Integrating Existing Assertions With Contracts

Document #:	P3290R0
Date:	2024-05-22
Project:	Programming Language C++
Audience:	SG21 (Contracts)

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`, have semantics that allow contract checks to legitimately spread across many independent assertions and associated preprocessor-guarded blocks of code. In so doing, they also necessarily greatly limit their own utility in new code where the target is to substantially minimize undefined behavior and improve safety by reducing security vulnerabilities and generally improving correctness. In this paper, we propose three additions to the Contracts MVP; these three additions will allow current users of the standard C `assert` macro, as well as similar homegrown assertion facilities, to seamlessly integrate with and exploit a maximal subset of our burgeoning C++ Contracts MVP without compromising its ability to improve security and correctness.

Contents

Revision History

Revision 0

- 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.

An important distinguishing feature of the SG21 Contracts proposal is that each individual evaluation of a contract assertion it proposes to provide — whether using `pre`, `post`, or `contract_assert` — is, when checked, independent of every other assertion check, including previous and future evaluations of the same assertion. Thanks to this important property, the semantics with which assertions can be evaluated are completely implementation defined[1] and, importantly, can vary (at run time) from one evaluation to the next.

Historical contract-checking facilities, however, such as the standard `assert` macro or various customized macro-based improvements on it, have a subtly different model due to their fundamental macro-based nature. Individual assertion macros reliably have the same semantic — based on one or more configuration macros, such as `NDEBUG`, referenced when the assertion macro itself is defined — any time they are used within a translation unit (TU). Keeping the semantic consistent across all local uses of the same macro creates an implicit dependency among the evaluations of their associated predicates, which can be safely relied upon, often to good effect. Of course, being a macro, any change of behavior that is not consistent across all definitions of a function throughout an entire program leads to violations of the One Definition Rule (ODR).

The contract assertions of the SG21 MVP make such implicit dependencies problematic; they do not require, or even allow, the semantic of a given contract assertion to be knowable at compile time. In so doing, the MVP removes ODR concerns that accompany the classic preproccessor-based model for varying semantics by expanding to distinct sequences of code. An implementation merely defines how the user's choice of configurations and linking options will result in a final semantic when the predicate of a given contract is evaluated. To maintain a sound logical model on which we can build, any single contract assertion must be a single, independent contract check.

This same model of contract assertions being independent of one another means that each individual contract assertion can be added or removed from a program without changing the correctness of that program. This property is the logical foundation of incrementally improving a program's correctness using Contracts.

[1] This model for semantics, where implementations are given complete freedom to define arbitrarily flexible mechanisms to control the semantics of contract assertion evaluation, is the result of the adoption of [P2877R0] by SG21.

This very different design approach lends itself to many important new use cases while not precluding any existing ones — although some may require additional features not present in the MVP before they can be realized. As a result, predicates that worked as intended in `assert` macros in legacy programs will not necessarily work correctly if simply migrated unchanged to `contract_assert`. In practice, however, most education on the use of assertions includes instructions to avoid predicates that have side effects when evaluated, meaning the overwhelming majority of assertions will be supported with ease.

To enable the many essential features for C++ Contracts to eventually be usable at scale in industrial settings with more than just a few programmers, discouraging any side effects in predicates becomes necessary and desirable as does prohibiting destructive side effects; i.e., any side effects of evaluation that actually alter the correctness of a program, including the results of any subsequent evaluations of contract assertions.

For reasons relating to consistency, code mobility, and effective testing for destructive side effects, the compiler is permitted to repeat the evaluation of any contract assertion up to an implementation-defined number of times, but we expect most implementations to define this number as a function of the number of repetitions requested when compiling the program.[2]

To further reinforce the notion that, unlike with the standard `assert` macro, side effects in C++ `contract_assert` are not well supported, we have introduced the novel notion of *`const`-ification*; i.e., any identifier representing a local variable, a member variable, `*this`, or (for `post` only) the result object, has `const` added to its type and is thus made nonmodifiable. The underlying object remains modifiable if it has not been declared `const`; hence, in cases where an API must be used that makes no modifications but does not work on `const`-qualified objects, a `const_cast` is a viable and well-defined mechanism to allow the use of such APIs. Conversely, any code that fails to compile as a result of `const`-ification is highly likely to be an inadvertent error by well-meaning developers and to be trivially fixed (see [P3268R0]).

These properties of the C++ `contract_assert` make it inherently incompatible with some stateful uses of `assert` where a single checking mechanism is spread out over multiple contract assertions (e.g., setting a flag in one assertion and unsetting it in another assertion) or multiple evaluations of the same contract assertion (e.g., incrementing a counter inside an assertion and checking whether the counter's value is below some fixed number). Thus, plenty of legacy assertions cannot not be reliably migrated to MVP Contracts as defined in [P2900R7] without depending on the implementer to provide a special build mode. Even then, such a mode would fork the language, which we have worked diligently to avoid; libraries that depend on this build mode would be unsuitable for users who wish to make use of any other build mode.

Since presenting our nascent MVP to EWG in Japan (c. April, 2024), an ongoing debate has arisen over which (mutually exclusive until now) option the MVP should aim to provide:

1. a self-consistent new facility primarily focused on making new and existing code safer
2. a *fancy* version of `assert` that removes some of `assert`'s rough edges but supports a complete superset of its valid uses

Requiring both is in direct conflict with the default behavior for `contract_assert`. Therefore, perhaps

[2] See [P3119R1], Section 4.

we should set aside discussion of what exactly `contract_assert` should do and instead creatively collaborate to find a solution (or set of solutions) that will address everyone's needs yet allow C++ Contracts to evolve to reach its full potential post-MVP.

This paper proposes three separate, mutually compatible ways in which the needs of those who advocate for option 2 over option 1 can be met in the MVP without permanently hindering the ability to smoothly support large subsets of the use cases identified by SG21 in [P1995R1]. Having a long-term solution that satisfies all those goals consistently and in an easily understandable fashion is far more likely to result in a robust, safe, scalable C++ Contracts facility than simply aiming to build a fancy `assert` replacement would do now.

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.
3. **Add a new form of contract assertion with more familiar semantics:** Introduce a distinct but parallel form of contract assertion that aims to fully address precisely the same use cases as the standard `assert` macro which `contract_assert` does not serve. Originally discussed with the name `contract_c_assert` as a potential mechanism to improve consensus, it is clear that the primary discerning feature of this new assertion is that it allows for presenting contract assertions that are only a small part of a larger contract-checking algorithm, and therefore we propose an initial name of `partial_contract_assert`. This new assertion statement would allow side effects and would not perform `const`-ification by default.

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 more of these three independent proposals, we hope to significantly advance the Contracts MVP since the vast majority of concerns regarding lack of support for a drop-in C-assert replacement will have been addressed.

2 Proposals

In this section, we provide three 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 obviously initial suggestions, with proposed reasoning, but still highly likely to be subject to future changes during the standardization process.

2.1 Support Directly Invoking the Contract-Violation Handler

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, 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 an *enforced* or *observed* contract, 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 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 semantic is `enforce`.
- Perhaps a feasible approach would 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`.

- A different function we might propose would use the same semantic as is configured for other contract assertions, but that is not a clearly well-defined value that could be accessed, and the flexibility of that choice of semantic is a big part of the source of problems when just 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 — the existing macro-based controls and the controls which 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 instead are focusing 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 tied into builds where these recommended practices were 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 existing assertion facility has decided to invoke the handler, it 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.
- Source location can be detected using a defaulted `std::source_location` function parameter, but this option need not be specified explicitly. Instead, we can simply dictate that the generated `contract_violation` object's `location` property has the location of the call site. This approach also leaves leeway, in some builds, to discard such information where it is deemed private.
- 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 present 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 of these considerations together we suggest the following initial minimal proposal for an API.

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);
  void handle_observed_contract_violation(const char* comment);
}
```

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 of the function invocation, though as usual it may also be a default-constructed `std::source_location` or have any other value.
  - 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. We could consider, as a second proposal on top of the above, a third function, which would have the semantics of a contract violation with 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;
```

- 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 need only 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 comes 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
  void handle_observe_contract_violation(const char* comment,
                                         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 there is no invocation of a contract-violation handler that 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.

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 the [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. We, therefore, propose this change to be implementation defined and will likely see this change take effect only when users explicitly request it, e.g., via command-line options.

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

When the evaluation of the expression in an `assert` macro yields `false`, which happens only when `NDEBUG` is undefined, the implementation defines whether the currently specified diagnostic is output to the standard error stream or the contract-violation handler is invoked. If the contract-violation handler is invoked, the behavior is equivalent to a call to the function `handle_enforced_contract_violation(#__VA_ARGS__,std::nothrow)`, but the `kind` property of the generated `contract_violation` object will be a new enumerator, `cassert`, and the `detection_mode` will be the value `predicate_false`.

2.3 Introduce a New Contract Assertion Statement With More Familiar Semantics

Like many other aspects of the C++26 MVP, a seemingly simple problem isn't always that simple. A question that often comes up is why `contract_assert` has semantics that differ from those of the more simple model inherent in the `assert` macro, a tool with longevity.

Central to the concept of contract checking is that introducing contracts into a program should not alter the correctness of the program, but should simply identify the cases where the program is correct or incorrect.

Each of the contract assertion kinds in the MVP today might select a different evaluation semantic every time it is evaluated in a program, independently of both the semantic of other contract assertions and even the semantic of earlier evaluations of the same contract assertion. Due to this ability to change semantic on each evaluation, it is essential that each evaluation of a contract assertion not alter the correctness of the program independently of all other contract assertion evaluations.

By contrast, the semantic with which `assert` macros are invoked is chosen based on the state of the `NDEBUG` macro when the header `<assert.h>` or `<cassert>` is processed. If `NDEBUG` is defined, that semantic is effectively *ignore*; otherwise, it is *enforce*. Therefore, it is perfectly reasonable for evaluations of assertion macros to depend on earlier evaluations of assertion macros or on blocks guarded by `#ifndef NDEBUG`, within the same translation unit.

If `pre`, `post`, and `contract_assert` preserved the same stateful properties as `assert` for its direct users, many fundamental tools would be removed from those who are building programs augmented with contract assertions.

- Arbitrary subsets of assertions could not be enabled or disabled independently while maintaining a correct program. This restriction would prevent commonly identified use cases, such as enforcing all preconditions and ignoring all postconditions, or more eclectic yet still valid options, such as enforcing a fixed (yet randomly selected) percentage of all contract assertions and ignoring others.
- Runtime selection of the evaluation semantic could not change while a program is running because modifying such a flag to enable assertions that would be correct only if earlier assertions had been evaluated would result in immediately encountering falsely identified contract violations.
- We could never safely assume individual contract assertions because their correctness might depend on earlier evaluations of contract assertions that would not have taken effect if they had been assumed.

We would hope to eventually see, layered on top of the Contracts MVP, new syntax that allows users to clearly specify when a relationship occurs between distinct contract assertions such that their evaluations could not be made independent of one another — i.e., grouping assertions and supporting code into user-defined groups that can be individually controlled.[3] Sadly, that design is out of scope for the Contracts MVP targeting C++26.

[3] See section 2.2.20 of [P2755R1] for an initial suggestion as to how such a facility might be expressed and behave, though there is recognizably far more to explore in this space before a complete proposal can be offered.

A possible interim solution is to provide a new syntax that places assertion-statement-like constructs that are controlled as a single unit within a translation unit. Normal `contract_assert` statements could remain entirely independent of themselves and one another, which serves the vast majority of use cases, while this new construct would help bridge the gap with the subset of `assert` predicates that are not independently evaluable.

Let's now take a closer look at what use cases the standard `assert` macro enables over the current MVP and why.

The stronger evaluation guarantees of `assert` allow a programmer to reason about how assertion evaluations relate to one another due to their behavior being tied to the state of macro definitions. Consider, for example, making use of `assert` to protect against recursive invocation of a function by using an RAII type that does all modifications and checks inside `assert` macro invocations so as to have no checking overhead when `NDEBUG` is not defined:

```
#include <cassert>
class NoRecursionGuard
{
private:
  bool* d_inFunction_p;
public:
  NoRecursionGuard(bool* inFunction_p)
    : d_inFunction_p(inFunction_p)
  {
    assert( !*d_inFunction_p );
    assert( *d_inFunction_p = true );
  }
  ~NoRecursionGuard()
  {
    assert( *d_inFunction_p );
    assert( ! ( *d_inFunction_p = false ) );
  }
};

void nonrecursive()
{
  static bool inFunction = false;
  NoRecursionGuard guard(&inFunction);

  // Do stuff.
}
```

None of the assertions within `NoRecursion`'s constructor or destructor will work as intended if elided or repeated or if any mix of them is *ignored* while others are checked. With `assert`, however, this concern is never an issue because all these assertion predicates are either checked or unchecked as a single unit based on the value of `NDEBUG` when `assert` is defined. With `NDEBUG` defined, everything involved in the check against recursion is optimized away.

Conceptually, each of these assertions is only a piece of a larger compound contract check that maintains internal state as the different `assert` expressions are reached. To support this kind of use case, we must consider several points.

- Assertions such as those above maintain state entirely by having side effects of evaluations of the `assert` macro. In general, however, the state being maintained and the code to maintain are often placed inside blocks guarded by `#ifndef NDEBUG`.
- Typical legacy assertion facilities provide no syntactic indication of when assertions are or are not tied together. Therefore, the safest approach to replicate the expected semantics of such facilities is to keep the semantics of our proposed replacement consistent through all uses within definitions from the same TU. Just like macro-based facilities, we also need to extend the ODR to make ill-formed, no diagnostic required those cases where the same use gets different semantics in different inlined definitions; if we don't, we lose guarantees that an individual assertion will be evaluated or not evaluated consistently throughout the life of a program.
- Importantly, `const`-ification — i.e., treating each identifier in the assertion's predicates as if it were passed by `const` *lvalue* reference — is more of a hindrance to writing these kinds of assertions than a help, so it should not apply to this new piece of the Contracts facility we present here.
- Elision and duplication, similarly, break the semantics of a facility where we depend on the side effects of evaluating contract predicates. Therefore, we should not support those either.
- The name for this new facility should convey its intended purpose and semantics and be clearly distinct from `contract_assert`.
  - While the semantics are intended to mimic those of `assert`, the option `contract_cassert` is far too similar to `contract_assert`. Though `contract_c_assert` is sufficiently visually distinct, it is describing the behavior of a new and eminently useful C++ Contracts feature in terms of a facility that many are actively trying to obsolete and eventually deprecate and where the difference in meaning from `contract_assert` is subtle to understand and not obviously inherent in `assert` to those who are not well studied in its behavior.
  - So, if the meaning we ascribe to a `contact_c_assert` is that it is a part of a bigger contract assertion that applies to the entire TU (and, by extension, program), we would like to choose a name that indicates that.
    - Some short-sighted names might tie this construct to just its nature as having a consistent semantic throughout a translation unit, with varying degrees of accuracy:
      - `contract_global_assert`
      - `contract_tu_assert`
      - `contract_program_assert`
      - `contract_ubiquitous_assert`
    - But such wide-spread interdependence need not be the case. Post-MVP, we might decide to allow labels to create smaller cliques of assertions within a given TU to be treated as atomically checked or not. A better, more durable name would reflect that a particular contract assertion is part of a larger cohesive (atomic) whole, which for

now is the entire TU or program. To that end, we might consider more a expressive name:

- `partial_contract_assert`
- `dependent_contract_assert`
- `connected_contract_assert`

Though `partial_contract_assert` might sound like its a partial `assert` that happens to be a contract `assert`, that's not the intent; it is an `assert` that part of a larger contract check ("partial-contract assert") of which there are typically more (or else this one is stateful), and hence we cannot elide or repeat predicates. Other, more intuitive spellings are certainly possible:

* `contract_part_assert`

(which means "contract-part assert" or "assert for part of a contract"). For now, we'll use `partial_contract_assert` as a working name and save the nuance of naming for another day.

Therefore, we propose our remaining suggested name for the new facility, partial-contract assertions, and the corresponding new keyword, `partial_contract_assert`, since each assertion specified with this new facility is semantically part of a single large contract check consisting of all such assertions in the TU.

- As suggested above, future evolution of this facility might consider grouping the evaluations of partial-contract assertions by more than just the TU in which they are defined.
  - The simplest case is a partial-contract assertion that depends on only its own earlier evaluations to remain correct, such as one that increments a counter when checked and must not be invoked more than a specified number of times. This requirement might apply within a single function invocation (if the incremented variable is automatic) or across all function invocations (if the stateful variable has static storage duration).
  - In some cases, the only needed requirement is that all evaluations within a single function call are made with either all checked semantics (*observe*, *enforce*, or *quick_enforce*) or all unchecked ones (*ignore* and, post-MVP, *assume*).
  - Other situations require that matching assertions across different invocations of the enclosing function be evaluated together but are otherwise independent of any other partial-contract assertion.

  Each of the use cases above that require subgroupings within the TU would require additional syntax to be added to this proposal to specify the groupings themselves and to exist at the appropriate scope and for the appropriate duration.

Putting this together, we propose a new type of assertion statement, `partial_contract_assert`.

Proposal 3: `partial_contract_assert`

Add a new `kind` of *contract assertion*, `partial_contract_assert`, which mirrors `contract_assert` but with the following differences.

- A partial-contract assertion is introduced by the new keyword, `partial_contract_assert`.
- Each partial-contract assertion in a TU will be evaluated with the same implementation-defined contract-evaluation semantic.
- Extend the ODR so that a partial-contract assertion must always have the same evaluation semantic; otherwise, the program is ill-formed, no diagnostic required.
- When a checking semantic (*enforce*, *quick_enforce*, or *observe*) is used to evaluate a partial-contract assertion, the predicate is evaluated as normal and may not be elided.
- Partial-contract assertions do not form parts of contract-assertion sequences and thus will always be evaluated in lexical order with respect to all other contract assertions.
- The evaluation of a partial-contract assertion may not itself be repeated.
- Within a partial-contract assertion, the type of *id-expressions* referring to objects with automatic storage duration is not implicitly made `const` as is done within the predicates of other contract assertions.

This facility provides a clear mechanism to specify a contract assertion that is part of an implicit larger whole. Not coincidentally, such a partial-contract assertion is precisely what a standard `assert` macro provides. The more limited choices of evaluation semantics, removal of elision, duplication, and `const`-ification of `contract_assert` that led to `partial_contract_assert` remove several potential benefits but provide a mechanism that is a direct replacement for many more use cases currently served by the standard `assert` macro and similar macro-based tools.

3 Wording Changes

4 Conclusion

The core building block of the Contracts MVP which we have sought to provide — the ability to simply and nonintrusively state that, at a particular point of evaluation in a program, a certain predicate must be `true` — is a tool that serves many needs quite well. In particular, such predicates form a key basis for documenting and checking the correctness of a program at run time or compile time, and any effects a predicate might have that alter the correctness of a program must be considered destructive and avoided. What contract assertions in the Contracts MVP do not provide and until now have not sought to provide is a complete drop-in replacement for all the use cases that might be covered by existing macro-based assertion facilities. Much of the friction arises from uses of those facilities that check contract checks as a compound operation of multiple evaluations of contract predicates, often in practice combined with additional blocks of code enclosed in preprocessor guards such as `#ifndef NDEBUG`. Making `contract_assert` satisfy all these use cases on its own would necessarily sacrifice other use cases we have been working to satisfy.

In this paper, we have put forth three independent yet mutually compatible proposals, any or all of which might be a valuable addition to the Contracts MVP.

1. Provide utility functions that enable the violation handler to be invoked from bespoke legacy assertion facilities.
2. Modify the definition of the C++ `assert` macro to conditionally support invoking the C++ Contracts violation handler (instead of printing the currently specified diagnostics) before aborting to allow existing use of C `assert` to be expanded (only if explicitly requested) without having to rework a massive amount of legacy code.
3. Provide a parallel construct to `contract_assert` that more closely mirrors the semantics of the standard `assert` macro, effectively creating a drop-in replacement for that and many other existing contract-checking facilities.

Adopting these three solutions will be highly beneficial.

- Each supports additional use cases that are known to be unsupported by the current MVP as defined by [P2900R7].
- Collectively, they help clarify the need for novel behavior — benefiting those who use it and those who don't — simply by learning why they were incorporated into the MVP.
- We hope that the Contracts MVP, by clearly providing mechanisms to suppport known use cases for existing tools, will achieve increased consensus in WG21.

Any feedback or suggestions that might further help improve consensus are welcome.

Acknowledgements

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

Bibliography

[P1995R1] Joshua Berne, Andrzej Krzemieński, Ryan McDougall, Timur Doumler, and Herb Sutter, "Contracts — Use Cases", 2020
http://wg21.link/P1995R1

[P2755R1] Joshua Berne, Jake Fevold, and John Lakos, "A Bold Plan for a Complete Contracts Facility", 2024
http://wg21.link/P2755R1

[P2877R0] Joshua Berne and Tom Honermann, "Contract Build Modes, Semantics, and Implementation Strategies", 2023
http://wg21.link/P2877R0

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

[P3119R1] Joshua Berne, "Tokyo Technical Fixes to Contracts", 2024
http://wg21.link/P3119R1

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

[P3268R0] Peter Bindels, “C++ Contracts Constification Challenges Concerning Current Code”, 2024
http://wg21.link/P3268R0