Contracts for C++

Document #:	P2900R1
Date:	2023-10-09
Project:	Programming Language C++
Audience:	SG21 (Contracts)

Reply-to:	Joshua Berne <jberne4@bloomberg.net>
	Timur Doumler <papers@timur.audio>
	Andrzej Krzemieński <akrzemi1@gmail.com>

Abstract

After long 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-checking annotations that may be leveraged in their ecosystems in numerous ways.

Contents

Revision History

Revision 1 (October 2023 mailing)

- Added new sections 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 contract-checking 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 two primary sections. The first, Section 2, describes the design of the Contracts facility carefully, clearly, and precisely. The second, Section 3, 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 Contracts will do, and contain the needed wording to match that information.

Our intent is to make this the final product of this phase of the SG21 plan that can be further reviewed and vetted by all needed groups before Plenary successfully decides to amend the standard with this wording and finally deliver a long-desired feature to C++.

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 Design

Contracts provide the ability to add annotations to and within functions that will be used to identify when incorrect behavior has occurred. Based on how the program has been translated, this may or may not result in a runtime check. When defects are detected during such runtime checks a user-replaceable contract-violation handler will be invoked and, potentially, program execution will be terminated.

There are three kinds of contract checks that can be added to a function — preconditions, postconditions, and assertions:

> **TODO**: The syntax below for adding preconditions, postconditions, and assertions to a function is a strawman that is not expected to be the final syntax. This same strawman syntax will be reused throughout this paper.

```
int f(const int x)
  __PRE__[{ x != 1 }]
  __POST__[{ r : r != 2 }]
{
  __ASSERT__[{ x != 3 }];
  return x;
}
```

In the above example a contract-check will be violated if `f` is called with a value of `1`, `2`, or `3`:

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

The behavior of this violation of a contract check will be dependent on implementation-defined details of how the program is built.

2.1 Terminology

We will begin by providing the general terminology that will be used throughout this section (and hopefully, in general, many of the other papers discussing these topics).

A *contract-checking annotation*, or *CCA*, is a syntactic construct that contains the information needed to specify an algorithm to detect defects in C++ software.

Each CCA has a *kind* that is one of *precondition*, *postcondition*, or *assertion*. The kind largely dictates where the CCA may appear, and when exactly it will be evaluated.

Each CCA has a *point of evaluation* based on its kind and syntactic position. Preconditions evaluate immediately after function parameters are initialized. Postconditions evaluate immediately after

local variables in the function are destroyed when a function returns normally. Assertions evaluate at the point in the function where control flow reaches them.

Each CCA has a *predicate* which is a potentially evaluated expression that will be contextually converted to `bool` in order to identify violations of a contract. It is generally expected that, in a correct program, the predicate of each CCA would evaluate to `true` at the appropriate point of evaluation.

The evaluation of a CCA does not necessarily mean evaluating any expressions, such as its predicate. The exact meaning depends on the CCA semantic chosen, which is done in an 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.

When a CCA is evaluated with a semantic that requires that it be *checked*, 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.

There is a function, `::handle_contract_violation`, that will be invoked when a contract violation has been detected. 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.

Each evaluation of a CCA is done with a *semantic* that is chosen in an implementation-defined manner.

The *ignore* semantic does nothing. Note that even though the predicate of an ignored CCA 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 so, the contract-violation handler will be invoked. If the contract-violation handler returns normally, the program will be terminated in an implementation-defined manner. If there is no contract violation, program execution will continue from the point of evaluation of the CCA.

The *observe* semantic determines if there has been a contract violation. If so, 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 CCA.

Because both the *observe* and *enforce* semantics must identify if there has been a contract violation, these are called *checked* semantics.

Because the *ignore* semantic does *not* need to, nor is it even allowed to, detect a contract violation,

it is called an *unchecked* semantic.

2.2 CCAs

2.2.1 Preconditions

A precondition CCA is attached to a function declaration (see Section 2.2.6 for which declarations) and will be evaluated whenever the function is invoked, immediately following the initialization of function parameters.

Multiple preconditions are invoked in lexical order, immediately after the initialization of function arguments and prior to the function body. Note that a constructor's member initializer list and a function-try block are considered to be part of the function body.

Name lookup in the precondition's predicate is performed as if the predicate came immediately before the function's body, but with the declaration to which the CCA is attached instead of the declaration that is part of the function's definition, i.e. using the parameter names that are visible to the CCA and not those visible to the function definition. Access control is applied based on that behavior, i.e. the precondition's expression may reference anything that might be referenced from within the function body.

2.2.2 Postconditions

A postcondition CCA is attached to a function declaration (see Section 2.2.6 for which declarations) and will be evaluated whenever the function returns normally. This evaluation is after the return value has been initialized and local automatic variables have been destroyed, but prior to the destruction of function parameters.

A postcondition may give a name to the return value. As with a variable declared within a function body, this name cannot shadow function parameter names. In such cases, that name will designate a variable of type `const& R` where `R` is the return type of the function. This variable will be initialized prior to evaluating the predicate to designate the actual return value. Note that this variable is visible only in the predicate to which it applies, and does not introduce a new name into the scope of the function.

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
```

Multiple postconditions are evaluated in lexicographical order.

Name lookup in the postcondition first considers its return value name, if any, to be in a synthesized enclosing scope around the precondition. All other names are looked up as if the postcondition came immediately prior to the first statement in the function's body, but with the first declaration instead of the declaring declaration. Access control is applied just as in a precondition, with the same privileges to access protected and private data that a statement in the function's body would have.

If a function parameter is odr-used by a postcondition CCA'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):

```
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
```

2.2.3 Assertions

TODO: The possible position of an assertion CCA depends on the syntax we end up choosing. In attribute-like syntax [P2935R0], an assertion CCA is attached to the statement that contains it (and there can generally be only one assertion per such statement), and will be evaluated when that statement would be evaluated. In "natural" syntax [P2961R0], an assertion CCA is an expression, and will be evaluated according to the usual rules for evaluating expressions.

Name lookup and access control in an assertion CCA's predicate occurs as if the predicate's expression were located where the attached assertion statement or assertion expression is located.

2.2.4 Access to private variables and friend types

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

For preconditions and postconditions, access control for names that are found in the predicate is therefore enforced as-if the predicate were part of the function body.

2.2.5 Not part of the immediate context

Precondition and postcondition predicates, while they are lexically a part of a function declaration, are 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>>); // ill-formed: 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 annotation.

2.2.6 Multiple Declarations

Any function declaration is a *first declaration* if there are no other declarations of the same function reachable from that declaration. Preconditions and postconditions 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 CCAs.

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

TODO: When comparing CCAs on different function declarations, the comparison is done by applying the one-definition rule (ODR) to an imaginary function body that contains the CCA 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.

2.2.7 Virtual Functions

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

A function that overrides a function with CCAs will inherit those CCAs. The CCAs 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 preconditions or postconditions.[1]

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

[1] See [P2954R0].

2.2.8 Trivial Special Member Functions

TODO: It remains to be decided whether it should be well-formed to add a CCA to a trivial special member function, and whether the function will remain trivial as a result; see [P2932R0].

2.2.9 Lambdas

TODO: The ability to attach preconditions and postconditions to a lambda still remains to be added. While that ability in itself is uncontroversial, it must also be determined whether implicit captures should be allowed from within a CCA's predicate, which would violate the zero-overhead principle of [P2834R1]; see also [P2896R0].

2.2.10 Coroutines

TODO: It is currently undecided whether coroutines may have preconditions or postconditions and what their semantics should be; see [P2932R0] and [P2957R0].

2.2.11 Function Pointers

A CCA may not be attached to a function pointer. The CCAs on a function have no impact on its type, and thus no impact on what happens what a function with contracts is converted into.

When a function *is* invoked through a function pointer its precondition and postconditions must still be evaluated.

2.3 Syntax

TODO: The final major task in the SG21 plan is to decide on a syntax. Once that is complete, a description of the syntax will appear here.

TODO: The final syntax will make the following clear:

- Each precondition or postcondition will appertain to the first declaration of a function.
- There may be any number of preconditions or postconditions, in any order, specified for a function.
- Whether an assertion may be used as a statement within a function body or as an expression anywhere a C++ expression is syntactically allowed.

2.4 Evaluation and contract-violation handling

Each evaluation of a CCA is done with a semantic that is implementation-defined to be one of *ignore*, *enforce*, or *observe*. Chains of consecutive evaluations of CCAs may have individual CCAs

repeated any number of times (with certain restrictions and limitations — see Section 2.4.5).

2.4.1 Contract Semantics: *ignore*, *enforce*, *observe*

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 CCA 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 so, the contract-violation handler will be invoked. If the contract-violation handler returns normally, the program will be terminated in an implementation-defined manner. If there is no contract violation, program execution will continue from the point of evaluation of the CCA.

The *observe* semantic is a *checked* semantic. It determines if there has been a contract violation. If so, 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 CCA.

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.

2.4.2 Selection of Semantics

The semantic a CCA 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 CCAs can have different semantics, even in the same function. The same CCA may even have different semantics for different evaluations.

The semantic a CCA 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 CCA is checked or unchecked, or which concrete semantic it has. This is another important difference between CCAs and the `assert` macro.

It is expected that there will be various compiler flags to choose globally the semantics that will be assigned to CCAs, and that this flag does not need to be the same across all translation units. Whether the CCA semantic choice 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 CCAs to have, at translation time, the *enforce* or the *ignore* semantic. Other additional flags are implicitly encouraged.

When nothing else has been specified by a user, it is recommended that a CCA will have the *enforce* semantic. 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.

2.4.3 Detecting a violation

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

The CCA'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 CCA.
- If the predicate evaluates to `false`, there is a contract violation and the contract-violation invocation process will begin with a `detection_mode` of `predicate_false`.
- If the evaluation of the predicate results in an exception escaping that exception will be caught and, within the handler for that exception, 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 `std::longjmp` or program termination that process continues as normal.

If it can be determined that the predicate *would* evaluate to `true` or `false` then the predicate does not need to be evaluated. In other words, 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 CCA. When this results in a determination that the predicate would return `false` the contract-violation invocation process will begin with a `detection_mode` of `predicate_false`.

2.4.4 Undefined behavior

If evaluating the predicate of a checked CCA results in undefined behavior, the entire program has undefined behavior. 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 CCA has been checked, the CCA 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;  // UB
}

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, in this case because the program as a whole has undefined behavior. 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 CCAs evaluated with the *observe* semantic.

2.4.5 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. Two contract-checking annotations shall be considered consecutive when they are separated only by vacuous operations. 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

A *CCA sequence* is a sequence of CCAs where all operations between adjacent elements of the sequence are vacuous. These will naturally include:

- all precondition checks on a single function when invoking that function
- all postcondition checks on a single function when that function returns normally
- any assertion CCAs that are consecutive
- the preconditions of a function and assertion ccas that are at the beginning of the body of the function
- the preconditions of a function (`f1`) and the preconditions of the first function invoked by `f1` (`f2`), when preparing the arguments to the invoked function (`f2`) involves no non-trivial operations
- the postconditions of a function (`f1`) and the preconditions of the next function invoked immediately after `f1` returns (`f2`), when the destruction of the arguments of the first function

(`f1`) and the preparation of the arguments of the second function (`f2`) involve no non-trivial operations

Within a CCA sequence, after evaluation a CCA with a checked semantic additional evaluations of the same CCA may be inserted at any point within the sequence. These are independent evaluations and may thus have different semantics.

2.4.6 Side effects

The predicate of a CCA is allowed to be an expression that has observable side effects when evaluated, such as logging. However, due to the rules for violation detection (see 2.4.3) and for consecutive and repeated evaluations (see 2.4.5), it is unspecified whether these side effects will be observed zero, one, or multiple times when the CCA is checked. The usage of predicates with side effects is generally discouraged.

2.4.7 Compile-time Evaluation

TODO: The semantics of CCAs during constant evaluation remain to be determined; see [P2932R0] and [P2894R0].

2.4.8 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.

This function

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

The implementation-provided version of this function 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, 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 name and a signature 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 name and signature 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.

2.4.9 The Contract-Violation Handling Process

Upon detection of a contract violation, the contract-violation handling process will begin.

A `contract_violation` object will be produced in an unspecified manner — it may already exist in read-only memory, it may be populated at runtime on the stack.

The value of the `location` property of the `contract_violation` is unspecified. It is recommended that it be the source location of the caller of a function when a precondition is violated. For other CCA kinds, or when the location of the caller is not used, it is recommended that the source location of the CCA itself is used. Based on implementation and user choices it may be empty or have some other value.

The value of the `comment` property is unspecified. It is recommended that it contain a textual representation of the CCA's predicate. Based on implementation and user choices it may be empty, have a truncated or otherwise modified version of the CCA's predicate, or contain some other message intended to identify the CCA for the purpose of aiding in diagnosing the bug. The value of the comment should be a null-terminated multi-byte string (NTMBS) in the string literal encoding.

The value of the `detection_mode` property indicates the shape of the event that led to invocation of the contract-violation handler.

- A value of `predicate_false` means the predicate either was evaluated and produced a value of `false` or the predicate would have produced a value of `false` if evaluated.
- A value of `evaluation_exception` indicates that the predicate was evaluated and an exception escaped that evaluation.
- A value of `evaluation_undefined_behavior` indicates that the implementation has chosen to invoke the contract-violation handler within a program execution with undefined behavior. This is informative.
- Implementation-defined values indicate an alternate method in which a contract-violation handler was detected.

The value of the `semantic` property indicates the semantic with which the CCA was being evaluated when a contract-violation was detected.

The value of the `will_continue` property indicates whether control flow will resume after the point of evaluation of the CCA should the contract-violation handler return normally. In general this will have a value of `true` if the CCA was being evaluated with a semantic of `observe`, and `false` if the semantic was *enforce*. For other implementation-defined semantics this value must still be determined.

2.4.10 Evaluation summary

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

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 CCA. This may be a compile-time value or, based on what the platform provides, even a runtime evaluation.
- `__check_predicate(`X `)`: Determine the result of the predicate X — 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 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.
- `__terminate_on_enforced_violation()`: The implementation-defined mechanism via which program execution is terminated when an enforced contract is violated.

Building from these intrinsics, the evaluation of a CCA 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

    // 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) {
        __terminate_on_enforced_violation();
    }
}
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 CCA's predicate must be done within the `catch` block for that exception. The invocation when *no* exception is thrown must be done *outside* the `try` block that would catch that exception. There are many ways 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 contract-checking facilities, the meaning of the CCA 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 CCA evaluated with different semantics at different times by a single program.

2.4.11 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 CCA evaluation with the *enforce* semantic. For preconditions and postconditions, the contract-violation handler is treated as if the exception had been thrown inside the function body. Therefore, when a precondition or postcondition is violated, a user-defined contract-violation

handler ends up throwing an exception, and the function that the precondition or postcondition applies to is `noexcept`, `std::terminate` will be called, regardless of whether the semantic is *enforce* or *observe*.

2.5 Standard Library API

2.5.1 The <contracts> Header

A new header `<contracts>` is added to the C++ Standard Library. The facilities provided in this header have a very specific intended usage audience — those writing user-defined contract-violation handlers and, in future Standards, 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 CCAs.

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

```
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 future Standards */
  /* 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 in a future Standard
  /* to be extended with implementation-defined values and by future Standards */
  /* 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 future Standards */
  /* 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;
};

}
```

2.5.2 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.
  - `predicate_false` : To indicate that the predicate was evaluated, or would have been evaluated, and the resulting value was `false`.
  - `evaluation_exception` : To indicate that, during the evaluation of a CCA predicate, an exception was not handled. This may be extended to other kinds of CCAs in the future that might not actually be predicates (but might instead evaluate a block statement of some sort).
  - `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.
- `enum class contract_semantic : int`: A reification of the semantic that can be chosen for the evaluation of a CCA when that CCA 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 CCAs do not invoke the contract-violation handler.
- `enum class contract_kind : int`: Identifies one of the three potential kinds of CCA, with implementation-defined alternatives a possibility for when something invokes the contract-violation handler outside the purview of a CCA with one of those kinds.

TODO: The enumerator names should be consistent with the names used by the chosen syntax (even though it might not be possible for them to be exactly the same if the chosen syntax claims any of the below names as full keywords).

- `pre`: A precondition CCA.
- `post`: A postcondition CCA.
- `assert`: An assertion CCA.

2.5.3 The class `std::contracts::contract_violation`

The `contract_violation` object is provided to the `handle_contract_violation` function when a contract violation has occurred. 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 that is populated is a recommended practice only — it should be conforming to (generally based on how the compiler was invoked) provide an empty object as a return value. If the CCA is a precondition, the location of the call site if possible is ideal; otherwise the location of the CCA itself should be provided.
- `const char* comment() const noexcept`: It is recommended that this value contain the string representation of the CCA's predicate. Pretty-printing, truncating, or providing the empty string are all conforming implementations.
- `contract_kind kind() const noexcept`: The kind of the CCA which has been violated.
- `contract_semantic semantic() const noexcept`: The semantic with which the violated CCA was being evaluated.
- `detection_mode detection_mode() const noexcept`: The method by which a violation of the CCA 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 CCA evaluated with the *enforce* semantic. For a CCA 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.

2.5.4 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 CCAs 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 CCA 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.

3 Proposed wording

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

4 Conclusion

The idea of having a contract-checking 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 WG21 study group 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 Tom Honermann and Ville Voutilainen for repeated requests to see this paper come to light. Thanks to Jens Maurer for feedback on this paper.

Bibliography

[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

[P2834R1] Joshua Berne and John Lakos, "Semantic Stability Across Contract-Checking Build Modes", 2023
http://wg21.link/P2834R1

[P2894R0] Timur Doumler, "Constant evaluation of Contracts", 2023
http://wg21.link/P2894R0

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

[P2932R0] Joshua Berne, "A Principled Approach to Open Design Questions for Contracts", 2023
http://wg21.link/P2932R0

[P2935R0] Joshua Berne, "An Attribute-Like Syntax for Contracts", 2023
http://wg21.link/P2935R0

[P2954R0] Ville Voutilainen, "Contracts and virtual functions for the Contracts MVP", 2023
http://wg21.link/P2954R0

[P2957R0] Andrzej Krzemieński and Iain Sandoe, "Contracts and coroutines", 2023
http://wg21.link/P2957R0

[P2961R0] Jens Maurer and Timur Doumler, "A natural syntax for Contracts", 2023
http://wg21.link/P2961R0