Contracts for C++

Document #: P2900R6
Date: 2024-02-29
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>
Tom Honermann <tom@honermann.net>
Lisa Lippincott <lisa.e.lippincott@gmail.com>
Jens Maurer <jens.maurer@gmx.net>
Jason Merrill <jason@redhat.com>
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 the highest bar possible for consensus. The proposal includes syntax for specifying three kinds of contract assertions — precondition assertions, postcondition assertions, and assertion statements. In addition, we specify the range of runtime semantics these assertions may have – ignoring, observing, or enforcing the truth of their predicates — 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 6 (Forwarded to EWG and LEWG for design review)

- Allowed attributes in general, and `[[maybe_unused]]` specifically, to appertain to the result name
- Made it ill-formed for an *await-expression* or *yield-expression* to appear in the predicate of `contract_assert`
- Clarified that evaluating a predicate with side effects during constant evaluation may lead to an ODR violation
- Expanded section Design Principles
- 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 `pre` and `post` on virtual functions ill-formed
- Removed function `contract_violation::will_continue()`
- Removed enum value `detection_mode::evaluation_undefined_behavior`
- Introduced distinct terms *function contract specifier* for the syntactic construct and *function contract assertion* for the entity it introduces
- Added rules for equivalence of function contract specifier sequences
- Allowed repeating the function contract specifier sequence 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 section Design Principles
- 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 only be elided 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 `pre` and `post` on deleted functions ill-formed
- 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 2023-11 Kona Meeting SG21 Feedback)

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

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 2023-06 Varna Meeting SG21 Feedback)

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

1 Introduction

There is a long and storied history behind the attempts to add a Contracts facility to C++. The current step we, collectively, are on 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.

In this paper you will find three primary sections. The first, Section 2, introduces the general concepts and the terminology that will be used throughout this paper and provides an overview over the scope of the proposal. The second, Section 3, describes the design of the proposed Contracts facility carefully, clearly, and precisely. The third, Section 4, 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, and contain the needed wording to match that information.

What this paper is explicitly *not* is a collection of motivation for the use of contracts, instructions on how to use them, the history of how this design came to be, or an enumeration of alternative designs that have been considered. To avoid excessive length of this paper, we have factored out all of this information into a sister-paper to this one, [D2899R0] — Contracts for C++ — Rationale. That paper will contain, for each section or subsection of the design section of this paper, as complete a history as possible for the decisions in that section. That paper will also, importantly, contain citations to the *many* papers written by the valiant 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 hopefully, in general, many of the other papers discussing these topics. After that, we will go over the basic features and scope of the proposed Contracts facility.

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 on how the component interoperates with other components in a correct program, 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 runtime, while contract violations can usually only be addressed by fixing the bug in the code.

A *precondition* is a part of a function contract where the responsibility for satisfying it rests in the hands of the caller of the function. Generally, these are requirements placed on the arguments passed to a function and/or the global state of the program upon entry into the function.

A *postcondition* is a part of a function contract where the responsibility for satisfying the condition lies in the hands of the callee, i.e. the implementer of the function itself. These 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 a condition that a class type maintains throughout the lifetime of an object of that type between calls to its public member functions. There are other kinds of invariants, such as loop invariants. Often these are 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.

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

Often, some provisions of a plain language contract can 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* 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++. There are a number of powerful use cases for such a facility:

1. Runtime checking of contract assertions to identify violations
2. Static analysis and formal verification
3. Documenting contracts in code consumable by both readers and tooling
4. Guiding optimization to improve performance

The C++ Standard primarily concerns itself with how C++ programs behave, and so the specification of runtime checking of contract assertions dominates this proposal. With that behavior well specified, however, all of the other above use cases are enabled as well. Fully optimizing based on contract assertions — i.e., allowing the compiler to *assume* that some subset of contract assertions are not violated — is not, however, proposed here, though this will be addressed in a future paper.

The existing `assert` macro in header `<cassert>` is arguably also a Contracts facility, albeit a very rudimentary one. It does not offer any contract-violation handling strategy other than calling `std::abort()` or any kind of customization of its behavior other than token-removal of the entire assertion by defining `NDEBUG`. For this reason, many code bases resort to custom assertion macros that offer more options. However, these macros have no language support and thus suffer from a number of limitations (cannot be placed on function declarations, difficult to use across different libraries and code bases, plus all the known issues with macros). It is a main goal of this proposal to provide a superior alternative to the usage of such assertion macros.

2.2 Proposed Features

The Contracts facility we propose will enable placing precondition and postcondition assertions on a function declaration, and assertion statements in function bodies:

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

Precondition and postcondition assertions are collectively called *function contract assertions*. In most cases, precondition and postcondition assertions are used to check preconditions and postconditions, respectively, although there are some cases where one might use an assertion statement at the beginning of a function body or even a postcondition assertion to check a precondition.

Each contract assertion has a *predicate* which is a potentially evaluated expression that will be contextually converted to `bool` in order to verify compliance with the provision of the contract expressed by the contract assertion or to identify a contract violation. When the predicate evaluates to `true`, compliance has been verified. When the predicate evaluates to `false`, or evaluation of the predicate exits via an exception, there is a contract violation. 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, there will be a contract violation 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
}
```

The syntactic construct that introduces a function contract assertion is called a *function contract specifier* (a *precondition specifier* or *postcondition specifier*, respectively). The function contract specifiers of a function introduce, and effectively define, the function contract assertions of that function. These must be placed on first declarations and may be repeated or omitted on redeclarations. All parts of a program that can reach a declaration of a function must see the equivalent sequence of function contract assertions for that function.

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 done with a specific contract *semantic*, which may or may not evaluate the predicate. Which semantic is chosen is implementation-defined. In practice, it will most likely be controlled by a command-line option to the compiler, although platforms might provide other avenues for selecting a semantic, and the exact forms and flexibility of this selection is not mandated by this proposal. The semantic may vary from one evaluation of a contract assertion to the next or, possibly, may be the same across all evaluations.

The *ignore* semantic does nothing. Note that even though the predicate of an ignored contract assertion is not evaluated, it is still parsed and is a *potentially evaluated* expression, i.e. it ODR-uses entities that it references. Therefore it must always be a well-formed, evaluable expression.

The *enforce* semantic verifies compliance and identifies contract violations. If a contract violation happens at compile time, the program is ill-formed; at runtime, the contract-violation handler will be invoked. If the contract-violation handler returns normally, the program will be terminated by calling the function `std::abort()`. If compliance has been verified, program execution will continue from the point of evaluation of the contract assertion.

The *observe* semantic verifies compliance and identifies contract violations. If a contract violation happens at compile time, the compiler will emit a warning; at runtime, the contract-violation handler will be invoked. If compliance has been verified, or if the contract-violation handler returns normally, program execution will continue from the point of evaluation of the contract assertion.

Because both the *enforce* and *observe* semantics must verify compliance with the provision of the contract expressed by the contract assertion and identify contract violations, these are called *checked* semantics. Because the *ignore* semantic does *not* need to, nor is it even allowed to, evaluate the predicate and detect a contract violation, it is called an *unchecked* semantic. The evaluation of a contract assertion with a checked semantic is also called a *contract check*; evaluating a contract assertion with a checked semantic is also called *checking* the contract assertion.

When checking a contract assertion, the value of the predicate is determined either by evaluating the predicate expression or by evaluating a side-effect free expression that produces the same result (in other words, the side effects of evaluating the predicate can be elided). Contract assertions can also be checked during constant evaluation; in this case, evaluating a predicate that is not a core constant expression is also considered a contract violation.

Finally, there is a function `::handle_contract_violation`, called the *contract-violation handler*, that will be invoked when a contract violation has been detected at runtime. The implementation-provided version of this function, the *default contract-violation handler*, has implementation-defined effects; the recommended practice is that the default contract-violation handler outputs diagnostic information about the contract violation. It is implementation-defined whether this function is replaceable, giving the user the ability to 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>` 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 contract semantic.

Note that not all parts of a contract can be specified via contract assertions, and of those who can, some cannot be checked at runtime 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, cover the entire plain-language contract of a function; however, they should always specify a *subset* of the plain-language contract.

2.3 Features Not Proposed

To keep the scope of this MVP proposal minimal (while still viable), the following features are intentionally not included in this proposal; however, we expect these features to be proposed as post-MVP extensions at a later time:

- The ability to specify precondition and postcondition assertions for virtual functions
- The ability to refer to "old" values of parameters and other entities (from the time when the function was called) in the predicate of a postcondition
- The ability to assume that an unchecked contract predicate would evaluate to `true`, and allow the compiler to optimize based on that assumption, i.e. the *assume* semantic
- The ability to express the desired contract semantic directly on the contract assertion
- The ability to assign an assertion level to a contract assertion, or more generally 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, for example 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 runtime (for example, because it calls a function with no definition)
- The ability to express invariants

Most of the above features were part of previous Contracts proposals in some shape or form; however, as a general rule, nothing in these 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 correct choices for how the facility should work and how it should integrate with the full breadth of the C++ language.

The primary principle that leads to many of those decisions, and which should be kept in mind for future language changes and how they will, in turn, interact with Contracts, is that contract assertions must enable observation of the correctness of existing programs. This means that the facility itself must not fundamentally force a program to change in such a way that it is arguably no longer doing the same thing outside of the contract assertions themselves.

From this basic principle, we can find three actionable principles to follow:

- **Concepts do not see Contracts** — if the mere presence of a contract assertion, independent of the predicate within that assertion, on a function or in a block of code would change when the satisfiability of a concept then a contained program could be substantially changed by simply using contracts in such a way. Therefore, we remove the ability to do this. As a corollary, the addition or removal of a contract assertion must not change the result of SFINAE, the result of overload resolution, the result of the `noexcept` operator, or which branch is selected by an `if constexpr`.

- **Zero Overhead** — The presence of a contract check that is not actually checked — i.e., that is *ignored* — must not have impact on how a program behaves, for example by triggering additional lambda captures that result in the addition of additional member variables to closure objects.
- **Chosen Semantic Independence** — Which contract semantic will be used for any given evaluation of a contract assertion, and whether it is a checked semantic, must not be detectable at compile time, as such detection may result in different programs being executed when contract checks are enabled.

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

- **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 it is attached so that it will verify the plain language contract (or some subset of the plain language contract) of *that* function and not of some other function.[1]
- **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:
  - Callers promise to satisfy a function's preconditions, resulting in callers being able to rely upon those preconditions being true.
  - 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 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*.
- **Contract Assertions are not Flow Control** — A contract assertion provides an algorithm to validate correctness, but importantly nothing about a contract assertion guarantees always associating any particular runtime behavior with that syntactic construct. An unadorned contract assertion[2] might *enforce* the associated condition, terminating if it is violated, but it might equally do nothing at all in another build, allowing violations to happen.

[1]In particular, the function contract assertions attached to a virtual function must not implicitly be applied to all overriding functions, but rather should apply only when invoking the function through a pointer or reference to that particular base class.

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

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

The design of this proposal has also been guided by an additional principle about how to address open design questions for which solutions are not yet agreed upon or known:

- **Explicitly Define all New Behavior** — For any behavior that we define as part of a Contracts facility, there are many cases where certain rules must be followed. Enforcing those rules can be done 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, and thus have chosen to never explicitly introduce new undefined behavior when evaluating contract assertions.
- **Choose Ill-Formed to Enable Flexible Evolution** — When clear consensus on the proper solution to a problem that Contracts could address has not become apparent, the choice has been made to leave the relevant constructs ill-formed rather than giving them unspecified or undefined behavior. This choice enables conforming extensions to explore possible options while leaving open all options for an eventual solution being incorporated into the C++ Standard.

3.2 Syntax

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

```
int f(const int x)
  pre (x != 1)                // precondition specifier
  post (r : r != 2)           // postcondition specifier
{
  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 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 *assignment-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

[3]See [P2053R1].

[4]The distinction between precondition and postcondition specifiers on the one hand and precondition and postcondition assertions on the other hand is analogous to the distinction between `noexcept`-specifiers and exception specifications: The former refers to the syntactic construct, while the latter refers to the conceptual entity that is a

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 only parsed as part of a function contract specifier when they appear in the appropriate syntactic position for the latter. 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[5], after trailing return types and requires clauses, immediately before the semicolon in a declaration:

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

When function contract specifiers appear on a definition, they appear in the corresponding location in the declaration part of the definition, immediately prior to the function body (noting 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; };
```

property of a function. The distinction is important because a function that has function contract assertions may have multiple declarations, some of which may not have function contract specifiers (see Section 3.3.1 for details). Note that no such distinction is necessary for assertion statements.

[5]Should function contract specifiers be allowed on virtual functions in the future, the intention is to add an exception to the above rule to place contract checks prior to the pure-specifier `= 0` when it is present, for visual consistency with `= default`.

```
  return f(42);
}
```

There may be any number of function contract specifiers, in any order, 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 postconditions will still be done in their respective lexical order, see Section 3.5.1.

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. This is necessary in order to be able to disambiguate an assertion statement 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>`.

Unlike the `assert` macro, `contract_assert` is not an expression:

```
const int j = (contract_assert(i > 0), i);  // syntax error
```

Using `contract_assert` in places that require an expression rather than a statement can be accomplished by wrapping it into a lambda. Alternatively, such usages are often better expressed with a precondition assertion (for example, replacing a subexpression in the member initializer list of a constructor with a precondition assertion on that constructor).

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, however this permission can be useful for vendor-specific extensions to the functionality provided by this proposal. The syntactic location for such contracts-specific attributes is in between the `pre`, `post` or `contract_assert` and the predicate:

```
bool binary_search(Range r, const T& value)
  pre [[vendor::message("A non-sorted 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 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 Restrictions on the Placement of Contract Assertions

3.3.1 Multiple Declarations

Any function declaration is a *first declaration* if there are no other declarations of the same function 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 there are multiple first declarations for the same function 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 where a function might be used or defined see a consistent and unambiguous view of what the sequence of function contract specifiers of that function is.

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 that the names of function parameters, names of template parameters, and the result name may be different[6] (the entities found by name lookup will be the same).

3.3.2 Virtual Functions

It is ill-formed for a declaration of a virtual function to have precondition or postcondition specifiers. The intention is to add support for virtual functions in a future extension.

3.3.3 Defaulted and Deleted Functions

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

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

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

3.3.4 Coroutines

It is ill-formed for a declaration of a coroutine — i.e. a function containing a `co_yield`, `co_return`, or `co_await` statement — to have precondition or postcondition specifiers. This restriction might be relaxed in a future extension.

It is valid to use `contract_assert` within the body of a coroutine, however an *await-expression* or *yield-expression* may not appear in the predicate of `contract_assert` as a subexpression that is in the suspension context of that coroutine:

[6]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.

```
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_assert` if it is not in the suspension context of that coroutine, for example because it appears inside an immediately invoked lambda which is not suspending the evaluation of the function or coroutine evaluating the predicate itself:

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

3.3.5 Function Pointers

A contract specifier may not be attached to a function pointer. The contract assertions on a function have no impact on its type, and thus no impact on the type of its address, nor what types of function pointers that address may be assigned to.

When a function *is* invoked through a function pointer, its function contract assertions must still be evaluated as normal.

3.4 Semantic Rules for Contract Assertions

3.4.1 Name Lookup and Access Control

For precondition assertions, name lookup in the predicate is generally performed as if the predicate came at the beginning of the body of the function or lambda expression. This name lookup occurs as if there were a function body 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 may 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 (however, there is a special rule that the program is ill-formed if such references trigger implicit lambda captures; see Section 3.4.7). 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 of Local Entities

A contract check is supposed to observe the state of the program, not change it, exceptions such as logging notwithstanding. To prevent accidental bugs due to unintentional modifications of entities inside a contract predicate, identifiers referring to local variables and parameters inside a contract predicate are `const` lvalues. This is conceptually similar to how identifiers referring to members are implicitly `const` lvalues inside a `const` member function. In particular, in a contract predicate,

- an identifier that names a variable with automatic storage duration of object type `T`, a variable with automatic storage duration of type reference to `T`, or a structured binding of type `T` whose corresponding variable has automatic storage duration, is an lvalue of type `const T`;
- `*this` is implicitly `const`.

These `const` amendments are shallow (on the level of the lvalue only); attempting to invent "deep const" rules would make raw pointers and smart pointers likely behave differently, which is not desirable. The type of lvalues referring to namespace-scope or local static variables is not changed; such accesses are more likely to be intentionally modifying, e.g. for logging or counting:

```
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))             // OK
{
  contract_assert((x = 0));     // error: assignment to const lvalue
  int var = 42;
  contract_assert((var = 42)); // error: assignment to const lvalue

  static int svar = 1;
  contract_assert((svar = 1)); // 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, as it still produces the declared type of the entity denoted by `x`. However, `decltype((x))` yields `const T&`, where `T` is the type of the expression `x`.

Modifications of local variables and parameters inside a contract predicate are possible — although discouraged — via applying a `const_cast`, except that 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, arguably such an overload set that yields different results depending on the `const`-ness of the parameter is in itself a bug.

When a lambda inside a contract predicate captures a non-function 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
    {}
```

```
};
```

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 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 a lvalue of type `const T`; `decltype(r)` is `T`, while `decltype((r))` is `const T&`. This is consistent with the implicit `const`-ness of identifiers naming local entities and parameters in contract predicates (see Section 3.4.2).

Modifications of the return value in the postcondition assertion predicate are possible via applying a `const_cast`, although they are strongly discouraged. Note that even in the case that 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; // non—copyable non—movable
  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
```

It might be useful to clarify the relevant existing wording to make this intent more clear; such a clarification is being proposed in [CWG2841].

The address of the result name refers to the address of the result object, except for trivially copyable types, for which it may also refer to a temporary object created by implementation that will later be used to initialise the return object; this dispensation exists to make sure that adding a postcondition assertion does not alter a function's ABI by making it impossible to pass the return value in a register.

This means that for non-trivially copyable types, we now have a reliable way to obtain the address of the result object inside a postcondition assertion, something that was previously not possible:

```
X f(X* ptr)
  post(r: &r == ptr)  // guaranteed to pass (for the call from main below)
                      // if X is not trivially copyable
{
  return X{};
}

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

If a postcondition names the return value on a non-templated 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 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
```

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, it would be impossible to reason about postcondition predicates on a function declaration 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;
}
```

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

3.4.5 Not Part of The Immediate Context

The predicate of a function contract assertion, while lexically a 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 a function body can — causing a program to be ill-formed in an unrecoverable fashion (i.e., not subject to SFINAE) if those requirements are not met for a given set of function template arguments.

3.4.6 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
```

3.4.7 No Implicit Lambda Captures

For lambdas with default captures, contract assertions that are part of the lambda need to be prevented from triggering lambda captures that would otherwise not be triggered. Otherwise, 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 the Zero Overhead principle described in Section 3.1. 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 And Contract-Violation Handling

3.5.1 Point of Evaluation

All precondition assertions attached to a function are evaluated after the initialization of function parameters and before the evaluation of the function body begins. Note that a constructor's member initializer list and a function-try block are considered to be part of the function body.

All postcondition assertions attached to a function are evaluated after the return value has been initialized and local automatic variables have been destroyed, but prior to the destruction of function parameters.

Multiple precondition or postcondition assertions are evaluated in the order in which they are declared.

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

3.5.2 Contract Semantics: *Ignore*, *Enforce*, *Observe*

Each evaluation of a contract assertion is done with a contract *semantic* that is implementation-defined to be one of *ignore*, *enforce*, or *observe*.[7] This is true for evaluations during runtime as well as for evaluations during constant evaluation (at compile time). Chains of consecutive evaluations of contract assertions may have individual contract assertions repeated any number of times (with certain restrictions and limitations — see Section 3.5.5), and may involve evaluating the same contract assertion with different semantics.

The *ignore* semantic does not attempt to verify compliance or detect contract violations. It is therefore an *unchecked* semantic. The only effects of an *ignored* contract are that the predicate is parsed and the entities it references are ODR-used. Note that this makes an ignored contract assertion different from an ignored `assert` macro (if `NDEBUG` is defined): in the former case, the predicate is never evaluated, but it still needs to be a well-formed, evaluable expression, while in the latter case, the tokens comprising the predicate are entirely removed by the preprocessor.

The *enforce* semantic is a *checked* semantic. It verifies compliance and detects contract violations. If there is a contract violation during constant evaluation, the program is ill-formed; otherwise, if the contract violation happens at runtime, the contract-violation handler will be invoked. If the contract-violation handler returns, the program will be terminated by calling the function `std::abort()`. If contract compliance has been verified, program execution will continue from the point of evaluation of the contract assertion.

The *observe* semantic is a *checked* semantic. It verifies compliance and detects contract violations. If there is a contract violation during constant evaluation, the compiler issues a diagnostic (a warning); otherwise, if the contract violation happens at runtime, the contract-violation handler will be invoked. If compliance has been verified, or the contract-violation handler returns normally, program execution will continue from the point of evaluation of the contract assertion.

In addition to the three contract semantics proposed in this paper, an implementation may provide additional contract semantics, with implementation-defined behavior, as a vendor extension.

3.5.3 Selection of Semantics

The semantic a contract assertion will have is implementation-defined. The selection of semantic (*ignore*, *enforce*, or *observe*) may happen at compile time, link time, load time, or runtime. Different contract assertions can have different semantics, even in the same function. The same contract assertion may even have different semantics for different evaluations. This is true both for evaluations during constant evaluations and for evaluations at runtime.

The semantic a contract assertion will have cannot be identified through any reflective functionality of the C++ language. It is therefore not possible to branch at compile time on whether a contract

[7] Implementation-defined semantics beyond these might also be available as compiler extensions.

assertion is checked or unchecked, or which concrete semantic it has. This is another important difference between contract assertions and the `assert` macro.

It is expected that there will be various compiler flags to choose globally the semantics that will be assigned to contract assertions, and that this flag does not need to be the same across all translation units. Whether the contract assertion semantic choice for runtime evaluation can be delayed until link or runtime is also, similarly, likely to be controlled through additional compiler flags.

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

When nothing else has been specified by a user, it is recommended that a contract assertion will have the *enforce* semantic at runtime. It is understood that compiler flags like `-DNDEBUG`, `-O3`, or similar might be considered to be "doing something" to indicate a desire to prefer speed over correctness, and these are certainly conforming decisions. The ideal, 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 will be provided by notifying that student of their own mistakes.

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

3.5.4 Checking the Contract Predicate

When a contract assertion is being checked, i.e. evaluated with a checked semantic, the result of evaluating the predicate must be determined.

If the result of the predicate can be determined, there are two possible such results:

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

If the predicate evaluates to `true`, compliance with the provision of the contract expressed by the contract assertion has been verified. Execution will continue normally after the point of evaluation of the contract assertion.

If the predicate evaluates to `false`, a contract violation has been detected. The contract-violation handling process will be invoked; if the contract violation occurs at runtime, the contract-violation handler will be called with the value `predicate_false` for `detection_mode` (see Section 3.5.8).

If evaluation of the predicate does not produce a value, there are two more possible outcomes of the contract check:

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 3.5.9).

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[8], and the contract-checking process will treat it as such. When this happens because of a thrown exception at runtime, the contract-violation handler will be called with the value `evaluation_exception` for `detection_mode`.

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.

3.5.5 Consecutive and Repeated Evaluations

A vacuous operation is one that should not, a priori, be able to alter the state of a program which a contract could observe, and thus 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
- performing trivial destruction, including destruction of scalars and invoking trivial destructors
- initializing reference variables
- transfer of control via function invocation or a return statement — though note the corresponding function parameter and result value initialization might not be vacuous

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 precondition assertions on a single function when invoking that function
- Checking all postcondition assertions on a single function when that function returns normally
- Checking consecutive assertion statements
- Checking the precondition assertions of a function and any assertion statements that are at the beginning of the body of that function

[8]It is possible that this situation 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 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.

- Checking 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` involves only vacuous operations
- Checking the postcondition assertions of a function `f1` and the precondition assertions of the next function `f2` invoked immediately after `f1` returns, when the destruction of the arguments of `f1` and the preparation of the arguments of `f2` involve 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 contract semantic.[9]

In practice, this means 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 first 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, #1 may repeated again any time later during the sequence. #2 will always be evaluated after #1 has been evaluated at least once, and after that, #2 too may be evaluated again. On many platforms, the simplest sequence #1 — #2 will be evaluated, with each precondition being evaluated exactly once, in order. In other situations, where both caller-side and callee-side checking is being performed, 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, etc.

while the following are not:

#2 — #1,
#2 — #2,
#1,
#1 — #1, etc.

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 that an evaluation with the *ignore* semantic also counts as an evaluation even though the predicate is not evaluated.

[9]Note that an equivalent formulation of this 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.

3.5.6 Predicate Side Effects

The predicate of a contract assertion is an expression that, when evaluated, follows the normal C++ rules for expression evaluation. It 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 program termination), it is allowed to elide all of 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 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 with 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 may also not introduce new side effects.

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

If the compiler cannot prove that evaluation of the predicate will not exit via an exception, it is not allowed to elide the evaluation of the predicate, because the thrown exception must be available via `std::current_exception` in the contract-violation handler (see Section 3.5.8).

Likewise, if the compiler cannot prove that evaluation of the predicate will not call `longjmp` or cause program termination, it 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 3.5.5, 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, this may result in multiple violations happening:

```
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 one 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 therefore a contract check is as-if-equivalent to evaluating the predicate once.

3.5.7 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 detected at runtime.

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. There is no user-accessible declaration of the default contract-violation handler provided by the Standard Library, and no way for the user to call it directly. Whether the default contract-violation handler itself is `noexcept` is implementation-defined, though the recommended implementation certainly could be.

Whether this function 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` — by defining a function with the correct signature (function name and argument type) and return type that 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)`. This is a primary reason that there is no declaration for `::handle_contract_violation` provided in the Standard Library, as 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 it is ill-formed, no diagnostic required to provide a function with the signature and return type needed to attempt to replace the default contract-violation handler. This allows platforms to 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.

3.5.8 The Contract-Violation Handling Process

When a contract violation (see Section 3.5.4) is detected at runtime, 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), `contract_kind` (the kind of contract assertion — `pre`, `post`, or `contract_assert`), and `semantic` (the contract semantic of the evaluation 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, except that the memory for it is not allocated via `operator new` (similar to the memory for exception objects). It may already exist in read-only memory, or it may be populated at runtime on the stack. The lifetime of this object will continue at least through the point where 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. Therefore, inside the contract-violation handler, that exception is the currently handled exception and is available via `std::current_exception`.

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

- `std::contracts::contract_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 3.5.9) or at runtime. The result may be a compile-time value (for example, controlled by a compiler flag or a platform-specific annotation on the contract assertion) or, for a contract evaluation at runtime, may even be a value determined at runtime based on what the platform provides.
- `__check_predicate(`X`)`: Determine the result of the predicate X at runtime — by either 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(contract_semantic, detection_mode)`: Handle a runtime contract violation of the current contract. This 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 of its properties must last through the invocation of the contract-violation handler.

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

```
contract_semantic _semantic = __current_semantic();
if (contract_semantic::ignore == _semantic) {
  // do nothing
}
else if (contract_semantic::observe == _semantic
      || contract_semantic::enforce == _semantic)
{
  // checked semantic

  if consteval {
    // see Section 3.5.9
  }
  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 (...) {
      // Handle violation within exception handler
      _violation = true;
      __handle_contract_violation(_semantic,
                                  detection_mode::evaluation_exception);
      _handled = true;
    }
    if (_violation && !_handled) {
      __handle_contract_violation(_semantic,
                                  detection_mode::predicate_false);
    }

    if (_violation && contract_semantic::enforce == _semantic) {
      abort();
    }
  }
}
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 only used 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. There are many ways in which these could be accomplished, the exposition-only boolean variables above are just one possible solution.

One important takeaway from having the semantic of evaluation being effectively unspecified until runtime is that, unlike a macro-based solution, a contract assertion's definition is the same even though individual evaluations may have different semantics. This means that an implementation which supports mixing translation units where contract assertions are configured to have different semantics is not, in and of itself, an ODR violation.

3.5.9 Compile-time Evaluation

Contract assertions may be evaluated during constant evaluation (at compile time). During constant evaluation, the three possible contract semantics have the following meaning:

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

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

- The result is `true` — compliance has been verified.
- The result is `false` — a contract violation has been detected.
- The predicate is not a core constant expression — a contract violation has been detected.

In order to satisfy the "Concepts do not see Contracts" design principle described in Section 3.1, the presence of a contract assertion must not alter whether containing expressions are or are not eligible to be constant expressions, in particular because it is possible to SFINAE on whether an expression is a core constant expression. Therefore, evaluating a contract assertion never makes an expression not eligible to be a core constant expression — although if its predicate is ineligible to be evaluated, that will result in a contract violation.

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 runtime) depending on whether the initializer is a core constant expression:

```
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 firsts determines whether the initializer is a core constant expression by performing trial evaluation[10] 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*, *enforce*, or *observe* chosen in an implementation-defined manner. This evaluation behaves normally with regards to possible contract violations.

This rule is again derived from the Zero Overhead principle in Section 3.1. 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 runtime (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 avoids aggressive enforcement of contract checks at compile time for functions that would otherwise be evaluated at runtime (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 call to `compute_at_runtime`) and `j` will therefore be initialized at runtime, at which point the precondition passes. The above program therefore contains no contract violations.

[10]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.

The rules regarding elision and duplication of side effects described in Section 3.5.6 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 for the function `g` — an ODR violation. Considering that such ODR violations only happen when function-contract assertions are already doing something ill-advised by jumping through `const_cast` hoops to modify function parameters, this is a recognized but not significant 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 (*observe* or *enforce* semantic) or not evaluated (*ignore* semantic) when determining the size of the array `a`.

3.6 Noteworthy Design Consequences

3.6.1 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 it can be hard to reason about which declarations are first declarations and which are redeclarations, 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 only becomes reachable from the point where 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 it has a precondition specifier, that specifier is compared to the previous declaration of `f` and we determine that it matches (it is, 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 as 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, 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 such hard-to-reason-about cases, when using a friend declaration of a function with function contract assertions inside a template, it is recommended to always do one of the following:

- Befriend functions that have already been declared further above, 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).

3.6.2 Recursive Contract Violations

There is no dispensation 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 checked semantic, the contract-violation handler will be called recursively. It is the responsibility of the user to handle this case explicitly if they wish to avoid overflowing the call stack (if the evaluation happens at runtime) or reaching the resource limits of the compiler (during constant evaluation).

3.6.3 Throwing Violation Handlers

There are no restrictions 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, for example terminating, calling `longjmp`, etc. In all cases, evaluation happens as described above. The same applies to the case when 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*.

3.6.4 Undefined Behavior

As stated in the design principles in Section 3.1, the design of this proposal has deliberately not introduced any new explicitly undefined behavior into the C++ language — and hopefully fails to introduce any other undefined behavior through new holes in the specification. At the same time, there is also no special protection against the evaluation of a predicate expression that has undefined behavior due to the existing rules for C++ expressions. In other words, if a contract assertion is evaluated with a checked 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(false)`.

With regards to undefined behavior occurring elsewhere *after* a contract assertion has been checked, the contract assertion does not formally constitute an optimization barrier that guards against "time travel optimization" as the C++ Standard does not specify such things. Consider:

```
int f(int* p) pre ( p != nullptr )
{
  std::cout << *p;   // undefined behavior
}

int main()
{
  f(nullptr);
}
```

This program has defined behavior if the contract semantic chosen for the precondition is *enforce* — a contract violation will be detected and control flow will not continue into the function. If the selected semantic is *ignore* this program will have undefined behavior — control flow will always reach the null pointer dereference within `f`. If the semantic is *observe* the program will have undefined behavior whenever the contract-violation handler returns normally. Even though *observe* is a *checked* semantic, the implementation is theoretically allowed to optimize out the contract check whenever it can determine that the contract-violation handler will return normally. We do not expect this to occur in practice, as the contract-violation handler will generally be a function defined in a different translation unit, acting as a de-facto optimization barrier.

It is hoped that, should the Standard adopt an optimization barrier such as `std::observable()` from [P1494R2], that barrier will be implicitly integrated into all contract assertions evaluated with the *observe* semantic.

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 very 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++. As these uses are not intended to be frequent, everything in this header is declared in namespace `std::contracts` rather than namespace `std`. In particular, the `<contracts>` header does *not* need to be included in order to write contract assertions.

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

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

  enum class contract_kind : int {
    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 contract_semantic : int {
    enforce = 1,
    observe = 2,
    // ignore = 3, // not explicitly provided
    // assume = 4  // 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 : int {
    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
  public:
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    contract_kind kind() const noexcept;
    std::source_location location() const noexcept;
    contract_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` with an underlying type of `int` to guarantee sufficient room for implementation-defined values. Implementations will know the full range of potential values, so the `contract_violation` object itself need not use that same data type or the full size of an `int` to store the values.

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

The following enumerations are provided:

- `enum class contract_kind : int`: Identifies one of the three potential kinds of contract assertion, with implementation-defined alternatives a possibility for when something invokes the contract-violation handler 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.
- `enum class contract_semantic : int`: A reification of the semantic that can be chosen for the evaluation of a contract assertion when that contract assertion is checked.
    - `enforce` and `observe`: These enumerators are provided explicitly as they can result in the invocation of the contract-violation handler.
    - `ignore`: This enumeration is not explicitly provided as there is currently no explicit need for it as ignored contract assertions do not invoke the contract-violation handler.
    - Implementation-defined values indicate other contract semantics that may be available as a vendor extension.
- `enum class detection_mode : int`: 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 runtime.
    - `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 an exception escaped that evaluation; this exception is available in the contract-violation handler via `std::current_exception`.
    - Implementation-defined values indicate an alternate method provided by the implementation in which a contract violation was detected.

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. While the `assert` enumerator might appear to be in conflict with the function-like macro of the same name defined in `<cassert>`, in practice there will be no issues as the enumerator will not be used immediately prior to an opening parenthesis and therefore not expanded as the function-like macro.

For all of 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 runtime. This object cannot be constructed, copied, moved, or assigned to by the user. It is implementation-defined whether it is polymorphic — if so, 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 multibyte string (NTMBS) in the ordinary literal encoding; it is otherwise unspecified. It is recommended that this value contain a textual representation of the predicate of the contract assertion which 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 one could assume that this information is not present in generated object files and executables.
- `detection_mode detection_mode() const noexcept`: The method by which a violation of the contract assertion was identified.
- `contract_kind kind() const noexcept`: The kind of the contract assertion which has been violated.
- `std::source_location location() const noexcept`: The value returned is unspecified. It is recommended that it be the source location of the caller of a function when a precondition is violated. For other contract assertion kinds, or when the location of the caller is not used, it is recommended 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 which either a meaningful value or a default-constructed value is returned.
- `contract_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 only provided by the implementation during contract-violation handling, this function can only be called 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 Contracts, for example to check the preconditions and postconditions specified for Standard Library functions, however 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 Contracts themselves as soon as they are available.

It is important to 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 in order 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 case of any 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 [N4971] modified by [CWG2700] and [P0609R2].

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 checked 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 semantic in a manifestly constant-evaluated context resulting in a contract violation.

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.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 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 `f`, which in turn is sequenced before every expression or statement in the body of `f`.~~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.

[*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 three *contract semantics*: *ignore*, *enforce*, or *observe*. The ignore contract semantic is an *unchecked semantic*; enforce and observe are *checked semantics*.

Which contract 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 contract 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 contract assertion with a checked semantic (enforce or observe) is called a *contract check*. 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 occuring. — *end note*]

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, for example 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 contract semantic is enforce, the program is ill-formed. Otherwise, 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]) using `std::current_exception` ([except.special.general]). *— end note*]

If the contract-violation handler returns normally, and the contract semantic is enforce, the function `std::abort()` is called ([support.start.term]).

If the contract-violation handler returns normally, and the contract semantic is observe, control flow continues normally after the point of evaluation of the contract assertion. [*Note:* The observe semantic provides the opportunity to install a logging handler to instrument a code base without having to exit the program upon contract violation. *— end note*]

If a contract-violation handler invoked from checking 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 this happens 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 evaluation 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 effectively vacuous evaluations 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 contract semantic. [*Note:* For example, this allows evaluating all function contract assertions twice, both in the caller's translation unit before invoking the function and in the callee's translation unit as part of the function body. This allowance also extends to evaluations of contract assertions during constant evaluation. *— end note*]

[*Example:* Different contract 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 during constant evaluation:

```
constexpr int f(int i)
{
  contract_assert(++const_cast<int&>(i), true);  // #1
  return i;
}
inline void g()
```

```
{
  int a[f(1)];   // size dependent on the semantic and number of evaluations of #1
}
```

— end example]

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* appears in the predicate of a contract assertion ([basic.contract]) and the entity is

— the result object of (possibly deduced, see [dcl.spec.auto]) type T of a function call and the *unqualified-id* is the result name ([dcl.contract.res]) in a postcondition assertion,

— is a variable with automatic storage duration of object type T,

— a structured binding of type T whose corresponding variable has automatic storage duration, or

— a variable with automatic storage duration of type "reference to T",

then the type of the expression is const T. [*Note:* A function parameter is a variable with automatic storage duration. — *end note*]

[6] [*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*]

[7] 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 is intended to prevent situations where 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
    }()) {};
}
```

— *end example*]

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 ([dcl.contract.func]) 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, which are evaluated in sequence ([dcl.contract.func]).

Modify [expr.await], paragraph 2:

> An *await-expression* shall appear only in a potentially-evaluated expression within the *compound-statement* of a *function-body* 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. An *await-expression* shall not appear in the predicate of a contract assertion ([basic.contract]). 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 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 semantics, resulting in a diagnostic or ill-formed program if there is a contract violation. — *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 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 new paragraph after [stmt.return], paragraph 3:

All postcondition assertions ([dcl.contract.func]) of the function are evaluated in sequence. The destruction of all local variables within the function body is sequenced before the evaluation of any postcondition assertions. [*Note:* This in turn is 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, in turn, are sequenced before the destruction of local variables in each remaining enclosing block of the function, then the evaluation of postcondition assertions, 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-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*]

A coroutine ([dcl.fct.def.coroutine]), a virtual function ([class.virtual]), 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*.

Access control rules are applied to the predicate of a function contract assertion as if it were the first expression in the declared function. [*Example:*

```
class X {
private:
  int m;
public:
  void f() pre(m > 0);             // OK
  friend void g(X x) pre(x.m > 0); // OK
};

void h(X x) pre(x.m > 0);          // error: m is a private member
double i;
int j;
auto l1 = [i = j] pre(i > 0) {};   // OK, refers to captured int i
```

— *end example*]

If the predicate of a postcondition assertion of a function odr-uses ([basic.def.odr]) a non-reference parameter of that function, that parameter shall be declared const. [*Note:* This applies even to declarations that do not specify the *postcondition-specifier*. — *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;
}
```

— *end example*]

When a set of function contract assertions 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 the *function-contract-specifier* introducing X lexically precedes the one introducing Y.

[*Note:* The precondition assertions of a function are evaluated in sequence when the function is invoked ([intro.execution]). The postcondition assertions of a function are evaluated in sequence when a function returns normally ([stmt.return]). — *end note*]

[*Note:* The function contract assertions of a function are evaluated even when invoked indirectly, such as through a function pointer. Function pointers cannot have a *function-contract-specifier-seq* associated directly with them. — *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]), the result name may instead denote that temporary object. [*Note:* It follows that for objects that can be returned in registers, the address of the object referred to by the result name might be a temporary materialized to hold the value before it is used to initialize the actual result object. Modifications to that temporary's value are still expected to be retained for the eventual result object. — *end note*] [*Example:*

```
  struct A {};  // trivially copyable

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

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

  int main() {
    A a = f(&a);  // postcondition check may fail
    B b = f(&b);  // 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 h(int& p)
  post (p >= 0)       // OK
  post (r: r >= 0)    // OK
{
  return p = 0;
}
```

— *end example*]

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

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 implementation 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
  declarator requires-clause$_{opt}$ *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 1:

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 [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-specifier*s ([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-specifier*s of a function template specialization are instantiated when the function contract assertions of that function are needed (see [dcl.contract.func]). ~~The *noexcept-specifier* 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 checking 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 [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 t~~The following dynamic memory allocation function~~ signature~~s 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&)
```

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.

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 contract_kind : int {
    pre = 1,
    post = 2,
    assert = 3
  };

  enum class contract_semantic : int {
    enforce = 1,
    observe = 2
  };

  enum class detection_mode : int {
    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;
    detection_mode detection_mode() const noexcept;
    contract_kind kind() const noexcept;
    source_location location() const noexcept;
    contract_semantic semantic() const noexcept;
  };

  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

Enum class `<contract_kind>` **[support.contracts.kind]**

The type `contract_kind` specifies the syntactic form of the contract assertion ([basic.contract]) whose check resulted in the contract violation. Its enumerated values and their meanings as as follows:

— `contract_kind::pre`: the evaluated contract assertion was a precondition assertion.

— `contract_kind::post`: the evaluated contract assertion was a postcondition assertion.

— `contract_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 `<contract_semantic>` **[support.contracts.semantic]**

The type `contract_semantic` specifies the contract semantic ([basic.contract.eval]) of the evaluation that resulted in the contract violation. Its enumerated values and their meanings as as follows:

— `contract_semantic::enforce`: the contract assertion was evaluated with the enforce contract semantic.

— `contract_semantic::observe`: the contract assertion was evaluated with the observe contract 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`.

[*Note:* No enumeration value for the ignore semantic is provided because evaluations with the ignore semantic cannot result in a contract violation. — *end note*]

Enum class `<detection_mode>` **[support.contracts.detection]**

The type `detection_mode` specifies the manner in which a contract violation was detected ([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 evaluation 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 only be created by the implementation. Whether the destructor is virtual is implementation-defined.

```
const char* comment() const noexcept;
```

Returns: An implementation-defined null-terminated multi-byte 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, reformatted, represent the code before or after preprocessing, or summarized. An implementation can return an empty string if storing a textual representation of violated predicates is undesired.

```
detection_mode detection_mode() const noexcept;
```

Returns: The manner in which the contract violation was detected.

```
contract_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.

`contract_semantic semantic() const noexcept;`

Returns: The contract semantic with which the violated contract assertion was evaluated.

`<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 having a Contracts facility in the C++ Standard has been worked on actively for nearly two decades. This proposal represents the culmination of significant efforts to reach consensus in the Contracts study group (SG21). We feel that it 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 it 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 that helped shape this paper. Thanks to Andrei Zissu, Oliver Rosten, and Lewis Baker for reviewing an earlier draft of this paper and pointing out errors and omissions.

Bibliography

[CWG2700] Richard Smith, "#error disallows existing implementation practice"
https://wg21.link/cwg2700

[CWG2841] Tom Honermann, "When do const objects start being const?"
https://wg21.link/cwg2841

[D2899R0] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, "Contracts for C++ — Rationale", 2023
http://wg21.link/D2899R1

[N4971] Thomas Köppe, "Working Draft, Programming Languages – C++", 2023
http://wg21.link/N4971

[P0609R2] Aaron Ballman, "Attributes for Structured Bindings", 2023
http://wg21.link/P0609R2

[P1494R2] S. Davis Herring, "Partial program correctness", 2021
http://wg21.link/P1494R2

[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