Controlling Contract-Assertion Properties

Document #:	P3400R3
Date:	2026-05-11
Project:	Programming Language C++
Audience:	EWG

Reply-to: Joshua Berne <jberne4@bloomberg.net>

Abstract

C++26 Contracts, adopted with [P2900R14], provides a framework for specifying and checking contract assertions whose behavior is controlled through implementation-defined build configurations.

This paper introduces *assertion-control objects* — compile-time objects, composed from `constexpr` *labels*, that let developers customize contract-assertion behavior directly in C++ source code. Labels participate in independently opt-in *facets* that expose hooks to customize every aspect of contract-assertion evaluation and violation handling behavior. Accompanying Standard Library additions provide ready-made labels covering the most common use cases, while the underlying framework is designed to be extended with user-defined labels and new facets over time. The functionality enabled by this proposal is essential to the unhindered and widespread adoption of Contracts across the many domains in which C++ is used.

Contents

Revision History

Revision 3

— Major restructuring: new Introduction, Design, Glossary, expanded Standard Library Additions sections

— Clearer presentation of possible layering of adoption

— Extensive editorial cleanup

— Assertion-control using declarations (of identifiers, not namespaces) removed

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 for expressing conditions that must be `true` at specific program points. With C++26 Contracts, however, the behavior of the program when a contract assertion is evaluated is controlled entirely in an implementation-defined manner, i.e. through command line options and build configurations outside of the source code.

In this paper, we propose *assertion-control objects*: compile-time objects, composed from one or more `constexpr` *labels*, that allow developers to customize individual contract assertions directly in source code. Each label can participate in one or more independently opt-in dimensions of behavior called *facets*. Among other things, facets allow labels to:

— restrict or compute the evaluation semantic,

— identify assertions as part of named groups for build-system control,

— provide local violation handlers,

— manipulate the comment or message in the `contract_violation` object, and

— expose data to the violation handler through a type-erased query interface.

Building on this core-language facility, we add a variety of ready-made labels to the Standard Library `<contracts>` header covering the most common use cases. These tools provide a power user-facing facility alongside common building blocks that will aid those who need to write their own labels.

The primary goal of this effort is not to simply allow complete in-source control over the behavior of contract assertions, but to enable the full spectrum of flexibility in how the developer and build engineer can collaborate to maximize the effectiveness of how they apply Contracts.

1.1 Making Use of Labels

The needs of a user leveraging Contracts in their software evolve over time as they attempt to check more of their code and use Contracts in more places. As these needs evolve, labels provided by the `<contracts>` header will solve many of the limitations that arise with C++26 Contracts.

Grouping Contract Assertions Labels can identify that a contract assertion is part of a particular named group that our compiler will allow us to control through our build configuration:

```
void h(int *p)
  pre<"mylib::pointers"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.

Combining Multiple Groups Groups are often formed for very different and orthogonal reasons, which naturally leads to assertions belonging to more than one group. Labels can be combined using the pipe (`|`) operator, so placing an assertion in multiple groups is straightforward:

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

Controlling Cost of Checking. One primary axis that leads to wanting to group assertions separately is based on the cost of evaluating a predicate — very fast checks are frequently not worth the risk of turning off, while very expensive checks might never be viable to run in real deployments.

To capture that nuance, the Standard Library provides labels to identify these groups so that a build engineer can control assertions accurately:

```
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
```

The behaviors of the above assertions will then be tied to additional configuration options provided by a compiler, generally defaulting the `opt` assertions to being checked while the `audit` ones are only evaluated in specific builds where the user has explicitly asked to pay the performance cost.

Deploying New Assertions Safely. Adding new assertions to existing code is one of the most challenging aspects of using Contracts as part of a large-scale workflow. A newly introduced contract assertion can be marked so that it will be *observed* if it would otherwise be *enforced* or *quick-enforced*, allowing it to be more easily deployed to production environments:

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

Note that such a label will not force the contract to be checked at all; that choice is still made based on the build configuration. What `review` does is change the semantic to *observe* when an enforcing semantic would have been chosen.

Combining Labels. The same `|` operator also combines labels with different purposes. For example, combining `audit` and `review` produces an assertion that is only checked in specific builds *and* is observed rather than enforced in those builds:

```
void f(int *begin, int *end)
  pre<audit | review>( std::is_sorted(begin,end) );
    // Slow check is on in only some builds
    // and is observed in those builds.
```

Requiring Enforcement. Some development processes demand that control never flow into code after a contract violation. To facilitate that, the `terminating` label requires that an assertion always uses an enforcing (i.e., *enforce* or *quick-enforce*) semantic:

```
void g(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.

Library Hardening. Assertions can be configured to use a terminating semantic when a hardened Standard Library has been selected — including when compiling the Standard Library implementation itself:

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

When using a hardened Standard Library the above precondition will be enforced — the same guarantee provided by all hardened preconditions in the Standard Library.

1.2 Building New Labels

As the use of Contracts widens, and when supporting larger shared codebases or widely used libraries, a wider set of possibilities opens if you dig deeper and begin to write your own label types.

The Simplest Label. The starting point is the label that does nothing. All assertion-control objects must satisfy the `assertion_control_object` concept, which requires only a nested member typedef. The simplest valid label is the empty label, which participates in no facets:

```
struct empty_label_t {
  using assertion_control_object = empty_label_t;
};
constexpr empty_label_t 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:

```
void f1()
  pre<empty_label>(true)
  post<empty_label>(true)
{
  contract_assert<empty_label>(true);
}
```

Since it is the empty label we are applying here, the behaviors of the above contract assertions are the same as the behaviors with no assertion-control object present.

Implementing a Facet. To make a label that actually *does* something, its type must satisfy one or more facet concepts. For example, the `review` label models `semantic_computation_label` by providing a `compute_semantic` member function:

```
namespace std::contracts::labels {
struct review_t {
  using assertion_control_object = review_t;

  constexpr evaluation_semantic
  compute_semantic(evaluation_semantic in) const {
    if (is_terminating(in))
      return evaluation_semantic::observe;
    return in;
  }
};
constexpr review_t review{};
}
```

The type introduced above participates in the facet, and the `constexpr` variable named `review` gives us an easy way to get an object of that type to use in our assertion-control expressions.

Restricting Semantics. Labels can also restrict rather than compute semantics. The `allowed_semantics_t` template provides a compile-time-fixed set of allowed semantics. The `terminating` label, for example, is simply a specialization that allows only enforcing semantics (and fails to compile if combined with another label that would produce a non-enforcing semantic):

```
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...};
};
constexpr allowed_semantics_t<
    evaluation_semantic::quick_enforce,
    evaluation_semantic::enforce> terminating;
}
```

By using a label that restricts the allowed semantics we get an easily auditable statement of the checking behavior our program will have. Even when combined with other labels with `operator|` the chosen semantic for a contract assertion with these labels will always be one of those in the allowed set.

How Labels Combine. When labels are combined using `operator|`, the resulting assertion-control object participates in the union of the facets of its constituents, and fail to compile when attempting to combine incompatible labels. How exactly the facets are combined — intersection of allowed semantics, chaining of `compute_semantic` calls, and so on — is governed by each facet's combination rule, as described in later sections.

Implementing Library Hardening. 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 {
  using assertion_control_object = hardened_t;

  constexpr evaluation_semantic
  compute_semantic(evaluation_semantic in) const {
#if   _LIBCPP_ASSERTION_SEMANTIC == _LIBCPP_ASSERTION_SEMANTIC_IGNORE
    return evaluation_semantic::ignore;
#elif _LIBCPP_ASSERTION_SEMANTIC == _LIBCPP_ASSERTION_SEMANTIC_OBSERVE
    return evaluation_semantic::observe;
#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();
#endif
  }
};
constexpr hardened_t hardened{};
}
```

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

1.3 More Advanced Uses

As part of the core language feature that facilitates the Standard Library's labels, and for even more use cases, there is of course more that can be done with labels to apply those more advanced features to your own use cases.

Name Lookup for Labels. Standard library labels are provided in the `std::contracts::labels` namespace. The `<contracts>` header includes a new kind of using directive that makes these names available within assertion-control expressions without polluting the enclosing namespace:

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

This directive introduces names only into assertion-control expression lookup, not into normal name lookup.[1]

Working With Labels Outside Assertions. Because assertion-control expressions are evaluated at compile time, we can capture their values and give different names to labels using all the normal facilities of C++. To make that easier, we also introduce 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 f2(int *p)
  pre<general_group>(p != nullptr);
}
```

Outside of the `mylib` namespace we would fully qualify the reference to `general_group`:

```
namespace other {
void f3(int *p)
  pre<mylib::general_group>(p != nullptr);
}
```

This operator not only lets you declare your own labels easily to share with clients, it also provides a great tool for testing the behavior of label types.

Symbolic Predicates. Labels can also be used to write symbolic predicates that do not have a runtime implementation. Such predicates will never be checked at run time but might be checked by an external static analyzer that understands the predicate:

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

Since there's (currently, on most platforms) no reasonable way to implement a check like `is_reachable` in C++, the above predicate must not be enabled at runtime. A wide range of tools, however, do enough analysis to identify potential violations of such predicates at compile time — and the precondition gives them exactly the hook they need to know the potential problems to look for without needing to parse implementations that might not even be available.

2 Design

This section discusses high-level design considerations that shaped this proposal.

2.1 Building Labels and Facets

A central goal of this proposal is that users can define their own labels with arbitrary logic — not just selecting from a fixed menu of compiler-provided options, but fully participating in that process to whatever extent they desire. To achieve this, labels must be objects whose types can

[1] The `contract_control` keyword is used to identify using directives that apply to assertion-control expressions as well as for the operator described next.

carry member functions and data, and the logic they encode must execute at compile time so that it can influence code generation. These requirements lead naturally to using compile-time evaluation as the mechanism: a label is a `constexpr` object, and the compiler evaluates its member functions during compilation to determine how the contract assertion will behave.

An equally important goal is that labels must compose. A user who wants an assertion to be both *reviewed* and part of a named *group* must be able to write `pre<review | "mygroup"group>(...)` without either label needing to know about the other. This composability requirement is what gives rise to *facets*: each facet is an independent dimension of behavior, identified by a concept, and the combining operator knows how to merge each facet independently. A label that only restricts the evaluation semantic does not need to provide — or even know about — the group-identification interface, and vice versa.

Because participation in each facet is opt-in via a concept, new facets can be added in the future without breaking any existing labels. A label written today that satisfies only `semantic_computation_label` will continue to work unchanged when a new facet is introduced tomorrow — it simply will not participate in the new facet, and the default behavior for that facet will apply.

2.2 Restricting vs. Computing Semantics

Two of the most important facets — `allowed_semantics_label` and `semantic_computation_label` — both influence the evaluation semantic, but they do so in fundamentally different ways and serve different purposes.

The `allowed_semantics` facet *constrains*: it declares which evaluation semantics are acceptable for the associated contract assertion. This is a static, unconditional property of the label. When two labels with allowed-semantics sets are combined, the result is the intersection of both sets — the combined assertion is at least as restrictive as either constituent. If the intersection is empty, the program is ill-formed. This facet is essential for labels like `terminating` (which must never be evaluated with an unchecked semantic) and `symbolic` (which must never be checked at run time).

The `compute_semantic` facet *transforms*: it takes the configured semantic as input and produces a (possibly different) computed semantic as output. This is a function, not a set, and it can encode arbitrary compile-time logic. When two labels with `compute_semantic` are combined, their functions are chained left to right: the output of the first becomes the input to the second. This facet is essential for labels like `review` (which maps enforcing semantics to *observe*) and `hardened` (which selects a semantic based on the library's hardening configuration).

These two facets must coexist because neither subsumes the other. A label might want to both restrict *and* transform: for example, `review | terminating` restricts the allowed set to enforcing semantics (via `terminating`) and then maps those to *observe* (via `review`). The result is ill-formed because `observe` is not in the allowed set — exactly the right answer, detected at compile time. Making these two facets independent and composable ensures that such interactions are caught statically rather than silently producing surprising behavior.

2.3 The Semantic Selection Process

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. Labels also allow implementors to group related assertions by name so that they can be controlled in unison. 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_semantics_label` (see Section 3.3.1), 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 3.3.5), 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 *configured 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 configured 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 configured semantic to effective semantic.

1. If the assertion-control object is a `semantic_computation_label` (see Section 3.3.2), then the configured semantic will be passed to the `compute_semantic` member function of the assertion-control object, and its result — the *computed semantic* — becomes the *effective semantic*; otherwise (as is the case for all contract assertions in C++26), the effective semantic is the configured semantic.
2. If the assertion-control object is an `allowed_semantics_label` (see Section 3.3.1), 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.

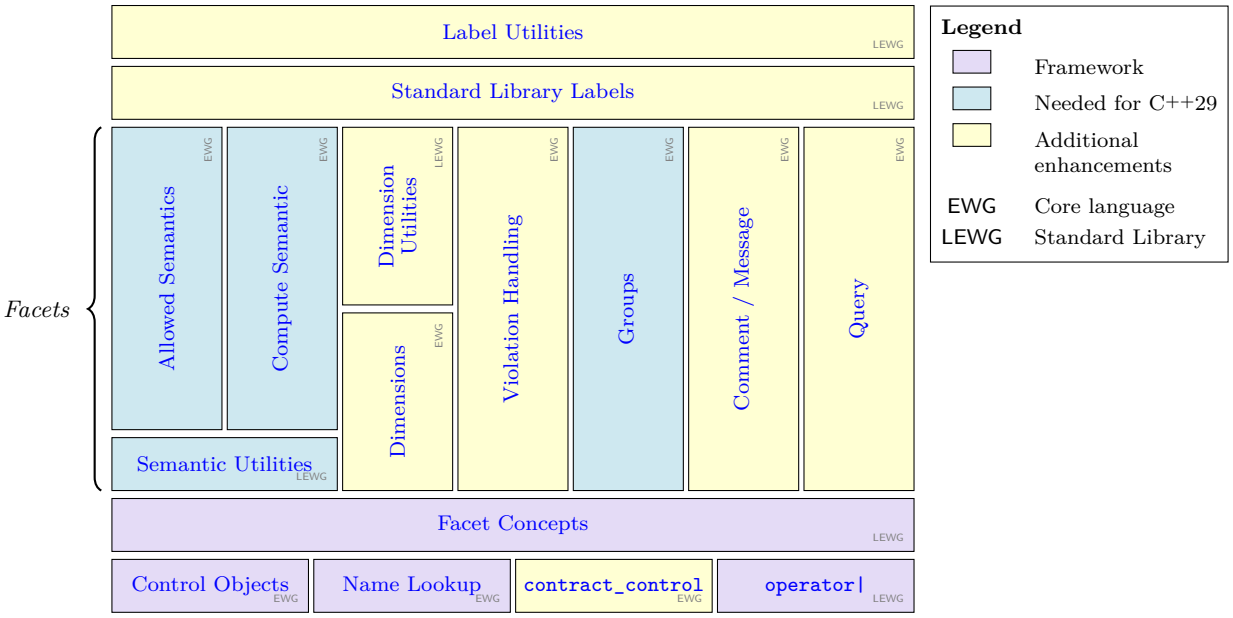
1. A `contract_violation` object will be populated.
2. If the assertion-control object is a `compute_comment_label` or a `compute_message_label` (see Section 3.3.6), the `comment` and `message` fields generated by the compiler for the `contract_violation` object will be passed through the corresponding 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.

3 Proposal

This proposal breaks down into a fairly large number of individual components that build on one another to provide a feature-rich user experience when using Contracts. The following diagram shows how they relate to one another:



At the base is the *framework*: the assertion-control expression syntax, the `assertion_control_object` concept, name-lookup rules, and the `operator|` combining mechanism. These are the core-language and library pieces needed to attach user-defined objects to contract assertions and combine them.

Connecting assertion-control objects produced with the framework to different aspects of the existing behavior of contract assertions are *facets* — independently opt-in dimensions of behavior that an assertion-control object may participate in by satisfying a particular concept. Each facet controls one aspect of the contract-assertion evaluation process, and each can be adopted on its own schedule.

Above the facets sit the *Standard Library additions*: ready-made labels, utility types, and concepts so that most users never need to write a label type of their own.

The components span two areas of WG21 responsibility: the core-language pieces (syntax, name lookup, facet semantics) require EWG approval, while the Standard Library components (concepts, `operator|`, labels, utilities) require LEWG review. Each box in the diagram is annotated with the responsible group.

Although this paper presents a complete design, these parts can be adopted incrementally.

— The core framework pieces are needed to deliver any part of this proposal, and importantly have been designed to enable all of the parts to be delivered eventually.

— In the immediate term as we decide what we want to adopt now, we should consider what features to prioritize for C++26. In [P3850R0] a plan for that set of features is being proposed, and those features that fully enable the control of evaluation semantics are a core part of that plan. Providing these facilities will close a significant gap in functionality that was left out of the C++26 Contracts MVP ([P2900R14]) precisely so the framework proposed here can deliver that functionality.

— The remaining features of this proposal all have real use cases that will improve the utility of Contracts in C++ and enable additional use cases — but none are as essential to target

immediately. These can be pursued as part of a single proposed deliverable or separated out to pursue independently where time and demand can dictate different schedules.

The three adoption categories are indicated by the background colors in the diagram above.

3.1 Assertion-Control Objects

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; in addition, the type 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 3.3.7) 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 3.3.2. 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 regard to name lookup, overload resolution, and other behaviors.

Proposal: 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 3.3). 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.

3.2 Selection

For assertion-control objects to be practical, the names used in assertion-control expressions must be easy to write without introducing conflicts in the enclosing code. This section describes the name-lookup mechanisms that make common labels available concisely, the `contract_control` operator for using the same lookup outside of assertions, and the (Standard Library) `operator|` mechanism for combining labels.

3.2.1 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. These

directives can include an entire namespace, such as the directive that is in `<contracts>`:

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

Grammatically, this is just a new optional addition to the existing using directive:

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

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.

— We previously considered adding the ability to also add the optional `contract_control` to using declarations, but that introduces significantly more challenge both in terms of specification, usage, and implementation. Using declarations share much of their behavior with `typedef`, and altering the set of discoverable types within a single namespace based on context is significantly more difficult for little clear benefit.

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: Contract Control Using Directives

Add the new keyword `contract_control` that can be optionally added to *using-directives* that will then be used only within assertion-control expressions.

3.2.2 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: 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.

3.2.3 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 chaining behavior 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 B.7, 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: 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.

3.3 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.

3.3.1 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;
  constexpr bool empty() const;

  // Modifiers
  constexpr void clear();
  constexpr void insert(evaluation_semantic semantic);
  constexpr void insert(std::initializer_list<evaluation_semantic> ilist);
  constexpr void remove(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 a configured 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 configured 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 configured 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: 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 3.5.1.

3.3.2 Choosing Evaluation Semantics

Throughout this section and the pseudocode that follows, we use three terms for the evaluation semantic at successive stages of the selection process:

— The *configured semantic* is the implementation-defined semantic selected by the build system (command-line flags, configuration files, etc.) before any label logic runs.

— The *computed semantic* is the result of passing the configured semantic through the label's `compute_semantic` function, if present.

— The *effective semantic* is the final semantic used to evaluate the contract assertion, after verification against the `allowed_semantics` set.

When no `semantic_computation_label` is involved, the configured semantic is used directly as the effective semantic, matching the current behavior of C++26 Contracts.

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 configured 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 transform the configured semantic into a computed semantic 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 the configured semantic has been determined, it is passed to the `compute_semantic` function of the assertion-control object, and the result — the computed semantic — becomes the effective 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, the configured semantic will be transformed through `compute_semantic` to produce the computed semantic. Other compilation choices, such as delaying the configured 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 configured semantic — to determine the full mapping from configured to effective 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 configured semantic is fully determined at compile time, and thus we can evaluate the entire pipeline from configured to effective semantic as a constant evaluation:

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

  // Determine the configured semantic based on command-line
  // configuration.  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 configured = get_configured_semantic(control_object);

  // Compute the effective semantic by passing through the label.
  constexpr evaluation_semantic effective = [&] {
    if constexpr (semantic_computation_label<decltype(control_object)>) {
      constexpr evaluation_semantic computed
                                    = control_object.compute_semantic(configured);

      // Verify the computed semantic is in the allowed set.
      if constexpr (allowed_semantics_label<decltype(control_object)>
                && !control_object.allowed_semantics.contains(computed)) {
        static_assert(false);
      }
      return computed;
    }
    return configured;
  }();

  // Evaluate the contract assertion with the effective semantic.
  switch (effective) { /* ... */ }
}
```

A very similar bit of pseudocode applies if the configured semantic is not known at compile time, but we instead must build the full mapping from every possible configured semantic to its

effective semantic (using 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 effective = get_configured_semantic(control_object);

  // At run time, we cannot pass effective to the constexpr
  // compute_semantic function.  Instead, expand over all
  // possible configured semantics and map each one at compile time.
  template for (constexpr auto s : std::meta::enumerators_of(^evaluation_semantic)) {

    // Skip semantics not in the allowed set.
    if constexpr (allowed_semantics_label<decltype(control_object)>
              && !control_object.allowed_semantics.contains([:s:])) {
      continue;
    }

    if (effective == [:s:]) {
      if constexpr (semantic_computation_label<decltype(control_object)>) {
        constexpr evaluation_semantic computed = control_object.compute_semantic([:s:]);
        if constexpr (allowed_semantics_label<decltype(control_object)>
                  && !control_object.allowed_semantics.contains(computed)) {
          static_assert(false);
        }
        effective = computed;
      }
      break;
    }
  }

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

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.[2]

Proposal: Compute Semantics

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

3.3.3 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:

[2] See the discussion of a build environment in Section 4.4 for a suggestion of a possible future proposal that might enable this kind of configuration without the use of the preprocessor.

```
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[3]:

```
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.

Proposal: 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.

3.3.4 Local Violation Handling

Customizing contract-violation handling for specific contexts and specific contract assertions is an oft-requested feature that arises in several distinct situations:

— Libraries that need to be in complete control of violation handling for security reasons — for example, to prevent any information about the violation from reaching a user-provided handler that might log or transmit it.

— Libraries that want to deliver library-specific diagnostic information — such as what request was being processed or what the subsystem configuration was — in addition to, or to a different endpoint from, the user's chosen contract-violation handler.

[3]This implementation assumes the presence of a Standard Library trait `is_specialization_of`, such as proposed in [P2098R1], or similar bespoke functionality.

— Libraries that want to alter the handling of specific categories of exceptions thrown from contract predicates — for example, rethrowing allocation failure exceptions (`std::bad_alloc`) that escape the evaluation of the predicate rather than treating those as contract violations.

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.[4]

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:

```
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

[4]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.

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: 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 rethrows `std::bad_alloc` exceptions that escape the contract predicate — rather than treating them as contract violations — can be implemented as a local violation handler that inspects the in-flight exception:

```
struct rethrow_bad_alloc_t {
  using assertion_control_object = rethrow_bad_alloc_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) {
      try {
        throw;  // rethrow the exception that escaped the predicate
      } catch (const std::bad_alloc&) {
        throw;  // propagate bad_alloc to the caller
      } catch (...) {
        // not a bad_alloc; fall through to not_handled
      }
    }
    return {};  // default-constructed value is not_handled;
                // delegate to the next handler in the chain
  }
};
constexpr rethrow_bad_alloc_t rethrow_bad_alloc{};
```

Note that the return type above is `std::integral_constant<violation_handled, violation_handled::not_handled>` rather than simply `violation_handled`. Because the return type carries the value statically, the implementation can determine at compile time that this handler never fully handles a violation. With that knowledge, the compiler need not emit a runtime branch to test the return value — it can unconditionally continue to the next handler in the chain. A label that *does* sometimes return `handled` would instead declare its return type as `violation_handled` so that the runtime value is inspected.

3.3.5 Grouping Assertions

One of the simplest use cases 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: 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 indicating 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.)

3.3.6 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 not be disclosed to 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 [P3099R2]. 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.[5] When we do adopt [P3099R2], 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 simplest implementation is to produce a fixed 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 [P3391R2]) 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)
  // consteval (not constexpr) forces compile-time-only calls; define_static_string
  // requires a constant-evaluation context, so runtime calls would be ill-formed.
  {
```

[5] The `compute_message_label` concept and functionality will be added to the wording of this paper if [P3099R2] lands first.

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

The string message introduced in [P3099R2] 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: 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.

3.3.7 Integration with `contract_violation`

Allowing the assertion-control object to communicate additional information to a contract-violation handler has many useful applications.

— 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 would 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.

[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.

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 3.3.3) so that it is not combined with other labels that answer for the same key.

The `queryable_label` concept requires an appropriate `query` member function on the 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 required 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 null-terminated strings:

```
class owner_label {
public:
  using assertion_control_object = owner_label;

  constexpr owner_label(const char* name, const char* address)
  : d_name(name)
  , d_address(address)
  {}

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

  // ...

private:
  const char* d_name;
  const char* 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* owner_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 char* name    = static_cast<const char*>(name_ptr);
    const char* address = (address_ptr == nullptr) ? "" :
                          static_cast<const char*>(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<const char*, const char*>> *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 char* name    = static_cast<const char*>(name_ptr);
    const char* address = (address_ptr == nullptr) ? "" :
                          static_cast<const char*>(address_ptr);
    owners->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: 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.

3.4 Standard Library Additions

The preceding sections describe the core-language mechanics of assertion-control objects and the facets through which they influence contract-assertion evaluation. For most users, however, the Standard Library is the primary surface of this proposal: including `<contracts>`, picking the right label, and combining labels with `|` is all that is needed to take advantage of the feature without writing any label types of one's own.

3.4.1 Namespace and Name Lookup

Standard library labels are placed in the `std::contracts::labels` namespace. The plural form reflects the nature of the namespace as a collection of distinct, independently usable objects — analogous to `std::ranges` or `std::views` — rather than a type whose members are accessed through a common prefix. A qualified name such as `std::contracts::labels::terminating` reads naturally as "the `terminating` label from the contracts labels library."

The `<contracts>` header includes the directive:

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

This makes all names in `std::contracts::labels` available unqualified within assertion-control expressions in any translation unit that includes `<contracts>`, without introducing those names into any enclosing scope where they might conflict with user-defined names. The mechanism is described in detail in Section 3.2.1.

Proposal: Using Directive in `<contracts>`

Add an assertion-control using directive to `<contracts>`.

3.4.2 Facet Concepts

Each facet is identified by a concept in `std::contracts::labels`. An assertion-control object opts into a facet by satisfying the corresponding concept; an object that does not satisfy a given concept simply retains the default behavior for that facet. The full definitions of these concepts appear in the facet sections of Section 3.3; we collect them here for reference.

For each concept, a corresponding `constexpr` variable template prefixed with `is_` is provided so that users can validate their labels with `static_assert` without resorting to `decltype`:

```
namespace std::contracts::labels {
template <auto control_object>
constexpr bool is_assertion_control_object =
  assertion_control_object<decltype(control_object)>;
}
```

The full set of facet concepts is:

— `assertion_control_object` — The base requirement for all assertion-control objects. Requires a nested `assertion_control_object` member typedef (see Section 3.3).

— `allowed_semantics_label` — Restricts the set of evaluation semantics that may be used for the associated contract assertion (see Section 3.3.1).

— `semantic_computation_label` — Provides a `compute_semantic` member function to transform the chosen evaluation semantic (see Section 3.3.2).

— `dimensioned_label` — Declares dimensions for mutual-exclusivity checking when combining labels (see Section 3.3.3).

— `local_violation_label` — Provides a `handle_contract_violation` member function to intercept or replace the global violation handler (see Section 3.3.4).

— `identification_label` — Provides `group_names` for tool-driven and configuration-driven grouping (see Section 3.3.5).

— `compute_comment_label` and `compute_message_label` — Provide compile-time transformation of the `comment` and `message` fields of the `contract_violation` object (see Section 3.3.6).

— `queryable_label` — Provides a type-erased `query` interface for communicating label data to the contract-violation handler at run time (see Section 3.3.7).

Proposal: Facet Concepts
Include Standard Library concepts for all adopted facets.

3.4.3 Combining Labels: `operator|` and `combined_label`

As described in Section 3.2.3, the Standard Library provides a free-function `operator|` in `std::contracts::labels` that combines two assertion-control objects into a single compound object. The returned object participates in the union of the facets satisfied by its operands and applies each facet's combination rule as described in the corresponding facet section.

The type returned by `operator|` is an unspecified class template, referred to here as `combined_label`. Its behavior is fully determined by the facet concepts its constituent labels satisfy:

— If either operand models `allowed_semantics_label`, the combined label's `allowed_semantics` is the intersection of those of its constituents.

— If either operand models `semantic_computation_label`, the combined label chains their `compute_semantic` functions left to right.

— If either operand models `identification_label`, the combined label's `group_names` is the sorted, unique union of both.

— If either operand models `dimensioned_label`, the combined label's `dimensions` is the union; overlapping dimensions make the combination ill-formed.

— If either operand models `local_violation_label`, the combined label chains their `handle_contract_violation` members from right to left.

— If either operand models `queryable_label`, the combined label delegates `query` to its constituents, using index arithmetic to iterate through results from both.

Users who need different combination behavior — such as replacing rather than chaining — can define their own overloads of `operator|` or use a named function instead.

Proposal: Combining Operator

Include the free function `operator|` in the `std::contracts::labels` namespace in the `<contracts>` header.

3.4.4 Evaluation Semantic Utilities

To help label authors write `compute_semantic` implementations that remain correct as new evaluation semantics are added in the future, we provide a set of free functions that categorize an `evaluation_semantic` value:

```
namespace std::contracts {

constexpr bool is_checked(evaluation_semantic s);
constexpr bool is_unchecked(evaluation_semantic s);
constexpr bool is_terminating(evaluation_semantic s);
constexpr bool is_continuing(evaluation_semantic s);

}
```

— `is_checked` returns `true` for *enforce*, *quick-enforce*, and *observe*.

— `is_unchecked` returns `true` for *ignore* (and, in the future, *assume*).

— `is_terminating` returns `true` for *enforce* and *quick-enforce*.

— `is_continuing` returns `true` for *observe* and *ignore*.

These predicates let label implementations avoid hard-coding comparisons against individual enumerators. For example, the `review` label's `compute_semantic` function could be written as:

```
constexpr evaluation_semantic compute_semantic(evaluation_semantic in) const
{
  if (is_terminating(in)) {
    return evaluation_semantic::observe;
  }
  return in;
}
```

Proposal: Semantic Categorization Functions
Provide the free functions listed above to categorize evaluation semantics.

3.4.5 Standard Library Labels

The following labels are provided by the Standard Library in the `std::contracts::labels` namespace. Each is a `constexpr` object whose type models the concepts indicated.

Core label.

— `empty_label` — An assertion-control object with no facet participation. Useful as an identity element for `operator|` and as a base for user-defined labels.

Semantic restriction labels. These labels model `allowed_semantics_label`.

— `terminating` — Allows only *enforce* and *quick-enforce*. A contract assertion with this label will never be evaluated with a continuing or unchecked semantic.

— `symbolic` — Allows only unchecked semantics (*ignore* and, in the future, *assume*). Because the assertion will never be checked at run time, its predicate is an unevaluated operand and may name functions that are declared but not defined.

— `always_enforce`, `always_quick_enforce`, `always_observe`, `always_ignore` — Each forces a single evaluation semantic. These labels are mutually exclusive; combining two distinct explicit-semantic labels is ill-formed.

— `never_enforce`, `never_quick_enforce`, `never_observe`, `never_ignore` — Each removes a single evaluation semantic from the allowed set.

Semantic computation labels. These labels model `semantic_computation_label`.

— `review` — Maps terminating semantics to *observe*; all other semantics pass through unchanged. Intended for newly introduced assertions that should be observed rather than enforced during an initial deployment period.

— `hardened` — Computes the evaluation semantic based on the Standard Library implementation's hardening configuration. When the library is built with hardening enabled, the semantic is set to the library's configured hardening level (typically *quick-enforce*); otherwise, the library's default applies.

Runtime-cost labels. These labels model `identification_label` and share a common `runtime_cost` dimension so that they are mutually exclusive.

— `opt` — Identifies contract assertions that are very inexpensive to evaluate. Implementations are encouraged to default these to a checked semantic even in optimized builds.

— `audit` — Identifies contract assertions that are expensive to evaluate. Implementations are encouraged to default these to an unchecked semantic and to require explicit opt-in from the user to enable checking.

Grouping labels.

— The user-defined literal operator `""group` returns an `identification_label` whose `group_names` member contains the string literal used to construct it.

Proposal: Standard Labels
Provide a basic set of Standard Library assertion-control labels as well as general utilities for manipulating and accessing 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.

3.4.6 Label Utilities

To simplify the implementation of user-defined labels, the Standard Library provides several concrete types and helpers.

General utilities.

— `empty_label_t` — The type of `empty_label`; can be used as a base class for user-defined labels that need only the `assertion_control_object` member typedef.

Semantic utilities.

— `allowed_semantics_t<evaluation_semantic...>` — A label type that models `allowed_semantics_label` with a compile-time-fixed set of allowed semantics. `terminating`, `symbolic`, and the explicit semantic labels are specializations of this template.

— `evaluation_semantic_set` — The set type used to represent allowed semantics, with `constexpr` set operations (`&`, `|`, `~`, `contains`) and factory functions (`all()`, `none()`).

— `is_checked`, `is_unchecked`, `is_terminating`, `is_continuing` — Free functions for categorizing semantics (described in the Evaluation Semantic Utilities subsubsection).

Combination utilities.

— `dimension_list<typename...>` — A template used to declare the dimensions of a label for mutual-exclusivity checking.

Violation-handling utilities.

— `violation_handled` — The enumeration (`not_handled`, `handled`) returned by local violation handlers.

Identification utilities.

— `assertion_group_label<int N>` — The type returned by the `""group` user-defined literal operator.

Comment and message utilities.

— `fixed_message_label_t` — A label type that replaces the violation message with a compile-time string literal.

Some or all of the above facilities can be adopted in order to ease the use of labels in different situations, though they all build on core-language functionality and could be written by users themselves.

Proposal: Standard Library Labels
Provide the utilities for writing user-defined labels as specified in this section.

3.5 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.

3.5.1 Always-Checked and Always-Enforced 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.

— Having an always-enforced function-contract assertion on a function is a property that can easily become baked into an ABI. As soon as compilers of caller and callee TUs come to an agreement as to which one will evaluate an always-checked contract assertion the *other* compiler can begin to optimize based on the fact that that assertion will be enforced. For example, consider a function that passes a pointer to another function with an always-enforced precondition:

```
void f(int* p) pre<terminating>(p != nullptr);
void g(int* p)
{
  f(p);
  if (p) { std::cout << "p is not null\n"; }  // test of branch can be elided
}
```

Given the above declaration of `f`, the compiler that translates `g` can know that `f` will never return normally if `p` is `nullptr`. With that knowledge, the test to branch in the `if` statement can be elided, resulting in the output being printed unconditionally. This optimization, however, depends on that contract assertion and its label being known and unchanging — therefore this build of `g` will not behave correctly if compiled against a version of `f` that removes the precondition (or simply changes its label to one that happens to end up not being enforced). That is the definition of not being ABI-compatible, and such coupling between translation units is something we strive to minimize with our language design choices.

— Implicit contract assertions, as proposed in [P3100R6], 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 [P3271R1]), 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, and barring a massive ABI change, the statements above imply that 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.

3.5.2 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 [P3100R6] 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<symbolic>( 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, however, 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.[7]

Proposal: 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.

[7] Because this option needs to be experimented with, the current wording in this paper does not reflect this proposal being selected.

3.5.3 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.

For example, two translation units that declare the same function with different assertion-control expressions would be ill-formed:

```
// TU 1:
void f(int *p) pre<terminating>(p != nullptr);

// TU 2:
void f(int *p) pre<review>(p != nullptr);  // Error: ODR violation
```

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. Annotations are compile-time values attached to declarations via `[[=expr]]` syntax and accessible through reflection (as introduced by [P3394R4] and suggested for this proposal by [P3831R0]). They 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 gaining a clear picture of the assertion-control object that might be applicable to any given evaluation becomes challenging or impossible 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.

3.5.4 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.

4 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.

4.1 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 particular time. 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.

4.2 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 their enclosing 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 ambient control-object declaration that can be used at class or namespace scope:

ambient-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.

4.3 Core-Language Control Labels

Implicit contract assertions for core-language constructs, as explained in [P3100R6] 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 offer two key capabilities:

   (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 were to adopt 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, bounds checking in a localized context — such as an inner loop — might unacceptably degrade runtime performance. By adding the `audit` label to all implicit

bounds checks in that local context, we can recover the performance needed in that piece of hot-path code while leaving those relatively expensive checks available to be enabled in slower builds that are used exclusively for testing.

This contextual control of implicit behavior could be done at a scope using another variation of the *ambient-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 3.3.5) 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.

Alternatively, using 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.

4.4 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.

4.5 The Assume Semantic

The C++ Standard Library currently specifies its preconditions such that violating them results in core-language undefined behavior — and that undefined behavior is precisely what enables compilers to optimize callers based on the assumption that the preconditions and postconditions of such functions hold. To migrate those specifications to contract-assertion syntax without weakening the guarantees they currently provide, we need an *assume* semantic: one that permits the implementation to optimize under the assumption that the predicate holds without requiring a runtime check. Without such a semantic, expressing existing library preconditions as contract assertions would be a strict regression in the optimization properties that callers currently enjoy.

In particular, the assume semantic enables a class of optimization — eliding redundant checks in callers who can rely on a callee's precondition having been satisfied — that many in the community consider an essential feature of contracts and a primary motivation for adopting them in the first place.

[P3100R6] specifies the assume semantic and makes it available for implicit contract assertions, providing the mechanism by which core-language undefined behavior can be expressed using contracts. Cross-language precedent also exists: WG14's contracts proposal for C ([N3867]) effectively evaluates all contract assertions twice — the second time with the assume semantic — demonstrating that this is a design point shared across C and C++.

On the other hand, the assume semantic carries known risks — as documented thoroughly in [P2064R0] — and any design that introduces it must carefully account for the possibility of widening safety vulnerabilities that come with increased undefined behavior.

We see two broad approaches for integrating the assume semantic with assertion-control objects. The first would add *assume* as a generally available evaluation semantic alongside *ignore*, *observe*, *enforce*, and *quick-enforce*, making it selectable through the same build-configuration mechanisms that control other semantics. The second would make it available only when a label explicitly opts in via a new facet — keeping the default contract-assertion behavior unchanged and requiring that authors deliberately choose assumption semantics for specific assertions. Both approaches have merit; the first provides uniformity with existing semantics while the second provides an additional layer of protection against accidental misuse.

We defer the detailed design discussion of these options to a future paper and expect that compiler implementations will explore both approaches in the interim, providing the practical experience needed to make a well-informed choice between them.

5 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.

5.1 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.

5.2 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.

5.3 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 [P3261R2]) 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.

6 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.

6.1 Core Proposal Wording

These wording changes are relative to the C++29 working draft with git hash 2a8305d4, last modified on Mon, 27 Apr 2026 12:27:09 +0100.

5 Lexical conventions [lex]

5.12 Keywords [lex.key]

Modify [lex.key], paragraph 1:

1 The identifiers shown in ([tab:lex.key]) are reserved for use as keywords (that is, they are unconditionally treated as keywords in phases 6 and 7) except in an *attribute-token*([dcl.attr.grammar]).

[*Note*: The `register` keyword is unused but is reserved for future use. — *end note*]

Table 1: Keywords [tab:lex.key]

alignas	constinit	extern	protected	throw
alignof	const_cast	false	public	true
asm	continue	float	register	try
auto	contract_assert	for	reinterpret_cast	typedef
bool	contract_control	friend	requires	typeid
break	co_await	goto	return	typename
case	co_return	if	short	union
catch	co_yield	inline	signed	unsigned
char	decltype	int	sizeof	using
char8_t	default	long	static	virtual
char16_t	delete	mutable	static_assert	void
char32_t	do	namespace	static_cast	volatile
class	double	new	struct	wchar_t
concept	dynamic_cast	noexcept	switch	while
const	else	nullptr	template	
consteval	enum	operator	this	
constexpr	explicit	private	thread_local	
	export			

6 **Basics** [**basic**]

6.4 **Scope** [**basic.scope**]

6.4.10 **Contract-assertion scope** [**basic.scope.contract**]

Modify [basic.scope.contract], paragraph 1:

1 Each contract assertion([basic.contract]) C introduces a *contract-assertion scope* that includes the predicate of C.

Modify [basic.scope.contract], paragraph 2:

2 If a *result-name-introducer*([dcl.contract.res]) that is not name-independent([basic.scope.scope]) and whose enclosing postcondition assertion is associated with a function `F` potentially conflicts with a declaration whose target scope is

(2.1) — the function parameter scope of `F` or

(2.2) — if associated with a *lambda-declarator*, the nearest enclosing lambda scope of the postcondition assertion([expr.prim.lambda]),

the program is ill-formed.

6.4.($10 + a$) **Assertion-control scope** [**basic.scope.contract.control**]

Add a new section 6.4.(10+a) ([basic.scope.contract.control]) after [basic.scope.contract]

6.4.(10+a) Assertion-control scope [basic.scope.contract.control]

1 Each *assertion-control-specifier*([basic.contract.control]) and *assertion-control-expression*([expr.contract.control]) introduces a *assertion-control scope* that includes its *constant-expression*.

[*Note*: Within an assertion-control scope name lookup will search additional namespaces introduced by *using-directives* with the `contract_control` keyword([namespace.udir,basic.lookup]). — *end note*]

6.5 Name lookup [basic.lookup]

6.5.3 Unqualified name lookup [basic.lookup.unqual]

Modify [basic.lookup.unqual], paragraph 1:

1 A *using-directive* U is *active* in a scope S at a program point P if it

— if U precedes P and inhabits either S or the scope of a namespace nominated by a *using-directive* that is active in S at P, and

— if S has the `contract_control` keyword then P is within an assertion-control scope.

6.11 Contract assertions [basic.contract]

6.11.(1 + a) Control [basic.contract.control]

Add a new section 6.11.(1+a) ([basic.contract.control]) after [basic.contract.general]

6.11.(1+a) Control [basic.contract.control]

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

Multiple strategies were considered for how to specify facets:
1. Define the concepts in the standard library and have the core wording reference them by name, saying "if the assertion control object models std;:contracts::labels::..." to identify when the assertion control object participates in a particular facet.
2. Define facet as a first-order term and describe the facets, and define what it means to participate in a facet. Standard library concepts will refer back to that definition.
3. Define facets as a first-order term and include the satisfaction of the Standard library concept along with that definition, ensuring that we retain consistency between the core and library behavior.
4. Define new adjectives for each facet and do not use the term facet in the Standard. Say things along the lines of "If the assertion control object is an allowed semantics control object..." Standard library concepts refer back to these definitions.
5. Define the adjectives and include modeling the concept in that definition keeping the library dependency but in only one place.
For now, we are going with option 4 for certain reasons:
1. This keeps the core wording from having to depend on the Standard library definitions, especially when
2. We can keep the library concept definitions sufficiently in synch as we update the Standard, we do not have to bake in that requirement.
3. We don't need to deliberate on introducing "facet" as a new term of art. If we change our mind on this last one, we could conceivably reuse it for things like various aspects of the coroutine promise object.

1 A type is an *assertion control type* if the search for the name `assertion_control_object` in its scope finds a declaration of a type.

2 Every contract assertion C may have associated with it an *assertion control object* that is a constexpr variable.

(2.1) — If C has an *assertion-control-specifier* S then it has an assertion-control-object O whose type is deduced from S and which is initialized by S. The type of O must be an assertion-control type.

(2.2) — Otherwise, the contract assertion has no assertion-control object.

We are avoiding having a default assertion-control object because to do so we would have to define all of the properties that its unspecified type does not have.

3 [*Note*: S is manifestly constant-evaluated([expr.const.defns]), O must have a literal type([basic.types.general]), and S is not part of the immediate context([temp.deduct.general]). — *end note*]

4 A type is an *allowed-semantics control type* if

— it is an assertion-control type, and

— it has a static member named `allowed_semantics` that is

— usable in constant expressions, and

— can be converted to the type `std::contracts::labels::evaluation_semantic_set`.

5 A type T is an *computed-semantics control type* if

— it is an assertion-control type, and

— the expression `o.compute_semantic(s)` has the type `std::contracts::evaluation_semantic` where `o` is an object of type `T` and `s` is a prvalue of type `std::contracts::evaluation_semantic`.

6 a type T is a *local-violation control type* if

— it is an assertion-control type, and

— the expression `o.handle_contract_violation(v)` is valid where `o` is an object of type T and `v` is a `const` lvalue of type `std::contracts::contract_violation` and its type is `void` or is convertible to `std::contracts::violation_handled`.

6.11.2 **Evaluation** [**basic.contract.eval**]

Remove [basic.contract.eval] paragraph 2:

2 It is implementation-defined which evaluation semantic is used for any given evaluation of a contract assertion.

[*Note*: The range and flexibility of available choices of evaluation semantics depends on the implementation and need not allow all four evaluation semantics as possibilities. The evaluation semantics can differ for different evaluations of the same contract assertion, including evaluations during constant evaluation. — *end note*]

Add a new paragraph after [basic.contract.eval], paragraph 2:

2 A contract evaluation semantic is an *allowed contract evaluation semantic* of a contract assertion C if

— C does not have an assertion-control object,

— the assertion-control object O of C is not an allowed-semantics control type([basic.contract.control]), or

— the expression `o.allowed_semantics.contains(S)` is `true` where `S` is the corresponding enumerator for the evaluation semantic in `std::contracts::evaluation_semantic`.

2+a If a contract assertion has no allowed evaluation semantics the program is ill-formed. The *configured contract evaluation semantic* for an evaluation of a contract assertion is an allowed contract evaluation semantic chosen in an implementation-defined manner.

2+b [*Note*: The range and flexibility of available choices of configured evaluation semantics depends on the implementation and need not allow all four evaluation semantics as possibilities. The evaluation semantics can differ for different evaluations of the same contract assertion, including evaluations during constant evaluation. — *end note*]

2+c The *effective contract evaluation semantic* for an evaluation E of a contract assertion C is

— the configured contract evaluation semantic of E if

— C has no assertion-control object, or

— if the assertion control object O of C is not a computed-semantics control type([basic.contract.control]),

or

— the contract evaluation semantic corresponding to the value of the expression `o.compute_semantic(S)` where `o` is the assertion control object of C and `S` is the enumerator corresponding to the configured contract evaluation semantic of E.

The effective contract evaluation semantic shall be an allowed contract evaluation semantic.

Modify [basic.contract.eval], paragraph 11:

11 If a contract violation occurs in a context that is not manifestly constant-evaluated and the evaluation semantic is enforce or observe, the contract-violation handler([basic.contract.handler]) is invoked with an lvalue referring to an object `v` of type `const std::contracts::contract_violation`([support.contract.violation]) containing information about the contract violation. Storage for `v` is allocated in an unspecified manner except as noted in [basic.stc.dynamic.allocation]. The lifetime of `v` persists for the duration of the invocation of then the contract-violation handler handling occurs.

Remove [basic.contract.eval] paragraph 12:

12 If the contract violation occurred because the evaluation of the predicate exited via an exception, the contract-violation handler is invoked from within an active implicit handler for that exception([except.handle]). If the contract-violation handler returns normally and the evaluation semantic is observe, that implicit handler is no longer considered active.

[*Note*: The exception can be inspected or rethrown within the contract-violation handler. — *end note*]

Add a new paragraph after [basic.contract.eval], paragraph 15:

15 Contract-violation handling consists of a number of steps evaluated in order.

— An object `V` of type `std::contracts::contract_violation`([support.contract.violation]) is created with information about the contract assertion and evaluation that triggered the violation. Storage for `V` is allocated in an unspecified manner except as noted in [basic.stc.dynamic.allocation]. The lifetime of `V` persists until contract-violation handling completes.

(15.1) — If the contract assertion has an assertion-control object o whose type is a local-violation control type, the expression `o.handle_contract_violation(v)` is evaluated where `v` is a `const` lvalue referring to `V`. If this expression has a value equivalent to `std::contracts::violation_handled::handled` then contract-violation handling is complete.

(15.2) — The contract-violation handler is invoked with a `const` lvalue referring to `v`.

Remove [basic.contract.eval] paragraph 13:

13 If the contract-violation handler returns normally and the evaluation semantic is enforce, the program is contract-terminated; if violation occurred as the result of an uncaught exception from the evaluation of the predicate, the implicit handler remains active when contract termination occurs.

Add a new paragraph after [basic.contract.eval], paragraph 16:

16 If the contract violation occurred because the evaluation of the predicate exited via an exception, the contract-violation handling occurs within an active implicit handler for that exception([except.handle]). If contract-violation handling completes normally and the evaluation semantic is observe, that implicit handler is no longer considered active.

[*Note*: The exception can be inspected or rethrown within the contract-violation handler. — *end note*]

Remove [basic.contract.eval] paragraph 14:

14 [*Note*: If the contract-violation handler returns normally and the evaluation semantic is observe, control flow continues normally after the point of evaluation of the contract assertion. — *end note*]

Add a new paragraph after [basic.contract.eval], paragraph 17:

17 If contract-violation handling completes normally and the evaluation semantic is enforce, the program is contract-terminated; if violation occurred as the result of an uncaught exception from the evaluation of the predicate, the implicit handler remains active when contract termination occurs.

Modify [basic.contract.eval], paragraph 15:

15 There is an observable checkpoint([intro.abstract]) C that happens after the contract-violation handler returns normally such that any other evaluation that happens after the contract-violation handler returns also happens after C.

[*Note*: If contract-violation handling completes normally and the evaluation semantic is observe, control flow continues normally after the point of evaluation of the contract assertion. — *end note*]

Add a new paragraph after [basic.contract.eval], paragraph 15:

15+d There is an observable checkpoint([intro.abstract]) C that happens after contract-violation handling completes normally such that any other evaluation that happens after the contract-violation handler returns also happens after C.

Modify [basic.contract.eval], paragraph 17:

17 If a contract-violation handler invoked from handling during the evaluation of a function contract assertion([dcl.contract.func]) exits via an exception, the behavior is as if the function body exits via that same exception.

[*Note*: A *function-try-block*([except.pre]) is the function body when present and thus does not have an opportunity to catch the exception. If the function has a non-throwing exception specification, the function `std::terminate` is invoked([except.terminate]). — *end note*]

[*Note*: If a contract-violation handler invoked from handling during the evaluation of an *assertion-statement*([stmt.contract.assert]) exits via an exception, the search for a handler continues from the execution of that statement. — *end note*]

7 Expressions [expr]

7.6 Compound expressions [expr.compound]

7.6.2 Unary expressions [expr.unary]

7.6.2.(10 + a) Contract-control expression [expr.contract.control]

Add a new section 7.6.2.(10+a) ([expr.contract.control]) after [expr.reflect]

7.6.2.(10+a) Contract-control expression [expr.contract.control]

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

1 An *assertion-control-expression* yields the same value and has the same value category as its *constant-expression*.

[*Note*: The *constant-expression* is manifestly constant-evaluated. Note that the *assertion-control-expression* introduces a new assertion-control scope([basic.scope.contract.control]) in which name lookup will search namespaces referred to by *using-directives* that have the optional `contract_control` keyword([namespace.udir,basic.lookup]). — *end note*]

8 Statements [stmt]

8.9 Assertion statement [stmt.contract.assert]

Modify [stmt.contract.assert] at the start of the section:

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

9 Declarations [dcl]

9.4 Function contract specifiers [dcl.contract]

9.4.1 General [dcl.contract.func]

Modify [dcl.contract.func] at the start of the section:

function-contract-specifier-seq:
function-contract-specifier function-contract-specifier-seq$_{opt}$

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* `)`

Modify [dcl.contract.func], paragraph 5:

5 A *function-contract-specifier-seq* S_1 is the same as a *function-contract-specifier-seq* S_2 if S_1 and S_2 consist of the same *function-contract-specifier*s in the same order. A *function-contract-specifier* C_1 on a function declaration D_1 is the same as a *function-contract-specifier* C_2 on a function declaration D_2 if

(5.0+a) — both C_1 and C_2 have *assertion-control-specifiers* or neither do,

(5.0+b) — if present, their assertion-control specifiers O_1 and O_2 would satisfy the one-definition rule([basic.def.odr]) if placed as the arguments of contract-control expressions([expr.contract.control]) in function definitions on the declarations D_1 and D_2, respectively, except for

(5.2.1) — renaming of the parameters of f, and

(5.2.2) — renaming of template parameters of a template enclosing f

if D_1 and D_2 are in different translation units, corresponding entities defined within each predicate behave as if there is a single entity with a single definition, and,

(5.1) — their predicates P_1 and P_2 would satisfy the one-definition rule([basic.def.odr]) if placed in function definitions on the declarations D_1 and D_2, respectively, except for

(5.1.1) — renaming of the parameters of f,

(5.1.2) — renaming of template parameters of a template enclosing f, and

(5.1.3) — renaming of the result binding([dcl.contract.res]), if any,

and, if D_1 and D_2 are in different translation units, corresponding entities defined within each predicate behave as if there is a single entity with a single definition, and

(5.2) — both C_1 and C_2 specify a *result-name-introducer* or neither do.

If this condition is not met solely due to the comparison of two *lambda-expression*s that are contained within P_1 and P_2, no diagnostic is required.

[*Note*: Equivalent *function-contract-specifier-seq*s apply to all uses and definitions of a function across all translation units. — *end note*]

[*Example*:

```
bool b1, b2;

void f() pre (b1) pre ([]{ return b2; }());
void f();                       // OK, function-contract-specifiers omitted
void f() pre (b1) pre ([]{ return b2; }()); // error: closures have different types.
void f() pre (b1);              // error: function-contract-specifiers only partially repeated

int g() post(r : b1);
int g() post(b1);       // error: mismatched result-name-introducer presence
```

```
namespace N {
  void h() pre (b1);
  bool b1;
  void h() pre (b1);     // error: function-contract-specifiers differ according to
                         // the one-definition rule([basic.def.odr]).
}
```

— *end example*]

9.9 Namespaces [basic.namespace]

9.9.4 Using namespace directive [namespace.udir]

Modify [namespace.udir] at the start of the section:

using-directive:
attribute-specifier-seq$_{opt}$ `using` `contract_control`$_{opt}$ `namespace` *nested-name-specifier*$_{opt}$ *namespace-name* `;`
attribute-specifier-seq$_{opt}$ `using` `contract_control`$_{opt}$ `namespace` *splice-specifier* `;`

Add a new paragraph after [namespace.udir], paragraph 2:

2+a The optional `contract_control` keyword, if present, indicates a using directive that will be considered only within assertion-control scopes([basic.scope.contract.control,basic.lookup]).

17 Language support library [support]

17.10 Contract-violation handling [support.contract]

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

Modify [contracts.syn], paragraph 1:

1 The header `contracts` defines types for reporting information about contract violations([basic.contract.eval]) and controlling the behavior of contract assertions([basic.contract.control]).

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

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

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

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

  class contract_violation {
```

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

    see below ~contract_violation();

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

  void invoke_default_contract_violation_handler(const contract_violation);    void* query_-
control_object(const void* key,
                               size_t index = 0) const;
  };

  void invoke_default_contract_violation_handler(const contract_violation&);

  // [{]support.contract.control.vhandled}, enum {violation_handled}
  enum class violation_handled;

  // [{]support.contract.control.semset}, class {evaluation_semantic_set}
  class evaluation_semantic_set;

  // [{]support.contract.control.semfun}, evaluation semantic classification
  constexpr bool is_checked(evaluation_semantic s);
  constexpr bool is_unchecked(evaluation_semantic s);
  constexpr bool is_terminating(evaluation_semantic s);
  constexpr bool is_continuing(evaluation_semantic s);

  namespace labels {
    // [{]support.contract.control.aco}, concept {assertion_control_object}
    template<class T>
      concept assertion_control_object = see below;

    // [{]support.contract.control.allowed}, concept {allowed_semantics_label}
    template<class T>
      concept allowed_semantics_label = see below;

    // [{]support.contract.control.compute}, concept {semantic_computation_label}
    template<class T>
      concept semantic_computation_label = see below;

    // [{]support.contract.control.dim}, concept {dimensioned_label}
    template<class T>
      concept dimensioned_label = see below;

    // [{]support.contract.control.lvl}, concept {local_violation_label}
    template<class T>
      concept local_violation_label = see below;

    // [{]support.contract.control.ident}, concept {identification_label}
    template<class T>
      concept identification_label = see below;

    // [{]support.contract.control.comment}, concept {compute_comment_label}
    template<class T>
      concept compute_comment_label = see below;

    // [{]support.contract.control.query}, concept {queryable_label}
    template<class T>
```

```
    concept queryable_label = see below;

  // [{]support.contract.control.dimlist}, class template {dimension_list}
  template<class... Dims>
    struct dimension_list;

  // [{]support.contract.control.combine}, label combination
  template<assertion_control_object LHS, assertion_control_object RHS>
    constexpr assertion_control_object auto operator|(const LHS& lhs,
                                                      const RHS& rhs);

  // [{]support.contract.labels.empty}, label {empty_label}
  struct empty_label_t;
  inline constexpr empty_label_t empty_label;

  // [{]support.contract.labels.allowed}, label template {allowed_semantics_t}
  template<evaluation_semantic... Allowed>
    struct allowed_semantics_t;

  // [{]support.contract.labels.sem}, semantic restriction labels
  inline constexpr see below terminating;
  inline constexpr see below symbolic;
  inline constexpr see below always_enforce;
  inline constexpr see below always_quick_enforce;
  inline constexpr see below always_observe;
  inline constexpr see below always_ignore;
  inline constexpr see below never_enforce;
  inline constexpr see below never_quick_enforce;
  inline constexpr see below never_observe;
  inline constexpr see below never_ignore;

  // [{]support.contract.labels.review}, label {review}
  inline constexpr see below review;

  // [{]support.contract.labels.hardened}, label {hardened}
  inline constexpr see below hardened;

  // [{]support.contract.labels.cost}, runtime-cost labels
  inline constexpr see below opt;
  inline constexpr see below audit;

  // [{]support.contract.labels.group}, label template {assertion_group_label}
  template<int N>
    struct assertion_group_label;
  template<assertion_group_label Label>
    constexpr auto operator""group();
 }
}
```

17.10.3 Class `contract_violation` [support.contract.violation]

Add a new paragraph after [support.contract.violation], paragraph 2:

2+a If the contract violation is the result of a contract assertion that has an assertion-control object that is a computed-comment control type([support.contract.control.comment]) then the return value is instead the result of `o.compute_comment(r)` where `o` is the assertion-control object and `r` is the return value that would have been returned.

Add a new paragraph after [support.contract.violation], paragraph 9:

```
void* query_control_object(const void* key, size_t index = 0) const;
```

9+b *Returns:* If the assertion-control object of the violated contract assertion satisfies `queryable_label`([support.contract.control.query]), the result of invoking `query(key, index)` on the assertion-control object; otherwise, `nullptr`.

17.10.(4 + *a*) Assertion-control concepts and utilities [support.contract.control]

Add a new section 17.10.(4+a) ([support.contract.control]) after [support.contract.invoke]

17.10.(4+a) Assertion-control concepts and utilities[support.contract.control]

This section defines a mix of concepts and language definitions for library-only assertion control types. Where the concept represents a core-language assertion control type we reference back to [basic.contract.control]

17.10.(4 + *a*).1 General [support.contract.control.general]

17.10.(4+a).1 General [support.contract.control.general]

1 This subclause defines concepts, types, and utilities related to controlling the behavior of contract assertions([basic.contract.control]). All concepts and variable templates defined in this subclause are declared in the namespace `std::contracts::labels`.

2 For each concept `C` defined in this subclause, a corresponding variable template is provided:

```
namespace std::contracts::labels {
  template<auto control_object>
    constexpr bool is_C = C<decltype(control_object)>;
}
```

17.10.(4 + *a*).2 Concept `assertion_control_object` [support.contract.control.aco]

17.10.(4+a).2 Concept `assertion_control_object`[support.contract.control.aco]

```
template<class T>
  concept assertion_control_object =
    requires { typename T::assertion_control_object; };
```

1 The `assertion_control_object` concept specifies the base requirement for all assertion-control objects.

[*Note*: Every assertion-control object must satisfy this concept. As a general convention, label types name themselves with this member typedef. — *end note*]

17.10.(4 + *a*).3 Concept `allowed_semantics_label` [support.contract.control.allowed]

17.10.(4+a).3 Concept `allowed_semantics_label` [support.contract.control.allowed]

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

```
  };
```

1 A type `T` models `allowed_semantics_label` if it is an allowed-semantics control type([basic.contract.control]).

2 [*Note*: Combined assertion-control objects intersect the allowed semantics sets of their constituents. — *end note*]

17.10.(4 + *a*).4 Concept `semantic_computation_label` [support.contract.control.compute]

17.10.(4+a).4 Concept `semantic_computation_label` [support.contract.control.compute]

```
template<class T>
  concept semantic_computation_label =
    assertion_control_object<T> &&
    requires (const T t, evaluation_semantic s) {
      { t.compute_semantic(s) } -> same_as<evaluation_semantic>;
    };
```

1 A type `T` models `semantic_computation_label` if it is a computed-semantics control type.([basic.contract.control]).

2 [*Note*: Combined assertion-control objects chain their `compute_semantic` functions from left to right. — *end note*]

17.10.(4 + *a*).5 Concept `dimensioned_label` [support.contract.control.dim]

17.10.(4+a).5 Concept `dimensioned_label` [support.contract.control.dim]

```
template<class T>
  concept dimensioned_label =
    assertion_control_object<T> &&
    is-specialization-of<typename T::dimensions, dimension_list>;
```

1 A type `T` models `dimensioned_label` if it declares dimensions via a nested `dimensions` member that is a specialization of `dimension_list`.

2 A type is a *dimensioned control type* if it models `dimensioned_label`.

3 [*Note*: Combined assertion-control objects have the union of the dimensions of their constituent objects, and it is ill-formed if the dimensions of their constituents have a non-empty intersection. — *end note*]

17.10.(4 + *a*).6 Concept `local_violation_label` [support.contract.control.lvl]

17.10.(4+a).6 Concept `local_violation_label` [support.contract.control.lvl]

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

1 A type `T` models `local_violation_label` if it has a member function `handle_contract_violation` that accepts a `const contract_violation&`. The return type shall be `void` or a type convertible to `violation_handled`.

2 When a contract violation occurs for an assertion whose control object satisfies this concept, `handle_contract_violation` is invoked before the global contract-violation handler([basic.contract.handler]). If the return type is not `void` and the return value converts to `violation_handled::handled`, the global handler is not invoked.

[*Note*: Combined assertion-control objects chain their `handle_contract_violation` members from right to left. — *end note*]

17.10.(4 + *a*).7 Concept `identification_label` [support.contract.control.ident]

17.10.(4+a).7 Concept `identification_label` [support.contract.control.ident]

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

1 A type `T` models `identification_label` if it has a `group_names` member that is a range of values convertible to `string_view`.

2 A type is a *identification control type* if it models `identification_label`.

3 [*Note*: The value of an identification label's `group_names` can be used as input to the implementation-defined choice of evaluation semantic when evaluating a contract assertion.([basic.contract.eval]). — *end note*]

4 [*Note*: Combined assertion-control objects produce the sorted, unique union of the group name sets of their constituents. — *end note*]

17.10.(4 + *a*).8 Concepts `compute_comment_label` [support.contract.control.comment]

17.10.(4+a).8 Concepts `compute_comment_label` [support.contract.control.comment]

```
template<class T>
  concept compute_comment_label =
    assertion_control_object<T> &&
    requires (T t, const char* comment) {
      { t.compute_comment(comment) } -> convertible_to<const char*>;
    };
```

1 Types satisfying `compute_comment_label` have a `constexpr` member function `compute_comment` that are used to transform at compile time the `comment` fields of the `contract_violation` object([support.contract.violation]) before they are stored.

2 A type is a *computed-comment control type* if it models `computed_comment_label`.

[*Note*: Combined assertion-control objects chain their `compute_comment` functions from left to right. — *end note*]

17.10.(4 + *a*).9 Concept `queryable_label` [support.contract.control.query]

17.10.(4+a).9 Concept `queryable_label` [support.contract.control.query]

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

1 A type `T` models `queryable_label` if it has a `query` member function providing a type-erased interface for the contract-violation handler to query data from the assertion-control object at runtime. The `key` parameter identifies the data to query; the `index` parameter allows returning multiple results for the same key. `query` returns `nullptr` if no result is available for the given key and index.

2 A type `T` is a *queryable control type* if it models `queryable_label`.

3 [*Note*: Combined assertion-control objects delegate `query` to their constituents, using index arithmetic to iterate through results from both. — *end note*]

17.10.(4 + *a*).10 Enum `violation_handled` [support.contract.control.vhandled]

17.10.(4+a).10 Enum `violation_handled` [support.contract.control.vhandled]

```
namespace std::contracts {
  enum class violation_handled {
    not_handled,
    handled
  };
}
```

1 The type `violation_handled` is used as a return type by local violation handlers([support.contract.control.lvl]) to indicate whether a contract violation has been fully handled.

17.10.(4 + *a*).11 Class `evaluation_semantic_set` [support.contract.control.semset]

17.10.(4+a).11 Class `evaluation_semantic_set` [support.contract.control.semset]

17.10.(4 + *a*).11.1 General [support.contract.control.semset.general]

17.10.(4+a).11.1 General [support.contract.control.semset.general]

```
namespace std::contracts {
  class evaluation_semantic_set {
  public:
    // [support.contract.control.semset.cons], constructors
    constexpr evaluation_semantic_set();
    constexpr evaluation_semantic_set(const evaluation_semantic_set&);
    constexpr evaluation_semantic_set(initializer_list<evaluation_semantic>);
```

```
    // [support.contract.control.semset.access], accessors
    constexpr bool contains(evaluation_semantic) const;
    constexpr size_t size() const;
    constexpr bool empty() const;

    // [support.contract.control.semset.mod], modifiers
    constexpr void clear();
    constexpr void insert(evaluation_semantic);
    constexpr void insert(initializer_list<evaluation_semantic>);
    constexpr void remove(evaluation_semantic);

    // [support.contract.control.semset.setops], set operations
    constexpr evaluation_semantic_set& operator&=(const evaluation_semantic_set&);
    constexpr evaluation_semantic_set& operator|=(const evaluation_semantic_set&);
    constexpr evaluation_semantic_set operator~() const;

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

    // [support.contract.control.semset.cmp], comparisons
    friend constexpr bool operator==(const evaluation_semantic_set&,
                                     const evaluation_semantic_set&);

    // [support.contract.control.semset.factory], factory functions
    static constexpr evaluation_semantic_set all();
    static constexpr evaluation_semantic_set none();
  };
}
```

1 An object of type `evaluation_semantic_set` represents a set of `evaluation_semantic` values.

17.10.(4 + a).11.2 Constructors [support.contract.control.semset.cons]

17.10.(4+a).11.2 Constructors [support.contract.control.semset.cons]

```
constexpr evaluation_semantic_set();
```

1 *Postconditions*: `contains(s)` is `false` for every enumerator `s` of `evaluation_semantic`.

```
constexpr evaluation_semantic_set(initializer_list<evaluation_semantic> ilist);
```

2 *Postconditions*: For each enumerator `s` of `evaluation_semantic`, `contains(s)` is `true` if and only if `s` is an element of `ilist`.

17.10.(4 + a).11.3 Accessors [support.contract.control.semset.access]

17.10.(4+a).11.3 Accessors [support.contract.control.semset.access]

```
constexpr bool contains(evaluation_semantic s) const;
```

1 *Returns:* `true` if `s` is an element of the set; otherwise, `false`.

```
constexpr size_t size() const;
```

2 *Returns:* The number of elements in the set.

```
constexpr bool empty() const;
```

3 *Returns:* `true` if the set contains no elements; otherwise, `false`.

17.10.(4 + *a*).11.4 **Modifiers** [**support.contract.control.semset.mod**]

17.10.(4+a).11.4 Modifiers [support.contract.control.semset.mod]

```
constexpr void clear();
```

1 *Postconditions*: `empty()` is `true`.

```
constexpr void insert(evaluation_semantic s);
```

2 *Postconditions*: `contains(s)` is `true`.

```
constexpr void insert(initializer_list<evaluation_semantic> ilist);
```

3 *Postconditions*: For each element `s` of `ilist`, `contains(s)` is `true`.

```
constexpr void remove(evaluation_semantic s);
```

4 *Postconditions*: `contains(s)` is `false`.

17.10.(4 + *a*).11.5 **Set operations** [**support.contract.control.semset.setops**]

17.10.(4+a).11.5 Set operations [support.contract.control.semset.setops]

```
constexpr evaluation_semantic_set& operator&=(const evaluation_semantic_set& rhs);
```

1 *Postconditions*: For each enumerator `s` of `evaluation_semantic`, `contains(s)` is `true` if and only if, immediately before the call, both `contains(s)` and `rhs.contains(s)` were `true`.

2 *Returns:* `*this`.

```
constexpr evaluation_semantic_set& operator|=(const evaluation_semantic_set& rhs);
```

3 *Postconditions*: For each enumerator `s` of `evaluation_semantic`, `contains(s)` is `true` if and only if, immediately before the call, `contains(s)` or `rhs.contains(s)` was `true`.

4 *Returns:* `*this`.

```
constexpr evaluation_semantic_set operator~() const;
```

5 *Returns:* An `evaluation_semantic_set r` such that, for each enumerator `s` of `evaluation_semantic`, `r.contains(s)` is `true` if and only if `contains(s)` is `false`.

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

6 *Returns:* `evaluation_semantic_set(lhs) &= rhs`.

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

7 *Returns:* `evaluation_semantic_set(lhs) |= rhs`.

17.10.(4 + a).11.6 Comparisons [support.contract.control.semset.cmp]

17.10.(4+a).11.6 Comparisons [support.contract.control.semset.cmp]

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

1 *Returns:* `true` if, for every enumerator `s` of `evaluation_semantic`, `lhs.contains(s) == rhs.contains(s)`; otherwise, `false`.

17.10.(4 + a).11.7 Factory functions [support.contract.control.semset.factory]

17.10.(4+a).11.7 Factory functions [support.contract.control.semset.factory]

```
static constexpr evaluation_semantic_set all();
```

1 *Returns:* An `evaluation_semantic_set r` such that `r.contains(s)` is `true` for every enumerator `s` of `evaluation_semantic`.

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

2 *Returns:* `evaluation_semantic_set()`.

17.10.(4 + a).12 Evaluation semantic classification [support.contract.control.semfun]

17.10.(4+a).12 Evaluation semantic classification [support.contract.control.semfun]

```
constexpr bool is_checked(evaluation_semantic s);
```

1 *Returns:* `true` if `s` is `evaluation_semantic::observe`, `evaluation_semantic::enforce`, or `evaluation_semantic::quick_enforce`; otherwise, `false`.

```
constexpr bool is_unchecked(evaluation_semantic s);
```

2 *Returns:* `!is_checked(s)`.

```
constexpr bool is_terminating(evaluation_semantic s);
```

3 *Returns:* `true` if `s` is `evaluation_semantic::enforce` or `evaluation_semantic::quick_enforce`; otherwise, `false`.

```
constexpr bool is_continuing(evaluation_semantic s);
```

4 *Returns:* `!is_terminating(s)`.

17.10.(4 + a).13 Class template `dimension_list` [support.contract.control.dimlist]

17.10.(4+a).13 Class template `dimension_list` [support.contract.control.dimlist]

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

1 The class template `dimension_list` is used to declare the dimensions of an assertion-control object for mutual-exclusivity checking when combining labels([support.contract.control.combine]).

[*Note*: Each template argument represents a distinct dimension. Labels with overlapping dimensions cannot be combined. — *end note*]

17.10.$(4+a)$.14 **Label combination** **[support.contract.control.combine]**

17.10.(4+a).14 Label combination **[support.contract.control.combine]**

```
template<assertion_control_object LHS,
         assertion_control_object RHS>
  constexpr assertion_control_object auto operator|(const LHS& lhs,
                                                    const RHS& rhs);
```

1 Let `C` be the type of the returned object.

2 *Constraints*:

(2.1) — If both `LHS` and `RHS` model `dimensioned_label`, the intersection of their `dimensions` is empty.

3 *Returns:* An object whose type `C` models `assertion_control_object` and whose behavior is determined as follows:

(3.1) — `C` models `allowed_semantics_label` if `LHS` or `RHS` does. If both do, the `allowed_semantics` member is the intersection of those of `lhs` and `rhs`.

(3.2) — `C` models `semantic_computation_label` if `LHS` or `RHS` does. The `compute_semantic` member function chains the functions of the constituent objects from left to right.

(3.3) — `C` models `dimensioned_label` if `LHS` or `RHS` does. The `dimensions` member is the union of those of the constituent objects.

(3.4) — `C` models `local_violation_label` if `LHS` or `RHS` does. The `handle_contract_violation` member function chains the handlers of the constituent objects from right to left. If any handler returns `violation_handled::handled`, subsequent handlers are not invoked.

(3.5) — `C` models `identification_label` if `LHS` or `RHS` does. The `group_names` member is the sorted, unique union of the group names of the constituent objects.

(3.6) — `C` models `compute_comment_label` if `LHS` or `RHS` does. The `compute_comment` member function chains the functions of the constituent objects from left to right.

(3.7) — `C` models `queryable_label` if `LHS` or `RHS` does. The `query` member function delegates to the constituent objects, using index arithmetic to iterate through results from both.

17.10.(4 + *b*) Standard labels [support.contract.labels]

17.10.(4+b) Standard labels [support.contract.labels]

17.10.(4 + *b*).1 General [support.contract.labels.general]

17.10.(4+b).1 General [support.contract.labels.general]

1 This subclause describes assertion-control label objects and label type templates provided by the Standard Library in the namespace `std::contracts::labels`.

17.10.(4 + *b*).2 Label `empty_label` [support.contract.labels.empty]

17.10.(4+b).2 Label `empty_label` [support.contract.labels.empty]

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

1 The type `empty_label_t` models `assertion_control_object` and no other facet concepts.

[*Note*: `empty_label` serves as an identity element for `operator|` and as a base class for user-defined labels. — *end note*]

17.10.(4 + *b*).3 Label template `allowed_semantics_t` [support.contract.labels.allowed]

17.10.(4+b).3 Label template `allowed_semantics_t` [support.contract.labels.allowed]

```
namespace std::contracts::labels {
  template<evaluation_semantic... Allowed>
    struct allowed_semantics_t {
      using assertion_control_object = allowed_semantics_t;
      static constexpr evaluation_semantic_set allowed_semantics{Allowed...};
    };
}
```

1 The type `allowed_semantics_t<Allowed...>` models `allowed_semantics_label`. Its `allowed_semantics` member contains exactly the enumerators `Allowed...`.

17.10.(4 + *b*).4 Semantic restriction labels [support.contract.labels.sem]

17.10.(4+b).4 Semantic restriction labels [support.contract.labels.sem]

```
namespace std::contracts::labels {
  inline constexpr allowed_semantics_t<evaluation_semantic::enforce,
                                       evaluation_semantic::quick_enforce> terminating;
  inline constexpr allowed_semantics_t<evaluation_semantic::ignore> symbolic;

  inline constexpr allowed_semantics_t<evaluation_semantic::enforce> always_enforce;
  inline constexpr allowed_semantics_t<evaluation_semantic::quick_enforce> always_quick_enforce;
  inline constexpr allowed_semantics_t<evaluation_semantic::observe> always_observe;
  inline constexpr allowed_semantics_t<evaluation_semantic::ignore> always_ignore;
```

```
  inline constexpr see below never_enforce;
  inline constexpr see below never_quick_enforce;
  inline constexpr see below never_observe;
  inline constexpr see below never_ignore;
}
```

1 The type of each `never_`*X* object models `allowed_semantics_label`. Its `allowed_semantics` member contains all enumerators of `evaluation_semantic` except the one indicated by its name.

17.10.(4 + *b*).5 Label `review` [support.contract.labels.review]

17.10.(4+b).5 Label `review` [support.contract.labels.review]

```
namespace std::contracts::labels {
  struct review_t {
    using assertion_control_object = review_t;
    constexpr evaluation_semantic compute_semantic(evaluation_semantic s) const;
  };
  inline constexpr review_t review;
}

constexpr evaluation_semantic compute_semantic(evaluation_semantic s) const;
```

1 *Returns:* `evaluation_semantic::observe` if `is_terminating(s)` is `true`; otherwise, `s`.

17.10.(4 + *b*).6 Label `hardened` [support.contract.labels.hardened]

17.10.(4+b).6 Label `hardened` [support.contract.labels.hardened]

```
namespace std::contracts::labels {
  inline constexpr unspecified hardened;
}
```

1 The type of `hardened` models `semantic_computation_label`. The behavior of its `compute_semantic` member function is implementation-defined.

[*Note*: Implementations are encouraged to compute the evaluation semantic based on the Standard Library implementation's hardening configuration. — *end note*]

17.10.(4 + *b*).7 Runtime-cost labels [support.contract.labels.cost]

17.10.(4+b).7 Runtime-cost labels [support.contract.labels.cost]

```
namespace std::contracts::labels {
  struct runtime-cost-dimension;          // exposition only
  inline constexpr unspecified opt;
  inline constexpr unspecified audit;
}
```

1 The types of `opt` and `audit` each model `identification_label` and `dimensioned_label`. Their `dimensions` members each contain *`runtime-cost-dimension`*, so they are mutually exclusive.

2 The `group_names` member of `opt` is a range containing a single element with value `"opt"`. The `group_names` member of `audit` is a range containing a single element with value `"audit"`.

[*Note*: Implementations are encouraged to default `opt` assertions to a checked semantic even in optimized builds, and to default `audit` assertions to an unchecked semantic. — *end note*]

17.10.(4 + *b*).8 Label template `assertion_group_label` [support.contract.labels.group]

17.10.(4+b).8 Label template `assertion_group_label` [support.contract.labels.group]

```
namespace std::contracts::labels {
  template<int N>
    struct assertion_group_label {
      using assertion_control_object = assertion_group_label;
      unspecified group_names;
      constexpr assertion_group_label(const char (&name)[N]);
    };

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

1 The type `assertion_group_label<N>` models `identification_label`.

```
constexpr assertion_group_label(const char (&name)[N]);
```

2 *Postconditions*: `group_names` is a range containing a single element whose value as a `string_view` compares equal to `string_view(name, N)`.

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

3 *Returns:* `Label`.

C Compatibility [diff]

C.(0 + *a*) C++ and ISO C++ 2026 [diff.cpp26]

Add a new subclause C.(0+a) ([diff.cpp26]) after [diff]

C.(0+a) C++ and ISO C++ 2026 [diff.cpp26]

C.(0 + *a*).1 General [diff.cpp26.general]

C.(0+a).1 General [diff.cpp26.general]

1 Subclause [diff.cpp26] lists the differences between C++ and ISO C++ 2026, by the chapters of this document.

C.(0 + *a*).2 [lex]: lexical conventions [diff.cpp26.lex]

C.(0+a).2 [lex]: lexical conventions [diff.cpp26.lex]

Affected-subclause: lex.key
Change: New keywords.
Rationale: Required for new features.

(0.1) — The `contract_control` keyword is added to enable functionality specific to assertion-control expressions.

Effect on original feature: Valid C++ 2026 code using `contract_control` as an identifier is not valid in this revision of C++.

7 Conclusion

In this paper, we have presented a comprehensive extension to C++26 Contracts (adopted from [P2900R14]). By allowing developers to attach compile-time assertion-control objects to contract assertions, composed from independently opt-in facets, we provide a single extensible framework that addresses a wide range of real-world deployment needs: controlling evaluation semantics, identifying groups for build-system integration, providing local violation handlers, manipulating violation messages, and exposing label data to violation handlers at run time.

The design is intentionally layered. The core framework — the assertion-control expression syntax, name lookup, and combining operator — is the foundation that all other parts build on and is the minimum that must be adopted first. The individual facets can then be adopted incrementally, with the semantic-control facets and group identification being the most urgent for closing the gap left by the C++26 Contracts MVP. The remaining facets and the full suite of Standard Library labels and utilities are valuable additions that can follow on their own schedules.

The core-language components of this proposal require EWG review, while the Standard Library components — concepts, labels, and utility types — require LEWG review. In [P3850R0], we propose a plan for prioritizing the features that are most needed for C++26.

Once adopted, C++ will have a Contracts facility with the flexibility, expressivity, and extensibility that a programming language used by millions — to write software that is used by billions — demands.

A Glossary

The following terms are used throughout this paper with the meanings defined here.

assertion-control expression A C++ expression, appearing in angle brackets after the introducer of a contract assertion (`pre`, `post`, or `contract_assert`), that is evaluated at compile time to produce an *assertion-control object*. Name lookup within an assertion-control expression finds names in `std::contracts::labels` without explicit qualification; this behavior is provided by the `<contracts>` header.

assertion-control object The compile-time value produced by evaluating an *assertion-control expression*. A type models the `assertion_control_object` concept by providing a nested `assertion_control_object` member typedef (whose value is irrelevant). An assertion-control object influences how the contract assertion to which it is attached is evaluated, without affecting the meaning of the predicate.

computed semantic The evaluation semantic produced by passing the *configured semantic* through the `compute_semantic` member function of the assertion-control object. When no se-

mantic computation label is involved, this step is skipped and the configured semantic is used directly as the *effective semantic*.

configured semantic The implementation-defined evaluation semantic selected by the build system (command-line flags, configuration files, etc.) before any label logic runs. This is the input to the semantic selection pipeline described in Section 3.3.2.

effective semantic The final evaluation semantic used to evaluate a contract assertion, after the configured semantic has been transformed by `compute_semantic` (if present) and verified against the `allowed_semantics` set (if present).

facet An independently opt-in dimension of behavior that an *assertion-control object* may participate in. Each facet is identified by the label's type satisfying a particular concept (e.g., `semantic_computation_label`, `allowed_semantics_label`, `compute_message_label`). A label need not satisfy any particular facet concept; the core requirement is only that the type model `assertion_control_object`. The set of standard facets is defined by the Standard Library and may be extended in future revisions of the standard. Each Standard facet also defines when and how a combined assertion-control object (using `operator|`) will participate in a facet if one or both of its constituent labels do.

label A `constexpr` object whose type models the `assertion_control_object` concept and that is intended to be used directly as, or as part of, an assertion-control expression. An object is a label not because it has a type named "label" but because its type satisfies the relevant concept. Standard library labels are provided in `std::contracts::labels` (e.g., `terminating`, `review`, `opt`, `audit`). Labels can be combined using `operator|` to form a combined assertion-control object that participates in the union of the facets of its operands.

B Example Implementation

This appendix presents key excerpts from a partial reference implementation of the Standard Library components described in this paper. The code is illustrative, not normative; a full, compilable version can be found at https://godbolt.org/z/xjKxfe9sf. All declarations in this appendix are within `namespace std::contracts::labels` as part of the `<contracts>` header unless otherwise noted.

B.1 `evaluation_semantic_set`

The `evaluation_semantic_set` type (in `namespace std::contracts`, not `labels`) is implemented as a thin wrapper around a fixed-size bitset, with one bit per enumerator of `evaluation_semantic`:

```
class evaluation_semantic_set {
public:
  // Constructors
  constexpr evaluation_semantic_set() = default;
  constexpr evaluation_semantic_set(const evaluation_semantic_set&) = default;
  constexpr evaluation_semantic_set(
      initializer_list<evaluation_semantic> ilist)
    : d_semantics() {
    for (evaluation_semantic s : ilist) {
      d_semantics.set(static_cast<int>(s) - 1);
    }
  }
```

```
  // Accessors
  constexpr bool contains(evaluation_semantic s) const {
    return d_semantics.test(static_cast<int>(s) - 1);
  }
  constexpr std::size_t size() const {
    return d_semantics.count();
  }
  constexpr bool empty() const {
    return d_semantics.none();
  }

  // Modifiers
  constexpr void clear() {
    d_semantics.reset();
  }
  constexpr void insert(evaluation_semantic s) {
    d_semantics.set(static_cast<int>(s) - 1);
  }
  constexpr void insert(initializer_list<evaluation_semantic> ilist) {
    for (evaluation_semantic s : ilist) {
      d_semantics.set(static_cast<int>(s) - 1);
    }
  }
  constexpr void remove(evaluation_semantic s) {
    d_semantics.reset(static_cast<int>(s) - 1);
  }

  // Set operations
  constexpr evaluation_semantic_set&
  operator&=(const evaluation_semantic_set& rhs) {
    d_semantics &= rhs.d_semantics; return *this;
  }
  constexpr evaluation_semantic_set&
  operator|=(const evaluation_semantic_set& rhs) {
    d_semantics |= rhs.d_semantics; return *this;
  }
  constexpr evaluation_semantic_set operator~() const {
    evaluation_semantic_set out; out.d_semantics = ~d_semantics; return out;
  }

  friend constexpr evaluation_semantic_set operator&(
      const evaluation_semantic_set& l, const evaluation_semantic_set& r) {
    auto out = l; out &= r; return out;
  }
  friend constexpr evaluation_semantic_set operator|(
      const evaluation_semantic_set& l, const evaluation_semantic_set& r) {
    auto out = l; out |= r; return out;
  }

  // Comparisons
  friend constexpr bool operator==(
      evaluation_semantic_set l, evaluation_semantic_set r) {
    return l.d_semantics == r.d_semantics;
  }

  // Utility Factories
  static constexpr evaluation_semantic_set all() {
    return {evaluation_semantic::ignore,   evaluation_semantic::enforce,
            evaluation_semantic::observe,  evaluation_semantic::quick_enforce};
  }
  static constexpr evaluation_semantic_set none() { return {}; }

private:
  bitset<4> d_semantics;
};
```

The categorization free functions described in Section 3.3.2 are straightforward switch statements over the enumerators. For example, `is_checked`:

```
constexpr bool is_checked(evaluation_semantic s) {
  switch (s) {
    case evaluation_semantic::enforce:
    case evaluation_semantic::quick_enforce:
    case evaluation_semantic::observe:
      return true;
    default:
      return false;
  }
}
```

The remaining functions (`is_unchecked`, `is_terminating`, `is_continuing`) follow the same pattern with their respective enumerator sets.

B.2 Facet Concepts

Each facet concept extends the base `assertion_control_object` concept with one additional member requirement. The pattern is uniform: check for a nested typename or a member with a specific signature.

The base concept, required by all assertion-control objects (see Section 3.3):

```
template <class T>
concept assertion_control_object =
  requires { typename T::assertion_control_object; };
```

Restricting the set of allowed evaluation semantics (see Section 3.3.1):

```
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});
  };
```

Computing the effective semantic from the configured semantic (see Section 3.3.2):

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

Providing a local violation handler (see Section 3.3.4):

```
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>);
  };
```

Declaring dimensions for mutual-exclusivity checking (see Section 3.3.3):

```
template <typename T>
concept dimensioned_label =
  assertion_control_object<T> &&
  is_specialization_of_v<typename T::dimensions, dimension_list>;
```

Identifying group membership for configuration-driven control (see Section 3.3.5):

```
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>;
  };
```

Filtering the comment and message fields of the `contract_violation` object (see Section 3.3.6):

```
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*>;
  };

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*>;
  };
```

Providing a type-erased query interface to the contract-violation handler (see Section 3.3.7):

```
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*>;
  };
```

B.3 Simple Labels

The simplest possible label satisfies only the base `assertion_control_object` concept and participates in no facets. It serves as an identity element for `operator|` and as a base for user-defined labels:

```
struct empty_label_t {
  using assertion_control_object = empty_label_t;
};
constexpr empty_label_t empty_label;
```

A label that participates in a single facet needs only one additional member. The `review` label, for instance, models `semantic_computation_label` by providing a `compute_semantic` member function that maps terminating semantics to *observe*:

```
struct review_t {
  using assertion_control_object = review_t;

  constexpr evaluation_semantic
  compute_semantic(evaluation_semantic in) const {
    if (is_terminating(in)) return evaluation_semantic::observe;
```

```
    return in;
  }
};
constexpr review_t review{};
```

B.4 `allowed_semantics_t`

The `allowed_semantics_t` template provides a convenient way to create labels that restrict the set of allowed evaluation semantics. Standard labels like `terminating` are specializations of this template:

```
template <evaluation_semantic... allowed>
struct allowed_semantics_t {
  using assertion_control_object = allowed_semantics_t<allowed...>;
  static constexpr evaluation_semantic_set allowed_semantics = {allowed...};
};

constexpr allowed_semantics_t<evaluation_semantic::quick_enforce,
                              evaluation_semantic::enforce> terminating;
```

B.5 `fixed_message_label_t`

A simple label that replaces the violation message with a compile-time string literal:

```
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;
};
```

B.6 Group Labels and `""group`

The `""group` user-defined literal operator provides a convenient way to turn a string literal into an `identification_label`. It is backed by an implementation class template parameterized on the string length:

```
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; }
```

The `group_names` member is a single-element array of character arrays, satisfying the `identification_label` concept directly. When two group labels are combined with `operator|`, the combined label's `group_names` will contain the union of both names (see Section 3.3.5).

B.7 `combined_label`

The `combined_label` class template is the return type of `operator|`. It uses a CRTP-like base-class model to compose facet behaviors independently: for each facet, a separate base class template has four partial specializations selected by `requires` clauses — one for when neither operand satisfies the concept, one for each operand alone, and one for when both do. This design lets each facet's combination logic be defined in isolation without any facet needing to know about the others.

The final `combined_label` class inherits from all facet bases and stores both constituent labels:

```
template <assertion_control_object LHS,
          assertion_control_object RHS>
struct __combined_label
: __combined_allowed_semantics_label<LHS, RHS>
, __combined_semantic_computation_label<LHS, RHS>
, __combined_local_violation_label<LHS, RHS>
, __combined_dimensioned_label<LHS, RHS>
, __combined_comment_label<LHS, RHS>
, __combined_message_label<LHS, RHS>
, __combined_queryable_label<LHS, RHS>
, __combined_identification_label<LHS, RHS>
{
  using assertion_control_object = __combined_label<LHS, RHS>;

  constexpr __combined_label(const LHS& lhs, const RHS& rhs)
    : __combined_allowed_semantics_label<LHS, RHS>(lhs, rhs)
    , _lhs(lhs), _rhs(rhs) {}

  LHS _lhs;
  RHS _rhs;
};
```

The following subsubsections show the implementation of each facet base class.

B.7.1 Allowed Semantics

When combining allowed semantics, the result is the intersection of the two sets. If neither operand has `allowed_semantics`, the combined label does not satisfy the concept.

```
template <typename LHS, typename RHS>
struct __combined_allowed_semantics_label {
  constexpr __combined_allowed_semantics_label(const LHS&, const RHS&) {}
};

template <typename LHS, typename RHS>
  requires allowed_semantics_label<LHS>
struct __combined_allowed_semantics_label<LHS, RHS> {
  decltype(LHS::allowed_semantics) allowed_semantics;
  constexpr __combined_allowed_semantics_label(const LHS& lhs, const RHS&)
    : allowed_semantics(lhs.allowed_semantics) {}
};

template <typename LHS, typename RHS>
  requires allowed_semantics_label<RHS>
struct __combined_allowed_semantics_label<LHS, RHS> {
  decltype(RHS::allowed_semantics) allowed_semantics;
  constexpr __combined_allowed_semantics_label(const LHS&, const RHS& rhs)
    : allowed_semantics(rhs.allowed_semantics) {}
};

template <typename LHS, typename RHS>
```

```
  requires allowed_semantics_label<LHS> && allowed_semantics_label<RHS>
struct __combined_allowed_semantics_label<LHS, RHS> {
  const evaluation_semantic_set allowed_semantics;
  constexpr __combined_allowed_semantics_label(const LHS& lhs, const RHS& rhs)
    : allowed_semantics(evaluation_semantic_set(lhs.allowed_semantics)
                      & evaluation_semantic_set(rhs.allowed_semantics)) {}
};
```

B.7.2 Semantic Computation

Semantic computation chains left to right: the configured semantic passes through the left operand's `compute_semantic` first, then through the right operand's. After each step, the result is verified against the allowed set (if present).

```
template <typename LHS, typename RHS>
struct __combined_semantic_computation_label {
  template <typename Self> requires semantic_computation_label<LHS>
  consteval evaluation_semantic compute_semantic(
      this const Self& self, evaluation_semantic s) {
    auto output = self._lhs.compute_semantic(s);
    if constexpr (allowed_semantics_label<Self>) {
      if (!evaluation_semantic_set(self.allowed_semantics).contains(output))
        throw "Non-allowed semantic from lhs";
    }
    return output;
  }

  template <typename Self> requires semantic_computation_label<RHS>
  consteval evaluation_semantic compute_semantic(
      this const Self& self, evaluation_semantic s) {
    auto output = self._rhs.compute_semantic(s);
    if constexpr (allowed_semantics_label<Self>) {
      if (!evaluation_semantic_set(self.allowed_semantics).contains(output))
        throw "Non-allowed semantic from rhs";
    }
    return output;
  }

  template <typename Self>
    requires semantic_computation_label<LHS> && semantic_computation_label<RHS>
  consteval evaluation_semantic compute_semantic(
      this const Self& self, evaluation_semantic s) {
    auto output = self._lhs.compute_semantic(s);
    if constexpr (allowed_semantics_label<Self>) {
      if (!evaluation_semantic_set(self.allowed_semantics).contains(output))
        throw "Non-allowed semantic from lhs";
    }
    output = self._rhs.compute_semantic(output);
    if constexpr (allowed_semantics_label<Self>) {
      if (!evaluation_semantic_set(self.allowed_semantics).contains(output))
        throw "Non-allowed semantic from rhs";
    }
    return output;
  }
};
```

B.7.3 Local Violation Handling

Local violation handlers are chained right to left: the right operand's handler runs first, and if it returns `handled`, the left operand's handler is skipped. When both handlers return `void`, the combined

handler also returns `void`.

```
template <typename LHS, typename RHS>
struct __combined_local_violation_label {};

template <typename LHS, typename RHS> requires local_violation_label<LHS>
struct __combined_local_violation_label<LHS, RHS> {
  template <typename Self>
  auto handle_contract_violation(this Self&& self, const contract_violation& v) {
    return self._lhs.handle_contract_violation(v);
  }
};

template <typename LHS, typename RHS> requires local_violation_label<RHS>
struct __combined_local_violation_label<LHS, RHS> {
  template <typename Self>
  auto handle_contract_violation(this Self&& self, const contract_violation& v) {
    return self._rhs.handle_contract_violation(v);
  }
};

template <typename LHS, typename RHS>
  requires local_violation_label<LHS> && local_violation_label<RHS>
struct __combined_local_violation_label<LHS, RHS> {
  using lhs_ret = decltype(std::declval<LHS>()
      .handle_contract_violation(std::declval<contract_violation>()));
  using rhs_ret = decltype(std::declval<RHS>()
      .handle_contract_violation(std::declval<contract_violation>()));
  using return_type = std::conditional_t<
      std::is_same_v<lhs_ret, void> && std::is_same_v<rhs_ret, void>,
      void, violation_handled>;

  template <typename Self>
  return_type handle_contract_violation(
      this Self&& self, const contract_violation& v) {
    if constexpr (std::is_convertible_v<rhs_ret, violation_handled>) {
      if (self._rhs.handle_contract_violation(v) == violation_handled::handled)
        return violation_handled::handled;
    } else {
      self._rhs.handle_contract_violation(v);
    }
    if constexpr (std::is_convertible_v<lhs_ret, violation_handled>) {
      if (self._lhs.handle_contract_violation(v) == violation_handled::handled)
        return violation_handled::handled;
    } else {
      self._lhs.handle_contract_violation(v);
    }
    if constexpr (std::is_same_v<return_type, void>) return;
    else return violation_handled::not_handled;
  }
};
```

B.7.4 Dimensions

Combining dimensions produces the union of both dimension lists. If the intersection is non-empty, the combination is ill-formed (enforced by a `static_assert`).

The `dimension_list` template and its helpers provide the compile-time list operations needed by this facet:

```
template <typename... Dims> struct dimension_list {};

template <> struct dimension_list<> {
```

```
  template <typename U>
  static constexpr bool contains() { return false; }
  static constexpr std::size_t size = 0;
};

template <typename T, typename... Dims>
struct dimension_list<T, Dims...> {
  template <typename U>
  static constexpr bool contains() {
    if constexpr (std::is_same_v<T, U>) return true;
    else return dimension_list<Dims...>::template contains<U>();
  }
  static constexpr std::size_t size = 1 + sizeof...(Dims);
};

template <typename... D1, typename... D2>
constexpr dimension_list<D1..., D2...>
concat_lists(dimension_list<D1...>, dimension_list<D2...>) {
  return {};
}

template <typename... D2>
dimension_list<> intersect_lists(dimension_list<>, dimension_list<D2...>);

template <typename T, typename... D1, typename... D2>
constexpr auto intersect_lists(
    dimension_list<T, D1...>, dimension_list<D2...> rhs) {
  auto rest = intersect_lists(dimension_list<D1...>{}, rhs);
  if constexpr (rhs.template contains<T>())
    return concat_lists(dimension_list<T>{}, rest);
  else
    return rest;
}
```

With those helpers, the combination base class checks for overlapping dimensions at compile time and concatenates the two lists:

```
template <typename LHS, typename RHS>
struct __combined_dimensioned_label {};

template <typename LHS, typename RHS> requires dimensioned_label<LHS>
struct __combined_dimensioned_label<LHS, RHS> {
  using dimensions = typename LHS::dimensions;
};

template <typename LHS, typename RHS> requires dimensioned_label<RHS>
struct __combined_dimensioned_label<LHS, RHS> {
  using dimensions = typename RHS::dimensions;
};

template <typename LHS, typename RHS>
  requires dimensioned_label<LHS> && dimensioned_label<RHS>
struct __combined_dimensioned_label<LHS, RHS> {
  static_assert(decltype(intersect_lists(
      typename LHS::dimensions{},
      typename RHS::dimensions{}))::size == 0);
  using dimensions = decltype(concat_lists(
      typename LHS::dimensions{},
      typename RHS::dimensions{}));
};
```

B.7.5 Comment and Message

Comment and message filters are chained left to right: the output of the left operand's filter becomes the input to the right operand's. The comment and message facets follow the same pattern; we show the comment version here (message is identical with `compute_message` substituted throughout).

```
template <typename LHS, typename RHS>
struct __combined_comment_label {};

template <typename LHS, typename RHS> requires compute_comment_label<LHS>
struct __combined_comment_label<LHS, RHS> {
  template <typename Self>
  constexpr const char* compute_comment(this Self&& self, const char* c) {
    return self._lhs.compute_comment(c);
  }
};

template <typename LHS, typename RHS> requires compute_comment_label<RHS>
struct __combined_comment_label<LHS, RHS> {
  template <typename Self>
  constexpr const char* compute_comment(this Self&& self, const char* c) {
    return self._rhs.compute_comment(c);
  }
};

template <typename LHS, typename RHS>
  requires compute_comment_label<LHS> && compute_comment_label<RHS>
struct __combined_comment_label<LHS, RHS> {
  template <typename Self>
  constexpr const char* compute_comment(this Self&& self, const char* c) {
    return self._rhs.compute_comment(self._lhs.compute_comment(c));
  }
};
```

B.7.6 Query

When both operands are queryable, the combined label delegates to the left operand first. If the left operand has no result for the given index, a binary search determines how many results the left operand has for that key, and the reduced index is passed to the right operand.

```
template <typename LHS, typename RHS>
struct __combined_queryable_label {
  template <typename Self> requires queryable_label<LHS>
  constexpr void* query(this Self&& self, const void* key,
                        std::size_t index = 0) {
    return self._lhs.query(key, index);
  }

  template <typename Self> requires queryable_label<RHS>
  constexpr void* query(this Self&& self, const void* key,
                        std::size_t index = 0) {
    return self._rhs.query(key, index);
  }

  template <typename T>
    requires queryable_label<LHS> && queryable_label<RHS>
  static constexpr std::size_t countAvailable(
      const T& label, const void* key, std::size_t max) {
    if (label.query(key, 0) == nullptr) return 0;
    std::size_t min = 0;
    while (min + 1 < max) {
      std::size_t mid = min + ((max - min) >> 1);
```

```
      if (label.query(key, mid) == nullptr) max = mid;
      else                                  min = mid;
    }
    return min + 1;
  }

  template <typename Self>
    requires queryable_label<LHS> && queryable_label<RHS>
  constexpr void* query(this Self&& self, const void* key,
                        std::size_t index = 0) {
    void* output = self._lhs.query(key, index);
    if (nullptr != output) return output;
    std::size_t lhsCount = countAvailable(self._lhs, key, index);
    return self._rhs.query(key, index - lhsCount);
  }
};
```

B.7.7 Identification

Combining identification labels should produce a label whose `group_names` is the sorted, unique union of the group names from both constituents. A full implementation would need to either copy the names into a dynamically sized compile-time array or provide a view that performs a merge over both ranges. The following shows the intended structure; a complete implementation is left for a future revision.

```
template <typename LHS, typename RHS>
struct __combined_identification_label {};

template <typename LHS, typename RHS> requires identification_label<LHS>
struct __combined_identification_label<LHS, RHS> {
  // group_names wraps self->_lhs.group_names
};

template <typename LHS, typename RHS> requires identification_label<RHS>
struct __combined_identification_label<LHS, RHS> {
  // group_names wraps self->_rhs.group_names
};

template <typename LHS, typename RHS>
  requires identification_label<LHS> && identification_label<RHS>
struct __combined_identification_label<LHS, RHS> {
  // group_names is the sorted, unique merge of both
};
```

B.8 `operator|`

The free-function `operator|` simply constructs a `__combined_label` from its two operands:

```
template <assertion_control_object LHS, assertion_control_object RHS>
constexpr assertion_control_object auto
operator|(const LHS& lhs, const RHS& rhs) {
  return __combined_label<LHS, RHS>(lhs, rhs);
}
```

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 earlier versions of this paper and providing editorial feedback.

Bibliography

[N3867] Gustedt, "C semantics for contracts, v3", 2026

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

[P2064R0] Herb Sutter, "Assumptions", 2020
http://wg21.link/P2064R0

[P2098R1] Walter E Brown and Bob Steagall, "Proposing `std::is_specialization_of`", 2020
http://wg21.link/P2098R1

[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
http://wg21.link/P3097R1

[P3099R2] Timur Doumler, Peter Bindels, and Joshua Berne, "Contracts for C++: User-defined diagnostic messages", 2026
http://wg21.link/P3099R2

[P3100R6] Timur Doumler and Joshua Berne, "Implicit contract assertions", 2026
http://wg21.link/P3100R6

[P3261R2] Joshua Berne, "Revisiting `const`-ification in Contract Assertions", 2024
http://wg21.link/P3261R2

[P3271R1] Lisa Lippincott, "Function Types with Usage (Contracts for Function Pointers)", 2024
http://wg21.link/P3271R1

[P3391R2] Barry Revzin, "constexpr std::format", 2025
http://wg21.link/P3391R2

[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

[P3850R0] Timur Doumler and Joshua Berne, "A proposed plan for extending Contracts in C++29", 2026
http://wg21.link/P3850R0