Integrating Existing Assertions with Contracts

Document #:	P3290R5
Date:	2026-07-08
Project:	Programming Language C++
Audience:	SG22, LEWG

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

Abstract

C++26 shipped with a contract checking facility introduced by [P2900R14]. C++ now includes three forms of contract assertion: `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. This paper proposes both a library API to perform that integration and changes to the `assert` macro to enable its interoperability.

Contents

Revision History

Revision 5, as presented to July 8, 2026 SG22 telecon

- Wording formatting and fixes
- Changed `assert` proposal to always `abort()` when the violation-handler completes
- Implementation experience with links on compiler explorer
- Proposing a more C-compatible control macro, `__STDC_WANT_ASSERT_USES_CONTRACTS__`

Revision 4, post-C++26, April 2026

- Rebased on C++26 as shipped
- Update to be ready to present to LEWG
- Wording cleanup and bug fixes
- Added feature test macros

Revision 3, post–Sofia meeting, June 2025

- Rebased on draft Standard after adoption of [P2900R14]
- Specified use of `source_location::current()` and discussed implementation choices
- Made `nothrow_t` parameters first parameters passed by value when used
- Discussed implementation experience
- Change `detection_mode::manual` to `detection_mode::unspecified`
- Addressed handling of `<assert.h>` vs. `<cassert>`

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

The C++ Standard now includes a Contracts feature, first introduced in [P2900R14] and shipped in C++26. 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 exactly 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* evaluation 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.

Because many existing assertion facilities will need to remain committed to behaviors and control mechanisms that may never be perfectly replicated by the core-language Contracts facility, this paper proposes two independent additions that allow for effective integration of any such 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 C++.
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 facility. With the adoption of one or both of these independent proposals, we hope to improve the use of Contracts in C++29, addressing many of the concerns that were raised in the C++26 cycle due to Contracts not providing an exact drop-in replacement for the `assert` macro.

Earlier versions of this paper were seen by SG21 and EWG, in both cases forwarding on these proposals to later groups:

SG21, Teleconference, 2024-09-05

Forward Proposals 1.1-1.3 of D3290R2 as presented, as a separate proposal on top of P2900, to EWG and LEWG for design review.

SF	F	N	A	SA
8	3	1	0	0

Result: Consensus

SG21, Teleconference, 2024-09-05

Forward Proposal 2 of D3290R2, with the polarity of NDEBUG switched and the suggested improvements in phrasing applied, as a separate proposal on top of P2900, to EWG and LEWG for design review.

SF	F	N	A	SA
5	7	0	0	0

Result: Consensus

EWG, Sofia, 2025-06-18

EWG approves the design of this paper and forwards this to LEWG and wants SG22 (C/C++ liaison group) to see the paper.

SF	F	N	A	SA
4	17	6	5	1

Result: Consensus

After the last poll in EWG, the majority of those against confirmed that implementation experience was the only concern. Implementations have been performed in both libc++ and libstdc++, although they have not yet been updated and upstreamed as the updates to the shared Itanium ABI that those implementations will call into are still in progress.

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 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 the C++ Contracts facility, 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 to integrate the contract-violation handler 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`. It's certainly likely that most implementations will have such a general function underlying those proposed here, but we do not want to prematurely restrict implementation options by designing and demanding that general facility.

- 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 (such as being `[[noreturn]]`) 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 C++. 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.

  Just as is expected by the violation handler, this should be a null-terminated multibyte string in the ordinary literal encoding.

  Because it is helpful to have these functions maintain a wide contract — reducing the chance of new bugs creeping in during the handling of a violation — a `nullptr` for `comment` should be interpreted as an empty string (although an implementation is free to provide a comment indicating the particular library function that was used to trigger the contract-violation handling process).
- 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, the defaulted `source_location` parameter is initialized to `std::source_location::current()`, producing the location where the function is invoked. Because this is a Standard Library function and thus the exact signature of the function is not guaranteed it is possible for an implementation to provide an overload with no defaulted parameter and instead acquire the source location of the caller in some other, implementation-defined manner. Such overloads would also be appropriate on platforms designed to strip out all identifying information from generated binaries.

  2. It is also possible to provide the `source_location` parameter explicitly. This parameter will be recommended for use in the `contract_violation` object that will be passed to the contract-violation handler.[1]

- 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.

  Introducing parameters of these types would also require that we answer the question of what happens if an invalid enumeration value is passed to these functions. Overloads avoid having to decide if we narrow the contract to preclude such cases or expect violation handlers to support arbitrary values from any source.

The names presented for all of these functions have been selected to be clear, distinct, and to capture the meaning of their intended behaviors.

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

[1] 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.

SG21, St. Louis, 2024-06-27

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,
      const std::source_location &location = std::source_location::current());
  void handle_observed_contract_violation(
      const char* comment,
      const std::source_location &location = std::source_location::current());
}
```

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

- Create and populate an object of type `std::contracts::contract_violation`.
  - The `comment` property is recommended to be the value provided as a function argument if not `nullptr`; Otherwise, a textual representation of the invoked overload is the recommended value.
  - 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, `unspecified`, 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,
    const std::source_location& location
        = std::source_location::current()) 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) 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.

3. The same wrapping approach can be taken while forwarding a `source_location` value that is defaulted at the call site of the wrapper:

   ```
   [[noreturn]] void my_handle_enforced_contract_violation(
       const char* comment,
       const std::source_location& location
           = std::source_location::current()) noexcept
   {
     std::contracts::handle_enforced_contract_violation(comment, location);
   }
   ```

This approach forwards the caller's source location, but also makes it much harder for a platform that wants to strip out uses of source location to do so, as the `source_location` is now baked in as part of a user-provided API that is not easily optimized away.

Therefore, we propose adding overloads to the proposed API that take an additional argument of type `std::nothrow_t` passed by value, similarly to how that type is used for nonthrowing operator `new`. Because the ordering of this parameter should not change with respect to a source location when provided, we make it the first parameter (similarly to how tag types are used through the rest of the standard library, although not how `nothrow_t` is used currently with `operator new` and `operator delete`). Because it is an empty and trivial type, we pass it by value to avoid needing to resolve a pointer to a static instance of this type (`std::nothrow`) whenever these functions are invoked.

Proposal 1.3: `noexcept` Overloads

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

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

Finally, there seems to be no compelling reason to implement the above library additions incrementally, so a single feature test macro suffices to indicate that the whole suite of free functions is available:

Proposal 2: Feature test macro for library API

The feature test macro `__cpp_lib_contracts_api` will have a value of `yyyymmL` if the free functions proposed above are available. This macro will be defined by both `<version>` and `<contracts>`.

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::contracts::handle_enforced_contract_violation(std::nothrow,#X);\
    }
#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:

```
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(std::nothrow,comment,location);
  } else {
    // When neither above configuration is selected, we log and continue.
    std::contracts::handle_observed_contract_violation(std::nothrow,comment,location);
  }
#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 default argument that is an immediate function, 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 C++ seems time-consuming and counterproductive. Therefore it is not feasible to consider redefining `assert` in terms of `contract_assert`.

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 `handle_enforced_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`, and `abort` is the only mechanism used to terminate the program after the violation handler completes.

As with the current state of the `assert` macro, this behavior when compiling in C++ should be consistent regardless of whether the macro came via including `<cassert>` or `<assert.h>`. This state of affairs is already the case in existing implementations and the behavior already differs from the C behavior of the macro since the adoption of [P2264R7].

An often cited concern with this integration is that it might enable the `assert` macro to suddenly become the source of an exception, something that legacy code is unlikely to be ready to deal with. Due to that, we do not allow exceptions from the invoked violation handler to escape and instead exit the program (through `std::abort()`, matching how `assert` historically terminates the program) if the violation handler does throw (similarly to calling one of the `nothrow_t` overloads from Proposal 1.3).

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.[2]

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

[2] Earlier versions of this paper proposed making the choice of integrating `assert` with the contract-violation handler one that was implementation-defined — i.e., a command line flag outside of the language. An alternative proposal, [P3311R0], proposed using a user-defined macro to control that choice (on the command line or in code).

SG21, when discussing these two proposals, 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.

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.

- 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 fundamentally this behavior is not C++-specific, and having `CPP` in its name would make it inappropriate as part of a WG14 proposal.
- Alternatively, because `assert` is primarily a C facility, a macro that begins with `STDC` could be considered, or a reserved name macro that begins with `__STDC` and ends with `__`. C23 introduced three control macros to control the behavior of `<math.h>` that can be used as a naming precedent:
  - `__STDC_WANT_IEC_60559_EXT__`
  - `__STDC_WANT_IEC_60559_EXT__`
  - `__STDC_WANT_LIB_EXT1__`

  Following that precedent, we could choose to make our control macro be `__STDC_WANT_ASSERT_USES_CONTRACTS__`, or alternatively `__STDCPP_WANT_ASSERT_USES_CONTRACTS__`.
- 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`.

Following the precedent of WG14 (where macro use is much more common in the first place, and which we hope will follow through with adopting a compatible Contracts facility and support integration in the same way), we suggest the use of the name `__STDC_WANT_ASSERT_USES_CONTRACTS__`.

SG21, St. Louis, 2024-06-27

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

After that poll was taken in SG21, this paper updated to propose the use of a control macro and that is the version which has been forwarded since then.

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 3: Integration of `assert` with the Contract-Violation Handler

When `NDEBUG` is not defined and `__STDC_WANT_ASSERT_USES_CONTRACTS__` is defined as a macro name at the point in the source file where `<cassert>` or `<assert.h>` 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::contracts::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, `std::abort()` will be invoked.

Note that the above description is roughly equivalent to an invocation of `handle_enforced_contract_violation(std::nothrow,#__VA_ARGS__)` with the caveat that different values for `kind` and `detection_mode` are passed through to the contract-violation handler as well, and any completion of the contract-violation handler results in calling `std::abort()`. This choice results in the smallest change possible to the behaviors of `assert()` while integrating it with the contract-violation handler.

As always, a feature test macro should also be provided to identify whether the current platform has implemented the associated functionality:

Proposal 4: Feature test macro for `assert` integration

The feature test macro `__cpp_lib_assert_can_use_contracts` will have a value of `yyyymmL` if the behavior `assert` depends on the value of `__STDC_WANT_ASSERT_USES_CONTRACTS__` as described above. This macro will be defined in `<version>`, `<assert.h>`, and `<cassert>`.

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

```
#ifdef NDEBUG
  #define assert(...) (static_cast<void>(0))
#elif defined __STDC_WANT_ASSERT_USES_CONTRACTS__
#include <source_location>  // for std::source_location
[[noreturn]] void __cxa_handle_cassert_violation(
    const char* comment,
    std::source_location = std::source_location::current() ) 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, then invokes std::abort()

  #define assert(...)                                                    \
    ((__VA_ARGS__)                                                       \
```

```
    ? static_cast<void>(0)                                                  \
    : __cxa_handle_cassert_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 error and abort

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

Note that in the above we made the choice to directly `#include` the `source_location` header. A compiler and Standard Library implementation need not do so necessarily, but could instead opt to introduce additional compiler builtins or internal headers to achieve the same net result.

3 Implementation Experience

The library API and `assert` integration proposed in this paper have been implemented in branches of GCC (libstdc++) and Clang (libc++) that are available on Compiler Explorer: [Compiler Explorer].

Both implementations provide the full set of programmatic violation-reporting functions — `handle_enforced_contract_viol` `handle_observed_contract_violation`, `handle_quick_enforced_contract_violation`, and their `std::nothrow_t` overloads — as well as integration of the C `assert` macro with the contract-violation handler, gated on `__STDC_WANT_ASSERT_USES_CONTRACTS__`. The library API populates the `contract_violation` object with `assertion_kind::manual` and `detection_mode::unspecified`, while a failed `assert` populates `assertion_kind::cassert` and `detection_mode::predicate_false`. Both compilers define the feature-test macros `__cpp_lib_contracts_api` and `__cpp_lib_assert_can_use_contracts` with matching values.

The two standard libraries share a single ABI entry point for the `assert` integration. This ABI is implemented in both compilers, and will be proposed as part of the Itanium ABI in the future. The `<cassert>` header in each library expands the violation branch in `assert(E)` to the same symbol, `__cxa_handle_cassert_violation`, which constructs the `contract_violation` object and invokes the handler. That branch replaces the invocations of `__assert_fail` (which in its implementation calls `std::abort()`) that occur when `NDEBUG` is not defined. Using one shared entry point keeps the observable behavior of `assert` identical across the two libraries and removes gratuitous differences between them.

We chose to have the shared contracts ABI perform enforced contract-termination with `std::abort()` rather than `std::terminate()`. Unlike `std::terminate()`, `std::abort()` is not affected by a user-installed terminate handler, so contract-termination is predictable and uniform regardless of program state; it is also the historical behavior of `assert`, which this integration preserves. The one place the ABI still uses `std::terminate()` is when a violation handler exits via an exception at a `noexcept` entry point, matching the language's own `noexcept` mechanism. For the `assert` entry point specifically we override even that case: it catches an escaping exception and calls `std::abort()`, so that a failed `assert` always

terminates via `abort()` on *any* completion of the handler. This decision avoids mixing C++-specific behavior with a macro primarily considered part of the C Standard Library.

The initial level of effort was low in both compilers. P3290 is almost entirely a library feature; the compiler's only role is to define the gate macros that enable the library API and the `assert` integration. The bulk of the work was in the standard library, constructing the `contract_violation` object from library code and routing it through the shared `__cxa_contract_violation` ABI — the same entry points the compiler emits for language contract checks — which also gives cross-compiler interoperability: a violation reported from libstdc++ can be handled by a handler built against libc++, and vice versa.

The two implementations differ only in incidental integration details. GCC required a change to `g++spec.cc` so that using the API implicitly links `-lstdc++exp`; Clang's driver did not need an analogous change. Both implementations needed differing amounts of rearranging of the `<assert.h>` and `<cassert>` headers so that the needed behavior and feature test macros was exposed. Both libraries needed to begin deploying their own version of `<assert.h>` in order to do this, using machinery similar to that already used for headers like `<stdlib.h>` and the corresponding `<cstdlib>`.

4 Wording Changes

These wording changes are relative to the C++29 working draft with git hash 9fd05919, last modified on Thu, 4 Jun 2026 21:25:15 +0200.

17 Language support library **[support]**

17.3 Implementation properties **[support.limits]**

17.3.2 Header `<version>` synopsis **[version.syn]**

Modify section 17.3.2[version.syn], paragraph 2:

2 Each of the macros defined in `<version>` is also defined after inclusion of any member of the set of library headers indicated in the corresponding comment in this synopsis.

[*Note 1*: Future revisions of this document might replace the values of these macros with greater values. — *end note*]

```
                                        ⋮  15 lines elided  ⋮
  // freestanding, also in <iterator>, <array>
#define __cpp_lib_as_const                          201510L // freestanding, also in <utility>
#define __cpp_lib_assert_can_use_contracts          yyyymmL // freestanding, also in <assert.h> and <cassert>↩
#define __cpp_lib_associative_heterogeneous_erasure 202110L
  // also in <map>, <set>, <unordered_map>, <unordered_set>


                                        ⋮  72 lines elided  ⋮
  // also in <vector>, <list>, <forward_list>, <map>, <set>, <unordered_map>, <unordered_set>,
  // <deque>, <queue>, <stack>, <string>
#define __cpp_lib_contracts                         202502L // freestanding, also in <contracts>↩
#define __cpp_lib_contracts_api                     yyyymmL // freestanding, also in <contracts>
#define __cpp_lib_copyable_function                 202306L // also in <functional>
#define __cpp_lib_coroutine                         201902L // freestanding, also in <coroutine>
```

⋮ *224 lines elided* ⋮

17.10 **Contract-violation handling** [**support.contract**]

17.10.1 **Header <contracts> synopsis** [**contracts.syn**]

Modify section 17.10.1[contracts.syn], paragraph 1:

1 The header <contracts> defines types for reporting information about contract violations (6.11.2[basic.contract.eval]).

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

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

  enum class evaluation_semantic : unspecified {
    ignore = 1,
    observe = 2,
    enforce = 3,
    quick_enforce = 4
  };

  enum class detection_mode : unspecified {
    unspecified = 0,
    predicate_false = 1,
    evaluation_exception = 2
  };

  class contract_violation {
    // no user-accessible constructor
  public:
    contract_violation(const contract_violation&) = delete;
    contract_violation& operator=(const contract_violation&) = delete;

    see below ~contract_violation();

    const char* comment() const noexcept;
    contracts::detection_mode detection_mode() const noexcept;
    bool is_terminating() const noexcept;
    assertion_kind kind() const noexcept;
    source_location location() const noexcept;
    evaluation_semantic semantic() const noexcept;
  };

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

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

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

17.10.2 **Enumerations** [**support.contract.enum**]

Modify section 17.10.2[support.contract.enum], Table 44 [support.contract.enum.kind]:

Table 44 — Enum `assertion_kind` [**tab:support.contract.enum.kind**]

Name	Meaning
`pre`	A precondition assertion
`post`	A postcondition assertion
`assert`	An *assertion-statement*
`manual`	A manually triggered violation
`cassert`	The `assert` macro (19.3.3[assertions.assert])

Modify section 17.10.2[support.contract.enum], Table 46 [support.contract.enum.detection]:

Table 46 — Enum `detection_mode` [**tab:support.contract.enum.detection**]

Name	Meaning
`unspecified`	The mode of detection was not provided to the contract-violation handling process
`predicate_false`	The predicate of the contract assertion evaluated to `false` or would have evaluated to `false`.
`evaluation_exception`	An uncaught exception occurred during evaluation of the contract assertion.

17.10.4 + *a* **Violation Invocation** [**support.contract.handle**]

Add a new section 17.10.4+a[support.contract.handle] after 17.10.4[support.contract.invoke]

17.10.4+a Violation Invocation [support.contract.handle]

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

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

1 *Effects*:

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

(1.1.1) — The `comment`, if populated, will be the `comment` passed to this function.

(1.1.2) — The `location`, if populated, will be the `location` passed to this function or the location of the function invocation.

(1.1.3) — The `kind` will be `manual`.

(1.1.4) — The `detection_mode` will be `unspecified`.

(1.1.5) — The `evaluation_semantic` will be `enforce`.

(1.2) — If the violation handler returns normally, the program is contract-terminated (6.11.2[basic.contract.eval]).

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

2 *Effects*:

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

(2.1.1) — The `comment`, if populated, will be the `comment` passed to this function.

(2.1.2) — The `location`, if populated, will be the `location` passed to this function or the location of the function invocation.

(2.1.3) — The `kind` will be `manual`.

(2.1.4) — The `detection_mode` will be `unspecified`.

(2.1.5) — The `evaluation_semantic` will be `observe`.

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

3 *Effects*: Contract-terminate the program (6.11.2[basic.contract.eval]).

19 Diagnostics library [diagnostics]

19.3 Assertions [assertions]

19.3.1 General [assertions.general]

Modify section 19.3.1[assertions.general], paragraph 1:

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 `__STDC_WANT_ASSERT_USES_CONTRACTS__`.

19.3.3 The `assert` macro [assertions.assert]

Modify section 19.3.3[assertions.assert], paragraphs 1-2:

1 If `NDEBUG` is defined as a macro name at the point in the source file where `<cassert>` or `<assert.h>` is included, the `assert` macro is defined as

```
#define assert(...) ((void)0)
```

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

(2.1) — `__VA_ARGS__` is evaluated and contextually converted to `bool`.

[*Note a*: If an exception escapes the evaluation of `__VA_ARGS__`, it propagates normally. — *end note*]

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

(2.2+a) — Otherwise, if `__STDC_WANT_ASSERT_USES_CONTRACTS__` is defined as a macro name at the point in the source file where `<cassert>` or `<assert.h>` is included, the `assert` macro's expression

(2.2+a.1) — invokes the contract-violation handler (6.11.3[basic.contract.handler]), and

(2.2+a.2) — then, if the contract-violation handler returns normally or exits via an exception, calls `abort()`.

The `contract_violation` object passed to the handler will be populated as follows:

(2.2+a.3) — The `kind` will be the value `cassert`.

(2.2+a.4) — The `detection_mode` will be the value `predicate_false`.

(2.2+a.5) — The `evaluation_semantic` will be the value `enforce`.

(2.2+a.6) — The `location` will, if populated, represent the source file, line number, and name of the enclosing function.

(2.2+a.7) — The `comment` will, if populated, contain `#__VA_ARGS__`.

(2.3) — Otherwise, the `assert` macro's expression creates a diagnostic on the standard error stream (ISO/IEC 9899:2024, 7.23.3) 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()`).

5 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 C++ contracts, require no major changes to legacy facilities' existing semantics, and open the door to integration with Contracts as soon as they are available.

- Existing facilities will have the ability to easily integrate with the violation-handling capabilities of C++.
- 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 Standard Contracts, and we hope to see this proposal adopted to facilitate a vastly widened integration of existing contract-checking facilities with the central violation-handling mechanisms of C++ Contracts.

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. Thanks to Eric Fiselier, Nina Ranns, Iain Sandoe, and Ville Voutilainen for excellent feedback when implementing this proposal. Thanks to Jens Gustedt for feedback on an earlier version of this paper.

Claude (Anthropic) was used for editorial assistance during the preparation of this paper, as well as significant parts of implementing the prototype implementations.

Bibliography

[P2264R7] Peter Sommerlad, "Make assert() macro user friendly for C and C++", 2023
http://wg21.link/P2264R7

[P2900R14] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, "Contracts for C++", 2025
http://wg21.link/P2900R14

[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