# Assertion-Control Objects (P3400R4)

## P4275R0 (for EWG)

Joshua Berne - jberne4@bloomberg.net

2026-06-10

# P3400R4: Assertion-Control Objects

- C++26 Contracts: `pre`, `post`, `contract_assert`

# P3400R4: Assertion-Control Objects

- C++26 Contracts: `pre`, `post`, `contract_assert`
  - They communicate "This predicate is expected to be `true`"

# P3400R4: Assertion-Control Objects

- C++26 Contracts: `pre`, `post`, `contract_assert`
  - They communicate "This predicate is expected to be `true`"
  - Nothing else is captured in the source code

# P3400R4: Assertion-Control Objects

- C++26 Contracts: `pre`, `post`, `contract_assert`
  - They communicate "This predicate is expected to be `true`"
  - Nothing else is captured in the source code
- All other effects are controlled outside the source

# P3400R4: Assertion-Control Objects

- C++26 Contracts: `pre`, `post`, `contract_assert`
  - They communicate "This predicate is expected to be `true`"
  - Nothing else is captured in the source code
- All other effects are controlled outside the source
  - Evaluation semantic

# P3400R4: Assertion-Control Objects

- C++26 Contracts: `pre`, `post`, `contract_assert`
  - They communicate "This predicate is expected to be `true`"
  - Nothing else is captured in the source code
- All other effects are controlled outside the source
  - Evaluation semantic
  - Static analysis

# P3400R4: Assertion-Control Objects

- C++26 Contracts: `pre`, `post`, `contract_assert`
  - They communicate "This predicate is expected to be `true`"
  - Nothing else is captured in the source code
- All other effects are controlled outside the source
  - Evaluation semantic
  - Static analysis
- P3400 proposes **assertion-control objects**: an object that carries extra information and logic

# P3400R4: Assertion-Control Objects

- C++26 Contracts: `pre`, `post`, `contract_assert`
  - They communicate "This predicate is expected to be `true`"
  - Nothing else is captured in the source code
- All other effects are controlled outside the source
  - Evaluation semantic
  - Static analysis
- P3400 proposes **assertion-control objects**: an object that carries extra information and logic
  - Composed with normal C++ expressions at compile time

# P3400R4: Assertion-Control Objects

- C++26 Contracts: `pre`, `post`, `contract_assert`
  - They communicate "This predicate is expected to be `true`"
  - Nothing else is captured in the source code
- All other effects are controlled outside the source
  - Evaluation semantic
  - Static analysis
- P3400 proposes **assertion-control objects**: an object that carries extra information and logic
  - Composed with normal C++ expressions at compile time
  - Individual components are `constexpr` objects we call **labels**

# P3400R4: Assertion-Control Objects

- C++26 Contracts: `pre`, `post`, `contract_assert`
  - They communicate "This predicate is expected to be `true`"
  - Nothing else is captured in the source code
- All other effects are controlled outside the source
  - Evaluation semantic
  - Static analysis
- P3400 proposes **assertion-control objects**: an object that carries extra information and logic
  - Composed with normal C++ expressions at compile time
  - Individual components are `constexpr` objects we call **labels**
  - Each dimension of behavior is called a **facet**

# Grouping Contract Assertions

- Labels can identify a contract assertion as part of a named group

# Grouping Contract Assertions

- Labels can identify a contract assertion as part of a named group
- The group can be controlled through build configuration

# Grouping Contract Assertions

- Labels can identify a contract assertion as part of a named group
- The group can be controlled through build configuration

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

# Grouping Contract Assertions

- Labels can identify a contract assertion as part of a named group
- The group can be controlled through build configuration

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

- *"mylib::pointers"*group is a user-defined literal operator from <contracts>

# Combining Multiple Groups

- Assertions often belong to more than one group

# Combining Multiple Groups

- Assertions often belong to more than one group
- Labels combine using the pipe (|) operator

# Combining Multiple Groups

- Assertions often belong to more than one group
- Labels combine using the pipe (|) operator

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

# Combining Multiple Groups

- Assertions often belong to more than one group
- Labels combine using the pipe (|) operator

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

# Combining Multiple Groups

- Assertions often belong to more than one group
- Labels combine using the pipe (|) operator

```
void get(int * const output, int const index)
  pre<"pointers"group>(output != nullptr)
  pre<"bounds"group>  (index >= 0 && index < size())
  post<"pointers"group | "bounds"group>
       (r : output != nullptr && r == output[index]);
```

# Controlling Cost of Checking

- Fast checks are rarely worth turning off

# Controlling Cost of Checking

- Fast checks are rarely worth turning off
- Expensive checks might never be viable in production

# Controlling Cost of Checking

- Fast checks are rarely worth turning off
- Expensive checks might never be viable in production
- Standard Library labels capture this nuance:

# Controlling Cost of Checking

- Fast checks are rarely worth turning off
- Expensive checks might never be viable in production
- Standard Library labels capture this nuance:

```
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
```

# Controlling Cost of Checking

- Fast checks are rarely worth turning off
- Expensive checks might never be viable in production
- Standard Library labels capture this nuance:

```
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
```

- Your build configurationstill controls the behavior

# Controlling Cost of Checking

- Fast checks are rarely worth turning off
- Expensive checks might never be viable in production
- Standard Library labels capture this nuance:

```
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
```

- Your build configurationstill controls the behavior
  - opt will generally default to checked

# Controlling Cost of Checking

- Fast checks are rarely worth turning off
- Expensive checks might never be viable in production
- Standard Library labels capture this nuance:

```
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
```

- Your build configurationstill controls the behavior
  - `opt` will generally default to checked
  - `audit` will require explicit opt-in

# Deploying New Assertions Safely

- Adding assertions to existing code is challenging at scale

# Deploying New Assertions Safely

- Adding assertions to existing code is challenging at scale
- A new assertion can be marked so it is *observed* rather than *enforced*:

# Deploying New Assertions Safely

- Adding assertions to existing code is challenging at scale
- A new assertion can be marked so it is *observed* rather than *enforced*:

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

# Deploying New Assertions Safely

- Adding assertions to existing code is challenging at scale
- A new assertion can be marked so it is *observed* rather than *enforced*:

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

- `review` does not force checking — that is still up to the build configuration

# Deploying New Assertions Safely

- Adding assertions to existing code is challenging at scale
- A new assertion can be marked so it is *observed* rather than *enforced*:

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

- `review` does not force checking — that is still up to the build configuration
- It changes the semantic to *observe* when an enforcing semantic would have been chosen

# Combining Labels

- Being `new` is orthogonal to being expensive, or belonging to some group

# Combining Labels

- Being **`new`** is orthogonal to being expensive, or belonging to some group
- We need to be able to apply `review`, `audit`, and *`"bounds"`*`group` in any combination

# Combining Labels

- Being **`new`** is orthogonal to being expensive, or belonging to some group
- We need to be able to apply `review`, `audit`, and *`"bounds"`*`group` in any combination
- The | operator can combine labels with different purposes

# Combining Labels

- Being **`new`** is orthogonal to being expensive, or belonging to some group
- We need to be able to apply `review`, `audit`, and *`"bounds"`*`group` in any combination
- The | operator can combine labels with different purposes
- `audit | review`: only checked in specific builds *and* observed rather than enforced

# Combining Labels

- Being **new** is orthogonal to being expensive, or belonging to some group
- We need to be able to apply `review`, `audit`, and *"bounds"*`group` in any combination
- The | operator can combine labels with different purposes
- `audit | review`: only checked in specific builds *and* observed rather than enforced

```
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 processes demand that control never flow past a contract violation

# Requiring Enforcement

- Some processes demand that control never flow past a contract violation
- The `terminating` label requires an enforcing semantic:

# Requiring Enforcement

- Some processes demand that control never flow past a contract violation
- The `terminating` label requires an enforcing semantic:

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

# Requiring Enforcement

- Some processes demand that control never flow past a contract violation
- The `terminating` label requires an enforcing semantic:

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

- `terminating` is not a keyword — it resolves to a **`constexpr`** variable from `<contracts>`

# Library Hardening

- Assertions can use a terminating semantic when a hardened Standard Library is selected:

# Library Hardening

- Assertions can use a terminating semantic when a hardened Standard Library is selected:

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

# Library Hardening

- Assertions can use a terminating semantic when a hardened Standard Library is selected:

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

- Same guarantee as existing hardened preconditions

# Library Hardening

- Assertions can use a terminating semantic when a hardened Standard Library is selected:

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

- Same guarantee as existing hardened preconditions
- Implementation can base behavior on its own configuration macros or intrinsics

# The Simplest Label

- All assertion-control objects must satisfy the `assertion_control_object` concept

# The Simplest Label

- All assertion-control objects must satisfy the `assertion_control_object` concept
- Only requires a nested member typedef:

# The Simplest Label

- All assertion-control objects must satisfy the `assertion_control_object` concept
- Only requires a nested member typedef:

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

# The Simplest Label

- All assertion-control objects must satisfy the `assertion_control_object` concept
- Only requires a nested member typedef:

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

- With this label, behavior is the same as with no assertion-control object

# Implementing a label: `review`

- A label that *does* something must satisfy one or more facet concepts

# Implementing a label: `review`

- A label that *does* something must satisfy one or more facet concepts
- `review` models `semantic_computation_label`:

# Implementing a label: `review`

- A label that *does* something must satisfy one or more facet concepts
- `review` models `semantic_computation_label`:

```
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{};
}
```

# Restricting Semantics: `terminating`

- Labels can restrict rather than compute semantics

# Restricting Semantics: `terminating`

- Labels can restrict rather than compute semantics
- `allowed_semantics_t` provides a compile-time-fixed set:

# Restricting Semantics: `terminating`

- Labels can restrict rather than compute semantics
- `allowed_semantics_t` provides a compile-time-fixed set:

```
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;
}
```

# Language Features

- **Syntax**: `pre|post|contract_assert` < *assertion-control-expression* > `(...)`

# Language Features

- **Syntax**: `pre|post|contract_assert` < *assertion-control-expression* > `(...)`
  - Optionally specify an assertion-control object for a contract assertion

# Language Features

- **Syntax**: `pre|post|contract_assert` < *assertion-control-expression* > `(...)`
  - Optionally specify an assertion-control object for a contract assertion
- **Name Lookup**: `using contract_control namespace`

# Language Features

- **Syntax**: `pre|post|contract_assert` < *assertion-control-expression* > `(...)`
  - Optionally specify an assertion-control object for a contract assertion
- **Name Lookup**: `using contract_control namespace`
  - Use namespace in assertion-control expressions

# Language Features

- **Syntax**: `pre|post|contract_assert < `*assertion-control-expression*` > (...)`
  - Optionally specify an assertion-control object for a contract assertion
- **Name Lookup**: `using contract_control namespace`
  - Use namespace in assertion-control expressions
  - Lets the Standard Library provide small label names from `<contracts>`

# Language Features

- **Syntax**: `pre|post|contract_assert <` *assertion-control-expression* `> (...)`
  - Optionally specify an assertion-control object for a contract assertion
- **Name Lookup**: **`using`** `contract_control` **`namespace`**
  - Use namespace in assertion-control expressions
  - Lets the Standard Library provide small label names from `<contracts>`
- **Reuse Name Lookup**: `contract_control(` *expression* `)`

# Language Features

- **Syntax**: `pre|post|contract_assert <` *assertion-control-expression* `> (...)`
  - Optionally specify an assertion-control object for a contract assertion
- **Name Lookup**: **`using`** `contract_control` **`namespace`**
  - Use namespace in assertion-control expressions
  - Lets the Standard Library provide small label names from `<contracts>`
- **Reuse Name Lookup**: `contract_control(` *expression* `)`
  - same name lookup, outside assertion-control expressions

# Proposed Facets

- Each facet is a concept that a label can model:

# Proposed Facets

- Each facet is a concept that a label can model:
  - `allowed_semantics` — restrict which semantics are permitted

# Proposed Facets

- Each facet is a concept that a label can model:
  - `allowed_semantics` — restrict which semantics are permitted
  - `compute_semantic` — transform the configured semantic

# Proposed Facets

- Each facet is a concept that a label can model:
  - `allowed_semantics` — restrict which semantics are permitted
  - `compute_semantic` — transform the configured semantic
  - `dimensions` — declare mutual exclusion between labels

# Proposed Facets

- Each facet is a concept that a label can model:
  - `allowed_semantics` — restrict which semantics are permitted
  - `compute_semantic` — transform the configured semantic
  - `dimensions` — declare mutual exclusion between labels
  - `group_names` — identify named groups for configuration

# Proposed Facets

- Each facet is a concept that a label can model:
  - `allowed_semantics` — restrict which semantics are permitted
  - `compute_semantic` — transform the configured semantic
  - `dimensions` — declare mutual exclusion between labels
  - `group_names` — identify named groups for configuration
  - `handle_contract_violation` — provide a separate violation handler

# Proposed Facets

- Each facet is a concept that a label can model:
  - `allowed_semantics` — restrict which semantics are permitted
  - `compute_semantic` — transform the configured semantic
  - `dimensions` — declare mutual exclusion between labels
  - `group_names` — identify named groups for configuration
  - `handle_contract_violation` — provide a separate violation handler
  - `compute_comment` / `compute_message` — customize diagnostic text

# Proposed Facets

- Each facet is a concept that a label can model:
  - `allowed_semantics` — restrict which semantics are permitted
  - `compute_semantic` — transform the configured semantic
  - `dimensions` — declare mutual exclusion between labels
  - `group_names` — identify named groups for configuration
  - `handle_contract_violation` — provide a separate violation handler
  - `compute_comment` / `compute_message` — customize diagnostic text
  - `query` — identify labels (and extract information) from the violation handler

# How Labels Combine

- `operator|` produces an object that participates in the union of its constituents' facets

# How Labels Combine

- `operator|` produces an object that participates in the union of its constituents' facets
- Each facet has its own combination rule:

# How Labels Combine

- `operator|` produces an object that participates in the union of its constituents' facets
- Each facet has its own combination rule:
  - `allowed_semantics`: intersection

# How Labels Combine

- `operator|` produces an object that participates in the union of its constituents' facets
- Each facet has its own combination rule:
  - `allowed_semantics`: intersection
  - `compute_semantic`: chained left to right

# How Labels Combine

- `operator|` produces an object that participates in the union of its constituents' facets
- Each facet has its own combination rule:
  - `allowed_semantics`: intersection
  - `compute_semantic`: chained left to right
  - `group_names`: sorted, unique union

# How Labels Combine

- `operator|` produces an object that participates in the union of its constituents' facets
- Each facet has its own combination rule:
  - `allowed_semantics`: intersection
  - `compute_semantic`: chained left to right
  - `group_names`: sorted, unique union
  - `dimensions`: union; overlapping is ill-formed

# How Labels Combine

- `operator|` produces an object that participates in the union of its constituents' facets
- Each facet has its own combination rule:
  - `allowed_semantics`: intersection
  - `compute_semantic`: chained left to right
  - `group_names`: sorted, unique union
  - `dimensions`: union; overlapping is ill-formed
  - `handle_contract_violation`: chained right to left

# How Labels Combine

- `operator|` produces an object that participates in the union of its constituents' facets
- Each facet has its own combination rule:
  - `allowed_semantics`: intersection
  - `compute_semantic`: chained left to right
  - `group_names`: sorted, unique union
  - `dimensions`: union; overlapping is ill-formed
  - `handle_contract_violation`: chained right to left
  - `query`: delegated with index arithmetic

# How Labels Combine

- `operator|` produces an object that participates in the union of its constituents' facets
- Each facet has its own combination rule:
  - `allowed_semantics`: intersection
  - `compute_semantic`: chained left to right
  - `group_names`: sorted, unique union
  - `dimensions`: union; overlapping is ill-formed
  - `handle_contract_violation`: chained right to left
  - `query`: delegated with index arithmetic
- Incompatible labels fail to compile

# How Labels Combine

- `operator|` produces an object that participates in the union of its constituents' facets
- Each facet has its own combination rule:
  - `allowed_semantics`: intersection
  - `compute_semantic`: chained left to right
  - `group_names`: sorted, unique union
  - `dimensions`: union; overlapping is ill-formed
  - `handle_contract_violation`: chained right to left
  - `query`: delegated with index arithmetic
- Incompatible labels fail to compile
  - Conflicting dimensions

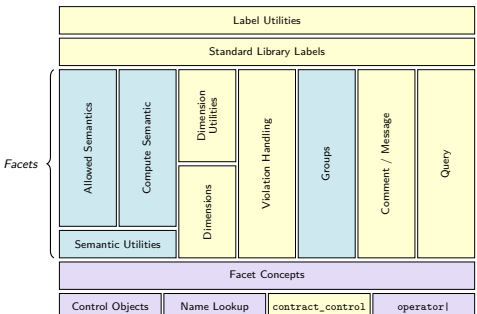
# How Labels Combine

- `operator|` produces an object that participates in the union of its constituents' facets
- Each facet has its own combination rule:
  - `allowed_semantics`: intersection
  - `compute_semantic`: chained left to right
  - `group_names`: sorted, unique union
  - `dimensions`: union; overlapping is ill-formed
  - `handle_contract_violation`: chained right to left
  - `query`: delegated with index arithmetic
- Incompatible labels fail to compile
  - Conflicting dimensions
  - Empty sets of allowed semantics

# How Labels Combine

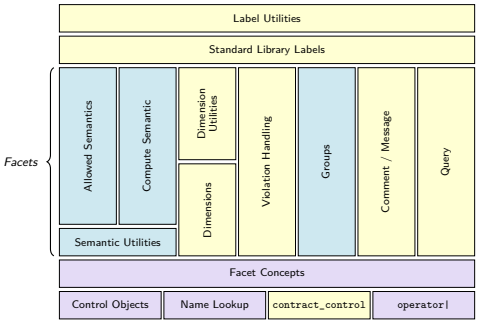
- `operator|` produces an object that participates in the union of its constituents' facets
- Each facet has its own combination rule:
  - `allowed_semantics`: intersection
  - `compute_semantic`: chained left to right
  - `group_names`: sorted, unique union
  - `dimensions`: union; overlapping is ill-formed
  - `handle_contract_violation`: chained right to left
  - `query`: delegated with index arithmetic
- Incompatible labels fail to compile
  - Conflicting dimensions
  - Empty sets of allowed semantics

# Physical Layering




## Proposal Components

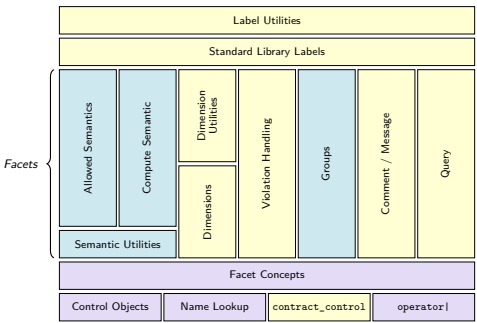
# Physical Layering




## Proposal Components

- Three categories of adoption

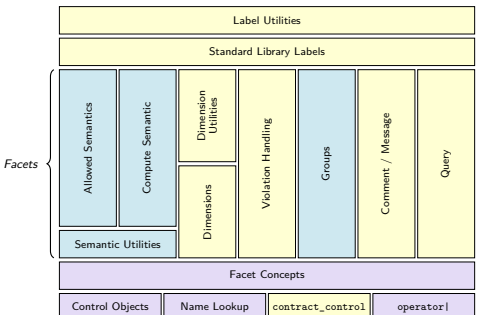
# Physical Layering




**Proposal Components**

- Three categories of adoption
- Framework — Central language feature

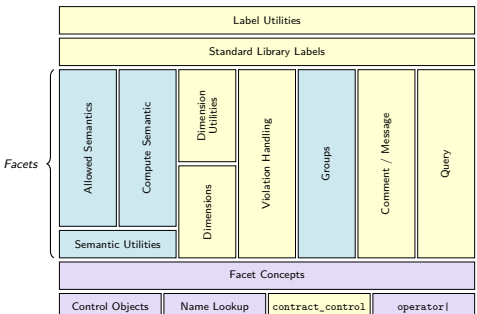
# Physical Layering




**Proposal Components**

- Three categories of adoption
- Framework — Central language feature
- Needed for C++29 — Priority facets

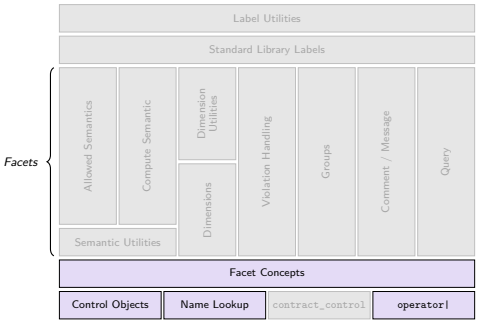
# Physical Layering




**Proposal Components**

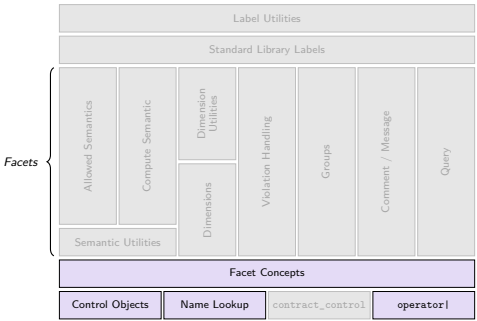
- Three categories of adoption
- Framework — Central language feature
- Needed for C++29 — Priority facets
- Additional enhancements — Further facets and standardizing expected uses

# Foundational Framework




## Essential Common Features

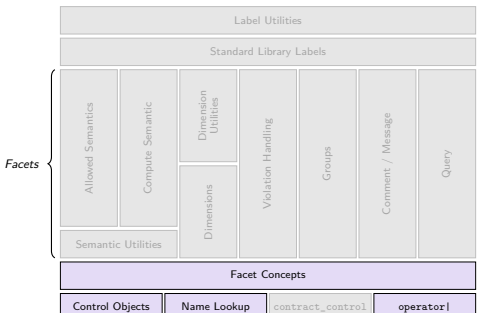
# Foundational Framework




## Essential Common Features

- Syntax for specifying assertion-control objects

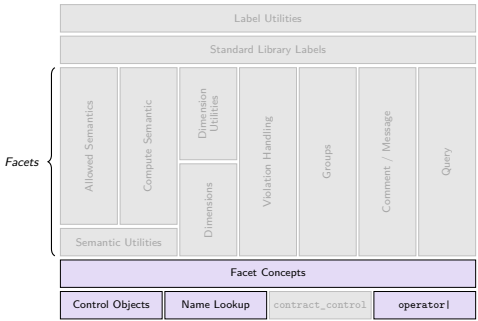
# Foundational Framework




**Essential Common Features**

- Syntax for specifying assertion-control objects
- Short names for common labels through name lookup

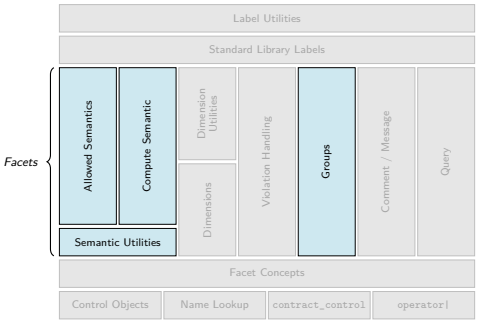
# Foundational Framework




**Essential Common Features**

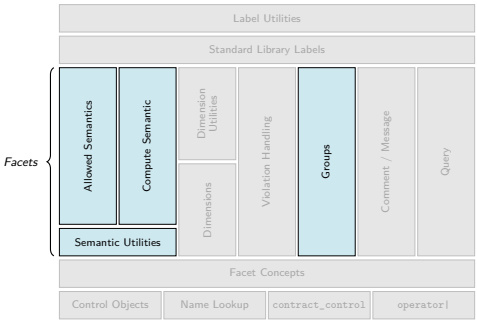
- Syntax for specifying assertion-control objects
- Short names for common labels through name lookup
- Combining labels with `operator|`

# Priority Features




**Needed for C++29**

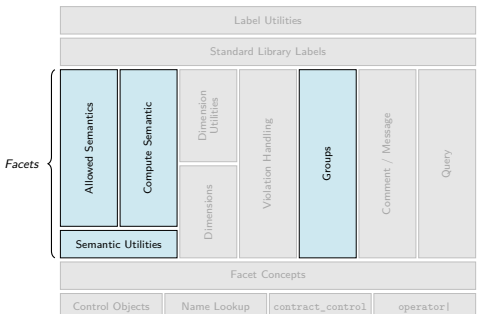
# Priority Features




**Needed for C++29**

- Three primary facets

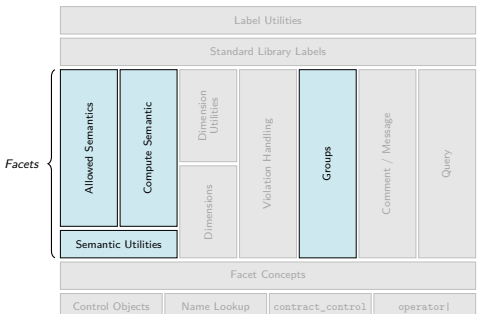
# Priority Features




**Needed for C++29**

- Three primary facets
  - Allowed semantics — `terminating`, `symbolic`, etc.

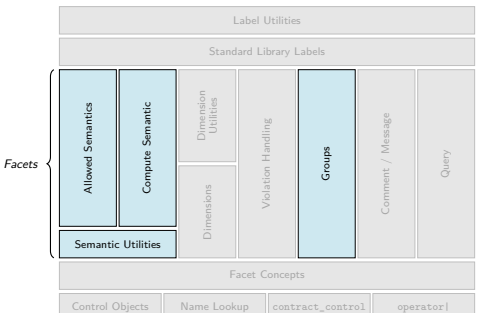
# Priority Features




## Needed for C++29

- Three primary facets
  - Allowed semantics — `terminating`, `symbolic`, etc.
  - Computed semantics — `review`, legacy control, etc.

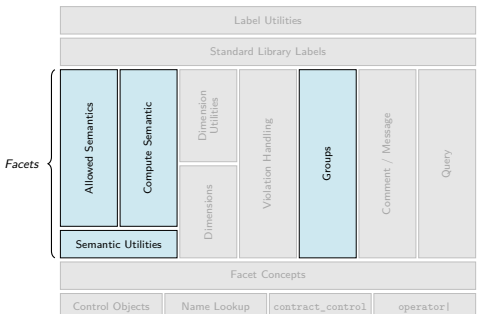
# Priority Features




**Needed for C++29**

- Three primary facets
  - Allowed semantics — `terminating`, `symbolic`, etc.
  - Computed semantics — `review`, legacy control, etc.
  - Groups — `audit`, `bounds`, etc.

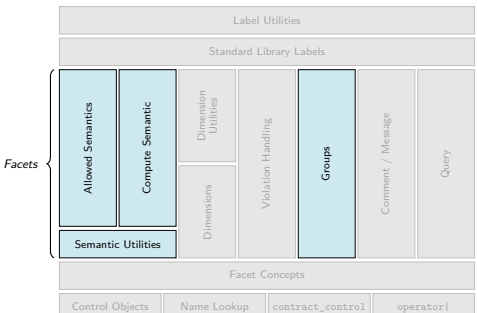
# Priority Features




**Needed for C++29**

- Three primary facets
  - Allowed semantics — `terminating`, `symbolic`, etc.
  - Computed semantics — `review`, legacy control, etc.
  - Groups — `audit`, `bounds`, etc.
- Satisfy most needed use cases for deployment at scale

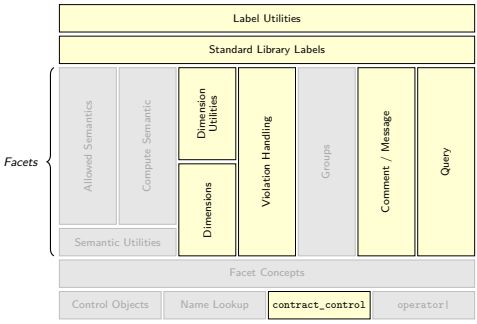
# Priority Features




**Needed for C++29**

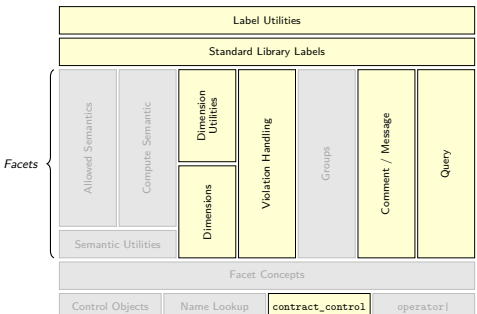
- Three primary facets
  - Allowed semantics — `terminating`, `symbolic`, etc.
  - Computed semantics — `review`, legacy control, etc.
  - Groups — `audit`, `bounds`, etc.
- Satisfy most needed use cases for deployment at scale
- Most commonly cited missing features of C++26 Contracts

# Orthogonal Additions




## Additional Enhancements

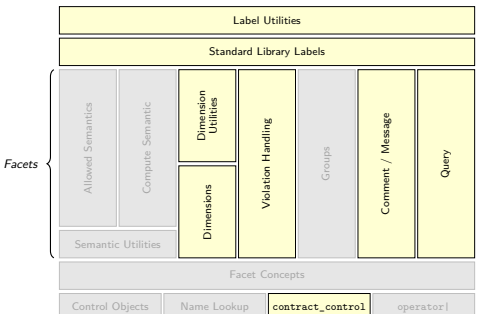
# Orthogonal Additions




## Additional Enhancements

- Many orthogonal features building on the framework

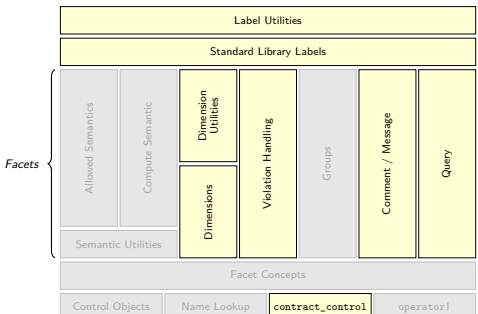
# Orthogonal Additions




**Additional Enhancements**

- Many orthogonal features building on the framework
- All serve different use cases

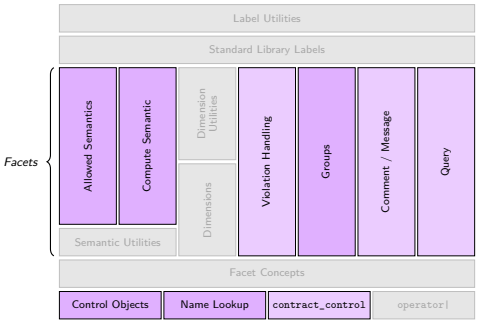
# Orthogonal Additions




**Additional Enhancements**

- Many orthogonal features building on the framework
- All serve different use cases
- Incremental adoption in any order is feasible

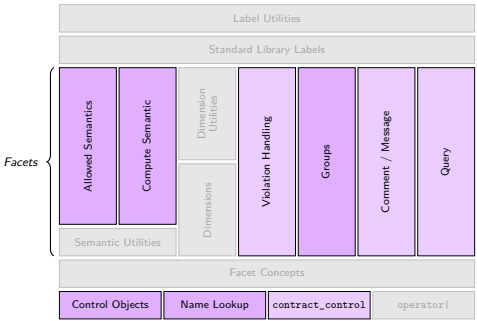
# Core-Language Features




**EWG (Core Language)**[a]

[a] Mauve selected as official EWG color by Erich Keane

# Core-Language Features

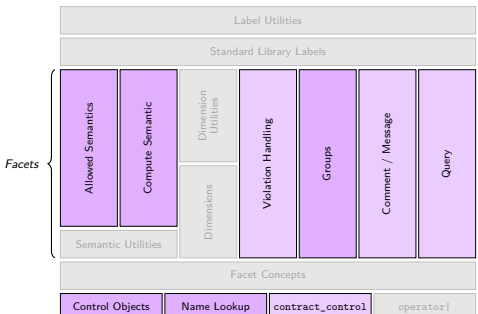



## EWG (Core Language)[a]

- Basic functionality has very little library dependency

[a] Mauve selected as official EWG color by Erich Keane

# Core-Language Features

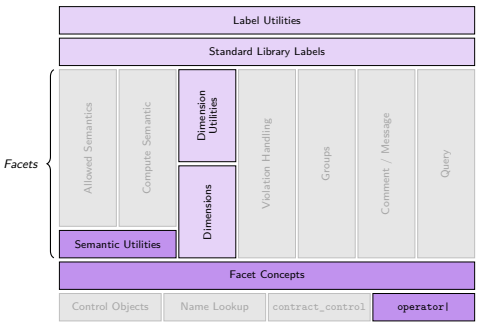



## EWG (Core Language)[a]

- Basic functionality has very little library dependency
- Additional features are all just a single point where a facet is defined and used

[a] Mauve selected as official EWG color by Erich Keane

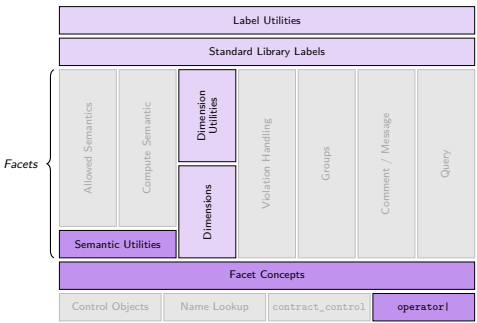
# Library Features




## LEWG (Standard Library)[a]

[a] Pantone 448 C selected as official LEWG color by Nevin Liber, Lavender (💜) selected by Inbal Levi.

# Library Features

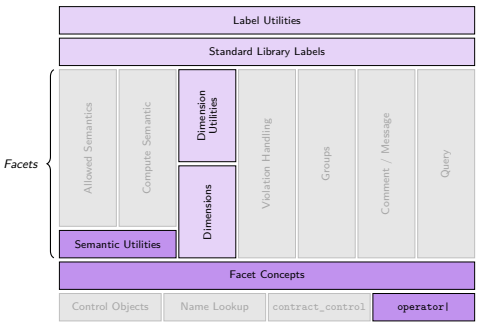



## LEWG (Standard Library)[a]

- Core functionality is guided completely by the feature design

[a] Pantone 448 C selected as official LEWG color by Nevin Liber, Lavender (💜) selected by Inbal Levi.

# Library Features

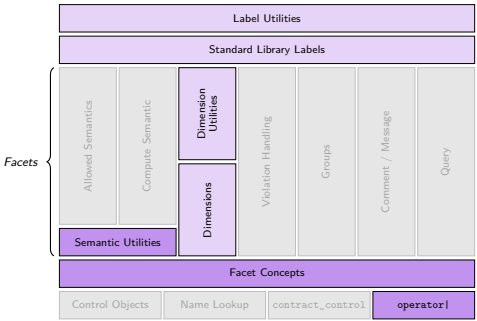



## LEWG (Standard Library)[a]

- Core functionality is guided completely by the feature design
- Additional features are the user-facing API of the feature

[a] Pantone 448 C selected as official LEWG color by Nevin Liber, Lavender (💜) selected by Inbal Levi.

# Library Features

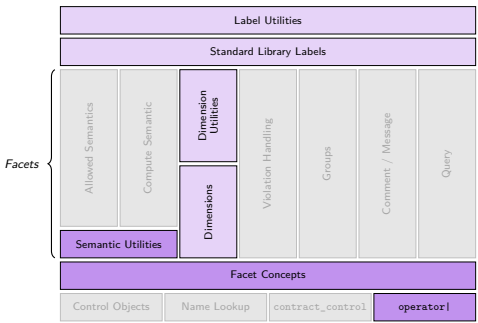



## LEWG (Standard Library)[a]

- Core functionality is guided completely by the feature design
- Additional features are the user-facing API of the feature
  - Most can be written by users themselves if needed

[a] Pantone 448 C selected as official LEWG color by Nevin Liber, Lavender (💜) selected by Inbal Levi.

# Library Features

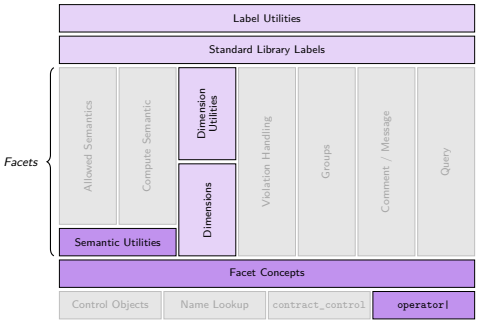



## LEWG (Standard Library)[a]

- Core functionality is guided completely by the feature design
- Additional features are the user-facing API of the feature
  - Most can be written by users themselves if needed
  - Standardize things we believe are going to be existing practice

[a] Pantone 448 C selected as official LEWG color by Nevin Liber, Lavender (💜) selected by Inbal Levi.

# Library Features




## LEWG (Standard Library)[a]

- Core functionality is guided completely by the feature design
- Additional features are the user-facing API of the feature
  - Most can be written by users themselves if needed
  - Standardize things we believe are going to be existing practice
  - Make taking advantage of certain facets easier with common utilities

[a] Pantone 448 C selected as official LEWG color by Nevin Liber, Lavender (💜) selected by Inbal Levi.

# Pause for Discussion

# Pause for Discussion

# Pause for Discussion

# Pause for Discussion

# Pause for Discussion

# Pause for Discussion

# The <> Syntax

- Contract assertions accept an assertion-control object in angle brackets:

# The <> Syntax

- Contract assertions accept an assertion-control object in angle brackets:

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

# The <> Syntax

- Contract assertions accept an assertion-control object in angle brackets:

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

- The expression inside <> must satisfy the `assertion_control_object` concept

# The <> Syntax

- Contract assertions accept an assertion-control object in angle brackets:

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

- The expression inside <> must satisfy the `assertion_control_object` concept
- No label means default behavior:

# The <> Syntax

- Contract assertions accept an assertion-control object in angle brackets:

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

- The expression inside <> must satisfy the `assertion_control_object` concept
- No label means default behavior:

```
void g(int *p)
  pre(p != nullptr);  // equivalent to pre<empty_label>(...)
```

# Name Lookup

- Labels live in `std::contracts::labels`

# Name Lookup

- Labels live in `std::contracts::labels`
- Inside <>, a special lookup rule finds names from that namespace automatically

# Name Lookup

- Labels live in `std::contracts::labels`
- Inside <>, a special lookup rule finds names from that namespace automatically
- Enabled by a **`using`** `contract_control` **`namespace`** declaration:

# Name Lookup

- Labels live in `std::contracts::labels`
- Inside <>, a special lookup rule finds names from that namespace automatically
- Enabled by a **`using`** `contract_control` **`namespace`** declaration:

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

# Name Lookup

- Labels live in `std::contracts::labels`
- Inside <>, a special lookup rule finds names from that namespace automatically
- Enabled by a **`using`** `contract_control` **`namespace`** declaration:

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

- The <contracts> header provides this declaration

# Name Lookup

- Labels live in `std::contracts::labels`
- Inside <>, a special lookup rule finds names from that namespace automatically
- Enabled by a **`using`** `contract_control` **`namespace`** declaration:

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

- The <contracts> header provides this declaration
- No need to qualify label names in assertions:

# Name Lookup

- Labels live in `std::contracts::labels`
- Inside <>, a special lookup rule finds names from that namespace automatically
- Enabled by a **`using`** `contract_control` **`namespace`** declaration:

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

- The <contracts> header provides this declaration
- No need to qualify label names in assertions:

```
#include <contracts>
void f(int *p)
  pre<terminating>(p != nullptr);  // found automatically
```

# The `contract_control` Operator

- `contract_control(expr)` applies the same lookup outside of assertions

# The `contract_control` Operator

- `contract_control(expr)` applies the same lookup outside of assertions
- Useful for building composite labels in user code:

# The `contract_control` Operator

- `contract_control(expr)` applies the same lookup outside of assertions
- Useful for building composite labels in user code:

```
constexpr auto my_label
    = contract_control(terminating | "safety"group);
```

# The `contract_control` Operator

- `contract_control(expr)` applies the same lookup outside of assertions
- Useful for building composite labels in user code:

```
constexpr auto my_label
    = contract_control(terminating | "safety"group);
```

- Makes label names available without full qualification

# The `contract_control` Operator

- `contract_control(expr)` applies the same lookup outside of assertions
- Useful for building composite labels in user code:

```
constexpr auto my_label
    = contract_control(terminating | "safety"group);
```

- Makes label names available without full qualification
- Same mechanism as inside <>, but usable anywhere

# operator|

- Labels combine using the pipe (|) operator:

# operator|

- Labels combine using the pipe (|) operator:

```
void f(int *begin, int *end)
  pre<audit | review>(std::is_sorted(begin, end));
```

# operator|

- Labels combine using the pipe (|) operator:

```
void f(int *begin, int *end)
  pre<audit | review>(std::is_sorted(begin, end));
```

- The result satisfies the union of its constituents' facets

# operator|

- Labels combine using the pipe (|) operator:

```
void f(int *begin, int *end)
  pre<audit | review>(std::is_sorted(begin, end));
```

- The result satisfies the union of its constituents' facets
- Each facet defines its own combination rule

# operator|

- Labels combine using the pipe (|) operator:

```
void f(int *begin, int *end)
  pre<audit | review>(std::is_sorted(begin, end));
```

- The result satisfies the union of its constituents' facets
- Each facet defines its own combination rule
- Incompatible labels fail to compile:

# operator|

- Labels combine using the pipe (|) operator:

```
void f(int *begin, int *end)
  pre<audit | review>(std::is_sorted(begin, end));
```

- The result satisfies the union of its constituents' facets
- Each facet defines its own combination rule
- Incompatible labels fail to compile:

```
pre<opt | audit>(...)  // Error: overlapping dimensions
```

# Allowed Semantics

- Some predicates must never be checked at run time (e.g., destructive input-iterator predicates)

# Allowed Semantics

- Some predicates must never be checked at run time (e.g., destructive input-iterator predicates)
- Some must always be enforced

# Allowed Semantics

- Some predicates must never be checked at run time (e.g., destructive input-iterator predicates)
- Some must always be enforced
- Some must never terminate

# Allowed Semantics

- Some predicates must never be checked at run time (e.g., destructive input-iterator predicates)
- Some must always be enforced
- Some must never terminate
- The `allowed_semantics_label` concept:

# Allowed Semantics

- Some predicates must never be checked at run time (e.g., destructive input-iterator predicates)
- Some must always be enforced
- Some must never terminate
- The `allowed_semantics_label` concept:

```
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});
  };
```

# Allowed Semantics — Behavior

- Implementation selects a configured semantic from the allowed set

# Allowed Semantics — Behavior

- Implementation selects a configured semantic from the allowed set
- If the configured semantic is not in the set, the implementation adjusts it (in a documented manner)

# Allowed Semantics — Behavior

- Implementation selects a configured semantic from the allowed set
- If the configured semantic is not in the set, the implementation adjusts it (in a documented manner)
- Combined labels: intersection of allowed-semantics sets

# Allowed Semantics — Behavior

- Implementation selects a configured semantic from the allowed set
- If the configured semantic is not in the set, the implementation adjusts it (in a documented manner)
- Combined labels: intersection of allowed-semantics sets
- Empty intersection is ill-formed

# Allowed Semantics — Behavior

- Implementation selects a configured semantic from the allowed set
- If the configured semantic is not in the set, the implementation adjusts it (in a documented manner)
- Combined labels: intersection of allowed-semantics sets
- Empty intersection is ill-formed
- Standard Library labels:

# Allowed Semantics — Behavior

- Implementation selects a configured semantic from the allowed set
- If the configured semantic is not in the set, the implementation adjusts it (in a documented manner)
- Combined labels: intersection of allowed-semantics sets
- Empty intersection is ill-formed
- Standard Library labels:
  - `terminating` — only *enforce* / *quick-enforce*

# Allowed Semantics — Behavior

- Implementation selects a configured semantic from the allowed set
- If the configured semantic is not in the set, the implementation adjusts it (in a documented manner)
- Combined labels: intersection of allowed-semantics sets
- Empty intersection is ill-formed
- Standard Library labels:
  - `terminating` — only *enforce* / *quick-enforce*
  - `symbolic` — only unchecked semantics

# Allowed Semantics — Behavior

- Implementation selects a configured semantic from the allowed set
- If the configured semantic is not in the set, the implementation adjusts it (in a documented manner)
- Combined labels: intersection of allowed-semantics sets
- Empty intersection is ill-formed
- Standard Library labels:
  - `terminating` — only *enforce* / *quick-enforce*
  - `symbolic` — only unchecked semantics
  - `always_enforce`, `always_observe`, etc. — single semantic

# Allowed Semantics — Behavior

- Implementation selects a configured semantic from the allowed set
- If the configured semantic is not in the set, the implementation adjusts it (in a documented manner)
- Combined labels: intersection of allowed-semantics sets
- Empty intersection is ill-formed
- Standard Library labels:
  - `terminating` — only *enforce* / *quick-enforce*
  - `symbolic` — only unchecked semantics
  - `always_enforce`, `always_observe`, etc. — single semantic
  - `never_enforce`, `never_observe`, etc. — remove one semantic

# Compute Semantic

- Labels can transform the configured semantic into a different effective semantic

# Compute Semantic

- Labels can transform the configured semantic into a different effective semantic
- Three stages:

# Compute Semantic

- Labels can transform the configured semantic into a different effective semantic
- Three stages:
  - **Configured**: chosen by the build system

# Compute Semantic

- Labels can transform the configured semantic into a different effective semantic
- Three stages:
  - **Configured**: chosen by the build system
  - **Computed**: result of `compute_semantic`

# Compute Semantic

- Labels can transform the configured semantic into a different effective semantic
- Three stages:
  - **Configured**: chosen by the build system
  - **Computed**: result of `compute_semantic`
  - **Effective**: verified against `allowed_semantics`

# Compute Semantic

- Labels can transform the configured semantic into a different effective semantic
- Three stages:
  - **Configured**: chosen by the build system
  - **Computed**: result of `compute_semantic`
  - **Effective**: verified against `allowed_semantics`
- The `semantic_computation_label` concept:

# Compute Semantic

- Labels can transform the configured semantic into a different effective semantic
- Three stages:
  - **Configured**: chosen by the build system
  - **Computed**: result of `compute_semantic`
  - **Effective**: verified against `allowed_semantics`
- The `semantic_computation_label` concept:

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

# Compute Semantic — Example

- `review | terminating`:

# Compute Semantic — Example

- `review | terminating`:
  - `terminating` restricts to enforcing semantics

# Compute Semantic — Example

- `review | terminating`:
  - `terminating` restricts to enforcing semantics
  - `review` maps those to *observe*

# Compute Semantic — Example

- `review | terminating`:
  - `terminating` restricts to enforcing semantics
  - `review` maps those to *observe*
  - Result: ill-formed — *observe* is not in the allowed set

# Compute Semantic — Example

- `review | terminating`:
  - `terminating` restricts to enforcing semantics
  - `review` maps those to *observe*
  - Result: ill-formed — *observe* is not in the allowed set
  - Caught at compile time

# Compute Semantic — Example

- `review | terminating`:
  - `terminating` restricts to enforcing semantics
  - `review` maps those to *observe*
  - Result: ill-formed — *observe* is not in the allowed set
  - Caught at compile time
- Library hardening via `compute_semantic`:

# Compute Semantic — Example

- `review | terminating`:
  - `terminating` restricts to enforcing semantics
  - `review` maps those to *observe*
  - Result: ill-formed — *observe* is not in the allowed set
  - Caught at compile time
- Library hardening via `compute_semantic`:

```
constexpr evaluation_semantic
compute_semantic(evaluation_semantic in) const {
#if _LIBCPP_ASSERTION_SEMANTIC == ...
  return evaluation_semantic::quick_enforce;
#else
  return in;
#endif
}
```

# Dimensions

- Some labels are mutually exclusive (e.g., `opt` and `audit`)

# Dimensions

- Some labels are mutually exclusive (e.g., `opt` and `audit`)
- Each such family forms a *dimension*:

# Dimensions

- Some labels are mutually exclusive (e.g., `opt` and `audit`)
- Each such family forms a *dimension*:

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

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

# Dimensions

- Some labels are mutually exclusive (e.g., `opt` and `audit`)
- Each such family forms a *dimension*:

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

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

- Combining labels with overlapping dimensions is ill-formed

# Dimensions

- Some labels are mutually exclusive (e.g., `opt` and `audit`)
- Each such family forms a *dimension*:

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

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

- Combining labels with overlapping dimensions is ill-formed
- Entirely implemented in the Standard Library (constraints on `operator|`)

# Groups

- The `identification_label` concept lets a label list, by name, the groups it belongs to:

# Groups

- The `identification_label` concept lets a label list, by name, the groups it belongs to:

```
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>;
  };
```

# Groups

- The `identification_label` concept lets a label list, by name, the groups it belongs to:

```
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>;
  };
```

- Combined labels: sorted, unique union of group names

# Groups

- The `identification_label` concept lets a label list, by name, the groups it belongs to:

```
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>;
  };
```

- Combined labels: sorted, unique union of group names
- Implementations document how groups interact with command-line configuration

# Groups

- The `identification_label` concept lets a label list, by name, the groups it belongs to:

```
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>;
  };
```

- Combined labels: sorted, unique union of group names
- Implementations document how groups interact with command-line configuration
- Static-analysis tools can use groups to selectively analyze assertions

# Groups — Configuration Example

- A configuration file can specify semantics per group:

# Groups — Configuration Example

- A configuration file can specify semantics per group:

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

# Groups — Configuration Example

- A configuration file can specify semantics per group:

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

- Applied to assertions:

# Groups — Configuration Example

- A configuration file can specify semantics per group:

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

- Applied to assertions:

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

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

# Violation Handling

- Sometimes the global violation handler isn't enough:

# Violation Handling

- Sometimes the global violation handler isn't enough:
  - Libraries needing complete control for security

# Violation Handling

- Sometimes the global violation handler isn't enough:
  - Libraries needing complete control for security
  - Delivering library-specific diagnostic information

# Violation Handling

- Sometimes the global violation handler isn't enough:
  - Libraries needing complete control for security
  - Delivering library-specific diagnostic information
  - Rethrowing specific exceptions (e.g., `bad_alloc`)

# Violation Handling

- Sometimes the global violation handler isn't enough:
  - Libraries needing complete control for security
  - Delivering library-specific diagnostic information
  - Rethrowing specific exceptions (e.g., `bad_alloc`)
- The `local_violation_label` concept:

# Violation Handling

- Sometimes the global violation handler isn't enough:
  - Libraries needing complete control for security
  - Delivering library-specific diagnostic information
  - Rethrowing specific exceptions (e.g., `bad_alloc`)
- The `local_violation_label` concept:

```
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);
  };
```

# Violation Handling — Behavior

- Return `void`: always chain to the next handler

# Violation Handling — Behavior

- Return `void`: always chain to the next handler
- Return `violation_handled::handled`: stop the chain

# Violation Handling — Behavior

- Return `void`: always chain to the next handler
- Return `violation_handled::handled`: stop the chain
- Return `violation_handled::not_handled`: continue to the next handler

# Violation Handling — Behavior

- Return `void`: always chain to the next handler
- Return `violation_handled::handled`: stop the chain
- Return `violation_handled::not_handled`: continue to the next handler
- Combined labels: handlers invoked right to left

# Violation Handling — Behavior

- Return `void`: always chain to the next handler
- Return `violation_handled::handled`: stop the chain
- Return `violation_handled::not_handled`: continue to the next handler
- Combined labels: handlers invoked right to left
- Example — rethrow `bad_alloc`:

# Violation Handling — Behavior

- Return `void`: always chain to the next handler
- Return `violation_handled::handled`: stop the chain
- Return `violation_handled::not_handled`: continue to the next handler
- Combined labels: handlers invoked right to left
- Example — rethrow `bad_alloc`:

```
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& v) {
    if (v.detection_mode()
                == detection_mode::evaluation_exception) {
      try { throw; }
      catch (const std::bad_alloc&) { throw; }
      catch (...) {}
```

# Comment / Message

- The `comment` field in `contract_violation` typically contains the predicate text

# Comment / Message

- The `comment` field in `contract_violation` typically contains the predicate text
- Sometimes this must be hidden or replaced

# Comment / Message

- The `comment` field in `contract_violation` typically contains the predicate text
- Sometimes this must be hidden or replaced
- The `compute_comment_label` concept:

# Comment / Message

- The `comment` field in `contract_violation` typically contains the predicate text
- Sometimes this must be hidden or replaced
- The `compute_comment_label` concept:

```
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*>;
  };
```

# Comment / Message

- The `comment` field in `contract_violation` typically contains the predicate text
- Sometimes this must be hidden or replaced
- The `compute_comment_label` concept:

```
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*>;
  };
```

- `compute_message_label` parallels this for user-provided messages (P3099R2)

# Comment / Message

- The `comment` field in `contract_violation` typically contains the predicate text
- Sometimes this must be hidden or replaced
- The `compute_comment_label` concept:

```
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*>;
  };
```

- `compute_message_label` parallels this for user-provided messages (P3099R2)
- Combined labels: functions chained left to right

# Comment / Message

- The `comment` field in `contract_violation` typically contains the predicate text
- Sometimes this must be hidden or replaced
- The `compute_comment_label` concept:

```
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*>;
  };
```

- `compute_message_label` parallels this for user-provided messages (P3099R2)
- Combined labels: functions chained left to right
- Can leverage `define_static_string` (P3491R3) for dynamic compile-time strings

# Query

- Labels may need to communicate data to the violation handler at run time

# Query

- Labels may need to communicate data to the violation handler at run time
- Challenge: handler is compiled separately, knows nothing about specific label types

# Query

- Labels may need to communicate data to the violation handler at run time
- Challenge: handler is compiled separately, knows nothing about specific label types
- Solution: type-erased query interface

# Query

- Labels may need to communicate data to the violation handler at run time
- Challenge: handler is compiled separately, knows nothing about specific label types
- Solution: type-erased query interface

```
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*>;
  };
```

# Query — Usage

- `contract_violation` gains a `query_control_object` member:

# Query — Usage

- `contract_violation` gains a `query_control_object` member:

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

# Query — Usage

- `contract_violation` gains a `query_control_object` member:

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

- Keys are addresses of constant objects — ensures link-time type safety

# Query — Usage

- `contract_violation` gains a `query_control_object` member:

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

- Keys are addresses of constant objects — ensures link-time type safety
- Index parameter supports combined labels with multiple results for the same key

# Query — Usage

- `contract_violation` gains a `query_control_object` member:

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

- Keys are addresses of constant objects — ensures link-time type safety
- Index parameter supports combined labels with multiple results for the same key
- Example: an `owner_label` carrying name and address:

# Query — Usage

- `contract_violation` gains a `query_control_object` member:

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

- Keys are addresses of constant objects — ensures link-time type safety
- Index parameter supports combined labels with multiple results for the same key
- Example: an `owner_label` carrying name and address:

```
if (auto* name = violation.query_control_object(
                     &owner_label::name_key)) {
  // send notification to owner
}
```

# Namespace and Name Lookup

- All labels live in `std::contracts::labels`

# Namespace and Name Lookup

- All labels live in `std::contracts::labels`
- `<contracts>` includes:

# Namespace and Name Lookup

- All labels live in `std::contracts::labels`
- `<contracts>` includes:

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

# Namespace and Name Lookup

- All labels live in `std::contracts::labels`
- `<contracts>` includes:

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

- Makes label names available unqualified in assertion-control expressions

# Namespace and Name Lookup

- All labels live in `std::contracts::labels`
- `<contracts>` includes:

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

- Makes label names available unqualified in assertion-control expressions
- Does *not* pollute the enclosing namespace

# Namespace and Name Lookup

- All labels live in `std::contracts::labels`
- `<contracts>` includes:

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

- Makes label names available unqualified in assertion-control expressions
- Does *not* pollute the enclosing namespace
- Users can also use `contract_control(expr)` to get the same lookup outside assertions

# Standard Library Labels

- **Core**: `empty_label`

# Standard Library Labels

- **Core**: `empty_label`
- **Semantic restriction** (`allowed_semantics_label`):

# Standard Library Labels

- **Core**: `empty_label`
- **Semantic restriction** (`allowed_semantics_label`):
  - `terminating`, `symbolic`

# Standard Library Labels

- **Core**: `empty_label`
- **Semantic restriction** (`allowed_semantics_label`):
  - `terminating`, `symbolic`
  - `always_enforce`, `always_observe`, ...

# Standard Library Labels

- **Core**: `empty_label`
- **Semantic restriction** (`allowed_semantics_label`):
  - `terminating`, `symbolic`
  - `always_enforce`, `always_observe`, ...
  - `never_enforce`, `never_observe`, ...

# Standard Library Labels

- **Core**: `empty_label`
- **Semantic restriction** (`allowed_semantics_label`):
  - `terminating`, `symbolic`
  - `always_enforce`, `always_observe`, ...
  - `never_enforce`, `never_observe`, ...
- **Semantic computation** (`semantic_computation_label`):

# Standard Library Labels

- **Core**: `empty_label`
- **Semantic restriction** (`allowed_semantics_label`):
  - `terminating`, `symbolic`
  - `always_enforce`, `always_observe`, ...
  - `never_enforce`, `never_observe`, ...
- **Semantic computation** (`semantic_computation_label`):
  - `review`, `hardened`

# Standard Library Labels

- **Core**: `empty_label`
- **Semantic restriction** (`allowed_semantics_label`):
  - `terminating`, `symbolic`
  - `always_enforce`, `always_observe`, ...
  - `never_enforce`, `never_observe`, ...
- **Semantic computation** (`semantic_computation_label`):
  - `review`, `hardened`
- **Runtime cost** (`identification_label` + shared dimension):

# Standard Library Labels

- **Core**: `empty_label`
- **Semantic restriction** (`allowed_semantics_label`):
  - `terminating`, `symbolic`
  - `always_enforce`, `always_observe`, ...
  - `never_enforce`, `never_observe`, ...
- **Semantic computation** (`semantic_computation_label`):
  - `review`, `hardened`
- **Runtime cost** (`identification_label` + shared dimension):
  - `opt`, `audit`

# Standard Library Labels

- **Core**: `empty_label`
- **Semantic restriction** (`allowed_semantics_label`):
  - `terminating`, `symbolic`
  - `always_enforce`, `always_observe`, ...
  - `never_enforce`, `never_observe`, ...
- **Semantic computation** (`semantic_computation_label`):
  - `review`, `hardened`
- **Runtime cost** (`identification_label` + shared dimension):
  - `opt`, `audit`
- **Grouping**: *"name"*`group` literal operator

# Utilities for Label Authors

- **Types**:

# Utilities for Label Authors

- **Types**:
  - `allowed_semantics_t<evaluation_semantic...>`

# Utilities for Label Authors

- **Types**:
  - `allowed_semantics_t<evaluation_semantic...>`
  - `evaluation_semantic_set`

# Utilities for Label Authors

- **Types**:
  - `allowed_semantics_t<evaluation_semantic...>`
  - `evaluation_semantic_set`
  - `dimension_list<typename...>`

# Utilities for Label Authors

- **Types**:
  - `allowed_semantics_t<evaluation_semantic...>`
  - `evaluation_semantic_set`
  - `dimension_list<typename...>`
  - `violation_handled` enum

# Utilities for Label Authors

- **Types**:
  - `allowed_semantics_t<evaluation_semantic...>`
  - `evaluation_semantic_set`
  - `dimension_list<typename...>`
  - `violation_handled` enum
  - `assertion_group_label<N>`

# Utilities for Label Authors

- **Types**:
  - `allowed_semantics_t<evaluation_semantic...>`
  - `evaluation_semantic_set`
  - `dimension_list<typename...>`
  - `violation_handled` enum
  - `assertion_group_label<N>`
  - `fixed_message_label_t`

# Utilities for Label Authors

- **Types**:
  - `allowed_semantics_t<evaluation_semantic...>`
  - `evaluation_semantic_set`
  - `dimension_list<typename...>`
  - `violation_handled` enum
  - `assertion_group_label<N>`
  - `fixed_message_label_t`
- **Semantic categorization functions**:

# Utilities for Label Authors

- **Types**:
  - `allowed_semantics_t<evaluation_semantic...>`
  - `evaluation_semantic_set`
  - `dimension_list<typename...>`
  - `violation_handled enum`
  - `assertion_group_label<N>`
  - `fixed_message_label_t`
- **Semantic categorization functions**:
  - `is_checked`, `is_unchecked`

# Utilities for Label Authors

- **Types**:
  - `allowed_semantics_t<evaluation_semantic...>`
  - `evaluation_semantic_set`
  - `dimension_list<typename...>`
  - `violation_handled` enum
  - `assertion_group_label<N>`
  - `fixed_message_label_t`
- **Semantic categorization functions**:
  - `is_checked`, `is_unchecked`
  - `is_terminating`, `is_continuing`

# Utilities for Label Authors

- **Types**:
  - `allowed_semantics_t<evaluation_semantic...>`
  - `evaluation_semantic_set`
  - `dimension_list<typename...>`
  - `violation_handled` enum
  - `assertion_group_label<N>`
  - `fixed_message_label_t`
- **Semantic categorization functions**:
  - `is_checked`, `is_unchecked`
  - `is_terminating`, `is_continuing`
- **Concept variable templates**:

# Utilities for Label Authors

- **Types**:
  - `allowed_semantics_t<evaluation_semantic...>`
  - `evaluation_semantic_set`
  - `dimension_list<typename...>`
  - `violation_handled` enum
  - `assertion_group_label<N>`
  - `fixed_message_label_t`
- **Semantic categorization functions**:
  - `is_checked`, `is_unchecked`
  - `is_terminating`, `is_continuing`
- **Concept variable templates**:
  - `is_assertion_control_object<obj>`, etc.

# Conclusion

Please discuss.