++ headers for freestanding implementations:

`[...]`
`<compare>`
`<contracts>`
`<coroutine>`
`[...]`

Modify [support.general], 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 contract-violation handling, support for exception processing, support for initializer lists, and other runtime support, as summarized in Table 38.

Modify [support.general], Table 38: Language support library summary:

	Subclause	Header
[...]		
[support.exception]	Exception handling	`<exception>`
[support.contracts]	Contract-violation handling	`<contracts>`
[support.initlist]	Initializer lists	`<initializer_list>`
[...]		

Add new section [contract.assertions] in [conforming], after [res.on.exception.handling]:

Contract assertions **[contract.assertions]**

Unless specified otherwise, an implementation is allowed but not required to check the specified preconditions and postconditions of a function in the C++ standard library using contract assertions ([basic.contract]).

Modify [replacement.functions]:

[support] through [thread] and [depr] describe the behavior of numerous functions defined by the C++ standard library. Under some circumstances, however, certain of these function descriptions also apply to replacement functions ([dcl.fct.def.replace]) defined in the program.

~~A C++ program may provide the definition for any of the~~ The following dynamic memory allocation functions ~~signatures~~ declared in header `<new>` ([basic.stc.dynamic], [new.syn]) are replaceable ([dcl.fct.def.replace]):

```
operator new(std::size_t)
operator new(std::size_t, std::align_val_t)
operator new(std::size_t, const std::nothrow_t&)
operator new(std::size_t, std::align_val_t, const std::nothrow_t&)

operator delete(void*)
operator delete(void*, std::size_t)
operator delete(void*, std::align_val_t)
operator delete(void*, std::size_t, std::align_val_t)
operator delete(void*, const std::nothrow_t&)
operator delete(void*, std::align_val_t, const std::nothrow_t&)

operator new[](std::size_t)
operator new[](std::size_t, std::align_val_t)
operator new[](std::size_t, const std::nothrow_t&)
operator new[](std::size_t, std::align_val_t, const std::nothrow_t&)

operator delete[](void*)
operator delete[](void*, std::size_t)
operator delete[](void*, std::align_val_t)
operator delete[](void*, std::size_t, std::align_val_t)
operator delete[](void*, const std::nothrow_t&)
operator delete[](void*, std::align_val_t, const std::nothrow_t&)
```

~~A C++ program may provide the definition of t~~The following function ~~signature~~ declared in header `<new>` ([basic.stc.dynamic], [new.syn]) is replaceable:

```
bool std::is_debugger_present() noexcept
```

~~The program's definitions are used instead of the default versions supplied by the implementation ([new.delete]). Such replacement occurs prior to program startup ([basic.def.odr], [basic.start]). The program's declarations shall not be specified as inline. No diagnostic is required.~~

Modify [new.delete.single], paragraphs 2, 6, 13, and 21; [new.delete.array], paragraphs 2, 6, 12, and 18; and [debugging.utility]:

Replaceable: A C++ program may define a function with this function signature, and thereby displace the default version defined by the C++ standard library ([dcl.fct.def.replace]).

Add a new subclause [support.contracts] after [support.execution]:

Contract-violation handling [support.contracts]

Header `<contracts>` synopsis [contracts.syn]

The header `<contracts>` defines types for reporting information about contract violations ([basic.contract.eval]) generated by the implementation.

```
// all freestanding
namespace std::contracts {

  enum class assertion_kind : unspecified {
    pre = 1,
    post = 2,
    assert = 3
  };

  enum class evaluation_semantic : unspecified {
    ignore = 1,
    observe = 2,
    enforce = 3,
    quick_enforce = 4
  };

  enum class detection_mode : unspecified {
    predicate_false = 1,
    evaluation_exception = 2
  };

  class contract_violation {
    // no user-accessible constructor
  public:
    // cannot be copied or moved
    contract_violation(const contract_violation&) = delete;
    // cannot be assigned to
    contract_violation& operator=(const contract_violation&) = delete;

    /* see below */ ~contract_violation();

    const char* comment() const noexcept;
    std::contracts::detection_mode detection_mode() const noexcept;
    std::exception_ptr evaluation_exception() const noexcept;
    bool is_terminating() const noexcept;
    assertion_kind kind() const noexcept;
    source_location location() const noexcept;
    evaluation_semantic semantic() const noexcept;
  };

  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

Enum class `assertion_kind` [support.contracts.kind]

The type `assertion_kind` specifies the syntactic form of the contract assertion ([basic.contract]) whose evaluation resulted in the contract violation. Its enumerated values and their meanings are as follows:

- `assertion_kind::pre`: the evaluated contract assertion was a precondition assertion.
- `assertion_kind::post`: the evaluated contract assertion was a postcondition assertion.
- `assertion_kind::assert`: the evaluated contract assertion was an *assertion-statement*.

Recommended practice: Implementation-defined enumerators should have a name that is an identifier reserved for the implementation ([lex.name]) and a minimum value of `1000`.

Enum class `evaluation_semantic` [support.contracts.semantic]

The type `evaluation_semantic` specifies the evaluation semantic ([basic.contract.eval]) with which a contract assertion may be evaluated. Its enumerated values and their meanings are as follows:

— `evaluation_semantic::ignore`: the ignore evaluation semantic.

— `evaluation_semantic::observe`: the observe evaluation semantic.

— `evaluation_semantic::enforce`: the enforce evaluation semantic.

— `evaluation_semantic::quick_enforce`: the quick_enforce evaluation semantic.

Recommended practice: Implementation-defined enumerators should have a name that is an identifier reserved for the implementation ([lex.name]) and a minimum value of `1000`.

Enum class `detection_mode` [support.contracts.detection]

The type `detection_mode` specifies the manner in which a contract violation was identified ([basic.contract.eval]). Its enumerated values and their meanings are as follows:

— `detection_mode::predicate_false`: the contract violation occurred because the predicate evaluated to `false` or would have evaluated to `false`.

— `detection_mode::evaluation_exception`: the contract violation occurred because the evaluation of the predicate exited via an exception.

Recommended practice: Implementation-defined enumerators should have a name that is an identifier reserved for the implementation ([lex.name]) and a minimum value of `1000`.

Class `contract_violation` [support.contracts.violation]

The class `contract_violation` describes information about a contract violation ([basic.contract.eval]) generated by the implementation. Objects of this type can be created only by the implementation. Whether the destructor is virtual is implementation-defined.

```
const char* comment() const noexcept;
```

Returns: An implementation-defined null-terminated multibyte string in the ordinary literal encoding ([lex.charset]).

Recommended practice: The string returned should contain a textual representation of the predicate of the violated contract assertion. The source code produced may be truncated, be reformatted, represent the code before or after preprocessing, or be summarized. An implementation can return an empty string if storing a textual representation of violated predicates is undesired.

```
std::contracts::detection_mode detection_mode() const noexcept;
```

Returns: The manner in which the contract violation was identified.

`std::exception_ptr evaluation_exception() const noexcept;`

Returns: If the contract violation occurred because the evaluation of the predicate exited via an exception, an `exception_ptr` object that refers to that exception or a copy of that exception; otherwise, a null `exception_ptr` object.

`bool is_terminating() const noexcept;`

Returns: `true` if the current evaluation semantic is a terminating semantic ([basic.contract.eval]); `false` otherwise.

`assertion_kind kind() const noexcept;`

Returns: The syntactic form of the violated contract assertion.

`source_location location() const noexcept;`

Returns: An implementation-defined value.

Recommended practice: The value returned should represent a source location for identifying the violated contract assertion. For a precondition, the value returned should be the source location of the function invocation when possible; when the invocation location cannot be ascertained and on contract assertions other than preconditions, the value returned should be the source location of the violated contract assertion. The encoding of `file_name` should match the encoding in a `source_location` object generated in any other fashion. An implementation can return a default-constructed `source_location` object if storing information regarding the source location is undesired.

`evaluation_semantic semantic() const noexcept;`

Returns: The evaluation semantic with which the violated contract assertion was evaluated.

[*Note:* This member function is provided for logging purposes and to identify implementation-defined semantics. — *end note*]

`invoke_default_contract_violation_handler` **[support.contracts.invoke]**

`void invoke_default_contract_violation_handler(const contract_violation&);`

Effects: equivalent to invoking the default contract-violation handler ([basic.contract.handler]).

Add a new section to Annex C, [diff.cpp23], in the appropriate place:

Lexical conventions [diff.cpp23.lex]

Affected subclause: [lex.key]
Change: New keywords.
Rationale: Required for new features.

— The `contract_assert` keyword is added to introduce a contract assertion through an *assertion-statement* ([stmt.contract.assert]).

Effect on original feature: Valid C++ 2023 code using `contract_assert` as an identifier is not valid in this revision of C++.

5 Conclusion

The idea of a Contracts facility in the C++ Standard has been an area of active work and development for over two decades. This proposal represents the culmination of significant effort to reach consensus in the Contracts study group (SG21). We feel that this proposal will provide significant benefits to C++ users as it stands and that it will serve as a foundation that can grow to meet the needs expressed by our many constituents. We hope that this MVP proposal will be well received by the C++ community and that it will pave the way to a better, safer C++ ecosystem.

Acknowledgements

Thanks to everyone who participated in the many discussions at SG21 meetings and teleconferences and on the SG21 reflector and who thus helped shape this paper.

Thanks to everyone who participated in the EWG and LEWG reviews and provided feedback.

Thanks to Andrei Zissu and Lewis Baker for reviewing this paper and pointing out technical errors and omissions.

Thanks to Lori Hughes for reviewing this paper and providing editorial feedback.

Bibliography

[CWG2841] Tom Honermann, "When do const objects start being const?"
`https://wg21.link/cwg2841`

[N4981] Thomas Köppe, "Working Draft, Programming Languages – C++", 2024
`http://wg21.link/N4981`

[P1494R3] S. Davis Herring, "Partial program correctness", 2024
`http://wg21.link/P1494R3`

[P2053R1] Rostislav Khlebnikov and John Lakos, "Defensive Checks Versus Input Validation", 2020
`http://wg21.link/P2053R1`

[P2695R0] Timur Doumler and John Spicer, "A proposed plan for contracts in C++", 2022
`http://wg21.link/P2695R0`

[P2899R0] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, "Contracts for C++ — Rationale", 2024
`http://wg21.link/P2899R0`