Integrating Existing Assertions with Contracts

Document #:	P3290R2
Date:	2024-09-06
Project:	Programming Language C++
Audience:	SG21 (Contracts), EWG, LEWG

Reply-to:	Joshua Berne <jberne4@bloomberg.net>
	Timur Doumler <papers@timur.audio>
	John Lakos <jlakos@bloomberg.net>

Abstract

SG21 has been actively working on an MVP for a C++ Contracts facility, P2900. This novel facility aims to provide powerful contract assertions in three forms: `pre`, `post`, and `contract_assert`. Taken as a unit, these three standard names allow users to express, in a consistent syntax, individual, independent contract checks in their software without changing the meaning (i.e., essential behavior) of that software. Pre-existing contract-checking facilities, such as `assert`, occasionally have fundamentally different semantics and provide completely different control mechanisms for their behavior. Integrating these facilities with the contract-violation handler, however, can be an incredibly useful tool for enabling a gradual migration from older facilities to the newer tools provided by the language itself. With SG21 consensus, this paper proposes both a library API to perform that integration and changes to `assert` to enable its interoperability.

Contents

Revision History

- Revision 2, post September 5, 2024 SG21 telecon
  - Minor bug fixes and clarifications
  - Usage examples for library API
- Revision 1, post–St. Louis meeting, July 2024
  - Added `source_location` overloads to library API
  - Added control macro for change in behavior of `assert`
  - Removed proposal for `partial_contract_assert`
- Revision 0, for SG21 Telecon, May 2024
  - Original version of the paper for discussion during an SG21 telecon

1 Introduction

Over the past five years, SG21 has been diligently developing an MVP for a C++ Contract facility ([P2900R7]). This facility is exceedingly useful for *incrementally* enhancing safety, security, and correctness in both new and legacy code. Through its previous incarnations and especially within SG21, capturing the ability to express and take advantage of discrete contract checks in the language has been a priority, whereas replicating the preprocessor-derived semantics of the standard `assert` macro and similar bespoke macro-based facilities has not.

One of the central features of the Contracts MVP, however, has nothing to do with the semantics of predicate evaluation and simply addresses what happens when a violation occurs, i.e., the contract-violation handling process. For the *observe* and *enforce* semantics, this process involves invoking the contract-violation handler. For the *enforce* and *quick_enforce* semantics, the program is terminated in an implementation-defined manner. For all three checked semantics, this violation-handling behavior is quite useful to existing contract-checking facilities as well.

Rather than needlessly delay any integration with the Contracts facility until all use cases and needed semantics are included, this paper proposes two, independent additions that allow for effective integration of alternate tools with Contracts.

1. **Directly calling the contract-violation handler:** Provide a mechanism to integrate existing contract-checking facilities with the central contract-violation handling mechanism of [P2900R7] while retaining precisely the same (possibly incompatible) semantics.
2. **Conditionally integrating `assert` with Contracts:** Augment the specification for the standard `assert` macro to conditionally support invoking the (nonthrowing) C++ contract-violation handler (instead of outputting a message to the standard error stream) before aborting; this semantic would be similar to the *enforce* semantic.

Each of these individual proposals adds distinct value and is aimed at providing immediate support for easily allowing existing code to use the new C++ Contracts MVP upon release. With the adoption of one or both of these independent proposals, we hope to significantly advance the

Contracts MVP since the vast majority of concerns regarding lack of support for a drop-in `assert` replacement will have been addressed.

One additional proposal was considered in previous iterations of this paper: to introduce a new form of contract-assertion statement that more closely matched the semantics of legacy contract-assertion facilities. This proposal was discussed extensively but is not currently being pursued for three reasons.[1]

1. While a new type of statement might be able to make different choices than `contract_assert` (such as not supporting elision, repetition, or `const`-ification and not allowing exceptions to escape), any such choice would provide the right semantics for some legacy facilities but not for others. Thus, any choice would be incomplete and would disenfranchise some user subset of indeterminate size.
2. The introduction of a new type of contract assertion would make the Contracts facility produce an $N + 2$ problem in terms of the number of choices for expressing assertions in code. This ambiguity would lead to unclear messaging and a facility that is harder to teach.
3. Having part of the Contracts facility simply remove features that we know increase safety would imply that `contract_assert` itself is inadequate and ill suited for its purpose. In truth, `assert` is not the right tool for expressing a small part of a larger contract check, and we do need to provide such tools in the future, but such tools must take a different form to maximize their expressivity and safe usage.

2 Proposals

In this section, we provide two independent, mutually compatible proposals that provide support for immediately allowing existing contract-checking facilities to integrate with C++ Contracts in a variety of ways. Our goal in each case is to provide exactly what users of legacy assertion facilities need.

All names are merely initial suggestions, with proposed reasoning and are highly likely to be changed during the standardization process.

2.1 Support Directly Invoking the Contract-Violation Handling Process

One of the primary purposes of adopting a Contracts facility into the Standard in lieu of continuing to use bespoke solutions is to centralize the management, response, and mitigation approach to

[1] Again when discussing earlier revisions of this proposal, SG21 polled whether we should pursue a new kind of contract-assertion statement for this purpose, and no consensus was reached to do so.

> We want to spend more time considering [P3290R0] Proposal 3.
>
SF	F	N	A	SA
0	2	7	6	1
>
> Result: Not Consensus

Another poll was taken as to whether such a feature should be pursued after the Contracts MVP, and that poll, while having more favorable results, was still far from consensus.

detected bugs in large-scale software. By having a central and user-selectable contract-violation handler, those who assemble large programs can avoid having distinct libraries producing different bug responses that do not fit into a single and consistent diagnostic and mitigation strategy.

All uses of `pre`, `post`, and `contract_assert` will, when a contract violation is detected by evaluation with the *enforce* or *observe* semantic, invoke the same contract-violation handler regardless of where in the program the violated contract assertion might be. This centralized reporting facility is one of the core benefits of having a Contracts facility in the language itself. With [P2900R7], users of legacy contract-checking facilities do not (yet) have a mechanism to integrate with that same reporting mechanism.

We propose to address the current inability of the Contracts MVP to integrate with legacy assertion mechanisms by providing a library API to replicate the behaviors of the various contract-evaluation semantics when a contract violation has been detected. These utility functions then provide a direct mechanism to invoke the contract-violation handler as well as to terminate execution in a fashion that matches the termination behavior of a contract-assertion evaluation having the *enforce* or *quick_enforce* semantic. Several design considerations have been identified during the process of developing what we propose.

- Each distinct checked semantic has, associated with it, different behaviors related to how violations are handled. A contract assertion evaluated with the *observe* semantic will continue execution when the contract-violation handler returns; evaluations with the *enforce* semantic will instead terminate the program in an implementation-defined fashion; and evaluations with the *quick_enforce* semantic do not call the contract-violation handler and have a potentially distinct mechanism for terminating the program but might also record data about the violation in debug information that is not accessible at run time. To that end, we propose that the name of each semantic be embedded in the function name so that these properties can be annotated on the function when possible.
- A mechanism need not trigger the handling of a contract violation the way an evaluation with the *ignore* semantic would since that semantic never identifies contract violations and has no runtime behavior to emulate.
- A more targeted function that simply took all the properties of a `contract_violation` object, populated one, and invoked the contract-violation handler but did nothing else would have a more fundamental problem: The contract-violation handler would be unable to depend on any promises inherent in the values provided, such as a guarantee that the program will terminate if the violation handler returns when the evaluation semantic on the `contract_violation` object is `enforce`.
- Perhaps a feasible approach would be to pass to a more general function a semantic as a value of type `std::contracts::evaluation_semantic`, but that approach would bring along the need to answer the complex question of how it should behave when given unknown or implementation-defined values for the semantic. For the same reason, we carefully crafted

`std::contracts::contract_violation` so that it cannot be created by users, which would allow users to pass an arbitrary such object to `::handle_contract_violation`.

- Another suggestion that has been made is to introduce a function template with one nontype template parameter that is a value of type `std::contracts::evaluation_semantic`. This option seems to increase complexity while bringing only the slight benefit of not needing new names when new semantics are introduced. Because the properties of each semantic-specific overload set can be fundamentally different, this approach of a single function template seems a likely source of confusion. New standard semantics are expected to happen only infrequently, so the cost of one new library function in a rarely used namespace seems like a small concern.

- A different function we might propose would use the same semantic as is configured for other contract assertions, but that value is not a well-defined one. While compiling a translation unit such that all contract assertions within it will be evaluated with the same semantic is possible, the expectation that all code will be compiled in such a way is mistaken, and we do not want to build features that would work correctly only when programs are built that way. The flexibility of that choice of semantic is, in large part, a source of problems when migrating from older facilities directly to contract assertions. Such a utility function would also be confusing when integrated with an existing assertion facility because it would result in two layers of configuration — i.e., the existing macro-based controls and the controls that impact all other contract assertions — determining the resulting semantic of older macros. Rather than provide yet another point where implementation-defined controls can alter program behaviors, we are focusing instead on providing a more concrete building block to use as a foundation for existing facilities.

- Two recommended practices for contract-semantic configuration are put forth in [P2900R7]. Providing a function that ties into builds where these recommended practices are in play, however, might be possible.

  - Those recommended practices are just a minimum for what we expect, and any richer or nonglobal configuration of Contracts does not fit into that model.

  - If the global configuration is to *ignore* all contract assertions, then by the time an assertion from an existing facility has decided to invoke the handler, that assertion's evaluation has also already decided to forgo *ignoring* the assertion since the predicate in question has already been evaluated.

  - The only other recommended practice is to *enforce*, and for that we are providing explicit functions to execute the behavior of the *enforce* semantic.

- Converting a predicate expression to a comment field in the `contract_violation` object can be easily accomplished using the stringizing operator `#`, and often an expression is not even apt for capturing the form of violation that is being manually detected; hence, we propose that the comment be provided via a `const char*` function parameter.

- We could consider limiting the use of these functions to compile-time strings using the same functionality that is leveraged by `std::format`. While this option might lead to improved implementation behavior, it might also limit usability with some legacy assertion facilities that produce an error message with more detailed information. Nothing precludes an implementation

from providing different overloads that behave better when offered a compile-time string instead of a runtime one.

- To produce a `contract_violation` object, a `std::source_location` must be populated, which can occur in two distinct ways.

  1. When not provided, source location can be detected in an implementation-defined manner based on where the invocation of the handling function is located. This detection can be accomplished with a defaulted parameter of type `std::source_location`, compiler magic of some sort using a lookup table, or some form of stack-trace inspection when the function is invoked at run time. This approach also leaves complete leeway, in some builds, to discard such information where it is deemed private.

  2. Overloads to all functions are available and take an additional `const std::source_location&` parameter. This parameter will be recommended for use in the `contract_violation` object that will be passed to the contract-violation handler.[2]

- The enumerated `kind` and `detection_mode` values could be passed as arguments to our new functions, but doing so would greatly increase the number of overloads we would need to provide for each checked semantic that might invoke the handler and, therefore, increases the complexity of using this otherwise fairly straightforward facility. Hence, we instead suggest, for these enumerations, new values that simply capture that a manually detected contract violation was encountered.

Naming is generally hard, and gaining consensus on naming is even harder. Instead of presenting the names as final, we offer them to be as clear as possible for their intended use, which is to manually trigger the violation-handling behavior of the various contract-evaluation semantics.

Putting all these considerations together, we suggest the following initial minimal proposal for an API.

[2] SG21, when discussing this proposal, expressed a desire to provide a source location from the caller since the use of this API might be embedded in a legacy contract-checking facility's own violation handler and thus might be far from the location of the assertion itself.

Amend [P3290R0] Proposals 1.1–1.3 to support an optional additional function argument that is a `std::source_location`.

SF	F	N	A	SA
2	11	5	0	0

Result: Consensus

Proposal 1.1: Triggering *Enforce* and *Observe* Semantics

Add the following to the header `<contracts>`:

```
namespace std::contracts {
  [[noreturn]] void handle_enforced_contract_violation(
      const char* comment);
  [[noreturn]] void handle_enforced_contract_violation(
      const char* comment
      const std::source_location &location);
  void handle_observed_contract_violation(
      const char* comment);
  void handle_observed_contract_violation(
      const char* comment
      const std::source_location &location);
}
```

Each of these functions will perform similarly but has unique behavior.

- Create and populate an object of type `std::contract_violation`.
  - The `comment` property will be the value provided as a function argument.
  - The `location` property is recommended to be the `location` parameter if one is specified. Otherwise, the recommended value is the location where the `handle` function was invoked. As usual, these are recommended practices, and a conforming implementation might strip out some or all information that would be used to populate the `location` property of the `contract_violation` object.
  - The `kind` property will be a new value, `manual`, of the `std::contracts::assertion_kind` enumeration.
  - The `detection_mode` property will be a newly added value, `manual`, of the `std::contracts::detection_mode` enumeration.
  - The `evaluation_semantic` property will be the semantic value that matches the particular function being invoked.
- The installed contract-violation handler will be invoked with this generated `contract_violation` object.
- If the contract-violation handler returns normally within `handle_enforced_contract_violation`, the program will be terminated in an implementation-defined manner.
- If the contract-violation handler returns normally within `handle_observed_contract_violation`, this function returns normally.
- If an exception escapes from the contract-violation handler, it propagates normally.

The above proposal covers all semantics that invoke the contract-violation handler, which is the primary purpose of this proposal.

The most recently added semantic, however, does have some functionality that is not easily reproduced elsewhere. As a second proposal, we propose a third overload set that has the semantics of handling a contract violation for a contract-assertion evaluation having the *quick_enforce* semantic, introduced in [P3191R0].

Proposal 1.2: Triggering *Quick_Enforce* Violations

Add the following to the header `<contracts>`:

```
[[noreturn]] void handle_quick_enforced_contract_violation(
    const char* comment) noexcept;
[[noreturn]] void handle_quick_enforced_contract_violation(
    const char* comment,
    const std::source_location& location) noexcept;
```

- Terminate the program in an implementation-defined manner.

In addition to the specified runtime behavior, just as with a contract evaluated with the *quick_enforce* semantic, we can gain non-normative benefits from invoking the above function. If `comment` is a compile-time string, it may be embedded in debug information in a manner outside the purview of the abstract machine and the Standard itself but still provide useful information when applying some forms of diagnostic tools.

As a separate proposal on top of the above, we also suggest having `noexcept` overloads of the functions that might invoke the contract-violation handler.

This behavior can be achieved in (at least) two other ways that come with associated drawbacks.

1. Invocations of the `handle` functions can be placed inside `try`/`catch` blocks that then manually invoke `std::terminate()`:

   ```
   try {
     handle_enforced_contract_violation(comment);
   } catch (...) {
     std::terminate();
   }
   ```

   This approach achieves the goal of not allowing an exception to escape but at the cost of potentially significantly greater code-size overhead compared to a `noexcept` function that needs only to mark a stack frame as being a `noexcept` boundary. When a codebase has enough assertions, this overhead has shown to be a concern for some developers.

2. The `handle` function can be wrapped in a user-provided `noexcept` function:

   ```
   [[noreturn]] void my_handle_enforced_contract_violation(
       const char* comment) noexcept
   {
     std::contracts::handle_enforced_contract_violation(comment);
   }
   ```

   This approach will potentially have improved code generation but at the cost of the call location of the `handle` function always being within the same wrapper function, thereby losing valuable information that was intended to be conveyed to the contract-violation handler.

Therefore, we propose adding overloads to the proposed API that take an additional argument of type `std::nothrow_t`, just as is done for nonthrowing operator `new`.

Proposal 1.3: `noexcept` Overloads

Add the following to the header `<contracts>`:

```
namespace std::contracts {
  [[noreturn]] void handle_enforced_contract_violation(
      const char* comment,
      const std::nothrow_t&) noexcept;
  [[noreturn]] void handle_enforced_contract_violation(
      const char* comment,
      const std::source_location& location,
      const std::nothrow_t&) noexcept;
  void handle_observe_contract_violation(
      const char* comment,
      const std::nothrow_t&) noexcept;
  void handle_observe_contract_violation(
      const char* comment,
      const std::source_location& location,
      const std::nothrow_t&) noexcept;
}
```

If an exception escapes the invocation of the contract-violation handler made by these functions, `std::terminate` will be invoked. Otherwise, these functions behave identically to the corresponding overloads without the `std::nothrow_t` parameter.

No overload that takes a `std::nothrow_t` is necessary for `handle_quick_enforced_contract_violation` since, in this case, no invocation of a contract-violation handler could exit via an exception (and the function itself is already marked `noexcept`).

Should a new checking semantic be added to the Standard in the future, we would need to add, in a similar fashion, corresponding functions to manually trigger that semantic's behavior upon detecting a contract violation. Given that any new semantic potentially has a distinct interface, each is equally likely to result in a new function or set of functions to parallel those we propose here.

Using this API within an assertion macro provided by an existing contract-checking facility is fairly straightforward. Consider a simple facility that should be enforced when `ENFORCE_MY_ASSERTIONS` is defined and otherwise compiled out:

```
#ifndef ENFORCE_MY_ASSERTIONS
  #define MY_ASSERT(X) ((void)sizeof((~X))?true:false)
#elif __cpp_lib_contracts < 20260101
  #define MY_ASSERT(X)                                                    \
    if (!(X)) {                                                           \
      ::MyLib::invokeViolationHandler(#X,__FILE__,__LINE__);              \
    }
#else
  #define MY_ASSERT(X)                                                    \
    if (!(X)) {                                                           \
      std::contractshandle_enforced_contract_violation(#X,std::nothrow); \
    }
#endif
```

In another case, a facility might already be using `std::source_location` and determining runtime behavior based on runtime queries to free functions in a `Configuration` namespace, so a macro such as `MY_ASSERT` above might invoke the following function when a violation is detected: comments; they look like en dashes. Consider changing the comment font or at least the style (maybe try nonitalic).

```
namespace MyLib {

void invokeViolationHandler(
    const char *comment,
    std::source_location location = std::source_location::current()) noexcept
{
#if __cpp_lib_contracts < 20260101
  if (MyLib::Configuration::abortFastOnViolations()) {
    // Immediately abort, do not continue, and do not attempt to log.
    std::abort();
  } else if (MyLib::Configuration::enforceViolations()) {
    // Log a custom message and terminate.
    MyLib::logContractViolation(comment,location);
    std::abort();
  } else {
    // When neither above configuration is selected, we log and continue.
    MyLib::logContractViolation(comment,location);
  }
#else
  if (MyLib::Configuration::abortFastOnViolations()) {
    // Go directly to termination.
    std::contracts::handle_quick_enforced_contract_violation(comment,location);
  } else if (MyLib::Configuration::enforceViolations()) {
    // Hook into the user-defined contract-violation handler, and then terminate.
    std::contracts::handle_enforced_contract_violation(comment,location,std::nothrow);
  } else {
    // When neither above configuration is selected, we log and continue.
    std::contracts::handle_observed_contract_violation(comment,location,std::nothrow);
#endif
  }
}
```

Note that in the above code, we would be ill served by not being able to pass in a `std::source_location` since it would end up always being the location within the function `MyLib::invokeViolationHandler`, not the calling code where the assertion macro is used. Instead, through the magic of `location` being a function parameter with a `consteval` default value, the source location that will be used when the macro expands to `invokeViolationHandler(#X)` will be that of the call site where the macro is expanded.

2.2 Conditionally Integrate C `assert` with C++ Contracts

Direct use of the standard `assert` macro is commonly taught and used widely in industry for a variety of purposes. In our experience, the overwhelming majority of such uses of `assert` involve

no side effects whatsoever. The remaining side effects are often just temporary print statements or inadvertent errors, yet some practicable, valuable uses remain.

Requiring an organization to pore over all their legacy uses of `assert` to ensure that no destructive side effects occur before benefiting from a central contract-violation handler provided by [P2900R7] seems time-consuming and counterproductive.

Even given the ability for user-defined macro-based facilities to integrate with the contract-violation handler, as proposed in the previous section, direct users of the standard `assert` macro still have no similar mechanism, and requiring each organization to write their own assertion facility and then rename each `assert` to that new macro seems needlessly user hostile.

We recommend, as a simple change with vast potential benefit, an addendum to the C++ specification for the `assert` macro, allowing it to invoke the C++ contract-violation handler instead of merely outputting a diagnostic message to the standard error stream. By default, behavior would not change, and users would have to explicitly opt in, thereby making this a fully backward-compatible, *conditionally supported* extension. Note that the behavior would be almost equivalent to the two-argument overload of `enforce_contract_violation` (see section 2.1), with the change that `kind` will be a new enumeration value, `cassert`, and the `detection_mode` will be the value `predicate_false`.

We recommend this additional latitude for the standard `assert` macro to invoke the C++ violation handler be an allowance, not a requirement, due to the nature of `assert` being a facility shared between C and C++. Some platforms may find making any changes to the behavior of the `assert` facility to be difficult or ill advised. Just as the basic control of the behavior of `assert` is done through defining or not defining `NDEBUG`, we propose a similar macro that chooses whether `assert` integrates with the violation handler.[3]

The name of this control macro is an open question, and many possibilities can be considered.

- In another, similar proposal, [P3311R0], the name `ASSERT_USES_CONTRACTS` was proposed.
- Whatever choice we make, this new macro will sit alongside `NDEBUG` as one of two macros that control the behavior of the `assert` macro. Neither the authors of this paper nor any of the people with whom we have discussed this new macro have been able to come up with an alternative as obtuse as `NDEBUG`, so achieving consistency with `NDEBUG` is a nongoal. An attempt at something similarly obtuse was proposed in the form of `NASSERT_DOES_NOT_USE_CONTRACTS`, which is clear due to the large number of words it makes use of, confusing due to the double negative, and not a spelling that seems overly compelling.

[3] SG21, when discussing this proposal, expressed a preference for the use of a control macro along the same lines as `NDEBUG` instead of making this choice an implementation-defined behavior.

Amend [P3290R0] Proposal 2 to have a control macro to opt into the assert macro calling the contract-violation handler, rather than making this choice implementation-defined.

SF	F	N	A	SA
1	8	4	3	0

Result: Consensus

- We could choose a macro name that refers to `assert` as either `ASSERT` or `CASSERT`. Given that we hope WG14, the C Standard committee, also pursues adopting a compatible contract-checking facility and integrates it with the same contract-violation handler, we suggest using the common term `ASSERT` instead of the C++-specific spelling of `CASSERT`.
- The macro name could indicate that it is a C++ specifier by including `CPP`, but that notation would also suffer from a small chance of being adopted by WG14 in the future.
- The fundamental function of this macro is to change `assert` so that it uses the contract-violation handling mechanism of an *enforced* contract assertion. Therefore, we could consider `ASSERT_IS_ENFORCED` as a name.
- Alternatively, since we are integrating `assert` with the contract-violation handlers, the name could focus on that action and thus be something like `ASSERT_USES_CONTRACT_VIOLATION_HANDLER`.

For simplicity, we will move forward with the name proposed in [P3311R0], `ASSERT_USES_CONTRACTS`.

Note that, just like the `contract_violation` objects populated when a `pre`, `post`, or `contract_assert` detects a violation, some fields in the `contract_violation` object populated by the `assert` macro have recommended values that might, in practice, also be empty, truncated, or populated differently based on how the compiler is configured.

Proposal 2: Integration of `assert` with the Contract-Violation Handler

When `NDEBUG` is not defined and `ASSERT_USES_CONTRACTS` is defined as a macro name at the point in the source file where `<cassert>` is included, the `assert` macro will put into the program a diagnostic test that has the following effects when its evaluation yields `false`.

- The contract-violation handler will be invoked with an object of type `std::contract_violation` having the following properties.
  - `comment` has a recommended value of `#__VA_ARGS__`.
  - `location` has a recommended value of the location where the `assert` macro was expanded.
  - `kind` will be a new value of `std::contracts_assertion_kind`, `cassert`.
  - `detection_mode` will be the value `predicate_false`.
  - `evaluation_semantic` will be `enforce`.

If the contract-violation handler returns normally or an exception escapes the contract-violation handler's evaluation, the program will terminate in an implementation-defined manner.

Note that the above description is roughly equivalent to an invocation of `handle_enforced_contract_violation(#__VA_ARGS__,std::nothrow)` with the caveat that different values for `kind` and `detection_mode` are passed through to the contract-violation handler as well.

A possible implementation of `<cassert>` might be something like this:

```
#ifdef NDEBUG
  #define assert(...) (static_cast<void>(0))
#elif defined ASSERT_USES_CONTRACTS
```

```
[[noreturn]] void __handle_assert_violation(const char* comment) noexcept
  // ABI function that invokes the contract-violation handler with an appropriately
  // populated contract_violation object, using the source_location where this
  // function is invoked

  #define assert(...)                                                    \
    ((expr)                                                              \
     ? static_cast<void>(0)                                              \
     : __handle_assert_violation(#__VA_ARGS__)
#else
[[noreturn]] void __assert_fail(const char* comment,
                                const char* file,
                                unsigned int line,
                                const char* function);
  // ABI function to print "Assertion failed" message to standard output and abort

  #define assert(...)                                                    \
    ((expr)                                                              \
     ? static_cast<void>(0)                                              \
     : __assert_fail( #__VA_ARGS__,                                      \
                      __FILE__,                                          \
                      __LINE__,                                          \
                      __PRETTY_FUNCTION__ )
#endif
```

3 Wording Changes

Wording changes are relative to [P2900R7] and [N4981].

Modify [support.contracts] paragraph 2:

```
...
  namespace std::contracts {

    enum class assertion_kind : unspecified {
      pre = 1,
      post = 2,
      assert = 3,
      manual = 4
      cassert = 5
    };

    enum class evaluation_semantic : unspecified {
      enforce = 1,
      observe = 2
    };

    enum class detection_mode : unspecified {
      predicate_false = 1,
      evaluation_exception = 2,
```

```
    manual = 3
  };

...
  void invoke_default_contract_violation_handler(const contract_violation&);

  [[noreturn]] void handle_enforced_contract_violation(
      const char* comment);
  [[noreturn]] void handle_enforced_contract_violation(
      const char* comment
      const std::source_location &location);
  [[noreturn]] void handle_enforced_contract_violation(
      const char* comment,
      const std::nothrow_t&) noexcept;
  [[noreturn]] void handle_enforced_contract_violation(
      const char* comment,
      const std::source_location& location,
      const std::nothrow_t&) noexcept;

  void handle_observed_contract_violation(
      const char* comment);
  void handle_observed_contract_violation(
      const char* comment
      const std::source_location &location);
  void handle_observe_contract_violation(
      const char* comment,
      const std::nothrow_t&) noexcept;
  void handle_observe_contract_violation(
      const char* comment,
      const std::source_location& location,
      const std::nothrow_t&) noexcept;

  [[noreturn]] void handle_quick_enforced_contract_violation(
      const char* comment) noexcept;
  [[noreturn]] void handle_quick_enforced_contract_violation(
      const char* comment,
      const std::source_location& location) noexcept;

}

...
```

handle_enforced_contract_violation **[support.contracts.handle.enforced]**

```
  [[noreturn]] void handle_enforced_contract_violation(
      const char* comment);
  [[noreturn]] void handle_enforced_contract_violation(
      const char* comment
      const std::source_location &location);
  [[noreturn]] void handle_enforced_contract_violation(
      const char* comment,
      const std::nothrow_t&) noexcept;
```

```
[[noreturn]] void handle_enforced_contract_violation(
    const char* comment,
    const std::source_location& location,
    const std::nothrow_t&) noexcept;
```

Effects:

— Invoke the contract-violation handler with a `contract_violation` object populated as follows:

— The `comment` will be the `comment` passed to this function.

— The `location`, if populated, will be the `location` passed to this function or the location of the call sight.

— The `kind` will be `manual`.

— The `detection_mode` will be `manual`.

— The `evaluation_semantic` will be `enforce`.

— If the violation handler returns normally, terminate the program in an implementation-defined manner.

`handle_observed_contract_violation` **[support.contracts.handle.observed]**

```
void handle_observed_contract_violation(
    const char* comment);
void handle_observed_contract_violation(
    const char* comment
    const std::source_location &location);
void handle_observe_contract_violation(
    const char* comment,
    const std::nothrow_t&) noexcept;
void handle_observe_contract_violation(
    const char* comment,
    const std::source_location& location,
    const std::nothrow_t&) noexcept;
```

Effects:

— Invoke the contract-violation handler with a `contract_violation` object populated as follows:

— The `comment` will be the `comment` passed to this function.

— The `location`, if populated, will be the `location` passed to this function or the location of the call sight.

— The `kind` will be `manual`.

— The `detection_mode` will be `manual`.

— The `evaluation_semantic` will be `enforce`.

`handle_quick_enforced_contract_violation`

[support.contracts.handle.quickenforced]

```
[[noreturn]] void handle_quick_enforced_contract_violation(
    const char* comment) noexcept;
[[noreturn]] void handle_quick_enforced_contract_violation(
    const char* comment,
    const std::source_location& location) noexcept;
```

Effects: Terminate the program in an implementation-defined manner.

Modify [support.contract.kind] paragraph 1:

...

— `assertion_kind::assert`: the evaluated contract assertion was an *assertion-statement*.

— `assertion_kind::cassert`: the contract violation was detected during evaluation of the `assert` macro.

— `assertion_kind::manual`: the contract violation was triggered manually.

...

Modify [support.contracts.detection] paragraph 1

...

— `detection_mode::evaluation_exception`: the contract violation occurred because the evaluation of the predicate evaluation exited via an exception.

— `detection_mode::manual`: the contract violation was triggered by invoking a manual contract handling function.

...

Modify [assertions.general] paragraph 1:

The header `<cassert>` provides a macro for documenting C++ program assertions, ~~and~~ a mechanism for disabling the assertion checks through defining the macro `NDEBUG`~~.~~, and a mechanism to integrate with the contract-violation handler through defining the macro `ASSERT_USES_CONTRACTS`.

Modify [assertions.assert] paragraph 2:

Otherwise, the `assert` macro puts a diagnostic test into programs; it expands to an expression of type `void` that has the following effects:

— `__VA_ARGS__` is evaluated and contextually converted to `bool`.

— If the evaluation yields `true` there are no further effects.

— Otherwise, if `ASSERT_USES_CONTRACTS` is defined as a macro name at the point in the source file where `<cassert>` is included, the `assert` macro's expansion invokes the contract-violation handler ([basic.contract.handler]). The `contract_violation` object passed to the handler will be populated as follows:

— The `kind` will be the value `cassert`.

— The `detection_mode` will be the value `predicate_false`.

— The `evaluation_semantic` will be the value `enforce`.

— The `location` will, if populated, represent the source file, line number, and name of the enclosing function.

— The `comment` will, if populated, contain `#__VA_ARGS__`.

— Otherwise, the `assert` macro's expression creates a diagnostic on the standard error stream in an implementation-defined format and calls `abort()`. The diagnostic contains `#__VA_ARGS__` and information on the name of the source file, the source line number, and the name of the enclosing function (such as provided by `source_location::current()`).

4 Conclusion

Providing the beginnings of a migration path for users of legacy assertion facilities — both `assert` and homegrown solutions — is an essential part of making early use of the Contracts facility viable for many users. The hooks proposed in this paper allow for such legacy facilities to live side-by-side with the Contracts MVP as proposed in [P2900R7], require no major changes to legacy facilities' existing semantics, and open the door to integration with post-MVP Contracts as soon as it is available.

— Existing facilities will have the ability to easily integrate with the violation-handling capabilities of the Contracts MVP.

— The `assert` macro will expose this same kind of optional functionality.

Support for a widened set of use cases is a natural extension to the functionality provided by [P2900R7], and we hope to see this proposal adopted to increase consensus for the Contracts MVP.

Acknowledgements

Thanks to John Spicer, Tom Honermann, and the rest of SG21 for the discussions that led to these proposals and to Lori Hughes for helping to greatly increase the readability and quality of this paper.

Bibliography

[N4981] Thomas Köppe, "Working Draft, Programming Languages – C++", 2024
http://wg21.link/N4981

[P2900R7] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, "Contracts for C++", 2024
http://wg21.link/P2900R7

[P3191R0] Louis Dionne, Yeoul Na, and Konstantin Varlamov, "Feedback on the scalability of contract violation handlers in P2900", 2024
http://wg21.link/P3191R0

[P3290R0] Joshua Berne, Timur Doumler, and John Lakos, "Integrating Existing Assertions With Contracts", 2024
http://wg21.link/P3290R0

[P3311R0] Tom Honermann, "An opt-in approach for integration of traditional assert facilities in C++ contracts", 2024
http://wg21.link/P3311R0