Controlling Contract-Assertion Properties

Document #: P3400R2
Date: 2025-12-14
Project: Programming Language C++
Audience: SG21 (Contracts)
Reply-to: Joshua Berne <jberne4@bloomberg.net>

Abstract

The Contracts facility in C++26, adopted from [P2900R14], provides significant flexibility for tool vendors to enable control over contract-assertion behavior, along with defaults that enable powerful and principled use of contract assertions in a wide variety of contexts. Many other important use cases, however, require additional *in source* control over that flexibility. Yet other advanced features will be needed to enable wider adoption of Contracts in increasingly diverse applications and environments. To facilitate these controls, we present here a design for specifying the evaluation behavior of contract assertions in the program source using compile-time C++ objects, achieving flexibility, expressivity, and extensibility that is far more comprehensive, cohesive, and maintainable than any bespoke solution to each individual use case might offer.

Contents

1 Introduction **3**
1.1 Semantic Selection 8

2 Proposal **10**
2.1 Selection 12
2.1.1 Name Lookup For Control Expressions 12
2.1.2 Control-Expression Name Lookup Outside Assertions 14
2.1.3 Combining Assertion-Control Objects 15
2.2 Facets 16
2.2.1 Restricting Evaluation Semantics 17
2.2.2 Choosing Evaluation Semantics 19
2.2.3 Controlling Combination 22
2.2.4 Local Violation Handling 23
2.2.5 Grouping Assertions 25
2.2.6 Comment Manipulation 27
2.2.7 Integration with `contract_violation` 29
2.3 Key Considerations 34
2.3.1 Always-Checked Assertions 34
2.3.2 Never-Checked Assertions 36
2.3.3 ODR Applicability 37
2.3.4 Standard Library Labels 37

2.3.5 Header Dependency 39

3 Future Directions **39**
3.1 Runtime Selection of Semantics 39
3.2 Ambient Control Objects 39
3.3 Core-Language Control Labels 40
3.4 Build Environment 42

4 Alternate Design Considerations **42**
4.1 Syntax 42
4.2 Combination 44
4.3 Predicate Evaluation Behavior 44

5 Wording **45**

6 Conclusion **46**

A Example Implementation **46**
A.1 `std::contracts::labels::operator|` 46

Revision History

Revision 2 (2025-12 Mailing)

- Restructured introduction
- Fixed specification for handler access to the control object (through queries)
- Separated baseline facets from future work
- Clarifications on alternative design directions
- Addressed issues that arise with contract assertions that do not allow unchecked semantics

Revision 1 (For 2025-02 Hagenberg Meeting)

- Added group names to identification labels
- Explicit examples for attaching control objects to implicit contract assertions

Revision 0

- Original version of the paper

1 Introduction

C++26 contract assertions — `pre`, `post`, and `contract_assert` — provide a standard framework to express that particular conditions, expressed as boolean predicates, are expected to be `true` at specific program points and, more importantly, that a program is *incorrect* if any of those predicates fail to produce a value of `true` at those program points.

With C++26 Contracts, the behavior of the program when a contract assertion is evaluated is controlled entirely in an implementation-defined manner — generally through build configurations. In this paper, we introduce *assertion-control objects* comprising individual objects known as *labels*. These assertion-control objects allow local customization of the behavior when individual contract assertions are evaluated.

Assertion-control objects can be used to help customize a wide range of different aspects of the contract-assertion evaluation process. With this proposal, the Standard Library will support many of the following use cases.

- The header `<contracts>` defines many objects that can be used as components of an assertion-control object, which we call labels:

  ```
  #include <contracts>  // needed, but we won't repeat this directive in each example
  ```

- Labels are objects in the `std::contracts::labels` namespace that satisfy various concepts, such as the empty label that does nothing other than being a valid label:

  ```
  constexpr auto empty_label = std::contracts::labels::empty_label;
  static_assert(
    std::contracts::labels::assertion_control_object<decltype(empty_label)> );
  ```

- The assertion-control object is the value of the assertion-control expression that appears in angle brackets (`<>`) after the introducer (`pre`, `post`, or `contract_assert`) of a contract assertion (which, due to having included `<contracts>`, will find names in `std::contracts::labels` without needing to spell out the entire namespace):

  ```
  void f()
    pre<empty_label>( true )
    post<empty_label>( true )
  {
    contract_assert<empty_label>( true );
  }
  ```

- Labels let us control various aspects of the evaluation of a contract assertion, such as requiring that a particular assertion always uses a terminating (i.e., *quick-enforce* or *enforce*) contract evaluation semantic:

  ```
  void f(int *p)
    pre<terminating>(p != nullptr);
  ```

  Here, the identifier `terminating` is not a special new keyword; it resolves to a `constexpr` variable made available by the `<contracts>` header.[1]

- We can also use labels to identify that a contract assertion is part of a particular group that our compiler will allow us to control through our build configuration:

  ```
  void f(int *p)
    pre<"mylib::general"group>(p != nullptr);
  ```

  In this case, we are using a user-defined literal operator provided by the `<contracts>` header to generate a label indicating membership in the named group.

- Because labels are constant expressions, we can also capture their values and give different names to labels using all the normal facilities of C++. To make that easier, we can also make use of the `contract_control` function-like operator to get the same name lookup we would have in an assertion-control expression:

  ```
  namespace mylib {
  constexpr auto general_group = contract_control( "mylib::general"group );
  void f(int *p)
    pre<mylib::general_group>(p != nullptr);
  }
  ```

- Quick checks and expensive checks can be marked so that they can be configured based on the cost we are willing to pay for checking in any given build:

  ```
  int* binary_search(int *begin, int *end, int val)
    pre<opt>(nullptr != begin && nullptr != end)  // fast check, almost always on
    pre<audit>(std::is_sorted(begin,end));        // slow check, breaks complexity
  ```

[1]No names were permanently set during the design of this paper, and all additions to the Standard Library will have their naming choices duly considered once the basic low-level feature's scope is reasonably settled.

- Mark a newly introduced contract assertion so that it will be *observed* instead of *enforced* or *quick-enforced*, allowing it to be more easily deployed to production environments:

  ```
  double sqrt(double x)
    pre<review>(x >= 0)  // newly added
    pre(x > 0);          // old precondition
  ```

  Note that such a label won't force the contract to be checked at all; that is still controlled by the build configuration, using the same options that control the second precondition assertion with no label. What `review` does is change the semantic to *observe* when a different checking semantic would have been chosen.

- Use the overloaded pipe (`|`) operator provided by the Standard Library to combine the review label with other labels:

  ```
  void f(int *begin, int *end)
    pre<"mylib::general"group | review>( nullptr != begin && nullptr != end)
      // configured as part of the "mylib::general" group
      // observed instead of enforced or quick-enforced
    pre<audit | review>( std::is_sorted(begin,end) );
      // Slow check is on in only some builds and is observed in those builds.
  ```

- Configure assertions to use a terminating semantic when a hardened Standard Library has also been selected (including when implementing the Standard Library):

  ```
  namespace std {
  template <typename T>
  T* optional<T>::operator->() const
    pre<hardened>(has_value());  // checked if Standard Library is hardened
  }
  ```

  Depending on whether the Standard Library implementation supports a nonhardened build, this check might also be unchecked in some builds, but when a user selects a hardened library when their code is built, they will get corresponding checking of this function.

- Use symbolic predicates that do not have a runtime implementation, which will never be checked at run time but might be checked by static analysis that understands the predicate:

  ```
  int* binary_search(int *begin, int *end, int val)
    pre<symbolic>(std::is_reachable(begin,end));
      // no checked semantic allowed
  ```

Now one might ask how all of this works and how much core-language or Standard Library overhead will be involved in adding this kind of functionality to C++26 Contracts. The answer is fairly simple.

- After the introducer of a contract assertion (`pre`, `post`, or `contract_assert`), we may now add an expression, known as the assertion-control expression, enclosed in angle brackets (`<>`).
- Assertion-control expressions will be evaluated at compile time to produce a `constexpr` assertion-control object.
- Assertion-control expressions will generally be made up of objects we call *labels* that are in namespaces that are searched within the assertion-control expressions.

- Labels can be combined when needed using the pipe operator (`operator|`) provided in the `<contracts>` header.
- Label objects can participate in many different aspects of the assertion-evaluation process by satisfying various concepts we call *facets*.
- For each facet, if the assertion-control object satisfies the concept, we use that object appropriately, and if it does not, we maintain the current behavior.
- Assertion-control objects do not need to participate in all possible facets, allowing us to freely introduce new facets without breaking existing users.
- The implementation of `operator|` knows how to use labels that model some or all facets to produce a combined label that models the same set of facets or to reject incompatible combinations.

For example, the `review` label is a `constexpr` object defined in the `<contracts>` header:

```
namespace std::contracts::labels {
struct review_t {
  // Assertion-control object are identified by having this nested name.
  using assertion_control_object = review_t;

  // To alter the evaluation semantic used, provide a compute_semantic member function.
  // By having this function, review_t satisfies the semantic_computation_label concept.
  constexpr evaluation_semantic compute_semantic(evaluation_semantic in) const
  {
    switch (in) {
      case evaluation_semantic::enforce:
      case evaluation_semantic::quick_enforce:
        return evaluation_semantic::observe;  // observe if we would have terminated
      default:
        return in;  // In all other cases, don't change the semantic.
    }
  }
};
constexpr review_t review{};
}
```

When we use `review` in an assertion-control expression after we have included the `<contracts>` header, the `constexpr` variable above will be found because that header contains a directive (also part of this proposal) to search this namespace[2]:

```
using contract_control namespace std::contracts::labels;
```

Again, because of that using directive, we can combine the `review` label with any other label using `operator|`, which returns a combined assertion-control object that has the behavior of both of its operands. (How exactly it combines that behavior is based on the particular facet and objects, which we will describe below. Users are also free to define their own operators or to use other functions to combine labels with any other behaviors they might desire.)

[2]The `contract_control` keyword is used to identify using directives that apply to assertion-control expressions as well as for the operator that allows getting the same behavior outside of assertion-control expressions (see Section 2.1.2).

In other cases, we don't necessarily want to dictate the choice of semantic, but instead we want to limit the possible choices, even when combining our label with different labels. To do that, we can provide an `allowed_semantics` member to limit the possible choices of semantics. For example, to implement a label that must always use a terminating semantic (and fail to compile if combined with another label that makes the semantic nonterminating), we can make a label type that has a specific set of allowed semantics:

```
namespace std::contracts::labels {
template <evaluation_semantic... allowed>
struct allowed_semantics_t{
  using assertion_control_object = allowed_semantics_t<allowed...>;
  static constexpr evaluation_semantic_set allowed_semantics = {allowed...}
};
allowed_semantics_t<evaluation_semantic::quick_enforce,
                    evaluation_semantic::enforce>      terminating;
```

Standard Libraries looking to implement hardening with contract assertions can also do so entirely using labels. For example, libc++ could base the behavior of its label on the same configuration macros that currently control library hardening:

```
#include <__configuration/hardening.h>

namespace std::contracts::labels {
struct hardened_t {
  // Every assertion-control object must have this nested name.
  using assertion_control_object = hardened_t;

  // To alter the semantic used, we provide a compute_semantic member function.
  constexpr evaluation_semantic compute_semantic(evaluation_semantic in) const
  {
#if   _LIBCPP_ASSERTION_SEMANTIC == _LIBCPP_ASSERTION_SEMANTIC_IGNORE
    return evaluation_semantic::ignore;   // not a hardened implementation
#elif _LIBCPP_ASSERTION_SEMANTIC == _LIBCPP_ASSERTION_SEMANTIC_OBSERVE
    return evaluation_semantic::observe;  // not a hardened implementation
#elif _LIBCPP_ASSERTION_SEMANTIC == _LIBCPP_ASSERTION_SEMANTIC_QUICK_ENFORCE
    return evaluation_semantic::quick_enforce;
#elif _LIBCPP_ASSERTION_SEMANTIC == _LIBCPP_ASSERTION_SEMANTIC_ENFORCE
    return evaluation_semantic::enforce;
#else
    std::unreachable();  // will #error in <hardening.h> before this happens
#endif
  }
};
constexpr hardened_t hardened{};
}
```

Other Standard Library implementations might choose to have the behavior of their hardening macro based on different preprocessor macros or based on implementation-provided intrinsics that expose the choice of hardening mode.

With the full power of labels, we provide mechanisms to control many of the key facets of contract-assertion evaluation, which are described in detail in the following sections. Once this feature is available, C++ will have a Contracts facility that satisfies the wide range of deployment needs that a programming language used by millions — to write software that is used by billions — demands.

1.1 Semantic Selection

The goal of this proposal is not to replace the implementation-defined behavior that currently determines the semantic chosen for the evaluation of a contract assertion. That implementation-defined behavior is what captures the build flags and wide variety of options that compilers will provide users to leverage the contract assertions they have put into their code in a vast variety of fashions.

The purpose of labels is to augment that process with information that *is* known by the author of the source code so that build-time decisions can be made more generally and reliably. With C++26 Contracts, for example, *hardening* some contract assertions but not others would require maintaining supplementary configuration files that are passed with every build to dictate the semantics of specific contract assertions. With this proposal, however, a label (`hardened`) can be included in the source code to restrict certain contract assertions to checked semantics.

Different facets of the assertion-control object, however, are involved in the full determination of the evaluation semantic in different ways.

1. Before anything can be done with a contract assertion, the assertion-control expression must be evaluated, and an assertion-control object must be available.
2. If the assertion-control object is an `allowed_semantic_label` (see Section 2.2.1), then its `allowed_semantics` member will be used as input into the configuration handling.
3. If the assertion-control object is an `identification_label` (see Section 2.2.5), then its `group_names` member will be used as input into the configuration handling.
4. The compiler's configuration will be used to determine the initial semantic for the contract assertion. This happens in an implementation-defined manner that may include a wide variety of inputs.
    - User configuration can be provided to the compiler in the form of various compilation flags and, possibly, configuration files.
    - Some contract-assertion evaluations might be restricted in specific manners based on other compilation flags or the nature of the contract assertion. For example, the evaluation of postconditions within the caller's translation unit on platforms where function parameters are destroyed by the callee might be restricted to the *ignore* semantic.
    - The kind, source location, and module of the contract assertion might be referenced to guide the configuration of the semantic.
    - The source location and module of the caller might be referenced to guide the configuration of the semantic for function contract assertions or even for assertion statements within an inline function that is being inlined into a caller.
    - The group names and allowed semantics from the assertion-control object might be used to guide that configuration as well.

   - If the chosen semantic is not an allowed semantic (based on the assertion-control object), then the semantic will be adjusted (in some possibly configurable implementation-defined way) to be in the allowed set.
   - By delaying the full computation until after compile time, the build configuration might instruct the compiler to emit a link-time or runtime lookup of the initial semantic. In cases where this happens, later steps might be evaluated (at compile time) for *all* evaluation semantics for which code is being generated to determine the final mapping from platform-selected semantic to effective semantic.

5. If the assertion-control object is a `semantic_computation_label` (see Section 2.2.2), then the initial semantic will be passed to the `compute_semantic` member function of the assertion-control object, and its result will be the effective semantic; otherwise (as is the case for all contract assertions in C++26), the effective semantic is the initial semantic.
6. If the assertion-control object is an `allowed_semantic_label` (see Section 2.2.1), then the effective semantic will be verified to be in the `allowed_semantics` member of that object. If it is not in the `allowed_semantics` member of that object, then the program will not compile.

After the above steps are completed for an evaluation of a contract assertion, which will generally happen at compile time, the contract assertion will be evaluated with the effective semantic.

- If the semantic is *ignore*, nothing happens and control continues.
- If it is a checked semantic, the predicate is evaluated to see if a violation is detected.
- If a violation was detected and the effective semantic is *enforce* or *observe*, the violation handler will be invoked.
  1. A `contract_violation` object will be populated.
  2. If the assertion-control object is a `compute_comment_label` or a `compute_message_label` (see Section 2.2.6), the `comment` and `message` fields of the assertion-control object will be passed through the corresponding function (at compile time) to produce the value that will be placed into the `contract_violation` object.
  3. If the assertion-control object is a `local_violation_label`, then the violation object will be passed to the `handle_contract_violation` member of the assertion-control object. If that function returns `violation_handled::handled` normally, then the violation handling process is complete (skipping the global violation handler).
  4. Then, if violation handling is not yet complete, the global replaceable contract-violation handler (`::handle_contract_violation`) is invoked with the violation object.
- If a violation was detected and the effective semantic was *enforce* or *quick-enforce*, the program is then contract terminated.

Most of the above steps are part of the contract-assertion evaluation process in C++26, augmented with places where an assertion-control object that satisfies specific concepts can guide the behavior differently.

2 Proposal

For every contract assertion, we provide a mechanism to specify an *assertion-control object* that acts as a parameter for the assertion specification to control how that contract assertion will behave when evaluated. To specify these objects, we will leverage the well-known C++ syntax of using angle brackets (`<` and `>`) to surround the single expression that determines the value of the assertion-control object.

For `pre`, `post`, and `contract_assert`, we can easily add this syntax to the grammar that defines these constructs:

function-contract-specifier :
  precondition-specifier
  postcondition-specifier

precondition-specifier :
  `pre` *assertion-control-specifier*$_{opt}$ *attribute-specifier-seq*$_{opt}$ `(` *conditional-expression* `)`

postcondition-specifier :
  `post` *assertion-control-specifier*$_{opt}$ *attribute-specifier-seq*$_{opt}$ `(` *result-name-introducer*$_{opt}$ *conditional-expression* `)`

assertion-statement :
  `contract_assert` *assertion-control-specifier*$_{opt}$ *attribute-specifier-seq*$_{opt}$ `(` *conditional-expression* `)` `;`

assertion-control-specifier :
  `<` *constant-expression* `>`

The specified *constant-expression* is a manifestly constant-evaluated expression whose type must be a class type that is suitable to use as a `constexpr` variable and that must satisfy the `assertion_control_object` concept:

```
namespace std::contracts::labels {
template <class T>
concept assertion_control_object =
  requires ( typename T::assertion_control_object; );
}
```

Note the following caveats.

- We want to limit the types for assertion-control expressions to those that opt in so that we do not allow nonsensical assertion-control expressions that do nothing, such as `pre<5>` or `pre<true>`. Therefore, we require that the member `typedef` to identify such types.
- Special meaning might be given in the future to particular scalars (such as values of `std::contracts::evaluation_semantic`), but they will need specific core-language support that is not proposed in this paper. Requiring the member to `typedef` gives us more freedom to provide bespoke meanings later to expressions that will just be ill-formed with this proposal.
- Throughout this proposal, we will specify core-language behavior in terms of satisfying a particular concept whenever possible because doing so minimizes any direct dependency on the Standard Library from the core language. The core language will describe the concepts it will

use to identify a facet, but it does not need to name or directly use the concept defined in the Standard Library. Experience with implementing C++26 Contracts has clarified that this kind of independence is important since, within a single program, it is necessary to support different TUs and the violation handler being built with a different Standard Library implementation.

- Requiring that a member name be present to identify assertion-control objects can make some labels slightly more verbose than needed but adds the benefit of not introducing unintended interactions with existing or unrelated objects. This avoidance is particularly helpful due to the otherwise unconstrained `operator|` that we will be introducing in the `std::contracts::labels` namespace.

For every contract assertion, an assertion-control object will be created that is initialized by the *constant-expression* in the assertion-control specifier.

- To avoid requiring that these objects get names that are mangled, we leave unspecified whether the same object is used for all instances of the same contract assertion, or even for different contract assertions. An intentional goal of our design is to be able to minimize the impact on the size of generated code and static storage when using labels. This property will become observable when we expose the assertion-control objects to the contract-violation handler later (in Section 2.2.7) for the cases in which we want to use an assertion-control object to maintain state.

- Naturally, if the assertion-control expression is not a constant expression or if it does not have literal type, the program is ill-formed, and this error is not subject to SFINAE (just as the rest of a function contract specifier is not subject to SFINAE, preventing the presence of contract assertions from altering overload-resolution behavior and allowing the delay of instantiation of contract assertions until after overload resolution is complete).

- We do not want to support assertion-control objects being different in different translation units for the same contract assertion, so we make having them be different in different locations an ODR (one definition rule) violation. A function contract specifier is not a definition, so the ODR cannot be directly applied, but we can apply the same tools we used for the contract predicate to the assertion-control object to define our requirement that they always be the same.

- Note that many facets are based on the results of member functions invoked on the assertion-control object, and those member functions might possibly produce different results in different evaluations, as described in Section 2.2.2. The proposed facets in this paper, however, all involve invocations that will happen during constant evaluation.

- The expression in all other ways behaves as any other constant expression with regards to name lookup, overload resolution, and other behaviors.

Proposal 1: Assertion-Control Objects

Allow an optional *assertion-control-expression* to be specified as part of a *precondition-specifier*, *postcondition-specifier*, or *assertion-statement*.

Facets are what an assertion-control object can actually control (see Section 2.2). Some example facets include the ability to

- limit the evaluation semantic,
- alter the evaluation semantic using a `constexpr` function,
- identify a contract assertion as part of a named group or groups
- define a local violation handler to be invoked first

For each of the facets of an assertion-control object that we define, we will follow a few common design rules.

- A facet should be able to control the runtime (or nonruntime) behavior of a contract assertion but not the specific categories of bugs that are identified by that assertion. This restriction enables readability of the predicate without needing to investigate (and fully understand) the specific type and value of a control expression.
- As we said above, participation in a facet will be optional for each label and that opt-in will be based on satisfying a particular concept. No assertion-control expression is required to satisfy any particular concept other than the core `assertion_control_object` concept. This approach allows label objects to focus on the particular facet or facets they want to control without restricting the addition of new facets in the future.

2.1 Selection

Certain features are needed to make the use of assertion-control objects as straightforward as possible for the most common cases.

2.1.1 Name Lookup For Control Expressions

To minimize the redundant overhead of using labels as parts of assertion-control objects, the names of labels, especially commonly used ones, must be short. These names are also particularly useful only within the context of assertion-control expressions, so introducing short names into enclosing namespaces where they can do nothing but cause conflicts would be a clear downside. To facilitate using concise names without causing conflicts for enclosing code, we introduce a new kind of using directive and using declaration that introduce, into the enclosing scope, names that are found *only* within assertion-control expressions within that scope.

The directives are similar to a normal using directive but have the `contract_control` keyword inserted to indicate that the name is being considered only within assertion-control expressions. They can include an entire namespace, such as the directive that is in `<contracts>`:

```
using contract_control namespace std::contracts::labels;
```

A directive can also include just a single name, and then that name can be used without qualifiers in future assertion-control expressions:

```
using contract_control mylib::mygroup;

void f()
  pre<mygroup>(true);
```

Grammatically, these directives have the same structure as normal using directives:

> *assertion-control-using-directive :*
>
> *attribute-specifier-seq*$_{opt}$ `using` `contract_control` `namespace` *nested-name-specifier*$_{opt}$ *namespace-name* *';'*
>
> *assertion-control-using-declaration :*
>
> `using` `contract_control` *using-declarator-list*

Note a few caveats.

- We introduced a new keyword, `contract_control`, that is associated with the special behaviors and name lookup that are related to assertion-control expressions.
- A similar need for having libraries provide names that are used with just a single feature has arisen previously with user-defined literals. Since user-defined literals did not need to introduce any local form of name lookup, one might conclude that we should be able to do the same with assertion-control expressions.

  The difference, however, arises because user-defined literals segregate both their use and their names from all other expressions. A new user-defined literal operator brought into name resolution impacts only the interpretation of literals with the corresponding suffix and does not pollute any other expressions. Assertion-control expressions, however, are not special in that way and seek to leverage the normal syntax of the language, so we have to provide an alternate mechanism to avoid forcing name pollution on all users.
- Segregating names intended to be used as labels into their own namespace is helpful since unwanted names are not pulled into name lookup in the `std` or `std::contracts` namespaces. Therefore, we propose that Standard Library labels be placed in the namespace `std::contracts::labels`.
- The Standard Library labels must all be usable easily when importing or including the `<contracts>` header, so that header will behave as if it had an assertion-control using directive in the global namespace:

  ```
  // ... in <contracts>, in the global namespace:
  using contract_control namespace std::contracts::labels;
  ```

  By having this using directive, all Standard Library labels will become a common, simple, easily extensible vocabulary for expressing different properties of contract assertions.

Note, importantly, that the using directive itself is likely to be deployed very infrequently; most users will never need to know it is there and will simply leverage the benefits of having it present in the `<contracts>` header. Libraries that wish to provide their own suite of labels that provide a structure for their internal workflows will, however, benefit from having a similar shorthand available to match what the Standard Library can do.

Proposal 2: Contract Control Using Directives

Add the new keyword `contract_control` and introduce *assertion-control-using-directives* and *assertion-control-using-declarations* that make additional names available within assertion-control expressions.

2.1.2 Control-Expression Name Lookup Outside Assertions

For testing and for creating new labels that combine existing labels, the ability to evaluate expressions with those name-lookup rules outside of the context of an assertion-control expression can also be helpful. For that purpose, we introduce a new function-like operator that makes use of the `contract_control` keyword:

assertion-control-expression :
    `contract_control (` *constant-expression* `)`

Within the *constant-expression* in an *assertion-control-expression*, the same name-lookup rules apply that apply within the *assertion-control-specifier* on a contract assertion. This tool lets us write our own labels by combining existing Standard Library labels using the same expressions we can use within assertion-control expressions — avoiding the need to either add namespace-scope using directives or fully qualify all names.

For example, we could add a `using namespace` directive and then write our own label by combining Standard Library labels by short name, but that would bring all of the corresponding names into the enclosing namespace:

```
#include <contracts>
namespace my_lib1 {
  using namespace std::contracts::labels;
  constexpr auto my_audit_review_label = audit | review;
}
```

Similarly, we could fully qualify all our uses of the standard labels:

```
#include <contracts>
namespace my_lib2 {
  constexpr auto my_audit =
    std::contracts::labels::audit |
    std::contracts::labels::review;
}
```

With the addition of the *assertion-control-expression*, we can instead get the same name resolution and behavior outside a control expression that we would have in it:

```
#include <contracts>
namespace my_lib3 {
  constexpr auto my_audit = contract_control(audit | review);
}
```

This control expression also lets us validate properties of the type and value of an assertion-control expression outside of the context of a contract assertion in, for example, a test driver for the `review` label:

```
void testReviewWithAudit()
{
  static_assert( contract_control(audit|review)
    .compute_semantic(evaluation_semantic::quick_enforce)
                                                    == evaluation_semantic::observe);
  static_assert( contract_control(audit|review)
    .compute_semantic(evaluation_semantic::enforce) == evaluation_semantic::observe);
  static_assert( contract_control(audit|review)
    .compute_semantic(evaluation_semantic::observe) == evaluation_semantic::observe);
  static_assert( contract_control(audit|review)
    .compute_semantic(evaluation_semantic::ignore)  == evaluation_semantic::ignore);
}
```

The effects of this new operation could be replicated by manually replicating all of the assertion-control-using-directives as regular using directives within a block. In the case where only the Standard Library labels are being used and thus the only needed using directive is for the namespace `std::contracts::labels`, this could, for example, be done with a macro:

```
#define contract_control(...) \
  [](){ \
    using namespace std::contracts::labels; \
    return __VA_ARGS__; \
  }()
```

On the other hand, the preprocessor will be unable to know all assertion-control-using-directives that might be applicable in any given scope, and having a language construct that reproduces this effect for us is the most direct solution for these use cases.

Proposal 3: Assertion-Control Expression

Introduce the *assertion-control-expression* to evaluate expressions using the same name-lookup rules that apply within the assertion-control expression in a contract assertion.

2.1.3 Combining Assertion-Control Objects

A primary benefit of allowing arbitrary expressions to determine the values of assertion-control objects is that these expressions enable combining various labels in arbitrary ways. Consider, for example, labels having orthogonal purposes, such as to mark a contract assertion as being expensive to execute (and thus disabled in most builds) and as being newly introduced (and thus preferring to be observed instead of enforced).

To standardize the combining of labels, we choose an overloaded `operator|` as the mechanism that will be used when two assertion-control objects need to be combined. Other possibilities that we might consider are the comma operator (which is always confusing when overloaded) or bitwise `and` (which has less precedent for use in changin in the Standard). Because of the existing use of `operator|` with ranges to chain together multiple objects, we will continue to propose `operator|` as our default mechanism for combining assertion-control objects.

Importantly, a default implementation is provided in the namespace `std::contracts::labels` that will produce an assertion-control object having the combined properties of both its arguments:

```
namespace std::contracts::labels {
template <assertion_control_object LHS,
          assertion_control_object RHS>
constexpr assertion_control_object auto operator|(
  const LHS& lhs,
  const RHS& rhs);
}
```

This function will be found by overload resolution within a contract-control expression whenever `<contracts>` is included and by ADL for any object that is or inherits from a Standard Library label.

A full implementation of this function can be found in Appendix A.1, and for each individual property we propose, we will describe how the combined assertion-control object it returns will behave.

By implementing the combination logic as a library function, we make supporting vendor-provided extension facets easier and allow users to define their own combination semantics using a different operator or function if those provided by the Standard do not match their needs.

Proposal 4: Assertion-Combination Operator

Provide a free-function `operator|` in the namespace `std::contracts::labels` that returns an assertion-control object having all properties of both its operands.

2.2 Facets

Each facet that an assertion-control object can provide is optional, and integration with that facet is achieved by satisfying a particular concept. For each facet we describe here, we describe a few key aspects of the facet:

- what concept is used to opt-in to participation in the facet
- what the behavior will be if the assertion-control object does *not* satisfy the concept (i.e, what the *default* behavior is, which will in general match the current C++26 behavior)
- how the assertion-control object will be used when it does satisfy the concept
- Standard Library labels aid in making use of the facet, either directly in assertion-control expressions or as utilities for building user-defined labels
- how a combined assertion-control object (using `operator|`) will satisfy the facet if its constituent components do

For example, consider the simplest facet, which is the default facet that all assertion-control objects must satisfy.

- The concept that is checked is `assertion_control_object`, and it verifies that a nested typename `assertion_control_object` is within the type of the object:

  ```
  namespace std::contracts::labels {
    template <class T>
    concept assertion_control_object =
  ```

```
    requires ( typename T::assertion_control_object; );
}
```

As a general convention, label types will generally name themselves with this alias, but that is not usually required.

- If an assertion-control-object is specified that does not satisfy this concept, the program is ill-formed. (If no assertion control object is specified, the program remains well-formed, and no behavior changes.)
- If an assertion-control object is specified that does satisfy this concept, nothing happens.
- The Standard Library should provide a type `empty_label_t` that satisfies this concept as well as a `constexpr` object named `empty_label` of that type. The type will be empty other than the needed nested name and exists primarily for testing combined labels and possibly as a utility base class.

  ```
  namespace std::contracts::labels {
  struct empty_label_t {
    using assertion_control_object = empty_label_t;
  };
  constexpr empty_label_t empty_label;
  }
  ```

- A combined assertion-control-object requires that both its constituent labels satisfy this concept and will always itself satisfy this concept.

2.2.1 Restricting Evaluation Semantics

Three absolutely needed use cases arise where contract assertions must restrict the range of semantics that might be applied to a contract assertion.

1. When a contract-assertion predicate can be written only in a destructive manner, writing that contract assertion might still be useful. A common example is preconditions involving iterators that might be input iterators:

   ```
   template <typename IT>
   void f(IT begin, IT end)
     pre(std::distance(begin,end) > 5);  // destructive for input iterators
   ```

   Such contract assertions must be restricted from ever being evaluated with a *checked* semantic (i.e., *quick-enforce*, *enforce*, or *observe*).

2. Many codebases consider allowing narrow contracts that are unchecked to be unacceptable risks, and allowing the evaluation of contract assertions written in those contexts with any semantic that does not check and enforce the assertion is considered unacceptable.

   Such *hardened* or *terminating* contract assertions will thus limit the set of allowed semantics to only those that are enforced.

3. Certain codebases are equally allergic to the prospect of enabling unconditional termination and must always continue normally even when known bugs are present. In such cases, users would want to explicitly preclude the use of the *enforce* or *quick-enforce* semantics.

In addition, in some cases, finer control over assertion-evaluation semantics becomes important, such as when a particular semantic results in particularly unacceptable code generation or, worse, triggers a compiler bug.

To facilitate this control, we define a type in the `std::contracts` namespace to represent a *set* of evaluation semantics, with a simple `constexpr` subset of the API of `std::set<std::contracts::evaluation_semantic>`:

```
class evaluation_semantic_set {
public:
  // Constructors
  constexpr evaluation_semantic_set();
  constexpr evaluation_semantic_set(const evaluation_semantic_set& other);
  constexpr evaluation_semantic_set(std::initializer_list<evaluation_semantic> ilist);


  // Accessors
  constexpr bool contains(evaluation_semantic semantic) const;
  constexpr std::size_t size() const;

  // Modifiers
  constexpr void clear();
  constexpr void insert(std::initializer_list<evaluation_semantic> ilist);
  constexpr void erase(evaluation_semantic);

  // Set operations
  constexpr evaluation_semantic_set& operator&=(const evaluation_semantic_set& rhs);
  constexpr evaluation_semantic_set& operator|=(const evaluation_semantic_set& rhs);
  constexpr evaluation_semantic_set& operator~() const;

  friend constexpr evaluation_semantic_set operator&(const evaluation_semantic_set& lhs,
                                                     const evaluation_semantic_set& rhs);
  friend constexpr evaluation_semantic_set operator|(const evaluation_semantic_set& lhs,
                                                     const evaluation_semantic_set& rhs);


  // Comparisons
  friend constexpr bool operator==(const evaluation_semantic_set& lhs,
                                   const evaluation_semantic_set& rhs);

  // Utility Factories

  static constexpr evaluation_semantic_set all();
  static constexpr evaluation_semantic_set none();
};
```

An assertion-control object having an accessible `const` member named `allowed_semantics` that can initialize an `evaluation_semantic_set` will limit the possible set of evaluation semantics for the associated contract assertion to those in the set. In other words, assertion-control objects can opt in to this feature by modelling the following concept:

```
namespace std::contracts::labels {
template <typename T>
concept allowed_semantics_label  =
  assertion_control_object<T> &&
  requires (const T t) {
    requires std::is_const_v<decltype(T::allowed_semantics)>;
    evaluation_semantic_set({t.allowed_semantics});
  };
}
```

We require that the member be `const` to give the implementation the freedom to query the value at any time and use it asynchronously, relying on the elements of this set being unable to change once an assertion-control object has been constructed.

A combined assertion-control object will satisfy this concept if either of its constituent elements does, and its `allowed_semantics` member will have the intersection of all the corresponding members of its constituents.

When an implementation selects an implementation-defined semantic for a contract assertion, it shall always be from one of those semantics in the associated `allowed_semantics` set. If the set is empty or the chosen semantic is not in the allowed set, the program is ill-formed. Implementations are encouraged to document ways in which they will look at the set of allowed semantics and adjust — in a manner acceptable to users — the chosen semantic to be one of those in the set. Depending on the use and the compiler, this selection will involve some ordering of preferred semantics (such as by how strictly they enforce the predicate — *ignore* < *observe* < *enforce* < *quick-enforce*) and a search through that ordering starting at the chosen one until one that is allowed is found.

Proposal 5: Allowed-Semantics Control

Allow evaluation of contract assertions having only the `allowed_semantics` of the associated assertion-control object. Combined objects intersect the sets of allowed semantics.

Contract assertions that do not allow any unchecked semantics require special consideration and will be discussed in Section 2.3.1.

2.2.2 Choosing Evaluation Semantics

One of the most important use cases for labels is that of introducing logic — written with compile-time C++ — that can alter the semantic that will be used for a given contract assertion. Such logic augments the implementation-defined choice of semantic that is the purview of the build system with user-definable logic executed at compile time.

In particular, having a `review` label that can be used to mark contract assertions that are being newly introduced into a codebase is essential to deploying Contracts successfully at scale.

An assertion-control object having a `constexpr compute_semantic` member function can be used to alter the semantic that will be chosen for the associated contract assertion; in other words, assertion-control objects can opt in to this feature by modelling the following concept:

```
namespace std::contracts::labels {
template <typename T>
concept semantic_computation_label =
  assertion_control_object<T> &&
  requires(const T t, evaluation_semantic s) {
    evaluation_semantic(t.compute_semantic(s));
  };
}
```

A combined assertion-control object will implement the above member to pass its input to each constituent object in turn if either constituent object models `semantic_computation_label`.

When an implementation-defined semantic is chosen, it is then passed to the `compute_semantic` function of the assertion-control object, and the result of that function is used as the semantic for the evaluation.

- If the assertion-control object also models `allowed_semantics_label` and the result is not in that set, the program is ill-formed. Importantly, these two modes of influencing the semantic for a contract assertion are not mutually exclusive. Making them so would preclude the ability to use many compound assertion-control objects, such as `never_observe | review`.
- What values will be passed at compile time to `compute_semantic` is implementation defined. At a minimum, when a contract assertion is going to be evaluated with a particular compiler-chosen semantic, that semantic will be transformed first using `compute_semantic`. Other compilation choices, such as delaying the choice of semantic to link time or run time, can result in the compiler needing to call `compute_semantic` for a single contract assertion many times — for each possible evaluation semantic — to determine the set of actual semantics that will be available to evaluate that contract assertion. (In other words, when the final choice of evaluation semantic will not happen until after the compilation phase of a translation unit, the compiler may need to build a full mapping from user-chosen semantic to computed semantic; we will show below how that might be done).

One could model this process using pseudocode to get a good idea of how semantics will be determined. First, we can see the simplest case where the implementation-defined (i.e., chosen on the command line) semantic is fully determined at compile time, and thus we can evaluate the entire process of determining the semantic as a constant evaluation:

```
{
  constexpr auto control_object = /* the control object of this assertion */;

  // First determine, based on command line configuration,
  // what the semantic is.  This might vary based on groups that
  // the control object identifies and be adjusted to one of the
  // allowed semantics for the control object.
  constexpr evaluation_semantic chosen_semantic = get_chosen_semantic(control_object);

  // Allow the label to adjust the chosen semantic.
  if constexpr (semantic_computation_label<decltype(control_object)>) {
    constexpr evaluation_semantic computed_semantic
```

```
                                    = control_object.compute_semantic(chosen_semantic);

      // Check that the chosen semantic is an allowed one; fail if not.
      if constexpr (allowed_semantics_label<decltype(control_object>)
               && !(control_object.allowed_semantics.contains(computed_semantic))) {
        // Nonallowed semantic is produced.
        static_assert(false);  // Fail constexpr evaluation.
      }
      chosen_semantic = computed_semantic;
    }

    // Implement the evaluation of the contract assertion.
    switch (chosen_semantic) { /* ... */ }
  }
```

A very similar bit of pseudocode applies if the semantic is not known at compile time, but we instead must determine how any chosen semantic might map to a final semantic (using some reflection and expansion statements) while still invoking only the `constexpr compute_semantic` function:

```
  {
    constexpr auto control_object = /* the control object of this assertion */;
    evaluation_semantic chosen_semantic = get_chosen_semantic(control_object);

    // At run time, we wouldn't be able to pass chosen_semantic to the constexpr
    // compute_semantic function.
    template for (constexpr auto s : std::meta::enumerators_of(^evaluation_semantic)) {

      // A non-allowed semantic must not be produced by get_chosen_semantic above.
      if constexpr (allowed_semantics_label<decltype(control_object>)
               && !control_object.allowed_semantics.contains([:s:])) {
        continue;
      }

      if (chosen_semantic == [:s:]) {
        if constexpr (semantic_computation_label<decltype(control_object)>) {
          constexpr evaluation_semantic computed_semantic = control_object.compute_semantic([:s:]);
          if constexpr (allowed_semantics_label<decltype(control_object>)
                   && !control_object.allowed_semantics.contains(computed_semantic)) {
            static_assert(false);
          }
          chosen_semantic = computed_semantic;
        }
        break;  // End expansion statement.
      }
    }

    switch (chosen_semantic) { /* ... */ }
  }
```

An assertion-control object's `compute_semantic` member function might, for example, use legacy macros to determine its output or possibly even some other mechanism to enable user-defined configuration of a translation unit.[3]

Proposal 6: Compute Semantics

Compute the evaluation semantic of contract assertions using the `compute_semantic` member of the assertion-control object (if there is one).

2.2.3 Controlling Combination

Occasionally, users will need to make certain labels mutually exclusive. In some cases, this simple requirement is imposed because particular labels just do not make sense when used together. In other cases, a value is associated with which member of a mutually exclusive family of labels is being used, such as the *cost* of evaluation associated with `default` or `audit` from C++2a Contracts.

Since at least some of these mutually exclusive families of labels have values associated with them, we can think of each such family as a *dimension* that is associated with the resulting assertion-control object. The dimensions of a label can be expressed by a specialization of the following template:

```
namespace std::contracts::labels {
template<typename... Dims>
struct dimension_list {};
}
```

An assertion-control object has a dimension any time it declares a nested name that is a specialization of `std::contracts::labels::dimensions`, i.e., when it satisfies the following concept[4]:

```
namespace std::contracts::labels {
template <typename T>
concept dimensioned_label =
  assertion_control_object<T> &&
  is_specialization_of_v<typename T::dimensions, dimension_list>;
}
```

The primary purpose of these dimensions is to control what happens when multiple labels that have dimensions are combined.

- The dimensions of a combined label is the union of all dimensions of its constituent labels, if any.
- If any intersection occurs in the dimensions of the constituent labels, the combined label is ill-formed.

Within the Standard Library, the `opt` and `audit` labels, which identify contract assertions based on their cost to evaluate, would share a common dimension to avoid needing to determine how an assertion will behave if a user marks it as both *fast* and *slow*. Users will be able to provide their own mutually exclusive labels by specifying the same dimension.

[3] See the discussion of a build environment in Section 3.4 for a suggestion of a possible future proposal that might enable this kind of configuration without the use of the preprocessor.

[4] This implementation assumes the presence of a Standard Library trait `is_specialization_of`, such as proposed in [P2098R0], or some similar functionality.

Proposal 7: Label Dimensions

Recognize dimensions on labels, combine them, and make combining labels with overlapping dimensions ill-formed.

Unlike the other proposed facets, this facet is entirely implemented in the Standard Library. Constraints on `operator|` in `std::contracts::labels` will validate that the dimensions of its operands do not intersect, and the returned combined label object will have the union of both sets of dimensions.

2.2.4 Local Violation Handling

Customizing contract-violation handling for specific contexts and specific contract assertions is an oft-requested feature.

- Libraries and components might need to fail fast for safety or security reasons and to prevent the escape of additional information via a user-provided contract-violation handler but also still need to gather some diagnostics in a manner that *quick-enforce* would not allow.
- Other libraries might wish to log additional state information on certain failures — e.g., what request was being processed at the time of failure or what the subsystem configuration was — and to do so before delegating the primary logic of contract-violation handling to the user-chosen global violation handler.

To enable these use cases, we want to add a facet for control objects to provide their own contract-violation handler that will either precede the default one or replace it entirely.

- By default, we do not want to subvert the global violation handler, so a `handle_contract_violation` member function having a return type of `void` will indicate that, when it returns normally, the next violation handler in the chain (up to the replaceable global one) will be invoked with the same `contract_violation` object.[5]

  We could also consider disallowing a `void` return type entirely and forcing users to make a choice.
- To allow completely local violation handling, we also support returning a value of the new enumeration type `violation_handled`:

  ```
  enum class violation_handled {
    not_handled,
    handled
  };
  ```

  When `handled` is returned, the violation-handler invocation process is considered complete, and no further violation handlers will be invoked.

To achieve all that, an assertion-control object that models the `local_violation_label` concept will be able to insert its own contract-violation handling logic:

[5]Whether this behavior is the right default for `void`-returning local violation handlers is, of course, up for debate, like all defaults in C++. Our primary reasons for choosing this default, however, remain that the global violation handler is a key aspect of having Contracts as a language feature in the first place, so subverting that global violation handler should occur only when quite explicitly choosing to do so.

```
namespace std::contracts::labels {
template <typename T>
concept local_violation_label =
  assertion_control_object<T> &&
  requires(std::remove_const_t<T> t, const contract_violation& v) {
    t.handle_contract_violation(v);
    requires (std::is_same_v<decltype(t.handle_contract_violation(v)), void>
           || std::is_convertible_v<decltype(t.handle_contract_violation(v)),violation_handled>);
  };
}
```

When the associated contract assertion is violated, the `contract_violation` object will be passed first to the `handle_contract_violation` member of its assertion-control object. If the return type is non-`void` and converts to the `handled` value of `violation_handled`, violation handling will complete if the local violation handler returns normally. In this case, we say that the handler has *handled* the contract violation.

In all other cases, when the return type is `void` or when it converts to the `not_handled` enumerator, the global (replaceable) violation handler will be invoked next.

A combined assertion-control object will evaluate the `handle_contract_violation` members of its constituent objects from *right* to *left*. This can be thought of as control flowing back from the predicate to the *global* contract-violation handler, which is the default and which comes last. If any handler returns `handled`, indicating that the violation has been *handled*, the next handler in order will not be invoked.

Proposal 8: Local Violation Handlers

Invoke the `handle_contract_violation` member of the assertion-control object (if there is one), allowing it to be the exclusive handler or to be chained before the global replaceable handler.

Note that this design also supports some interesting use cases. For example, a label that causes any assertions from the contract predicate to be propagated back to the caller can be easily implemented by rethrowing exceptions when the cause of the violation is an exception from the predicate and, in all other cases, indicating that the violation has not yet been handled:

```
struct pass_through_exceptions_t {
  using assertion_control_object = pass_through_exceptions_t;

  std::integral_constant<violation_handled, violation_handled::not_handled>
      handle_contract_violation(const contract_violation& violation)
  {
    if (violation.detection_mode() == detection_mode::evaluation_exception) {
      throw;
    }
    return {}
  }
}
constexpr pass_through_exceptions_t pass_through_exceptions{}
```

2.2.5 Grouping Assertions

One of the simplest uses case for assertion-control objects and their constituent labels is to identify groups of contract assertions so that tools can manipulate those groups. Once we can group assertions with named groups, we can then teach our command-line configurations to use that grouping information to control the initial evaluation semantic for assertions in that group.

In addition, groups can then be used by, for example, static-analysis tools to identify those assertions that are to be subject to or excluded from that particular static-analysis tool (but which might otherwise have no impact on how the assertions are evaluated at run time).

For that purpose, we define a concept for identification labels that allows a label to list, by name, the groups to which it belongs:

```
namespace std::contracts::labels {
template <typename T>
concept identification_label =
  assertion_control_object<T> &&
  requires (const T t) {
    { t.group_names } -> std::ranges::range;
    { *(std::ranges::begin(t.group_names)) } -> std::convertible_to<std::string_view>;
  };
}
```

Combining such labels, using the pipe operator (`|`), would result in a label that is in all groups named by its constituents. The resulting object's `group_names` member will be the (sorted and unique) combination of the group names from each constituent object that satisfies this concept.

The simple case of turning a string literal into a label object satisfying the appropriate concept can be easily provided with a user-defined literal operator that returns an appropriately populated object:

```
namespace std::contracts::labels {
template <int N>
class assertion_group_label {
public:
  using assertion_control_object = assertion_group_label<N>;

  char group_names[1][N];

  constexpr assertion_group_label(const char(&name)[N])
  {
    std::copy(std::begin(name), std::end(name),
              std::begin(group_names[0]));
  }
};
}

template <assertion_group_label Label>
constexpr auto operator""group() { return Label; }
```

Now we can easily group our assertions so that they can be independently controlled from the command line:

```
void get(int *output, int index)
  pre<"pointers"group> ( output != nullptr )
  pre<"bounds"group>   ( index >= 0 && index < size() );
```

We can similarly have assertions that are in multiple groups, and our command-line configuration will dictate how that ends up resolving:

```
void f()
  pre                  (cond1())  // in no groups
  pre<"pointers"group> (cond2())  // in "pointers" group
  pre<"bounds"group>   (cond3())  // in "bounds" group
  pre<"pointers"group | "bounds"group>
                       (cond4()); // in both groups
```

Proposal 9: Group Identification Labels

Standardize the use of labels meeting the `identification_label` concept to identify groups of assertion by name, and add the user-defined literal operator "group" to the `std::contracts::labels` namespace to easily create labels with a named group.

Implementations must document how they will determine the evaluation semantic for contract assertions that belong to multiple groups. In general, we expect the more involved configurations that specify semantics at finer granularities to have an ordering for their specification that applies the first selection that matches. In other words, we might be passing a configuration file that *quick-enforces* all `hardening` assertions, ignores all `testing_only` ones, and enforces the remainder:

```
{ group="hardening" semantic="quick-enforce" }
{ group="testing_only" semantic="ignore" }
{ semantic="enforce" }
```

Given this, we can apply the configurations above to determine the semantic for a number of different assertions having varying control objects:

```
auto important = "important"group;
auto testing_only = "testing_only"group;

void f(...)
  pre<important>    (...)  // quick-enforce
  pre<testing_only> (...)  // ignore
  pre               (...); // enforce
```

The same labels, when combined, produce a semantic following a clear and reasonable algorithm:

```
void g(...)
  pre< important | testing_only >  (...)  // first config matches -> quick-enforce
  pre< testing_only | important >  (...)  // same -> quick-enforce
  pre< important | review >        (...); // first config -> quick-enforce, and
                                          // review changes that to observe
```

Note that we require that the group information be immutable so that no question remains about what value is used when an assertion is evaluated at run time when the group might have been changed. We certainly want to avoid an implied requirement indicates that compilers must support labels where the group is somehow dynamic. (Such support would require that the compiler's build-time logic to assign semantics to groups must persist, possibly all the way to run time, so that it can be reapplied if the groups have changed.)

2.2.6 Comment Manipulation

The `comment` property in the `contract_violation` object has an implementation-defined value but is generally expected to contain the program text of the assertion predicate. In some environments, such information must be restricted from users or log files, and for others a significantly more useful string might be available to present in logs (e.g., to aid in debugging and understanding).

For that purpose, we want to be able to manipulate the comment that will eventually be passed to the violation of a contract assertion. Another facet that looks for a `compute_comment` member function to use to filter the comment produced by the implementation can allow us to perform our desired manipulations of the `comment` property of the `contract_violation` object before it gets stored in application binaries and eventually passed to the violation handler:

```
namespace std::contracts::labels {
template <typename T>
concept compute_comment_label =
  assertion_control_object<T> &&
  requires (T t, const char* comment) {
    { t.compute_comment(comment) }-> std::convertible_to<const char*>;
  };
};
```

An additional property on `contract_violation`, `message`, has been proposed by [P3099R0]. This property allows for the specification of a user-defined message separate from the comment as an optional trailing parameter in the assertion specifier itself. If we have such adopt such a property, a facet to allow manipulation of that property should also be added to parallel the design of the facet allowing the manipulation of the comment:

```
namespace std::contracts::labels {
template <typename T>
concept compute_message_label =
  assertion_control_object<T> &&
  requires (T t, const char* message) {
    { t.compute_message(message) }-> std::convertible_to<const char*>;
  };
};
```

Because we will be specifying the interaction between the violation-handling process and this facet in the core language, we restrict ourselves to using `const char*` as the vocabulary type for the strings here. Historically, this would have been significantly limiting as we would have been restricted to return string literals, but we are now able to leverage the introduction of `define_static_string` by [P3491R3] to dynamically produce strings during constant evaluation.

The easiest case is, of course, to instead just produce a string literal as the message or comment to return, ignoring the value produced by the compiler (or earlier labels):

```
struct fixed_message_label_t {
  using assertion_control_object = fixed_message_label_t;

  constexpr fixed_message_label_t(const char* message) : d_message(message) {}

  constexpr const char* compute_message(const char*) { return d_message; }
private:
  const char* d_message;
};
```

To dynamically generate a string, we can use a wide range of Standard Library facilities (such as `constexpr std::format` as introduced by [P3391R1]) and then make a string to return using `define_static_string`:

```
struct message_suffix_label_t {
  using assertion_control_object = message_suffix_label_t;

  constexpr message_suffix_label_t(const char* suffix) : d_suffix(suffix) {}

  consteval const char* compute_message(const char* message)
  {
    std::string output = std::format("{}{}", message, suffix);  // Append the suffix.
    return std::define_static_string(output.c_str());           // Make a static string.
  }
private:
  const char* d_message;
};
```

The string message introduced in [P3099R0] can also be considered as implicitly adding another label to the start of the assertion-control expression that is implemented by `fixed_message_label_t` above. In other words, the two preconditions below will exhibit the same observable behavior when violated:

```
void f(int x)
  pre(x > 0, "x must be positive")
  pre<fixed_message_label("x must be positive")> (x > 0);
```

In the first precondition, we simply provide the user-provided message using the core-language syntax to do so. This syntax is functionally equivalent to (and could be syntactic sugar for) using the label object that provides the specified message in the second precondition.

Proposal 10: Comment and Message Filtering Labels

Enable a facet to manipulate (at compile time) the `comment` and `message` fields that will be made available in the `contract_violation` object.

Note that there is a related but much more involved use case, producing a custom message at run time based on data that is available in the context of the contract-assertion predicate. Such a calculation would by necessity require capturing (by reference) all the values that might be formatted into the resulting message, and a wide range of lifetime and safety concerns would need to be addressed as part of a proposal for such maximal flexibility. That form of functionality should be pursued as a separate feature that would need to introduce new work to be done at many points within the violation-handling process.

2.2.7 Integration with `contract_violation`

Allowing the assertion-control object to communicate additional information to a contract-violation handler has many obvious use cases.

- Indicate groups of assertions that are to receive some form of special treatment by a user-provided contract-violation handler, such as choosing not to throw from the handler or notifying particular endpoints about failures.
- Carry additional assertion-specific information that might guide the contract-violation handler, such as logging additional information.

Numerous considerations, however, must be carefully navigated when we contemplate functionality that spans both the TU in which code for a contract assertion is generated and the TU in which the contract-violation handler is defined.

- Any interface we provide to information emanating from an assertion-control object must work correctly for all permutations of compiler and Standard Library implementations that might be used in the two TUs.
- Because we can't depend on using the same Standard Library, we can't depend on using a polymorphic class and virtual functions to access information.
- As with all facets, we need an API that allows for the possibility of having multiple labels that do and do not overlap in the information they are presenting to the contract-violation handler; i.e., we must be able to combine multiple labels using the pipe operator and produce a new object that provides access to both behaviors in some fashion.
- It is far better for a contract-violation handler to be unaware of a label it won't understand than to be given access to objects that it might believe are of certain types but which aren't runtime compatible.
- Requiring that assertion-control objects always live until run time would introduce significant and possibly unwanted overhead on all contract assertions. Therefore, we should expose only those parts of assertion-control objects that have opted into such retention and integration.
- Details of the assertion-control objects that will be presented to the contract-violation handler will not be available until, at earliest, link time. Therefore, the violation handler must be able to inspect and reason about the control objects associated with a given violation using runtime executions, not compile-time evaluations.

To achieve that, there are a number of potential alternatives that will *not* work.

- Erasing the entire type of the contract-violation object and exposing it to the handler as a `void*` accessible through the `contract_violation` object will not work. Such a pointer cannot be turned into a usable object without knowing its type.[6]

- Requiring that participating contract assertions extend a particular base class and use `dynamic_cast` on that base class — or virtual functions provided by that base class — would be problematic because it would require that all contract assertions and the contract-violation handler make use of the same Standard Library. (It would also introduce an explicit dependency on the Standard Library from a core-language feature, another property we actively want to avoid).
- Using compile-time reflection facilities, including annotations ([P3394R4]), as suggested in [P3831R0], will not work because the contract-violation handler is compiled with absolutely no knowledge of either the specific assertion-control objects used for any given contract violation or even the full scope of such objects that might be in use within a complete linked program.

What will work, however, is to extend the design of `contract_violation` in the same way it has already progressed by using its member functions as a way to insulate details of the TU containing the assertion from the TU where the contract-violation handler is defined. Rather than provide direct access to a type-erased assertion-control object, we will provide a function that allows the contract-violation handler to query the assertion-control object for data.

Assertion-control objects will, of course, need to be able to respond to such queries. For that purpose, we'll look for a `query` member function that has two parameters.

1. The first is a `key` parameter of type `const void*`, used to provide a generic identifier for something to query. We do not want to make this parameter less opaque, such as a string or number, because that would vastly increase the risk of querying for one thing and, due to miscommunication between different libraries containing labels, getting a result back from a different label that is not in the expected format.

   By instead using pointers to constant objects, we ensure that the violation handler will be referencing a constant object that must match the one that the label itself was linked against. If the label provides a result for the given key, then it must have linked with a matching definition for that key object.

2. The second is an optional `index` parameter to allow for returning multiple different results for the same key. Although a single label is unlikely to produce results for many keys, a combined assertion-control object must facilitate that case, and hence the API must allow for it.

   In general, if the mere presence of a particular label is all that is important, the violation handler simply queries for the specified key using an `index` of `0` and acts if the returned result is not `nullptr`. When the contents of the result matter, the handler can iterate through them all until `nullptr` is returned from `query`. If the results matter and there is no clear mechanism to handle labels with multiple results for the same query, the label itself can also specify a dimension (see Section 2.1.3) so that it is not combined with other labels that answer for the same key.

[6]Previous versions of this paper suggested this option along with the use of `dynamic_cast` on the `void*`. Such a suggestion simply does not work, and having proposed such a nonviable approach is a source of eternal embarrassment for this author.

The concept we associated with this facet requires that we have an appropriate `query` member function on our assertion-control object:

```
namespace std::contracts::labels {
template <typename T>
concept queryable_label =
  assertion_control_object<T> &&
  requires (const T t, const void *key, std::size_t index) {
    { t.query(key,index) } -> std::same_as<void*>;
  };
}
```

Implementations that are generating code for a contract assertion having an assertion-control object that satisfies this concept would then store two distinct things in the data available to the `contract_violation` object.

1. The first is a pointer to the assertion-control object (as a `void*`).
2. The second is a function pointer having 3 parameters: the first a pointer to the assertion-control object and then the 2 expected parameters for the `query` function. When creating this pointer, the type of the assertion-control object will be known, and thus it can be a pointer to an instantiation of the following template:

   ```
   template <typename C>
   void* query_control_object_impl(const void* obj,
                                   const void* key,
                                   std::size_t index)
   {
     return static_cast<const C*>(obj)->query(key,index);
   }
   ```

Given that those two values will be available to the implementation of `contract_violation`, the public accessor for the violation handler to use will be straightforward to implement:

```
namespace std::contracts {
class contract_violation {
  // ...
  void* query_control_object(const void* key, std::size_t index = 0) const;
  // ...
};
}
```

Note that the key and return value must remain type-erased to `void*` because the violation handler must work correctly with violations from any TU in a program and cannot have compile-time type dependency on the specific types of control objects in use.

A combined label will pass on the query to its constituent objects when applicable, and can use a binary search to determine how many results the first constituent has for any given `key` to then send a reduced index in the query to the second constituent when needed.

As an example, we might define a label type that captures an owner's name and contact address as string views:

```
class owner_label {
public:
  constexpr owner_label(std::string_view name, std::string_view address)
  : d_name(name)
  , d_address(address)
  {}

  void* query(const void* key, std::size_t index = 0);

  // ...

private:
  std::string_view d_name;
  std::string_View d_address;
};
```

To allow for communication between a contract-violation handler and our label, we need to have something in common to which they can refer that can be used as a key. For that purpose, we'll define two static members of `owner_label` whose addresses will serve as keys:

```
// ...
  constexpr static int name_key = 1;
  constexpr static int address_key = 2;
// ...
```

With those keys, we can then implement our query function to return the name and address for these keys:

```
void* omner_label::query(const void* key, std::size_t index = 0)
{
  if (0 == index && key == &name_key) {
    return (void*)&d_name;
  }
  else if (0 == index && key == &address_key) {
    return (void*)&d_address;
  }
  else {
    return nullptr;
  }
}
```

Within the contract-violation handler, we can easily access the first name and address to see if an email should be sent:

```
void handle_contract_violation(const contract_violation& violation)
{
  if (auto* name_ptr = violation.query_control_object(&owner_label::name_key)) {
    auto* address_ptr = violation.query_control_object(&owner_label::address_key);

    const std::string_view name = (const std::string_view&)*name_ptr;
    const std::string_view address = (address_ptr == nullptr) ?
                                    "" : (const std::string_view&)*address_ptr;
    // Send message to name/address.
  }
}
```

If we want to support multiple names and addresses in our contract-violation handler, a simple utility function can be used to extract all of them:

```
void get_owners(const contract_violation& violation,
                std::vector<std::pair<std::string_view, std::string_view>> *owners)
{
  std::size_t index = 0;
  while (auto* name_ptr = violation.query_control_object(&owner_label::name_key, index)) {
    auto* address_ptr = violation.query_control_object(&owner_label::address_key, index);

    const std::string_view name = (const std::string_view&)*name_ptr;
    const std::string_view address = (address_ptr == nullptr) ?
                                     "" : (const std::string_view&)*address_ptr;
    output->emplace_back(name,address);

    ++index;
  }
}
```

Note that we could, possibly, consider not supporting indices and always simply returning the value returned by the first constituent label that answers for a query. This would, however, force a particular behavior when combining these labels that might not be generally optimal. The owners list above might, for example, be used to attach owners to many different unrelated labels that get combined based on other concerns, and we should be able to readily collate all the relevant owners:

```
namespace mylib {

constexpr auto my_review =
  contract_control(review | owner_label("review_committee", "rc@mycompany.com");
constexpr auto my_audit =
  contract_control(audit | owner_label("audit_committee", "ac@mycompany.com");

void f()
  pre<my_audit | my_review>(...);  // Mail both committees on violations.
}
```

Proposal 11: Query Interface Through `contract_violation`

Add the `query_control_object` member to `std::contracts::contract_violation` along with associated support for the `queryable_label` concept.

There is some additional complexity introduced to this design because of the support needed for combining labels that respond to the same query (and thus the need to be able to pass in an index). A utility type in `std::contracts::labels` that provides the normal expected behavior for a simple queryable label would be helpful here, implementing `query` by delegating to another function that takes just a `key` when the `index` is `0`, and returning `nullptr` otherwise.

2.3 Key Considerations

Assertion-control objects introduce a new facet into the overall design of C++26 Contracts that must be analyzed carefully to ensure no issues will arise. In this section, we will consider a few of the subtler interactions.

2.3.1 Always-Checked Assertions

When we write a contract assertion in our code, we expect that predicate to be `true` when our programs run, and the exact times when we have that expectation are determined by which type of contract assertion we used (`pre`, `post`, or `contract_assert`). Our expectation that this predicate must be true gives the compiler the freedom it needs to evaluate the contract assertion with any of the available evaluation semantics.

Compilers then give us back the ability to control that choice of semantics in a variety of ways, largely through command-line parameters. Importantly, however, they are not *required* to give us the full range of alternatives, and in some cases, certain platforms will be unable to give us that full range.

- On some platforms (such as Windows), function parameters are destroyed by the callee before control is returned to the caller, which means that caller-side checking of postconditions that refer to a function parameter is nonviable. If asked to do only caller-side checking of postconditions, a compiler on such a platform would be unable to provide any choice of semantic other than *ignore* for those postconditions that reference function parameters.
- Implicit contract assertions, as proposed in [P3100R4], can often introduce a very large number of contract assertions in a very small amount of code. In some cases, a large number of program points might involve a check that would all be true (or not) under the same conditions, such as when a large expression makes use of an erroneous value, and thus each subexpression is similarly making use of further erroneous values. In these cases, the user experience again greatly improves by giving the compiler leeway to check the condition once for the entire expression and then *ignore* all of the other implicit contract assertions within the expression. This freedom is most helpful when the user is otherwise interested in the *observe* semantic; little or no benefit is had by receiving many tightly correlated notifications of an observed violation after having received the first one.
- In the future, when doing indirect calls (through virtual functions as described in [P3097R1] or something like a function pointer as described in [P3271R0]), there might be distinct sets of caller-facing and callee-facing function contract assertions for any given function invocation. In most cases, the only simple implementation choice for these assertions is to evaluate caller-facing contract assertions on the caller side and callee-facing ones in the callee. Again, on platforms where postconditions cannot be readily evaluated in a caller, an implementation would be forced to say that most caller-facing postconditions can only be *ignored*.

All together, the ability to allow contract assertions to be evaluated at specific points without requiring that all implementations be able to do such evaluations under all conditions is important to maximizing our implementation choices and design freedoms. A problem, however, arises when an assertion-control object retracts design freedom by not allowing any unchecked semantics.

With C++26 Contracts as is, we do not need to aggressively make any changes to deal with these issues. Existing implementations all start by supporting callee-side checking, and that can reliably be checked. In the future, however, freedom has to be explicitly given in some scenarios to allow for the *ignore* semantic even when a label would not allow it. When such freedoms might be exploited, they will of necessity be implementation defined, where the conditions will be tightly dependent on when the platform supports alternative semantics beyond *ignore*.

One might also think that a contract assertion that is always checked might be eligible to be treated differently from other contract assertions in a more fundamental way — perhaps requiring exactly one evaluation or removing the use of `const`-ification within the predicate. This massive design change would, however, have major implications.

- Altering the interpretation and meaning of the contract-assertion predicate based on the possible semantics with which it can be evaluated would break one of our core guidelines for the design of labels in general: Understanding the meaning of the predicate must not depend on having to understand the meaning of the assertion-control object.
- In a similar vein, neither should changing the assertion-control object on a predicate alter the meaning of the predicate. Most importantly, profiling might easily lead to one's discovering that a particular predicate is both expensive to check and not being violated in practice, resulting in a desire to *ignore* that contract assertion in production builds by changing the label that has been applied to it. Taking action on that profiling information to make better decisions about production deployment need not result in fundamentally changing the nature of what the assertion itself is saying.
- A primary reason to always enforce a predicate is to ensure that any checks that come after it lexically are *never* evaluated after a violation of the earlier assertion. Consider, for example, a case where we have preconditions that use an always-terminating label before preconditions with no label:

  ```
  void f(int* p)
    pre<terminating>(p != nullptr)
    pre(*p > 17);
  ```

  If a compiler emits caller-side checks, then it must emit both of the above checks. The rules for duplication of function contract assertions require this property; the second assertion cannot be evaluated at all prior to having evaluated the first assertion at least once, and due to the `terminating` label, that evaluation cannot be with the *ignore* semantic.

  On the other hand, to make sure the preconditions are checked when invoking `f` through a function pointer, the default entry point for `f` must also evaluate both checks within the callee.

  Taken together, this means that (barring a massive ABI change), we cannot restrict the first assertion to evaluate exactly once without losing other features such as caller-side checking or the lexical ordering of the evaluation of contract assertions with different labels.

For these reasons, having an assertion-control object that forces checking and/or terminating must not make any additional changes to how such a contract assertion behaves.

2.3.2 Never-Checked Assertions

The flip side of the previous concern is the assertion-control object that never allows a checked semantic. In the current world, that means an assertion that is restricted to *only* the *ignore* semantic, but in the future that might also include the *assume* semantic (as described in [P3100R4] and elsewhere).

Such unchecked semantics will all share one important common property: The generated code for these assertions can be functionally identical to the code that would be generated if the assertion is not there at all. This property allows more leeway to remove code-generation implications from such assertions without risking fundamentally changing any semantics related to the contract assertion.

The primary use case is to allow writing contract assertions that are expressed in terms of functions that we cannot evaluate at run time and thus could not provide valid implementations for:

```
template <typename Iter>
void g(Iter begin, Iter end)
  pre<unchecked>( is_reachable(begin,end) );
```

What is `is_reachable` in the above precondition? It is a function that should return `true` if incrementing the first argument will eventually get to the second argument. Such a function is not, howevever, a function for which we can provide a nondestructive implementation for all possible iterator types, or even for many of them.

- For input iterators, attempting to determine if `end` is reachable from `begin` through forward iteration would also consume all values that the iterator provides, leaving nothing for the function to process.
- In the general case, one can never know by iterating if there won't be a future step that does reach the end. Some additional information about the meaning and structure of the iterators must therefore be available to produce a reliable result for `is_reachable`
- In many cases, such as where the iterators are raw pointers, that extra data is not available in the abstract machine at run time. Some sanitizers might track this extra ephemeral information, but that situation is rarely available in all builds so that it could be accessible at run time.

The predicate `is_reachable` is, however, often the property we are looking to check for with functions that take and process a range passed in as a pair of iterators.

To avoid writing code that might depend on a runtime-available `is_reachable`, we would like to have `is_reachable` declared but never defined. To allow that, we want contract assertions such as the above precondition to be able to name but not odr-use functions in the predicate, exactly as if the predicate were an unevaluated operation, such as the operand of a `sizeof` expression. We call such functions *symbolic* functions. Though not useful for runtime checking, static-analysis tools *can* recognize them and make use of their meanings to produce fruitful results.

In some sense, one might consider that this relaxation of the ODR requirement is a change in meaning of the predicate, but this would not match the reality of our software. In general, expressions that never get evaluated have a tendency to be optimized away before they cause a link-time error due to naming things that do not exist. Explicitly relaxing the requirement for the named function to exist allows us to standardize that existing practice and leverage names for which we cannot provide runtime implementations. If the name does resolve to a function, being unchecked will never change its meaning to a different function, and if it does not, we simply allow code to compile and link that *might* otherwise not be able to.

Proposal 12: Allow Symbolic Functions

When a label allows only *unchecked* semantics, the corresponding predicate is an unevaluated operand (and thus does not odr-use functions it names).

One might consider, just like always-checked predicates, rethinking other design aspects of contract assertions for such predicates because they have no possibility of their side effects altering the runtime state of the program. For many of the same reasons as we discussed in the previous section (such as minimizing the friction when changing labels), it seems unwise to pursue such modifications.

2.3.3 ODR Applicability

The other major way in which the ODR impacts assertion-control expressions is how we apply it to the expressions themselves. For the predicates of function contract assertions, we currently require that both clients of a function and the function definition always see ODR-equivalent lists of preconditions and postconditions. Our initial suggestion for labels is that we continue this same requirement for labels and expect that clients of a function and the function itself always know ODR-equivalent assertion-control expressions for each function contract assertion.

Having this consistency between callers and callees allows some ability for implementations to produce better code when, for example, they know that particular assertions will be evaluated with a terminating (or at least noncontinuing) semantic on the other side of a function call.

Achieving this consistency is also a reason to avoid making use of annotations as a mechanism for specifying assertion-control objects because annotations (as introduced by [P3394R4] and suggested for this proposal by [P3831R0]) explicitly don't enforce any consistency across multiple declarations of the same entity and instead simply append all annotations on all visible declarations together into one long sequence when accessed.

Some use cases that we might want to consider in the future do need to have different definitions and different assertion-control objects visible at different points in a program. Function contract assertions visible to only the definition, for example, allow the function definition to take advantage of having assertions where they can't be placed in the body (such as prior to the evaluation of the member initializer list in a constructor) without exposing those assertions to callers that see only an unadorned declaration. Because understanding the assertion-control object that might be applicable to any given evaluation becomes very hard to understand when they can differ based on which declarations have been seen (or not seen), we recommend that such features be carefully specified to opt-in in the future rather than simply removing the restriction that assertions and their control objects must always be consistent for all users of a function.

2.3.4 Standard Library Labels

Because many uses of assertion-control objects are simple applications of the basic functionality, making these labels a common vocabulary provided by the Standard Library is helpful. These common labels will include several basic label types.

- **Explicit Semantic Labels** — For each standard evaluation semantic, provide a label prefixed with `always_` that models `semantic_computation_label` and always returns the named semantic. Explicit semantic labels should be mutually exclusive.
- **Disallowed Semantic Labels** — For each standard evaluation semantic, provide a label prefixed with `never_` that models `allowed_semantics_label` whose `allowed_semantics` member is `evaluation_semantic_set::all()` with the named semantic removed.
- **Review Label** — A label named `review` models `semantic_computation_label` and returns `evaluation_semantic::observe` when a potentially terminating semantic is passed in as the chosen semantic (and otherwise returns the chosen semantic).
- **Always and Never Checked** — Labels `terminating` and `symbolic` specify `allowed_semantics` that contain all terminating and all unchecked semantics, respectively.
- **Runtime-Cost Labels** — Labels `opt`, `audit`, and possibly `default_cost`, that share a `runtime_cost` dimension, identify assertions that are either very inexpensive or very expensive to evaluate.
- **Other Feature Labels** — For each of the facets we support on assertion-control objects, a small library of simple utility types should be provided alongside that facet to aid in the implementation of labels that participate in that facet.

In addition, the header `<contracts>` will include a number of common utilities for dealing with the various facets of assertion-control-objects.

- **Concepts** — This utility will provide the concepts specified for each facet.
- **Concept Check Variables** — Rather than forcing users to use `decltype` to validate that their assertion-control objects satisfy particular concepts, a corresponding variable template prefixed with `is_` should be declared for each concept to easily test an assertion-control object for satisfying a particular type:

  ```
  namespace std::contracts::labels {
  template <auto control_object>
  bool is_assertion_control_object =
    assertion-control_object<decltype(control_object)>;
  static_assert(is_assertion_control_object(empty_label));
  }
  ```

- **Operator `|`** — This utility provides the pipe (`|`) operator to combine assertion-control objects into a compound assertion-control object.

Proposal 13: Standard Labels

Provide a basic set of Standard Library assertion-control labels as well as general utilities for working with assertion-control objects.

Other labels can easily be included in the set provided by the Standard Library, and the names proposed here are obviously subject to further discussion. We expect this list to be more fully specified in future revisions of this paper — or possibly in followup papers — but do not want to distract from the core-language feature by overly focusing now on the naming of Standard Library labels.

2.3.5 Header Dependency

Making use of assertion-control expressions, as we have specified, depends on having the names used in those expressions made explicitly available. The mechanism to do that is by using `#include <contracts>` or `import std`. This method introduces a dependency on the Standard Library when control over contract-assertion behavior is desired (although users can implement valid assertion-control objects themselves with a bit more manual effort and without directly depending on `<contracts>`).

Any feature that might *implicitly* make use of labels must also consider whether such implicit use should depend on having included `<contracts>` before such use. We would suggest that any such feature not force such an implicit dependency but instead simply specify, in the core language, the specific nature of the assertion-control object that would be used without making direct references to the Standard Library.

At some point in the future, we might look into supporting a separate module having *only* assertion-control functionality exported, such as `std.contracts`, to enable the use of assertion-control expressions without needing to include the entire `std` module. Whether such a module is desirable requires more experience with large-scale deployment of modules and with assertion-control objects themselves.

3 Future Directions

Several other possible additional features can be built on top of the proposal we have so far described. Each is useful and important, but we have chosen in this proposal to begin to focus on the basic functionality associated directly with specifying an assertion-control object using an assertion-control expression and to leave more advanced features for future evolution.

3.1 Runtime Selection of Semantics

Our proposed options for allowing control over evaluation semantics focus on manipulating the semantic and restricting it without demanding that the computation of the semantic happen at a time later than it currently happens. In particular, the platform is still in control over whether the determination of semantic happens at compile time, link time, or even at run time.

To have a label that explicitly removes this freedom and forces the semantic to be chosen at a later point (effectively giving up the performance gain for unchecked semantics by making the branch on semantic unremovable), we would want to use a distinct runtime-only member function that behaves much like `compute_semantic` but instead is not allowed to be `consteval` only.

3.2 Ambient Control Objects

A variety of use cases for assertion-control objects dictate that they be specifiable not only on single contract assertions, but also on a range of assertions identified by scope or context:

- all member functions of a particular class
- all functions declared within a particular namespace
- all functions invoked from a particular context

In each of these cases, attaching an assertion-control object to the corresponding context would be helpful. Such a feature, however, would result in multiple sources of assertion-control objects being possible for a single contract assertion since these scopes are not mutually exclusive and since an explicit assertion-control expression may also be present on the assertion. In all such cases, the assertion-control expression used for the contract assertion will be the result of combining the constituent ambient objects with the explicit one on the assertion using `operator|`. If overload resolution fails for that operator, the contract assertion is ill-formed.

As an example, we can introduce an implicit control-object declaration that can be used at class or namespace scope:

```
implicit-assertion-control-directive :
    contract_control ambient < expression > ;
```

Some requirements must be added to ensure that the final assertion-control expression used for each individual contract assertion is well defined and manageable.

- Having multiple such declarations in the same class definition or namespace scope is invalid.
- The ambient assertion-control objects of a function contract assertion are those of the namespace where the first declaration of the function occurs, followed by those of the class definition.
- The ambient assertion-control objects of an assertion statement are those of the enclosing namespace followed by those of the defining class.

3.3 Core-Language Control Labels

Implicit contract assertions for core-language constructs, as explained in [P3100R5] and [P3599R0], are a powerful way to provide standard mechanisms to manage and mitigate the risks of undefined behavior in the C++ language without any need to compromise on the available performance of C++ programs.

Such preconditions can interact with assertion-control objects in two ways.

1. Each type of implicit contract assertion introduced by the Standard should specify a Standard Library label that is an otherwise-empty (except as specified below) object that is the assertion-control object of that contract assertion. These Standard Library labels will then provide two key pieces of functionality:

   (a) a portable way to discuss and configure the evaluation semantic of implicit contract assertions

   (b) a mechanism to have other contract assertions controlled as part of the same group by specifying these labels as their assertion-control objects

   For example, assuming we adopted the implicit contract assertions proposed by [P3599R0], we would then define the following three objects in the `<contracts>` header:

```
namespace std::contracts::labels {
  constexpr auto array_bounds = "std::array_bounds"group;
  constexpr auto nullptr_indirection = "std::nullptr_indirection"group;
  constexpr auto arithmetic_range = "std::arithmetic_range"group
}
```

Compiler options that allow specifying semantics for contract assertions with specific labels could then name these labels to control the checking of core-language operations within a TU.

Note that we might also specify that other group labels, such as a more generic `"std::bounds"` group, are also applied to these implicit contract assertions. Because a contract assertion may be part of many groups, we can provide varying levels of granularity for the control of their checking.

2. Within particular contexts, it can be helpful to attach new labels to particular types of implicit contract checks. For example, in a particular context, bounds checking might be found to be excessively impactful to performance, so adding the `audit` label to all implicit bounds checks in a certain scope can help achieve the performance needed in a piece of hot-path code while leaving the check available to be enabled in slower builds that are used for testing.

   This contextual control of implicit behavior could be done at a scope using another variation of the *implicit-assertion-control-directive*:

   builtin-assertion-control-directive :
   `contract_control core` *assertion-group-expression* `|=` *control-expression* `;`

   The *assertion-group-expression* is a string literal that will be used to match the assertion-control group (see Section 2.2.5) of the implicit contract assertions within the scope.

   The *control-expression* is an expression that will be combined with all implicit contract assertions in the enclosing scope that are in the named assertion group. This combination, as with all other ambient assertion-control objects, will be done using the `|` operator.

   As with other assertion-control expressions, both the assertion-group-expression and the control-expression will use the name-lookup rules for control expressions, and thus they will respect *assertion-control-using-directives*.

   For example, let's say that we want to provide our own label that controls all three of the implicit contract-assertion types from [P3599R0]. To do so, we would add the following three directives at namespace scope near the top of our TU:

```
#include <contracts>

assert_group_label<"my::safety_checks"> my_safety_checks;

contract_control core array_bounds.name        |= my_safety_checks;
contract_control core nullptr_indirection.name |= my_safety_checks;
contract_control core arithmetic_range.name    |= my_safety_checks;
```

   Then, we can again turn to our compiler command line to select a semantic for anything with the `my_safety_checks` label.

A header that included these directives could, for example, encode the requirements of some safety standard (such as MISRA) by specifying those groups that must have checked semantics using the syntax above.

Similarly, if we instead used a semantic-computation label, we could more directly control the evaluation semantics of these implicit contract assertions within a scope.

When placed in a class definition, these directives will apply to all expressions within that class definition and any member function definitions of that class. When placed at namespace or function scope, these directives will apply to all expressions that occur lexically after the directive in the same scope.

3.4 Build Environment

One of the primary ways in which existing macro-based facilities benefit from being implemented using the C++ preprocessor is that their behavior can be *controlled* from the command line when compiling by providing definitions (or not providing such definitions) to various macros.

The most common example of such a control is the `NDEBUG` macro used to control the behavior of the `assert()` macro.

[P2755R1] described the idea of a *build environment* that would provide a map of values that could be specified on the command line and then accessed locally during constant evaluation using a new `consteval` API. Such an API would largely bridge the remaining gap between the abilities of the preprocessor and those of constant evaluation.

On the other hand, such an API would be hugely impactful in several ways that need exploration before it could be reasonably adopted. Its adoption can also be done at a later date, and assertion-control expressions could easily take advantage of it as soon as it does become available. We therefore do not consider such an API fundamental to the basic proposal of labels and will likely instead pursue it separately at a future time.

4 Alternate Design Considerations

A variety of alternate directions for this proposal have been suggested and considered. Some of those alternatives are worth saying a little more about, both to clarify our design and to avoid needing to revisit the same suggestions in the future.

4.1 Syntax

A range of syntax options have historically been considered for this proposal.

- Originally, a syntax that evolved more directly from the C++20 syntax for contract assertions was being pursued. In C++20, label-like functionality was available through the use of `default`, `audit`, and `axiom` in the syntactic space between the contract introducer (`pre`, `post`, and `assert`) and the : that preceded the predicate.

  ```
  int f(int *p)
    pre audit (p)
    post default new (r : r == *p);
  ```

This syntax had to contend with not allowing any more involved expressions and needing to disambiguate from the return name introducer that (in the C++20 Contracts syntax) used the same space for an arbitrary identifier. The syntax did, however, have the advantage of freely allowing the use of keywords as label names since they did not have any other meaning in this space.

Though achievable, attempting to introduce user-defined labels with the same amount of extensibility that using full expressions allows us to express requires inventing significant new technology for that purpose — how to map the identifiers used to types and functions, how to generalize with input values such as strings or numbers, and so on — and is not simple to design, implement, or understand. Using more normal name-lookup and expression rules gives us much greater flexibility and much more consistent behavior with the rest of the language.

- Other than `<>` that imply a parameterization of the behavior of the introducer, other syntax to surround the assertion-control expression has been suggested. Two options, `[]` and `{}`, might be viable, but neither has the same intuitive meaning that we believe `<>` properly conveys. The use of braces in particular would prohibit future features such as procedural interfaces (described in [P0465R0] and more recently in [P2755R1]). Admittedly, the use of `<>` inhibits a future new kind of contract assertion that consists of only an introducer followed by something enclosed in `<>`s, but at this stage, no one has suggested any such kind of contract assertion.
- The use of annotations has been suggested as well (by [P3831R0]):

  ```
  int f(int *p)
    pre [[=audit]] (p)
    post [[=review]] (r : r == *p);
  ```

  Since we already allow attributes in this location, annotations are also allowed and would appertain to the contract assertion. On the other hand, such annotations must retain all the same basic semantic and behaviors as annotations anywhere else in the language; bespoke rules for annotations on contract assertions would break all other non-control-object-related uses of annotations in this place.

  Those semantics of annotations are, however, problematic.

  – They allow any structural objects to be used and thus do not allow for any reasonable way to warn or indicate an error when a useless value is accidentally written. Distinguishing whether `[[=5]]` is meant as an instruction to some other annotation processing step or is a mistake is not possible because it is not a valid assertion-control object.

  – Redeclarations of entities with annotations simply append the redeclared annotations (in an arbitrary order) from all reachable declarations. Enforcing more stringent requirements is not possible because that would again prohibit other potential uses of annotations for unrelated reasons. Simply combining the repeated annotations would then be problematic because it would result in arbitrarily more complicated expressions that might be invalid or useless when combined. These differing combinations would also then result in, effectively, ODR violations whenever callers happened to see multiple declarations of the same function with contract assertions on them.

For the reasons above, we have thus continued with the template-like syntax for assertion-control objects we have described here.

4.2 Combination

Because we want to leverage the ability to define how combination of labels works by using regular C++ (and thus primarily within the Standard Library, not the core language), we need to pick a mechanism for combining labels that relies on either some Standard Library function or operator overloading. Using a function would lead to a much more cumbersome syntax and bring with it little benefit.

Other operators have been suggested, such as `&`, `^`, or others, but none convey a consistent meaning for all potential labels. Even `|` might be read as *or* and indicate that, for example, the allowed semantics of `a|b` are those of either `a` or `b`. On the other hand, `|` is the operator we use for connecting together ranges and multiple range adapters, and that is the precedent we are relying on by choosing `operator|` as our default combining operator for labels.

Another suggestion is that we allow multiple labels to be specified on a contract assertion separated by commas (`,`), but that choice simply introduces a second syntax to achieve the same goal. It is also our experience that any use of overloading `operator,` leads to nothing but increased frustration, so we are avoiding that approach entirely.

4.3 Predicate Evaluation Behavior

Contract assertions declare that something must be true. Needing to read and understand arbitrarily increasing amounts of code before knowing what a `pre` or `post` is actually saying must be true removes fundamental benefits of using Contracts in the first place. Therefore, any changes to Contracts that impact the meaning of a predicate or the places where the contract assertion might be evaluated should be accomplished with syntax orthogonal to the assertion-control expression.

- `const`-ification occasionally will add a significant burden on the users of Contracts when integrating with an existing library that is significantly not `const`-correct. One suggested potential workaround to this issue (discussed in [P3261R1]) is a way to annotate a contract assertion such that it does not apply `const`-ification in its predicate. Though having a scoped operator more selectively remove `const`-ification from single expressions would be better, any tool that removes `const`-ification from the entire predicate should be a separate keyword added to the contract assertion independently of the control expression.

- Whether exceptions escaping the predicate should be treated as violations again impacts the meaning of the predicate itself. Allowing in-code altering of this default behavior on individual contract assertions should, therefore, be done with a separate annotation.

  Note, however, that this behavior can also be altered with a label that introduces a local violation handler that rethrows exceptions *any time* the `detection_mode` is `evaluation_exception`. Given a label with such a local violation handler inlined, a compiler could reasonably remove any exception frame from such contract assertions and simply allow the exception to propagate further up the stack.

- Future proposals, such as [P3097R1], will allow for function calls where there are both caller-facing and callee-facing function contract assertions. As those proposals currently stand, these function calls will still consist of all the preconditions and postconditions on particular function declarations. A mechanism to mark `pre` and `post` as applying to just caller-facing contract assertions (i.e., when the declaration is used for virtual dispatch) or just callee-facing contract assertions (i.e., when that definition is selected as the final overrider) can be useful for certain designs. Such a mechanism would not, however, just be changing the semantics of certain evaluations to *ignore* but would instead be dictating that the assertions are not considered (and thus not evaluated) at all in some uses of the function. Because of this difference, a distinct syntax should be used.
- Generic code might often want to refrain from expanding certain contract assertions if they would not be applicable for particular template arguments. One suggestion states that adding `requires` clauses to contract assertions would enable this functionality, and we maintain that such functionality should remain separate from assertion-control objects. Again, the reason is that requires clauses would remove the emitting of the assertion entirely, not just have it be evaluated with the *ignore* semantic, and thus should be a separate syntax that must be understood along with the predicate to get the meaning of the contract assertion.
- The number of evaluations to which a contract assertion might be subjected is outside the control of the code itself and is entirely implementation defined. Because this property does not alter the meaning of the assertion (as long as the number of evaluations is not reduced to 0), adding a new facet to control this aspect of evaluation would be possible. Such a facet would, however, by necessity, remove certain lexical guarantees that might be assumed when reading the code, so we have avoided making this facet part of our proposal.
- Whether the violation handler, when evaluated, should be allowed to throw an exception is not about the meaning of the contract assertion or its predicate, but about the evaluation behavior. A facet to control this behavior would be reasonable, though again it does not have enough use cases to propose now.

5 Wording

Wording will be provided once consensus has been achieved on the shape and scope of the feature's design.

Other than the grammar changes, the core language changes will touch a few places.

- A new section, `[basic.contract.control]`, will describe the assertion-control expression, when assertion-control objects are created, and the concepts they must satisfy.
- In `[basic.contract.eval]`, we will describe the effects of the assertion-control object when it satisfies the `allowed_semantics_label`, `semantic_computation_label`, or `local_violation_label` concepts.
- A new section, `[expr.contract_control]`, will describe contract-control expressions.
- `[namespace.udir]` and `[namespace.udecl]` will add support for the optional `contract_control` to using directives and using declarations.

- `[basic.lookup]` will add those names introduced by using directives and declarations with `contract_control` to those contexts where they should be included, i.e., in assertion-control expressions and in the operand of a `contract_control` operator.

The library section will then need major changes.

- A new section in the library, `[support.contract.control]`, will describe all the concepts specified in this paper (though probably as exposition-only concepts) and how the combined label object returned by `operator|` will satisfy those concepts when its operands do.
- Another new section in the library, `[support.contract.labels]`, will describe all the Standard Library labels that we choose to introduce.

6 Conclusion

In this paper, we have presented many layers that result in a power extension to the Contracts facility proposed in [P2900R14]. These extensions vastly extend the use cases supported by Contracts to include those that will be necessary for extensive real-world deployment of Contracts in C++ codebases around the world for many years to come.

A Example Implementation

A.1 `std::contracts::labels::operator|`

An implementation will be provided in a future revision of this paper, making explicit what behaviors we expect when combining labels that satisfy the concepts proposed in this paper. An in-progress partial implementation that supports some of the concepts discussed here can be found at `https://godbolt.org/z/xjKxfe9sf`.

Acknowledgments

Thanks to the many people who have helped contribute to [P2900R14]; that foundation allows the proposals in this paper to be developed.

Thanks to Iain Sandoe, Timur Doumler, Peter Bindels, Ville Voutilainen, Yihe Li, Bengt Gustafsson, and John Lakos for feedback on this paper.

Thanks to Lori Hughes for reviewing this paper and providing editorial feedback.

Bibliography

[P0465R0] Lisa Lippincott, "Procedural function interfaces", 2016
`http://wg21.link/P0465R0`

[P2098R0] Walter E Brown and Bob Steagall, "Proposing `std::is_specialization_of`", 2020
`http://wg21.link/P2098R0`

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

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

[P3097R1] Timur Doumler and Joshua Berne, "Contracts for C++: Virtual functions", 2025
https://isocpp.org/files/papers/P3097R1.pdf

[P3099R0] Timur Doumler, Peter Bindels, and Joshua Berne, "Contracts for C++: User-defined diagnostic messages", 2025
http://wg21.link/P3099R0

[P3100R4] Timur Doumler and Joshua Berne, "A framework for systematically addressing undefined behaviour in the C++ Standard", 2025
http://wg21.link/P3100R4

[P3100R5] Timur Doumler and Joshua Berne, "A framework for systematically addressing undefined behaviour in the C++ Standard", 2025
http://wg21.link/P3100R5

[P3261R1] Joshua Berne, "Revisiting `const`-ification in Contract Assertions", 2024
http://wg21.link/P3261R1

[P3271R0] Lisa Lippincott, "Function Usage Types (Contracts for Function Pointers)", 2024
http://wg21.link/P3271R0

[P3391R1] Barry Revzin, "`constexpr std::format`", 2025
http://wg21.link/P3391R1

[P3394R4] Daveed Vandevoorde, Wyatt Childers, Dan Katz, and Barry Revzin, "Annotations for Reflection", 2025
http://wg21.link/P3394R4

[P3491R3] Barry Revzin, Wyatt Childers, Peter Dimov, and Daveed Vandevoorde, "`define_-static_{string,object,array}`", 2025
http://wg21.link/P3491R3

[P3599R0] Joshua Berne and Timur Doumler, "Initial Implicit Contract Assertions", 2025
http://wg21.link/P3599R0

[P3831R0] Yihe Li, "Contract Labels Should Use Annotation Syntax", 2025
http://wg21.link/P3831R0