A Framework For Contracts

Document #:	P3968R0
Date:	2026-01-16
Project:	Programming Language C++
Audience:	EWGI / SG21

Reply-to:	Bengt Gustafsson <bengt.gustafsson@beamways.com>

Contents

1 Overview

This proposal introduces a framework where various contract assertion types can be defined. The aim is to remove the dependency from the compiler to the standard library and delegate as much as possible of the functionality to library code for maximum flexibility. The `<contracts>` header implements the contract assertions according to the C++26 draft standard but different contract assertion types can also be specified.

This proposal strives to reconcile the seemingly opposing requirements that a) contract-assertion must be adaptable to the requirements of different codebases and b) it must be possible to verify that contract-assertions

according to those requirements are actually in place. With C++26 draft contracts many feel that the adaptability is favored making it is too hard to guarantee that the contracts-assertions have the desired semantics. Alternative proposals go in the other direction and want to remove flexibility in favor if easy verifiability.

With this proposal you can get an exact semantics of contract-assertions by giving them other names or redefine the `pre`, `post`, and `contract_assert` names to have well defined semantics that can't be changed with compiler switches.

In the proposed contracts framework the name of a contract-assertion refers to an object called a *assertion object*. A user defined class type is an *assertion object type* if it fulfills a certain concept and `constexpr` instances of this type are *assertion objects*. The values of specifically named data members of an *assertion object* controls how the compiler generates code for contract-assertions and an `operator()` of the assertion object is called when a contract assertion is violated. This idea bears resemblance with how co-routine `promise_type`s can be used to configure how the compiler generates the code for the co-routine.

To retain backwards compatibility to C++26 (except that `<contracts>` has to be included to use C++26 contracts) an *assertion object type* with three *assertion objects* called `pre`, `post`, and `contract_assert` are included in the proposal.

The configurability proposed in [P3400R2] is achieved by providing *assertion object*s with separate names instead of providing a label as a template argument. To get an always enforced pre-condition you would thus write something like `pre_enforce(cond)` instead of `pre<enforce>(cond)`.

This proposal agrees with the statement "*Various things that we've run into that should have been doable as third-party libraries, not language changes*" of [P3909R0] but without immediately drawing the conclusion that contracts as such should be withdrawn from the C++26 draft. This is as introducing this proposal later can be made in a backwards compatible way, removing the dependency from the compiler to the standard library. There are however some reasons to withdraw C++26 contracts in case this proposal is selected for C++29, mainly that the compiler options for selecting semantics are not needed, and with them the *semantic* parameter of the assertion object type's `operator()`. This removes one of the big risks for not getting the contract checks you asked for. Without [P3967R0] there is still a need for a compiler switch to select unchecked or checked contract assertions though.

2 Proposal

This proposal gives objects of certain types the ability to act as contract assertions. Below is an example type `MyAssertionType` which is shows the general idea, a precondition assertion object and a function using it:

```
namespace myns {

struct MyAssertionType {
    // The existence of the assertion_object_kind data member is used to
    // determine that objects of this type are usable as contract assertions. This can be
    // detected using a concept or innately by the compiler.
    int assertion_object_kind;  // pre, post or assert.

    // These data members, if they exist, modify the core language behavior, reverting to
    // these P2900-compatible defaults if they don't.
    bool constify;          // Constify condition if true.
    bool ignorable;         // Contract assertions are ignored in unchecked builds.
    bool assumable;         // Don't assume predicate was true in following code.

    void operator()(string_view comment, exception_ptr ep, int,
                    source_location loc = source_location::current()) const {
        if (ep == nullptr)
            std::println(f"Assertion {comment} at {loc} failed.");
```

```
        else
            std::println(f"Exception evaluating assertion {comment} at {loc}");

        abort();
    }
};

inline constexpr MyAssertionType myPre {
    .assertion_object_kind = 1, // Values are the same as in the assertion_kind enum.
    .constify = false,          // I don't want constification in my contract assertions.
    .ignorable = false,         // These contracts can't be ignored.
    .assumable = true           // Assume the contract was not violated in succeeding code.
}

void myFun(int v) myPre(v > 0);

}
```

In this example the `myPre` assertion object specifies that no constification is to be performed, that the contract condition is always to be evaluated even if the build is unchecked and that the optimizer is free to assume that no violation occurred.

The following paragraphs describe the aspects of assertion object types in detail.

2.1 Identifying that a type is an assertion object type

A concept can be used to detect that a type (such as `myAssertionType` above) is a valid *assertion object type*. The concept checks for the existence of the `assertion_object_kind` member variable convertible to `int`. The concept could be extended to also require an operator(), but I think it would be better to just make it an name lookup error if this doesn't exist.

Tentatively it is suggested that this concept is to be defined in `<type_traits>` the standard library, but it could also be built in to the compiler and not have a visible name, the compiler just checks for `assertion_object_kind` in any way it wants.

2.2 Identifying assertions and looking up their assertion objects

When an identifier is found after the closing parenthesis of a function parameter list the compiler checks that the identifier names an *assertion object*. When an *assertion object* is found its `assertion_object_kind` member is used to see if the assertion is a pre- or postcondition. If it is neither a compiler error results.

Assertions of the *assert* kind are looked up using the regular name lookup when doing semantic analysis of an expression statement. If the top level operation of an expression statement is invoking the function call operator of an *assertion object* of *assert* kind a contract-assertion has been detected. If the kind is not *assert* a compiler error results.

An assertion object and its type must be complete to be usable as a contract-assertion even when used on a function declaration. This ensures that the information needed for caller side contract checking is available.

It is not possible to form contract-assertions from references to assertion objects. This ensures that the values of the data members are available at compile time. It could be possible to allow references to assertion object types if the compiler can find the referred to object but this would be novel and not very useful so this is not proposed.

If different assertion object definitions are found in different declarations or definitions of the same function there is a regular ODR violation. Within a translation the compiler can easily detect if contract-assertions on different declarations of the same function refer to different assertion objects.

If during compilation of different TUs different assertion object definitions are found when compiling the same function declaration or definition there is an ODR violation which is hard to detect, as usual. This situation is equivalent with compiling C++26 code with different evaluation semantics in different translation units: Sometimes the same `pre` contract is treated in one way, sometimes in another.

To be able to emulate C++26 contracts per translation semantic assertion object type and assertion object definitions must be placed in an anonymous namespace. This ensures that even if the implementation of `operator()` for assertion object types is different in different translations function definitions in each translation unit are guaranteed to call the `operator()` it has compiled even if it is not inlined, as the assertion object type's functions have internal linkage. If different per-translation evaluation semantic selection is emulated using `#ifdef`s in the assertion object type's definition the result is the same as with C++26 contracts.

2.3 Data members controlling core language behavior

Three boolean data members of an assertion object affect how the compiler generates the code for contracts using it. As these values must be known at compile time contract control objects must be constexpr variables. It is conceivable that similar data members could be added to control for instance inheritance of assertions on virtual functions in the future (with reasonable defaults).

The bool data member `constify` controls whether the compiler performs constification of the function parameters in the contract-assertion predicate.

The data member `ignorable` controls whether the compiler includes the contract check when the semantic is *ignore* or not. If `ignorable` is false and the semantic is *ignore* the contract check is implemented and if the contract predicate evaluates to false the `operator()` of the assertion object is called with the semantic parameter set to `evaluation_semantic::ignore` and it is up to the `operator()` to handle this situation.

The data member `assumable` controls if the compiler is allowed to optimize code following the contract-assertion assuming it didn't fail.

Note that while the assertion objects must be declared constexpr to make sure that the compiler can use these data member values at compile time `mutable` can be used to be able to modify other data members that a particular assertion object type may need. For instance it would be possible to count how many times contracts are violated in some kind of *observe-like* assertion object type. While similar tasks can be performed using a user defined contract violation handler in P2900 contracts this handler is always global while an assertion object is used only when referred from a contract-assertion.

2.4 The function call operator

The `operator()` can have different signatures starting with any combination of `comment`, `ep` and, `semantic` parameters, in this order. The comment is first to differentiate this parameter from any user defined message parameter provided as extra arguments at the contract-assertion site.

The `comment` parameter must have a type implicitly convertible from `const char*` or `const char8_t*` and if it exists it is typically called with a textual representation of the source code for the contract-assertion predicate encoded in the narrow execution character set or UTF-8. The comment parameter is optional to allow the compiler to avoid consuming the storage for comment strings if they are not used.

The `ep` parameter must be of type `exception_pointers` as this is a magic type that can't be reimplemented outside of the standard library. Any codebase that uses this parameter must use the relevant parts of the standard library anyway. When this parameter is present the compiler encloses the code that evaluates the contract-assertion predicate in a try block to be able to capture thrown exceptions into an `exception_pointer` and pass it to the operator() should an exception occur.

The `semantic` parameter must be of type int or an enum type. It is assumed that the enumerator values of the enum type represent the semantics the compiler can handle, which (according to C++26) are the four enumerator values of `evalutation_semantic`. The type of the `semantic` parameter is not required to be exactly `evaluation_semantic` to allow for codebases not using the standard library. This parameter is optional just

for consistency with the other parameters, but if for instance all contract-assertions *quick_enforce* it saves some bytes to not send the semantic to the operator. This parameter exists to be able to emulate C++26 contracts. If we don't get contracts in C++26 it can be replaced by a `bool checked` parameter which is only set for violations of non-ignorable contract-assertions in unchecked builds (or with [P3967R0] compiles).

The `operator()` can take additional parameters. These parameters are matched to an argument list formed from any additional arguments after the contract assertion predicate. A typical such parameter is a `string_view message` used to provide a user defined message to the assertion object, but other arguments could convey for instance a file handle to a log file for a particular contract or group of contracts.

```
struct handler_t {
    int kind = 1;
    void operator()(const char* comment, evaluation_semantic semantic); // #1
    void operator()(evaluation_semantic semantic, string_view message); // #2
};

constexpr handler_t pre_m;

void f(int x) pre_m(x > 0), pre_m(x < 10, f"x must be less than 10 but is {x}"); // P3412
```

Here the first precondition of f calls #1 if x <= 0 and the second precondition calls #2 if x >= 10.

TODO: This doesn't work very well if the user provided argument is an `int` or an `exception_pointer`. It should not be possible to provide values for the three parameters listed above as additional arguments in a contract-assertion. One way to accomplish this is to make the `semantic` parameter mandatory and the last of the three predefined parameters. A special tag type is also a possiblity but this reinstates a dependency from the compiler to the standard library unless this tag type is just identified by its name. In either case there is a slight inefficiency compared to not having to send any semantic parameter value (or dummy tag type value) as no types can have zero size.

2.4.1 The value of the semantic argument

When the code for a contract assertion is generated a value for the `semantic` parameter of the function call must be found by the compiler. To keep backwards compatibility to C++26 this must be specified to happen in an implementation defined way. While many compilers will probably just have a compiler switch to select between the four semantics it is also possible to for instance provide a list of contracts to observe in a text file when compiling, by the means that was implemented for C++26 contracts in that particular compiler. It is even allowed to evaluate contracts randomly, but in C++26 any such possibility must be hardcoded into the compiler and that same logic can be applied with this proposal.

As it is the assertion object's responsibility to terminate or not this allows user defined *assertion object types* to for instance implement the "observe if you would otherwise enforce" semantic for newly added assertions by never terminating regardless of the value of the `semantic` parameter. A subsequent source code change is needed to turn such contract assertions into regular contract assertions just like with the labels of P3400.

Note: The value of the `semantic` parameter can only be *ignore* if the `ignorable` data member of the assertion object exists and is false. This indicates that the implementation specific semantic for the assertion was *ignore* but the predicate was evaluated anyway as `ignorable` was false for the assertion object.

If backwards compatibility is not needed due to contracts being retracted from the official C++26 standard the semantic argument may be replaced by a `bool checked` parameter. This is as there is no need for contract specific compiler options except for a checked/unchecked build mode flag. All other configuration such as enforce/quick-enforce can be done using the -D switch defining a macro that controls the semantics of the `operator()` when it comes to calling a handler etc. With [P3967R0] not even this compiler switch is strictly necessary.

3 Reimplementing C++26 contract assertion types

With this proposal the built in functionality of C++26 contracts can be reimplemented as standard library *assertion objects* without changing semantics.

Here is an *assertion object type* that can emulate C++26 contracts using this proposal:

```
namespace std::contracts {

struct assertion_object_t {
    assertion_kind assertion_object_kind;

    bool constify;
    bool ignorable;
    bool assumable;

    void operator()(string_view comment, evaluation_semantic semantic, exception_ptr ep,
                    source_location loc = source_location::current()) {
        if (semantic != evaluation_semantic::quick_enforce) {
            contract_violation violation(comment, ep, kind, loc, semantic);
            handle_contract_violation(violation);
        }

        if (semantic != evaluation_semantic::observe)
            abort();
    }
};

}
```

3.1 Standard assertion objects

The standard assertion objects are just instances of the `std::contracts::assertion_object_t` type in the global namespace, declared in the `<contracts>` header. To provide backwards compatibility to C++26 contracts the objects must be in the global namespace. This is also convenient to avoid repeating `std::contracts::` for each contract assertion.

```
// In the global namespace
inline constexpr std::contracts::assertion_object_t pre {
    .kind = std::contracts::assertion_kind::pre,
    .constify = true,
    .ignorable = true,
    .assumable = false
};

inline constexpr std::contracts::assertion_object_t post {
    .kind = std::contracts::assertion_kind::post,
    .constify = true,
    .ignorable = true,
    .assumable = false
};

inline constexpr std::contracts::assertion_object_t contract_assert {
    .kind = std::contracts::assertion_kind::assert,
    .constify = true,
    .ignorable = true,
```

```
    .assumable = false
};
```

4 Discussions and consequences

Here some consequences and possibilities of handling contracts by assertion objects rather than built in specifiers and keywords are discussed.

4.1 Name lookup rule for assertion objects

Assertion objects are looked up using regular name lookup rules as seen from the function declaration or definition where they are used. It would also be possible to add a rule that in name lookup after function declarations *assertion object*s are not hidden by other entities of the same name.

Also: Vice versa: *assertion object*s of the *pre* and *post* kind could be made invisible when looking up names for other purposes: This has the drawback that if for instance a violation counter that is incremented for each violation is needed it can't be member of the assertion object type as it could never be accessed when reporting the statistics, well, unless this is done by the destructor which runs when the program exists, but that's very limiting.

Using regular name lookup may look like a drawback at first glance but it has the advantage that contracts in a certain namespace can get handled differently than contracts in another namespace even if they have the same name. This allows library vendors to ensure that contract violations in their code are handled according to their specification and not subjected to a global contract violation handler that can subvert the semantics that the library depends on. Note that this includes any use of this library even if the contracts are on inline functions as name lookup is performed as seen from the declaration of the function. It is also possible to ensure specific contract handling in a class hierarchy by defining the assertion objects as static members of the base class. Using a class just providing assertion objects as a mixin may turn out to be a very convenient way to opt in to specific contract violation handling semantics. This is essentially the same rules as for [P3400R2] label lookup.

4.2 Customization of standard assertion objects

It is debatable if it is a drawback or advantage to be able to customize the standard assertion objects. This proposal takes the stance that predictable behavior is more important than customization possibility. This means that as long as you use the standard library you have the standard behavior of the three standardized assertion objects. Customized assertion objects can however be introduced in namespace scope, even if named `pre`, `post`, or `contract_assert`. This poses the same risk as P3400 labels in namespace scope: You could always declare a label called `enforce` in some namespace that doesn't actually enforce.

To allow customization of the standard assertion objects but still allow them to create a `contract_violation` object the `<contracts>` header would have to be subdivided into one header with the current contents and one header containing the contract control objects. This can be done by just adding a `<contract_objects>` file (which includes the current `<contracts>` header) but it seems more logical to put the assertion objects in `<contracts>` and relegate its current contents to a subordinate `<contract_violation>` header. This would be problematic with `import std;` and is not proposed.

4.3 No core language dependency on `<contracts>`

With this proposal the compiler knows only about the names of the data members in a assertion object that controls the semantics of the contracts. This includes the `assertion_object_kind` member which is converted to `int` by the compiler before use. By this definition the dependency from the compiler to the standard library header `<contracts>` is removed and contracts can be used without using the standard library. The actual integer constants selected are compatible with the enumerator values specified for the `std::contracts::assertion_kind` enum:

```
// From N5032:
enum class assertion_kind : unspecified {
    pre = 1,
    post = 2,
    assert = 3
};
```

The other data members that the compiler looks for: `constify`, `ignorable` and `assumable` are converted to bool before use by the compiler and if they don't exist a default value is used.

The `semantic` parameter of the `operator()` is defined to have an int or enum type, assuming the compiler can convert its internal integer value to any enum type. This is to avoid depending on the definition of the `evaluation_semantic` enum but in principle any enum would match, which allows an alternative library to define some other enum with matching enumerator values.

```
// From N5032:
enum class evaluation_semantic : unspecified {
    ignore = 1,
    observe = 2,
    enforce = 3,
    quick_enforce = 4
};
```

The only unavoidable dependencies on the standard library are for the magic types `source_location` and `exception_pointers` but using these types is optional for user defined assertion object types.

Note that this is in contrast from C++26 contracts where *everything* in `<contracts>` is used directly by the compiler, except `invoke_default_contract_violation_handler`.

4.3.1 Changes to the contract_violation type

To make backwards compatibility possible a constructor must be provided for the contract_violation class, removing the magical construction by the compiler of C++26. It is possible but not necessary to provide a constructor which takes the comment in UTF-8 and a getter returning it.

4.4 Additional standard assertion objects

In this first revision no additional standard assertion objects are proposed. However, it would be easy to define a number of such objects, the hard part is to figure out which ones are important enough to be standardized and how they should be named. Using a variable template it would even be possible to build the label system of P3400 on top of this proposal, but as the reason for this proposal is partly to get rid of the complexities of P3400 this is not proposed.

4.5 Static analyzer requirements

With this proposal based on a concept identifying an object as being a assertion object static analyzers would need to be able to detect such objects too. This is especially true for contract assertions when the syntax used looks just like a function call. The static analyzer, if it is to try to detect compile time detectable contract violations, has to do the same checking for a assertion object as the compiler does. This should be a very simple task compared to the complex semantic analysis that static analyzers regularly perform today.

4.6 Forwarding the source_location

The `std::assertion_object_t` captures the location of the failed contract using a defaulted source_location parameter. This does not allow contract checks to "spoof" the source location by explicitly providing an argument value for the parameter as the only way to construct a `source_location` is using the

`source_location::current` function. However this offers the benefit of being able to record the caller side source location even for callee checked contracts, but this possibility should not be over-used as it forces the construction and copying of `source_location` objects even if the contract doesn't fail.

Here is an example of how this is done, given the `assertion_object_t` type shown above.

```
void myFunction(int a, source_location loc = source_location::current()) pre(a > 0, loc);

void caller()
{
    myFunction(-1);
}
```

When the call to myFunction fails the source location created at the call site of myFunction in caller is recorded by the violation handler giving more information than the myFunction declaration.

4.7 Caller and callee side checking control

Additional data members could be added to allow customizing whether a contract is to be checked on the caller side, the callee side or both. This can be implemented with two bool variables but as this would allow both to be false, which is not really useful. This suggests an enum with values *caller, callee*, or *both*. This controllability is questionable: for non-inline functions we can't be sure that the values seen when compiling the TU containing the function definition is the same as the values seen when compiling some calling TU, which allows checks to be omitted in checked compiles.

With dual compiles according to [P3967] the callee is known to check all contracts which allows the compiler to omit caller side checks. Not specifying where contracts are checked gives the compiler much more leeway for optimizing the checking without having to come up with exception rules to explain why contracts may not get checked caller side even if the assertion object says so.

Thus this proposal does not include such caller and callee side checking control.

5 Specification alternatives

There are a number of changes that could be made to the specification that assertion object types have to adhere to.

— One possible change is to somehow tell the `operator()` that the function the contract is applied to is `noexcept`. However, it is unclear what good this would do as any exception thrown during condition evaluation or by a called handler would cause termination anyway. However an assertion object type could include a try block around the call to `handle_contract_violation` and report that it threw before termination.

— It would be possible to require the compiler to create a `contract_violation` object before calling the `operator()` of the assertion object but as some contract handlers may not use all of this information it seems better to provide a constructor to the `contract_violation` class that contract handlers can use. A assertion object type can also decide not to provide the source location information or qualify including it with some compiler define to be able to tune debuggability versus code size.

— It would be possible to require that the `operator()` of all assertion object types must take a comment and exception pointer as the first two parameters and let the compiler optimize away the comment string if it is not used and the try block if the exception pointer is immediately rethrown using `rethrow_exception`. This removes the special handling in the compiler to check which of the two parameter values are needed, by attempting overload resolution before compiling the contract checking code. This however requires that the `operator()` is inline and that the compiler actually is able to detect the optimization possibilities.

— To avoid these peculiar overload resolution attempts bool data members `catch_exceptions` and `include_comment` could be added but this results in new ways in which a assertion object type could

be erroneous, that is, if the signature of `operator()` does not match the setting of these variables. It would be possible to always require that the two first parameters are comment and exception_pointer but provide empty values if the bool data members are not true. Maybe this is the cleanest specification although it is a bit wasteful in that those dummy values must be provided to the `operator()` which knows that it can't use them. This takes a few extra code bytes for each contract-assertion.

— A possibility with a templated operator() taking the contract-assertion predicate that it could evaluate inside or not inside a try block to implement the `catch_exception` option was rejected as it would add a lot to compile times if each contract would involve synthesizing a lambda type and then specializing the `operator()` for it.

— A possibility to use *functions* instead of objects with function call operators was discarded as it would be hard to associate values for the contract control variables based on a function name. This is mainly that without [P3312R1] variable templates can't be be used as the instances are associated with the type of the function rather than the function itself. One idea which has not yet been explored is to retrieve the contract control variables from the return type of such a function, taking lead from co-routine return types.

5.0.1 A more P3400-like syntax

An alternative specification would be that the three names used for C++26 contracts are retained and named assertion objects are treated just like the labels of P3400. This has some advantages and still offers more flexibility with fewer novel ideas than P3400. The major reason this is not the promoted solution in this proposal is that the syntax is clumsier and invites defining macros to get back to the syntax the rest of this proposal. One obvious advantage of this alternative, if C++26 contracts is a starting point, is that you don't have to include `<contracts>` to use assertions as the three template functions are somehow built in to the compiler. Another possible advantage is that the three assertion kinds `pre`, `post` and `contract_assert` have known meanings when used with the default template parameter value. However, this meaning is so vaguely defined in C++26 that many codebases seem to not want to use C++26 contracts. With the promoted solution you can hardwire the meaning of one of these names or use other names for your assertion objects which provides an easier way to ensure that assertions do what you think they do.

A P3400-like syntax can be accomplished by actually defining the contract assertions as template functions, albeit with a magic touch. Here is a sketch of how the `pre` function could be defined. Note that this is not source code that is actually in a library, the compiler has it hidden somewhere or just generates code for it anyway:

```
template<auto Label = std_label, typename... Ts> void pre(function_ref<bool()> predicate, Ts&& ts)
{
    if constexpr (Label.ignorable && current_semantric() == evaluation_semantic::ignore)
        return;

    if constexpr (is_invocable_v<Label, exception_pointer, int, Ts...) {
        try {
            if (!predicate())
                Label(nullptr, current_semantric(), std::forward<Ts>(ts)...);
        }
        catch(...) {
            Label(current_exception(), current_semantric(), std::forward<Ts>(ts)...);
        }
    }
    else {
        if (!predicate())
            Label(current_semantric(), std::forward<Ts>(ts)...);
    }
}
```

While there is not necessarily anything magic about this function itself the data member `constification` of the Label object control the code generation for the callable shown as the `predicate` parameter here and the `assumable` data member controls if the compiler emits code under the assumption that the predicate is true.

Note: There are also additional if constexpr clauses not shown above. These are used to handle the other possible signatures of the `operator()`, the order of which control the priority. These were excluded for brevity.

The `current_semantic` function used above returns the semantic that the implementation specific mechanism gave the C++26 contracts. There will also have to be a `current_comment` function that returns the source code of the contract assertion predicate, possibly with a `current_comment_u8` variant. This requires the above pre function to always be inlined.

The compiler is free to accomplish this semantic in any which way it may, under the as-if rule.

6 Backwards compatibility to C++26 contracts

The main (only?) backwards compatibility issue is that the `<contracts>` header must be included to get the standard assertion object declarations. In C++26 contracts can be used without including any header, which is logical as the names of the contract assertion kinds are built in. Adding an include of this header is a refactoring that is somewhat tedious but not hard to do or error prone, and failing to do it causes compile time errors. A codebase can also select to change the build system to pre-include the contracts header to avoid excessive code changes. Most codebases have central headers that are included by all or most TUs so the work should be limited. If `import std;` is used there is no backwards compatibility issue.

If name lookup for assertion objects is not modified for pre- and postconditions the identifiers `pre` and `post` defined in the global namespace by `<contracts>` can be hidden by entities in intermediate namespaces that are not *assertion objects*. This causes a risk of backwards compatibility issues: If a program declares the name `pre` or `post` in a namespace also containing contract checks the contract checks fail to compile as name lookup for the contracts will now find the user defined entity instead of the assertion objects. The risk that such a name lookup result will not result in a compile error is very small as the entity found when looking up the name must be an object whose type has a data member `assertion_object_kind` of type convertible to `int` and a suitable `operator()`. Note that with C++26 contracts `contract_assert` is a keyword so no declarations of entities of this name can exist in C++26 code. This type of compiler error can be fixed by prefixing each `pre` or `post` contract assertion with `::` to explicitly select the assertion object or to rename the clashing entity.

Another risk is of course that a codebase declares entities called `pre` or `post` in the global namespace and also includes `<contracts>`. This will result in a compiler error with this proposal as global variables don't overload with any other entities. In this case the only solution is to rename the clashing entity or to move it into a namespace where contracts are not used.

Apart from these naming related issues there could be subtle semantic differences from C++26 in this proposal. If so this is unintended and can hopefully be corrected either by modifying the `assertion_object_t` implementation sketched above or by modifying the semantics of the remaining built in compiler functionality.

7 Reasons to not standardize C++26 contracts first

There are some reasons to not standardize P2900 for C++26 if this proposal is a future direction. One reason documented above is the possible code breakage, although minimal. The requirement to include `<contracts>` to be able to use contracts can be implemented easily as a warning or error by compilers as soon as this proposal is accepted to give users more time to add the include of the header.

The replaceable handler and all of the contents of the `<contracts>` header becomes more of a convenience than a necessity with this proposal but something similar would be needed anyway to avoid having to define new *assertion objects* and violation handling infrastructure in each codebase. There is nothing wrong with the `contract_violation` type or the replaceable handler that would benefit from not being standardized in C++26.

I think that the most compelling reason for removing P2900 from C++26 is its dependency from the core compiler to the contents of <contracts> which is novel and something we usually want to avoid, and which is avoided with this proposal. Another rather big advantage is that compilers need not implement (and then retain even with this proposal) compiler switches or other means to select an evaluation semantic for each TU. If functionality similar to C++26 contracts is desired this can be accomplished with the -D switch and some ifdefs in the <contracts> header.

8 References

[P3312R1] Bengt Gustafsson. 2025-04-16. Overload Set Types.
https://wg21.link/p3312r1

[P3400R2] Joshua Berne. 2025-12-14. Controlling Contract-Assertion Properties.
https://wg21.link/p3400r2

[P3909R0] Ville Voutilainen. 2025-11-03. Contracts should go into a White Paper - even at this late point.
https://wg21.link/p3909r0