Controlling Contract-Assertion Properties

Document #:	P3400R1
Date:	2025-02-28
Project:	Programming Language C++
Audience:	SG21 (Contracts)

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

Abstract

The Contracts facility proposed in [P2900R12] provides significant flexibility for tool vendors to enable control over contract-assertion behavior, along with defaults that enable power and principled use of contract assertions in a wide variety of contexts. Many users, however, demand in-source input and control over that flexibility, and other advanced features are needed for widespread adoption of Contracts in more environments. To facilitate these controls, we present here a design for how to configure contract assertions using compile-time C++ objects, achieving flexibility, control, and expressivity far better than any bespoke solution can offer.

Contents

Revision History

Revision 1

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

Revision 0

- Original version of the paper for discussion during an SG21 telecon

1 Introduction

A key feature missing from the Contracts MVP, [P2900R12], is the ability to distinguish contract assertions from one another by identifying common properties separate from their predicates. Up to this point, a number of features have been postponed rather than commit to the language a bespoke syntax for each individual property that might be specified for a contract assertion.

To provide that requested functionality, we propose a mechanism to add a place in the syntax to produce an *assertion-control object* from an arbitrary expression, effectively using these objects to parameterize the various kinds of contract assertions. The individual objects used to constitute an assertion-control object are, in general, referred to as *labels*:

```
struct my_label_t {};

constexpr my_label_t my_label;

void f(int i)
  pre<my_label>( i > 0 );
```

Much like how a coroutine's promise object controls the behavior of a coroutine, the type and value of the control object alters the behavior of a contract assertion. Because some properties that can be controlled are static and others are dynamic, the assertion-control object itself is the result of a constant expression (and hence why `my_label` above is a `constexpr` variable).

Multiple labels can be combined easily using `operator|` that is available when the `<contracts>` header is included (see Section 2.1.3):

```
struct my_label_2_t {};
constexpr my_label_2_t my_label_2;

void g(int i)
  pre<my_label | my_label_2>(i > 0 );
```

Once available, assertion-control objects allow features that will enable us to achieve the following fine-grained control over the properties of contract assertions.

- Guide the implementation-defined selection of contract-evaluation semantics or control other external tools that apply to source code (see Section 2.2.1).
- Restrict the set of allowed evaluation semantics for a contract assertion (see Section 2.2.2).

- Control the effective semantic with which a contract assertion will be evaluated (see Section 2.2.3).
- Use an alternate contract-violation handler, or wrap the default contract-violation handler (see Section 2.2.4).
- Denote whether a function contract assertion is caller facing or callee facing (see Section 2.2.6).

Consider, for example, a library that has the following requirements for its function contract assertions.

- Contract assertions should always be enforced or quick-enforced.
- When a contract-violation handler is invoked, a library-specific version should be invoked first.

The first task can be accomplished with a simple label that identifies the named evaluation semantics as the only ones that can be used:

```
struct enforce_or_quick_enforce_t {
  static constexpr evaluation_semantic_set allowed_semantics =
    evaluation_semantic_set(evaluation_semantic::enforce,
                            evaluation_semantic::quick_enforce);
};
constexpr enforce_or_quick_enforce = {};
```

Another label to provide a contract-violation handler that will be invoked first and then delegate to the globally installed one is equally simple to define:

```
struct my_library_violation_handler_t {
  std::true_type handle_contract_violation(
    const std::contracts::contract_violation& violation)
  {
    // Do stuff with violation object.
    return {};  // Return true value to continue on to next handler;
                // returning false or void would finish handling
                // of the violation.
  }
}
constexpr my_library_violation_handler = {};
```

These labels can then be combined into one label with both properties using the free function `std::contracts::combine_labels`:

```
constexpr auto my_lib_assertion =
  std::contracts::combine_labels(enforce_or_quick_enforce,
                                 my_library_violation_handler);
```

Then, this combined control object can be used for contract assertions to both limit the allowed evaluation semantics and hook in the library-specific violation handling:

```
void f()
  pre<my_lib_assertion>(true)
  post<my_lib_assertion>(true)
{
```

```
  contract_assert<my_lib_assertion> (true);
}
```

Other features described below would allow you to apply these assertion-control objects as the default for all contract assertions in a scope (see Section 2.1.5) or as the control object for implicit preconditions of core-language operations (see Section 2.2.8).

Tools to reduce verbosity when writing the expressions that initialize assertion-control objects are described in Section 2.1.2, and the ability to combine assertion-control objects using `operator|` within assertion-control expressions is described in Section 2.1.3.

A number of the use cases for assertion-control objects are simple and ubiquitous enough including such objects in the Standard Library itself is clearly justified so that simple wheels do not need to be regularly reinvented; these use cases are described in Section 2.2.7.

Finally, assertion-control objects provide a mechanism to describe and control the behaviors of implicit contract assertions as introduced by [P3100R1] or [P3599R0], which we describe in Section 2.2.8.

Put all together, pursing this design for labels will allow the Standard to provide the rich and effective set of controls needed in all environments where C++ is used.

2 Proposal

We will break down details of this proposal into two basic categories.

1. Section 2.1, the core features, explains how we attach *assertion-control objects* to contract assertions, describes ways to improve the usability of assertion-control objects, and discusses their various applications.
2. Section 2.2, the controllable properties, are those things that assertion-control objects can actually *do*. Each such property will specify
    - what concept an assertion-control object must satisfy to control the property in question
    - how assertion-combined assertion-control objects will determine if (and how) they satisfy the concept
    - what properties change based on the values and behavior of assertion-control objects that satisfy the concept

2.1 Core Features

2.1.1 Assertion-Control Objects

An assertion-control object can be thought of as a parameter for the type of a contract assertion, and we leverage the well-known C++ syntax of using angle brackets (`<` and `>`) to specify type parameters.

To begin with, we will modify the grammar productions for `pre`, `post`, and `contract_assert` to have an optional assertion-control expression:

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, so it must satisfy the `assertion_control_object` concept:

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

Note the following caveats:

- 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. To maintain freedom to adopt such proposals in the future, we restrict assertion-control objects to class types for now.
- To make some of the other features here more targeted and to avoid degenerate misuse of assertion-control objects, we might consider instead specifying this concept to require a particular tag be used to identify assertion-control objects:

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

  When using the feature, however, it is unfortunate if this name needs to be added to all labels used, even ones which would otherwise be empty, so for the rest of this proposal we will assume that the only requirement on assertion control objects is that they be class types.

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. This property will become observable when we expose the assertion-control objects to the contract-violation handler later (in Section 2.1.4).

- Naturally, if the assertion-control expression is not a constant expression, the program is ill-formed, and this error is not subject to SFINAE.
- We do not want to support assertion-control objects being different in different translation units for the same contract assertion, so we require that equivalent objects be produced by the expressions regardless of when they are evaluated. Of course, the same object can still produce different results when queried for different evaluations of the same contract, such as when determining the contract-evaluation semantic as described in Section 2.2.3 below.
- The expression in all other ways behaves as any other constant expression with regards to name lookup, overload resolution, and other behaviors.

Proposal 1: Assertion-Control Objects
Allow an optional *assertion-control-expression* to be specified as part of a *precondition-specifier*, *postcondition-specifier*, or *assertion-statement*.

2.1.2 Name Lookup in 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 only particularly useful 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:

assertion-control-using-directive :
attribute-specifier-seq$_{opt}$ *'contract_assert' 'using' 'namespace' nested-name-specifier*$_{opt}$ *namespace-name ';'*

assertion-control-using-declaration :
'contract_assert' 'using' using-declarator-list

Note the following caveats:

- Because a new keyword, `contract_assert`, is already unambiguously associated with contract assertions, we make use of that same keyword for these new kinds of statements. Alternative syntax proposals are welcome.
- Unlike user-defined literals, which make use of nested namespaces and normal using directives, no overlap occurs between the literal operations introduced into a namespace and other uses of entities within that namespace. In other words, adding a using directive to include `std::literals` does not pollute the results of name lookup for anything other than user-defined literal suffixes. Doing the same for contract-assertion labels would not be possible since they are simply regular C++ variables.
- Segregating names intended to be used as labels into their own namespace is still helpful so that unwanted names are not pulled into name lookup indiscriminately. Therefore, we propose that Standard-Library labels be placed in the namespace `std::contracts::labels`.

- The Standard-Library labels should all be usable easily when importing or including the `<contracts>` header, so that header should include a assertion control using directive:

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

Proposal 2: Assertion Using Directives

Introduce *assertion-control-using-directives* and *assertion-control-using-declarations* that make additional names available within assertion-control expressions.

2.1.3 Combining Assertion-Control Objects

A primary benefit of allowing arbitrary expressions to determine the values of assertion-control objects is that these expressions enable combining different labels in arbitrary ways. Consider, for example, that we might have labels with orthogonal purposes, such as to mark a label 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.

Importantly, a default implementation is provided in the namespace `std::contracts::labels` that will produce an assertion-control object with 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);
}
```

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

By implementing the combination logic as a library function, we make supporting vendor-provided extension properties easier and allowing users to define their own combination semantics simpler if the Standard-provided ones do not match their needs.

Proposal 3: Assertion-Combination Operator

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

2.1.4 Integration with `contract_violation`

By having a single assertion-control object for any contract assertion that might be violated, a natural way to expose such objects to the contract-violation handler appears: Provide a pointer to

that assertion-control object from the `contract_violation` object.

Such a pointer would, however, be somewhere between highly unsafe to use and totally useless for most objects. A category for which it is viable, however, is for polymorphic objects that can then be inspected with `dynamic_cast`. Therefore, we add a function to `contract_violation` to access the control object that returns `nullptr` if the control object was not polymorphic and a pointer to the control object otherwise:

```
namespace std::contracts {
class contract_violation {
  public:
  // ...
    void *control_object() const noexcept;
  // ...
}
```

A violation can then be inspected to see if it has a particular kind of assertion-control object:

```
struct my_dynamic_tag {
  virtual ~my_dynamic_tag() = default;
};

void handle_contract_violation(const contract_violation &violation)
{
  if (auto* dynamic_tag =
            dynamic_cast<my_dynamic_tag*>(violation.control_object()) != nullptr) {
    std::cout << "Dynamic Tag!\n";
  }
  else {
    std::cout << "No Dynamic Tag\n";
  }
}

void f()
  pre<my_dynamic_tag>( false )  // prints "Dynamic Tag!"
  pre( false );                 // prints "No Dynamic Tag"
```

For combined assertion-control objects, however, access to the object alone is not particularly useful. To facilitate identifying if any labels of a particular type are present, a utility function in `std::contracts` can search for such objects:

```
namespace std::contracts {
template <typename T>
T* get_constituent_label(void* control_object);
}
```

A combined control object will be polymorphic if and only if one of its constituent objects is polymorphic. The above function, `get_constituent_label`, will perform the following tasks.

- If `control_object` is an instance of `T`, return that.
- If `control_object` is a combined control object, recursively apply `get_constituent_label` to the constituent objects until one of them returns a non-`nullptr` value.

- Otherwise, return `nullptr`.

The violation handler shown earlier can be updated to handle combined objects as well:

```
#include <contracts>
void handle_contract_violation(const contract_violation &violation)
{
  if (auto* dynamic_tag =
            get_constituent_label<my_dynamic_tag>(violation.control_object())
                != nullptr) {
    std::cout << "Dynamic Tag!\n";
  }
  else {
    std::cout << "No Dynamic Tag\n";
  }
}

void f()
  pre<my_dynamic_tag>          ( false )  // prints "Dynamic Tag!"
  pre<my_dynamic_tag | review> ( false )  // prints "Dynamic Tag!"
  pre                          ( false ); // prints "No Dynamic Tag"
```

Proposal 4: Access-Control Object on Violation

Add an accessor for polymorphic assertion-control objects to `contract_violation`, add `get_constituent_label`, and make combined assertion-control objects conditionally polymorphic.

2.1.5 Ambient-Control Objects

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

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

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

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

implicit-assertion-control-directive :
'contract_assert' 'implicit' '<' 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.

Proposal 5: Ambient-Control Objects

Introduce *implicit-assertion-control-directives* that add ambient assertion-control objects to namespace and classes, combined with any explicit assertion-control object using `operator|`.

2.2 Controllable Properties

With great flexibility to specify arbitrary objects to control the behavior of contract assertions, we get absolutely nothing unless we also define behaviors associated with those objects. To that end, we define various concepts that assertion-control objects can satisfy and the associated behaviors that come when an assertion-control object does satisfy those concepts.

2.2.1 Identification Labels

The simplest use case for assertion-control objects and their constituent labels is to identify groups of contract assertions so that source-based tools can manipulate those groups. In particular, compilers can give users control over the default evaluation semantics chosen for such groups and how groups will interact when combined without impacting other groups when such control is exerted.

No additional proposal is needed for this functionality; the simple existence of assertion-control objects allows full command-line control of labelled contract assertions to be provided by vendors.

On the other hand, identifying labels from the command line or in other contexts based solely on object identity can be challenging, and it can instead be useful to give explicit names to categories of labels. For that purpose, we can define a concept for identification labels that allows a label to list the groups to which it belongs by name:

```
namespace std::contracts::labels {
template <typename T>
concept identification_label =
  requires (const T t) {
    { std::is_const_v<decltype(T::group_names)>;
      std::ranges::range<decltype(T::group_names)>;
      std::string(*t.group_names.begin());
    }
  };
}
```

Combining such labels using `|` 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 uniquified) combination of the

group names from each constituent object that satisfies this concept. If that list has a single element, it will also have a `group_name` member that is that single group name to which the control object belongs.

Of course, to make even easier use of this we would want a utility template for creating identification labels for specific named categories (using something like `static_string` as proposed in [P3491R1], and some compile-time use of `std::vector` and `std::string`):

```
namespace std::contracts::labels {
template <static_string GROUP_NAME>
class assertion_group_label {
public:
  constexpr static_string group_name = GROUP_NAME;
  constexpr static_vector<static_string> group_names = {group_name};
};
}
```

Proposal 6: Group Identification Labels

Standardize the use of labels meeting the `identification_label` concept to identify groups of assertion by name.

2.2.2 Allowed-Semantic Control

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.

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.

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

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

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

```
std::set<std::contracts::evaluation_semantic>:

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


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

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

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

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


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

    // Utility Factories

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

An assertion-control object with 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  =
    requires (const T t) {
      requires std::is_const_v<decltype(T::allowed_semantics)>;
      evaluation_semantic_set({t.allowed_semantics});
    };
```

```
}
```

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

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

When an implementation selects an implementation-defined semantic for a contract assertion, it shall always be from one of those semantics in the associated `allowed_semantics` set. If the set is empty or the chosen semantic is not in the allowed set, the program is ill-formed. Implementations are encouraged to document ways in which they will look at the set of allowed semantics and adjust — in a manner acceptable to users — the chosen semantic to be one of those in the set; in general, this adjustment will be the selection of a more conservative semantic.

Proposal 7: Allowed-Semantics Control

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

2.2.3 Chosen-Semantic Control

One of the most important use cases for labels is that of introducing logic that manipulates the semantic that will be used for a given contract assertion. 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 with a `constexpr compute_semantic` member function can be used to alter the semantic that will be chosen for the associated contract assertion; in other words, assertion-control objects can opt in to this feature by modelling the following concept:

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

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

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

- If the assertion-control object also models `allowed_semantics_label` and the result is not in that set, the program is ill-formed. Importantly, these two modes of influencing the semantic

for a contract assertion are not mutually exclusive. Making them so would preclude the ability to use many compound assertion-control objects, such as `never_assume | review`.

- What values will be passed at compile time to `compute_semantic` is implementation defined. At a minimum, when a contract assertion is going to be evaluated with a particular compiler-chosen semantic, that semantic will be transformed first using `compute_semantic`. Other compilation choices, such as delaying the choice of semantic to link-time or runtime, can result in the compiler needing to call `compute_semantic` for a single contract assertion many times — for each possible evaluation semantic — to determine the set of actual semantics that will be available to evaluate that contract assertion. (In other words, when the final choice of evaluation semantic will not happen until after the compilation phase of a translation unit, the compiler may need to build a full mapping from user-chosen semantic to computed semantic.)

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.[1]

Proposal 8: Compute Semantics

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

2.2.4 Local Violation Handlers

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

- Libraries and components might need to fail fast for safety or security reasons and to prevent the escape of additional information via a user-chosen contract-violation handler.
- Other libraries might wish to log additional state information on certain failures — e.g., what request was being processed at the time of failure or what the subsystem configuration was — and to do so before delegating the primary logic of contract-violation handling to the user-chosen global violation handler.

An assertion-control object that models the following concept will be able to insert its own contract-violation handling logic:

```
namespace std::contracts::labels {
template <typename T>
concept local_violation_label =
  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)),bool>);
  };
}
```

[1] For a suggestion of a possible future proposal that might enable this kind of configuration without the use of the preprocessor, see the *build environment* described in Sections 2.3.3–2.3.5 of [P2755R1].

In other words, the assertion-control object must have a member function that is a contract-violation handler that returns `void` or `bool`.

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 `void` or its value is `true`, violation handling will complete if the local violation handler returns normally, i.e., if the handler has *handled* the contract violation.

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

Proposal 9: Local Violation Handlers

Invoke the `handle_contract_violation` member of the assertion-control object (if there is one).

2.2.5 Dimensions and Mutual Exclusivity

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

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

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

An assertion-control object has a dimension any time it declares a nested name that is a specialization of `std::contracts::labels::dimensions`, i.e., when it satisfies the following concept[2]:

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

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

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

- The dimensions of a combined label is the concatenation 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.

Proposal 10: Label Dimensions

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

2.2.6 Virtual Function Interface and Implementation Assertions

The proper guarantees to provide when invoking a virtual function are to check the contract of both the statically invoked function and the final overrider selected by dynamic dispatch.[3] Therefore, precondition and postcondition specifiers on the declaration of a virtual function apply when both using that declaration for dynamic dispatch and when the declaration is invoked.

Sometimes, however, specifying distinct contract assertions for the caller-facing set and the callee-facing set is helpful, especially when a virtual function provides a significantly less specific interface than the actual definition of that virtual function does.

Both these properties are the default unless an assertion-control object with a `caller_facing` or `callee_facing` member is associated with the contract assertion, i.e., an assertion-control object that satisfies one (or both) of the following two concepts:

```
namespace std::contracts::labels {
template <typename T>
concept caller_facing_label =
  requires(T t) {
    typename T::caller_facing;
  };

template <typename T>
concept callee_facing_label =
  requires(T t) {
    typename T::callee_facing;
  };
}
```

A combined control object will satisfy either of these concepts if any of its constituent control objects do.

[3] See [P3097R0], which has been incorporated into [P2900R12].

Proposal 11: Caller-Facing and Callee-Facing Control

Recognize assertion-control objects with the `caller-facing` and `callee-facing` members to control whether the associated function contract assertion is considered part of the caller-facing or callee-facing sets of contract assertions when invoking a function, including virtual dispatch where different function declarations are used for the two sets.

2.2.7 Standard-Library Labels

The Standard Library should provide common labels that are either simple enough to preclude rewriting everywhere or ubiquitous enough to be useful as a vocabulary type for common uses of assertion-control objects. These common labels will include the following basic label types.

- **Explicit Semantic Labels** — For each standard evaluation semantic, provide a label prefixed with `always_` that models `semantic_computation_label` and always returns the named semantic. Explicit semantic labels should be mutually exclusive.
- **Disallowed Semantic Labels** — For each standard evaluation semantic, provide a label prefixed with `never_` that models `allowed_semantics_label` whose `allowed_semantics` member is `evaluation_semantic_set::all()` with the named semantic removed.
- **Review Label** — A label named `review` models `semantic_computation_label` and returns `evaluation_semantic::observe` when a potentially terminating semantic is passed in as the chosen semantic.
- **Other Feature Labels** — For any other proposed features, such as the ability to control whether a contract assertion is caller facing and callee facing, provide basic labels to opt in to those features.

Proposal 12: Standard Labels

Provide a basic set of Standard Library assertion-control labels.

2.2.8 Core-Language Control Labels

Implicit contract assertions for core-language constructs, as explained in [P3100R1] 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 object that is the assertion-control object of that contract assertion. These Standard-Library labels will then provide two key pieces of functionality:
   (a) A portable way to discuss and configure the evaluation semantic of implicit contract assertions

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

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

```
namespace std::contracts::labels {
  assertion_group_label<"std::array_bounds">
    array_bounds;
  assertion_group_label<"std::nullptr_indirection">
    nullptr_indirection;
  assertion_group_label<"std::arithmetic_range">
    arithmetic_range;
}
```

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

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

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

   builtin-assertion-control-directive :
   'contract_assert' 'builtin' 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 **??**) 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 lookup rules for control expressions, and thus they will respect *assertion-control-using-directives*.

   For example, let us 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_assert builtin array_bounds.name        |= my_safety_checks;
contract_assert builtin nullptr_indirection.name |= my_safety_checks;
contract_assert builtin 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.

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

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

Proposal 13: Implicit Contract Labels

Provide labels in the Standard Library that match the assertion-control objects that will be used for each kind of implicit precondition check adopted by the Standard.

Proposal 14: Ambient-Implicit Labels

Add the ability to specify builtin assertion control directives to add assertion control objets to implicit contract assertions at namespace, class, and function scope.

2.2.9 Profile Labels

Profiles, as specified in [P2816R0] and more concretely in [P3081R0], might be able to either introduce or be associated with implicit contract assertions. In such cases, the association is useful to achieve by associating a named label from the Standard Library with the associated implicit contract assertions.

For example, array bounds checking might have associated with it the assertion-control object `std::contracts::labels::bounds`. Just as with other labels on implicit contract assertions, these provide both a mechanism to manipulate the chosen evaluation semantics and a way to have other precondition checks (such as the preconditions of `std::vector::operator[]`) use the same label to be controlled with other aspects of the profile as a group.

Proposal 15: Profile Labels

Provide labels in the Standard Library that are associated with each Standard profile and are attached to any contract assertions associated with or introduced by the named profile.

Note that both this proposal and the previous one (Proposal 13) might add labels to the same contract assertions, and in such cases, the assertion-control object will be the combination of all added labels using `operator|`.

2.3 Alternative Proposals

Some functionality of assertion-control objects and labels is omitted from the proposals in this paper. In some cases, SG21 has not had consensus to pursue these features, and in other cases, the features are postponed for future work because they serve even more niche use cases.

2.3.1 Disable `const`-ification

As discussed in [P3261R2], one possible escape hatch from `const`-ification that contract-assertion predicates could have is the ability to simply turn off `const`-ification when a label modelling a particular concept is attached to a contract assertion.

This approach to `const`-ification, however, is heavy handed and avoids recording the actual specific objects that need to be manipulated as non-`const` objects without actually modifying them. An operator to control `const` objects is much easier to use and provides significantly better expressiveness.

In addition, this approach was discussed and polled in an SG21 telecon, where it had almost no support:

SG21, Teleconference, 2024-12-12, Poll 2

We want to spend more SG21 time considering an optional label to contract assertions that suppresses `const`-ification, as proposed in [P3261R2] Proposal E2.

SF	F	N	A	SA
1	1	1	11	2

Result: Consensus Against

SG21, Teleconference, 2024-12-12, Poll 3

We want to spend more SG21 time considering having labels for both suppressing and enabling `const`-ification, and making a contract assertion without such a label ill-formed, as proposed in [P3261R2] Proposal E3.

SF	F	N	A	SA
2	0	0	4	9

Result: Consensus Against

Given the above results, pursuing this option as a possible controllable property of contract assertions does not seem viable.

2.3.2 Legacy Label Syntax

C++2a contracts contained a nascent ability to provide some basic labels for contracts. This functionality included the `audit` and `axiom` labels that were initially proposed as well as the concrete semantics introduced by [P1429R3] and [P1607R1].

These labels were all individual identifiers that were placed as part of a contract assertion between the introducer and the expression, with only white space separating the labels themselves. Various expansions to this syntax are explored in many parts of [P2755R1], including an attribute-based mechanism for attaching types (with their associated behavior) to user-defined labels.

That approach had some basic benefits that this proposal does not have.

- Using labels had less syntactic overhead compared to the required template-like syntax (using `<` and `>`) of this proposal.

- By not being part of otherwise normal C++ expressions, user-defined labels were able to make use of identifiers that would be keywords in other contexts, such as `new` or `default`.
- The approach to labels from [P2755R1] has been implemented in a fork of GCC based on its C++2a Contracts implementation.

On the other hand, by moving to assertion-control objects, we gain significant advantages.

- Bespoke methods of combining labels do not need to be baked into the core language but instead are part of the Standard Library (through the definition of `std::contracts::labels::operator|()`), and users are freely able to customize their labels differently if the Standard's decisions are not appropriate.
- No new mechanism, such as the `contract_label_id` specifier described in [P2755R1], need be defined to attach user-provided types and values to labels that can be used on contract assertions.
- Disambiguation rules need not be defined nor future evolution limited due to potential conflicts with existing labels in use.
- No new syntax for, for example, parameterized labels need be defined since all the needed functionality is already freely available for use in constant expressions.

Due to these benefits, we are pursing an entirely expression-based mechanism for assertion control instead of the previous bespoke sequence of identifiers-based mechanism.

2.3.3 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.

Due to that, we do not consider such an API fundamental to the basic proposal of labels and will likely instead pursue it separately at a future time.

3 Conclusion

In this paper, we have presented many layers that result in a power extension to the Contracts facility proposed in [P2900R12]. These extensions vastly extend the use cases supported by Contracts to include those that will be necessary for extensive real-world deployment of Contracts in C++ codebases around the world for many years to come.

A Example Implementations

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

An implementation will be provided in a future revision of this paper, making explicit what behaviors we expect when combining labels that satisfy the concepts proposed in this paper. An in-progress partial implementation that supports some of the concepts discussed here can be found at https://godbolt.org/z/nGE9Ez9r4.

Acknowledgments

Thanks to the many people who have helped contribute to [P2900R12]; that foundation allows the proposals in this paper to be developed.

Thanks to Iain Sandoe, Timur Doumler, Peter Bindels, and Ville Voutilainen for feedback on this paper and to Lori Hughes for reviewing this paper and providing editorial feedback.

Bibliography

[P1429R3] Joshua Berne and John Lakos, "Contracts That Work", 2019
http://wg21.link/P1429R3

[P1607R1] Joshua Berne, Jeff Snyder, and Ryan McDougall, "Minimizing Contracts", 2019
http://wg21.link/P1607R1

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

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

[P2816R0] Bjarne Stroustrup and Gabriel Dos Reis, "Safety Profiles: Type-and-resource Safe programming in ISO Standard C++", 2023
http://wg21.link/P2816R0

[P2900R12] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, "Contracts for C++", 2024
http://wg21.link/P2900R12

[P3081R0] Herb Sutter, "Core safety Profiles: Specification, adoptability, and impact", 2024
http://wg21.link/P3081R0

[P3097R0] Timur Doumler, Joshua Berne, and Gašper Ažman, "Contracts for C++: Support for virtual functions", 2024
http://wg21.link/P3097R0

[P3100R1] Timur Doumler, Gašper Ažman, and Joshua Berne, "Undefined and erroneous behaviour are contract violations", 2024
http://wg21.link/P3100R1

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

[P3491R1] Barry Revzin, Wyatt Childers, Peter Dimov, and Daveed Vandevoorde, "define_static_string,object,array", 2025
http://wg21.link/P3491R1

[P3599R0] Timur Doumler and Joshua Berne, "Initial Implicit Contract Assertions", 2024
http://wg21.link/P3599R0