Contracts for C++

Document #:	P2900R4
Date:	2024-01-16
Project:	Programming Language C++
Audience:	SG21 (Contracts)

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>
	Jens Maurer <jens.maurer@gmx.net>
	Ville Voutilainen <ville.voutilainen@gmail.com>

Abstract

After long, careful, and hopefully thorough deliberation, SG21 is delivering in this paper a proposal for a Contracts facility that has been carefully considered with the highest bar possible for consensus. With the features proposed here, C++ users will at long last have the ability to add contract assertions that may be leveraged in their ecosystems in numerous ways.

Contents

Revision History

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 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 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 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. Do not fear if you are looking for such information, however, for it is all amply available in the 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.

TODO: Note that there are still currently a number of remaining open design questions about the design presented in this paper. More details on the ongoing discussions related to those decisions can be found in [P2896R0], and a brief description of each of those issues be presented here in a box like this one.

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) as well as an overview over the 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 global objects when a function 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 a *plain language contract*. For example, the C++ Standard defines plain language contracts — preconditions and postconditions — for the functions in the C++ Standard Library.

Conditions that are part of a contract can also be specified in code. A language feature that allows the programmer to specify contract conditions in code is called a *Contracts facility*. Programming languages such as Eiffel and D have a Contracts facility; this paper proposes a Contracts facility for C++. A Contracts facility can make contract conditions checkable — at runtime and at compile time — to detect contract violations, verifiable, usable to guide static analysis and optimization, and consumable by other tooling. When used correctly, this approach can significantly improve the safety and correctness of software.

2.2 Components of the proposed Contracts facility

A *contract assertion* is a syntactic C++ construct that contains the information needed to specify an algorithm to detect the violation of a condition that is part of a contract.

We propose three kinds of contract assertions:

- A *precondition specifier* is attached to a function and is evaluated after function parameters are initialized and before the function body begins.
- A *postcondition specifier* is attached to a function and is evaluated when a function returns normally.
- An *assertion expression* is an expression of type `void` that may be placed wherever a `void` expression may be placed, and is evaluated when control flow reaches it.

Precondition and postcondition specifiers are named what they are because, in general that is the kind of contract condition for which they detect violations. Precondition and postcondition specifiers can be added to the declarator of a function or lambda expression, and are collectively called *function contract specifiers*; assertion expressions are expressions of type `void` and can be placed anywhere such expressions are allowed.

Each contract assertion has a *predicate* which is a potentially evaluated expression that will be contextually converted to `bool` in order to identify contract violations. When it would evaluate to `true`, there is no contract violation. When it would evaluate to `false`, there is a contract violation. Other results that pass through the evaluation of the contract assertion, such as throwing, are treated as contract violations as well. Other results that do not pass 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.

Each contract assertion has *points of evaluation* based on its kind and syntactic position. Precondition specifiers evaluate immediately after function parameters are initialized. Postcondition specifiers evaluate immediately after local variables in the function are destroyed when a function returns normally. Assertion expressions evaluate at the point in the function where control flow reaches them.

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.

The evaluation of a contract assertion does not necessarily mean evaluating its predicate. The exact meaning depends on the contract assertion semantic chosen, which is done in an implementation-defined manner — most likely 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. Each individual evaluation of a contract assertion is done with a specific contract *semantic*, and the implementation-defined mechanism for choosing this semantic may vary from one evaluation to the next (or, possibly, may be the same across all evaluations — read your implementation documentation and look at how you configured your build to find out.)

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, thus it odr-uses entities that it references. Therefore it must always be a well-formed, evaluable expression.

The *enforce* semantic determines if there has been a contract violation. If the contract violation happens at compile time, the program is ill-formed; otherwise, 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 there is no contract violation, program execution will continue from the point of evaluation of the contract assertion.

The *observe* semantic determines if there has been a contract violation. If the contract violation happens at compile time, the compiler will emit a warning; at runtime, the contract-violation handler will be invoked. If there is no contract violation, or 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 identify if there has been a contract violation, these are called *checked* semantics. They will always either evaluate the contract predicate or

evaluate a side-effect free expression that provably produces the same result as the predicate would.

Because the *ignore* semantic does *not* need to, nor is it even allowed to, detect a contract violation, it is called an *unchecked* semantic. The predicate will never be evaluated.

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 must be determined, either by evaluating a side-effect free expression that produces the same result or by evaluating the expression itself. A side-effect free expression would be one that has no side effects observable outside of the cone of evaluation of the expression — i.e., the only core-language side effects it contains are those that modify (non-volatile) variables whose lifetime starts and ends within the expression itself. If the resulting value is `true`, control flow continues as normal; when the value is `false` or an exception is thrown, there is a *contract violation*. Other forms of abnormal exit from evaluation (such as `longjmp` or program termination) happen as normal. At compile time, evaluating a predicate that is not a core constant expression also counts as a contract violation.

Note that not all contract conditions can be specified via a contract assertion, 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. Therefore, we do not expect that function contract specifiers 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 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 class invariants
- 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

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:

1. 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`.
2. 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 causing additional copy operations to occur.
3. 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.

3.2 Syntax

We propose three kinds of contract assertions: precondition specifiers, postcondition specifiers, and assertion expressions, introduced with `pre`, `post`, and `contract_assert`, respectively, followed by the predicate in parentheses:

```
int f(const int x)
  pre (x != 1)
  post (r : r != 2)
{
  contract_assert (x != 3);
  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

A function contract specifier is a contract assertion that may be applied to the declarator of a function (see Section 3.3.1 for which declarations) or of a lambda expression. We propose two kinds of function contract specifiers: precondition specifiers and postcondition specifiers.

A precondition specifier is introduced with `pre`:

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

A postcondition specifier is introduced with `post`:

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

A postcondition specifier may introduce a name to the return object of the function, called the *return-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 return object of f
```

The exact semantics of the *return-name* are discussed in Section 3.4.3.

`pre` and `post` are contextual keywords. They are parsed as introducing a precondition or postcondition specifier, respectively, only in the syntactic position where these appear. In all other contexts, they are parsed as identifiers. This minimizes the possibility that the introduction of `pre` and `post` could break existing C++ code.

Function contract specifiers appear at the end of a function declarator, immediately before the semicolon (or, if the declaration is a definition, immediately before the function body):

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

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

The only exception to this is the pure-specifier `= 0`, which appears *after* the function contract specifiers:

```
struct X
{
  virtual void f() pre(c) = 0;
};
```

For lambda expressions, function contract specifiers appear at the end of the declarator, immediately before the opening brace:

```
int f() {
  auto f = [] (int i)
    pre (i > 0)
```

```
    { return ++i) };

  return f(42);
}
```

There may be any number of function contract specifiers, in any order, specified for a function. Precondition specifiers do not have to precede postcondition specifiers, rather they may be freely intermingled:

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

3.2.2 Assertion expressions

An assertion expression is a kind of contract assertion that is an expression of type `void` and may appear anywhere such an expression may appear. An assertion expression is introduced with `contract_assert` followed by the predicate in parentheses:

```
void f()
{
  contract_assert(i != 0);  // assertion (as a statement)
}

class X
{
  int* _p;
public:
  X(int* p)
  : _p(contract_assert(p), p))  // assertion (as a subexpression)
  {}
};
```

Unlike `pre` and `post`, `contract_assert` is a full keyword. This is necessary in order to be able to disambiguate an assertion expression from a function call expression. The keyword `contract_assert` is chosen instead of `assert` to avoid a clash with the existing `assert` macro from header `<cassert>`.

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. Function contract specifiers may only be attached to function declarations that are first declarations.

It is ill-formed, no diagnostic required (IFNDR) if there are multiple first declarations for the same function that have different lists of contract assertions.

In effect, all places where a function might be used or defined must see an equivalent list of function contract specifiers attached to that function declaration.

TODO: When comparing contract assertions on different function declarations, the comparison is done by applying the one-definition rule (ODR) to an imaginary function body that contains the contract assertion predicate (with a similarly imaginary declaration for the return value in scope). Function parameters, template parameters, and the return value may all have different names — the ODR requires that the entities found by name lookup be the same.

3.3.2 Virtual Functions

If a virtual function overrides another, it may not have function contract specifiers attached directly to it. In other words, only the root of a virtual function hierarchy may have function contract specifiers.

A function that overrides a function with contract assertions will inherit those contract assertions. The contract assertions will be evaluated in the context of the base class function declaration.

It is ill-formed for a function to override multiple functions from different base classes if any of them have function contract specifiers.

TODO: There is ongoing discussion about how to handle virtual functions for the MVP or whether contract assertions on virtual functions should even be allowed.

3.3.3 Defaulted and deleted functions

It is ill-formed for 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);    // OK (pre on function defaulted on non−first declaration)
};

Y::Y() = default;
```

Further, it is ill-formed for 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 coroutine — i.e. a function containing a `co_yield`, `co_return`, or `co_await` statement — to have precondition or postcondition specifiers. It is, however, valid to use `contract_assert`

within the body of a coroutine. This restriction might be relaxed in a future post-MVP extension.

3.3.5 Function Pointers

A contract assertion 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 specifiers must still be evaluated as normal.

3.4 Semantic rules for contract assertions

3.4.1 Name lookup and access control

For precondition specifiers, 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. For functions, the declaration to which the contract assertion is attached is considered instead of the declaration that is part of the function's definition, i.e. name lookup in the predicate uses the parameter names that are visible to the contract assertion and not those visible to the function definition.

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 3.4.6). When the precondition specifier is part of a member function, protected and private data members of that function's type may be accessed. When a precondition is part of a function that is a friend of a type, full access to that type is allowed.

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

For assertion expressions, name lookup and access control occurs as if the predicate's expression were located at the place where the assertion expression is located.

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 predicates are `const` lvalues. This is conceptually similar to how identifiers referring to members are implicitly `const` lvalues in 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 Postcondition specifiers: referring to the return value

A postcondition specifier may optionally specify a *return-name* introducing a name that refers to the return 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 *return-name* is an 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 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 callsite 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 *return-name* refers to the address of the return 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 specifier 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 return object inside a postcondition specifier, something that was previously not possible:

```
X f(X* ptr)
  post(r: &r == ptr)   // guaranteed to pass 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 Postcondition specifiers: referring to parameters

If a function parameter is odr-used by a postcondition specifier's 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 because the parameter value might have been modified in the definition.

Consider, for example the following declaration of a `clamp` function:

```
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 the parameters be `const` avoids such extreme failures and subtle variations on this theme, allowing the caller to reason about the meaning of the postcondition without the need to inspect the implementation to validate that it does not modify the variables on which the postcondition depends.

3.4.5 Not part of the immediate context

The predicate of a function contract specifier, 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, we may have a function template that works well for a given type, but stops working the moment we add a contract assertion.

3.4.6 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 design principles 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 expressions 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 specifiers 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 specifiers 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 specifiers are evaluated in the order in which they are declared.

An assertion expression will be evaluated at the point where control flow reaches the expression.

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*. 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 determine if there has been a contract violation. 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 determines if there has been a contract violation. 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 normally, the program will be terminated by calling the function `std::abort()`. If there is no contract violation, program execution will continue from the point of evaluation of the contract assertion.

The *observe* semantic is a *checked* semantic. It determines if there has been a contract violation. 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 there is no contract violation, 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 provided by the C++ Standard, 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 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. Other additional flags are implicitly encouraged.

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 Detecting a violation

When a contract assertion is being evaluated with a checked semantic it must be determined if the predicate will not return `true`.

The contract assertion's predicate may be evaluated, and a number of possible results may be considered:

- If the predicate evaluates to `true`, there is no contract violation and execution will continue normally after the point of evaluation of the contract assertion.

- If the predicate evaluates to `false`, there is a contract violation and, if the evaluation happens at runtime, the contract-violation invocation process will begin with a `detection_mode` of `predicate_false`.
- If the evaluation of the predicate exits via an exception, there is a contract violation and the contract-violation invocation process will begin with a `detection_mode` of `evaluation_exception`.
- If the evaluation of the predicate results in a call to `longjmp` or program termination, that call happens as normal.
- If the evaluation happens during constant evaluation (at compile time), and the predicate is not a core constant expression (and can therefore not be evaluated at compile time), there is a contract violation (see Section 3.5.9 for more details).

If it can be determined that the predicate *would* evaluate to `true` or `false` then the predicate does not need to be evaluated. The compiler cannot introduce new side effects to make this determination, but it may freely do anything that it is permitted to do under the as-if rule — i.e., the compiler may generate a side-effect free expression that provably produces the same results as the predicate and evaluate that expression instead of the predicate, effectively evaluating the predicate itself zero times. In such cases, the side effects of the predicate will not occur, even for an enforced or observed contract assertion (see Section 3.5.6).

When this results in a determination that the predicate would return `false` at runtime, the contract-violation invocation process will begin with a `detection_mode` of `predicate_false`. However, if a predicate would not evaluate to `true` or `false`, but exit in some other fashion, such as via an exception, invocation of `longjmp`, or termination, then the evaluation of that predicate, up to and including the `longjmp` or termination, happens normally, including any observable side effects.

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
- invoking functions as long as none of the function parameters require a nonvacuous operation to initialize
- returning from a function, as long as the initialization of the return value does not require a nonvacuous operation to be performed.

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 specifiers on a single function when invoking that function
- Checking all postcondition specifiers on a single function when that function returns normally
- Checking consecutive assertion expressions
- Checking the precondition specifiers of a function and any assertion expressions that are at the beginning of the body of that function
- Checking the precondition specifiers of a function `f1` and the precondition specifiers 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 no vacuous operations
- Checking the postcondition specifiers of a function `f1` and the precondition specifiers 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 no 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.[1]

In practice, this means that the preconditions and postconditions of a function may be checked, as a group, any number of times.

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 are proven correct.

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.

As described in Section 3.5.4, contract predicates may be elided if it can be determined that they will evaluate to a value of `true` or `false`. 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, observable side effects of the evaluation of a contract assertion's predicate 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:

[1] 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.

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

In other cases, a result of `true` or `false` may not be the only results possible, in which case the compiler cannot check the contract assertion without evaluating the contract predicate. For example, if the contract predicate might throw an exception, that exception must be available to the contract-violation handler (via `std::current_exception`) and so the predicate must be evaluated at least once:

```
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.

The same rules apply during constant evaluation:

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

std::array<int, f(0)> a;  // a.size() may be 0, 1, 17, etc.
```

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. 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 or may not 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 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*.

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

A single evaluation of a contract assertion involves determining the semantic with which to evaluate the contract assertion and then executing that semantic. When the semantic is a *checked* semantic, i.e. *enforce* or *observe*, the result of the contract assertion's predicate must be determined. If this result is not `true`, and the evaluation happens at runtime, the contract-violation handling process will be invoked.

A `contract_violation` object 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:

```
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 this equivalence is that, unlike most previous proposals and macro-based Contracts facilities, the meaning of the contract assertion does not change based on whether contracts are enabled or disabled, or any other aspect of compile-time contract configuration — therefore, it is not a violation of the one-definition rule (ODR) to have the same contract assertion evaluated with different semantics at different times by a single program.

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` — no contract violation.
- The result is `false` — contract violation.
- The predicate is not a core constant expression — contract violation.

A contract assertion is always eligible to be a core constant expression, even if its predicate is *not* a core constant expression (the latter is a contract violation, but does not render the program

ill-formed unless the contract semantic is *enforce*). In order to satisfy the design principles 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.

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[2] 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 design principles 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 specifier:

```
constexpr int compute(int n)
  pre (n == 0 || !std::is_constant_evaluated())  // passes for both i and j
{
```

[2]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.

```
  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.

3.6 Special cases

This section describes the behavior that follows from the rules specified above in some particular noteworthy cases.

3.6.1 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).

3.6.2 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 specifiers, 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*.

TODO: It is currently undecided what the `noexcept` operator should return when directly applied to an assertion expression or an expression that contains an assertion expression as a subexpression. It is further undecided how contract assertions should behave for special member functions defaulted on their first declaration that have a deduced exception specification that must be deduced from a default member initializer or function argument that includes a contract assertion.

3.6.3 Undefined behavior

If the evaluation of a contract assertion evaluates a predicate having undefined behavior, the evaluation of the contract assertion itself has undefined behavior. In other words, there is no special protection against predicates with undefined behavior. Given that the behavior is undefined, if an implementation is capable of detecting this case, it is allowed to treat it as a contract violation; the `detection_mode` of `evaluation_undefined_behavior` is provided for this purpose. Note that in this case, the program is still considered to have undefined behavior; any specification of the behavior in this mode would be a vendor-specific extension outside of the scope of the C++ Standard.

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 post-MVP 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 detection_mode : int {
  predicate_false = 1,
  evaluation_exception = 2,
  evaluation_undefined_behavior = 3
  /* to be extended with implementation-defined values and by post-MVP 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 post-MVP extension
  /* to be extended with implementation-defined values and by post-MVP extensions */
  /* Implementation-defined values should have a minimum value of 1000. */
};

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

class contract_violation {
  // No user-accessible constructor
public:
  std::source_location location() const noexcept;
  const char* comment() const noexcept;
  contract_kind kind() const noexcept;
```

```
  contract_semantic semantic() const noexcept;
  detection_mode detection_mode() const noexcept;
  bool will_continue() 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 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`.
  - `evaluation_undefined_behavior` : To indicate that the evaluation of the predicate had undefined behavior. There is no standard requirement to ever invoke the contract-violation handler with this value, and such invocations are considered to be undefined behavior and out of scope for the Standard, but this allows a platform to clearly communicate when this does end up being the case.
  - Implementation-defined values indicate an alternate method provided by the implementation in which a contract violation was detected.
- `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 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 specifier.
  - `post`: A postcondition specifier.
  - `assert`: An assertion expression.
  - Implementation-defined values indicate other kinds of contract assertions that may be available as a vendor extension.

Note that the enumerators `pre` and `post` match the contextual keyword that introduces the respective contract assertion kind; however, assertions use `assert` for the enumerator but `contract_assert` for the keyword as the latter needs to be a full keyword and therefore cannot be used as an enumerator name.

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, 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:

- `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.

- `const char* comment() const noexcept`: The value returned should be a null-terminated multibyte string (NTMBS) in the string 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.

- `contract_kind kind() const noexcept`: The kind of the contract assertion which has been violated.
- `contract_semantic semantic() const noexcept`: The semantic with which the violated contract assertion was being evaluated.
- `detection_mode detection_mode() const noexcept`: The method by which a violation of the contract assertion was identified.
- `bool will_continue() const noexcept`: `true` if flow of control will continue into user-provided code should the contract-violation handler return normally, `false` otherwise. Generally, this should always be `false` for a contract assertion evaluated with the *enforce* semantic. For a contract assertion evaluated with the *observe* semantic this will generally be `true`, but it *may* be `false` should the platform identify that user-provided code will never execute along the branch where this contract-violation has been detected.

  For example, consider a contract assertion where there is always undefined behavior in the code which follows it, should a violation occur:

  ```
  void f(int* p)
  {
    contract_assert(p != nullptr);
    *p = 5;
  }
  ```

  Even when the `contract_assert` above is *observe*, a compiler might be able to identify that all control flow paths which would be reached after the contract assertion is evaluated have undefined behavior. The compiler can inform that contract violation handler of this impending undefined behavior with a `will_continue` value of `false`.

  This value is based on what a compiler can determine when compiling a particular contract assertion, and one platform might identify a case where continuation will not happen while another might not.

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.

Just like with the default contract-violation handler itself, it is implementation-defined whether `invoke_default_contract_violation_handler` is `noexcept`.

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. Given that violation of a precondition when using a Standard Library function is undefined behavior Standard Library implementations are 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

Proposed wording will be added after all TODO items above have been resolved in SG21.

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 produce a consensus-driven proposal from within the Contracts study group (SG21) that provides a foundation that can grow to meet the needs of the many constituents that have participated in achieving that consensus. As a side effect, this somewhat minimal product can be seen to be highly viable for many use cases, and will enable a better, safer C++ ecosystem in the future.

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 and Oliver Rosten for reviewing the paper and pointing out errors.

Bibliography

[CWG2841] Tom Honermann, "When do `const` objects start being `const`?"
https://cplusplus.github.io/CWG/issues/2841.html

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

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

[P2695R0] Timur Doumler and John Spicer, “A proposed plan for contracts in C++”, 2022
http://wg21.link/P2695R0

[P2896R0] Timur Doumler, “Outstanding design questions for the Contracts MVP”, 2023
http://wg21.link/P2896R0