Slides for EWG presentation of P2900R6: Contracts for C++

Joshua Berne
Timur Doumler
Andrzej Krzemieński

Document #: P3190R0
Date: 2024-03-20
Audience: LEWG

Overview

- What are Contracts and what are they for?
- History and context
- Scope: what P2900 proposes and what it doesn't
- Design principles
- Language specification
  - Syntax
  - Semantic rules and restrictions
  - Evaluation and contract-violation handling
  - Noteworthy design consequences
- Library API specification

Overview

- What are Contracts and what are they for?
- History and context
- Scope: what P2900 proposes and what it doesn't
- Design principles
- Language specification
  - Syntax
  - Semantic rules and restrictions
  - Evaluation and contract-violation handling
  - Noteworthy design consequences
- Library API specification

Design by Contract

Design by contract (DbC) is an approach for designing software.

It prescribes that software designers should define formal, precise and verifiable interface specifications for software components, which extend the ordinary definition of software components with **preconditions**, **postconditions**, and **invariants**.

These specifications are referred to as **Contracts**, in accordance with a conceptual metaphor with the conditions and obligations of business contracts.

Terminology

- A **contract** is a set of conditions that expresses expectations on a correct program.
- A **function contract** is a contract that is part of the specification of a function.
  - A **precondition** is a part of a function contract where the responsibility for satisfying it is on the caller of the function. Generally, these are requirements placed on the arguments passed to a function and/or the global state of the program upon entry into the function.
  - A **postcondition** is a part of a function contract where the responsibility for satisfying the condition is on the callee, i.e. the implementer of the function itself. These are generally conditions that will hold true regarding the return value of the function or the state of objects modified by the function when it completes execution normally.

Terminology

- A **contract** is a set of conditions that expresses expectations on a correct program.
  - A **class invariant** is a condition that will hold true throughout the lifetime of an instance of that class (except during modification).
  - A **loop invariant** is a condition that will hold true at the beginning and end of every loop iteration.

Terminology

- A function with no preconditions has a **wide contract.**
- A function with preconditions has a **narrow contract.**
    - Calling a function with all preconditions satisfied: **call in-contract.**
    - Calling a function while failing to satisfy any precondition: **call out of-contract.**
- Failure to satisfy a contract is also called a **contract violation.**

Contract violations

- A contract violation is not an error.
- A contract violation is a **bug in the program.**
- **Who is responsible** for the contract violation?
  - Precondition: the caller of the function
  - Postcondition: the callee, i.e. the implementation of the function
  - Invariant: the implementation of the class
- **What happens** when there is a contract violation?
  - It depends...
  - ...but in general, **undefined behaviour**

3 Descriptions of function semantics contain the following elements (as appropriate):[143]

(3.1) — *Constraints*: the conditions for the function's participation in overload resolution ([over.match]).

[*Note 1*: Failure to meet such a condition results in the function's silent non-viability. — *end note*]

[*Example 1*: An implementation can express such a condition via a *constraint-expression* ([temp.constr.decl]). — *end example*]

(3.2) — *Mandates*: the conditions that, if not met, render the program ill-formed.

[*Example 2*: An implementation can express such a condition via the *constant-expression* in a *static_assert-declaration* ([dcl.pre]). If the diagnostic is to be emitted only after the function has been selected by overload resolution, an implementation can express such a condition via a *constraint-expression* ([temp.constr.decl]) and also define the function as deleted. — *end example*]

(3.3) — *Preconditions*: the conditions that the function assumes to hold whenever it is called; violation of any preconditions results in undefined behavior.

(3.4) — *Effects*: the actions performed by the function.

(3.5) — *Synchronization*: the synchronization operations ([intro.multithread]) applicable to the function.

(3.6) — *Postconditions*: the conditions (sometimes termed observable results) established by the function.

(3.7) — *Result*: for a *typename-specifier*, a description of the named type; for an *expression*, a description of the type of the expression; the expression is an lvalue if the type is an lvalue reference type, an xvalue if the type is an rvalue reference type, and a prvalue otherwise.

(3.8) — *Returns*: a description of the value(s) returned by the function.

(3.9) — *Throws*: any exceptions thrown by the function, and the conditions that would cause the exception.

(3.10) — *Complexity*: the time and/or space complexity of the function.

(3.11) — *Remarks*: additional semantic constraints on the function.

(3.12) — *Error conditions*: the error conditions for error codes reported by the function.

```
// narrow contract:
std::vector::operator[]
std::vector::front

// wide contract:
std::vector::at
std::vector::size
std::vector::empty

// narrow or wide contract (depending on type):
std::vector::swap
```

How do we specify a contract?

- In the documentation: **plain language contract**

How do we specify a contract?

- In the documentation: **plain language contract**
  - In source code comments

```
// The behaviour is undefined unless pos < size().
T& operator[] (size_t pos) const;
```

How do we specify a contract?

- In the documentation: **plain language contract**
  - In source code comments
  - In a separate specification document (e.g. the C++ Standard)

```
constexpr const_reference operator[](size_type pos) const;
```

1 *Preconditions*: `pos < size()`.

2 *Returns*: `data_[pos]`.

3 *Throws*: Nothing.

How do we specify a contract?

- In the documentation: **plain language contract**
  - In source code comments
  - In a separate specification document (e.g. the C++ Standard)
  - Implicit (e.g. via an agreed-upon coding convention)

How do we specify a contract?

- In the documentation: **plain language contract**
  - In source code comments
  - In a separate specification document (e.g. the C++ Standard)
  - Implicit (e.g. via an agreed-upon coding convention)
- In code: **contract assertion**

How do we specify a contract?

- In the documentation: **plain language contract**
  - In source code comments
  - In a separate specification document (e.g. the C++ Standard)
  - Implicit (e.g. via an agreed-upon coding convention)
- In code: **contract assertions**
  - A language feature that provides support for contract assertions is a **Contracts facility**
  - Can be a core language feature (D, Eiffel, Ada...) or a library feature
  - P2900R6 proposes a Contracts facility for C++ as a core language feature

C++ has a Contracts facility!

C++ has a Contracts facility!

```
#include <cassert>
void f(int i) {
  // The argument needs to be a positive number!
  assert(i > 0);
}
```

C++ has a Contracts facility!

```
#include <cassert>
void f(int i) {
  // The argument needs to be a positive number!
  assert(i > 0);
}
```

- Cannot go on function declarations, only in function bodies
- Behaviour not customisable (token-ignore or `std::abort`)
- Information about contract violation not programmatically accessible
- It's a macro (token-ignored if not evaluated, ODR violations, ...)

Why do we need a Contracts facility in C++ as a language feature

Why do we need a Contracts facility in C++ <u>as a language feature</u>

- Precondition and postcondition assertions on **declarations**
- Portably usable across different libraries and codebases
- Fully customisable behaviour without ODR violations
- Predicate expressions parsed even if not evaluated
- Information about the contract violation programmatically available
- Accessible for tooling

Contract assertions

```
T& operator[] (size_t pos) const
  pre (pos < size());
```

Contract assertions

```
T& operator[] (size_t pos) const
  pre (pos < size());
```

- A contract assertion typically expresses a particular **provision** of a contract rather than the entire contract
- A contract assertion specifies a C++ algorithm that allows to either:
  - Verify compliance with the provision, or
  - Identify violations of the provision.
- In P2900R6, this algorithm is a C++ expression contextually convertible to `bool` called a **contract predicate**.

Checking contracts with contact assertions

- Sometimes straightforward

```
T& operator[] (size_t pos) const
  pre (pos < size());
```

Checking contracts with contact assertions

- Sometimes straightforward
- Sometimes expensive, or even violates guarantees

```
void binary_search(Iter begin, Iter end)  // O(log N)
  pre (is_sorted(begin, end));            // O(N)
```

Checking contracts with contact assertions

- Sometimes straightforward
- Sometimes expensive, or even violates guarantees
- Sometimes impractical/impossible without additional instrumentation ("`ptr` points to an object that is within its lifetime")

Checking contracts with contact assertions

- Sometimes straightforward
- Sometimes expensive, or even violates guarantees
- Sometimes impractical/impossible without additional instrumentation ("`ptr` points to an object that is within its lifetime")
- Or outright impossible ("passed-in function `f` returns a value")

Checking contracts with contact assertions

- Sometimes straightforward
- Sometimes expensive, or even violates guarantees
- Sometimes impractical/impossible without additional instrumentation ("`ptr` points to an object that is within its lifetime")
- Or outright impossible ("passed-in function `f` returns a value")
- Or even entirely outside of the scope of the C++ program ("you paid your bill for this library this week")

Checking contracts with contact assertions

- Sometimes straightforward
- Sometimes expensive, or even violates guarantees
- Sometimes impractical/impossible without additional instrumentation ("`ptr` points to an object that is within its lifetime")
- Or outright impossible ("passed-in function `f` returns a value")
- Or even entirely outside of the scope of the C++ program ("you paid your bill for this library this week")
- Contract assertions in general specify only **a subset of the plain-language contract** of the function rather than the entire contract

Overview

- What are Contracts and what are they for?
- History and context
- Scope: what P2900 proposes and what it doesn't
- Design principles
- Language specification
  - Syntax
  - Semantic rules and restrictions
  - Evaluation and contract-violation handling
  - Noteworthy design consequences
- Library API specification

Contract annotations in Standard C++: A Drama in Four Acts

- 2004–2006: D-like Contracts – N1962
  (Thorsten Ottosen, Lawrence Crowl, et al)

Contract annotations in Standard C++: A Drama in Four Acts

- 2004–2006: D-like Contracts – N1962

  (Thorsten Ottosen, Lawrence Crowl, et al)

```
double sqrt(double x)
precondition
{
  x > 0.0;
}
postcondition(r)
{
  approx_equal(r * r, x);
}
```

Contract annotations in Standard C++: A Drama in Four Acts

- 2004–2006: D-like Contracts – N1962
  (Thorsten Ottosen, Lawrence Crowl, et al)
- 2013-2014: BDE-like Macro Contracts – N4378
  (John Lakos et al)

Contract annotations in Standard C++:
A Drama in Four Acts

- 2004–2006: D-like Contracts – N1962
  (Thorsten Ottosen, Lawrence Crowl, et al)
- 2013-2014: BDE-like Macro Contracts – N4378
  (John Lakos et al)

```
double sqrt(double x)
{
  contract_assert(x > 0.0);
}
```

Contract annotations in Standard C++: A Drama in Four Acts

- 2004–2006: D-like Contracts – N1962
  (Thorsten Ottosen, Lawrence Crowl, et al)
- 2013-2014: BDE-like Macro Contracts – N4378
  (John Lakos et al)
- 2014-2019: C++20 Contracts – P0542
  (Gabriel Dos Reis, J. Daniel Garcia, John Lakos, Alisdair Meredith, Nathan Myers, Bjarne Stroustrup)

Contract annotations in Standard C++: A Drama in Four Acts

- 2004–2006: D-like Contracts – N1962
  (Thorsten Ottosen, Lawrence Crowl, et al)
- 2013-2014: BDE-like Macro Contracts – N4378
  (John Lakos et al)
- 2014-2019: C++20 Contracts – P0542
  (Gabriel Dos Reis, J. Daniel Garcia, John Lakos, Alisdair Meredith, Nathan Myers, Bjarne Stroustrup)

```
double sqrt(double x)
  [[expects: x > 0.0]]
  [[ensures r: approx_equal(r * r, x)]]
{ [[assert: i != x ]]; }
```

Contract annotations in Standard C++: A Drama in Four Acts

- 2004–2006: D-like Contracts – N1962
  (Thorsten Ottosen, Lawrence Crowl, et al)
- 2013-2014: BDE-like Macro Contracts – N4378
  (John Lakos et al)
- 2014-2019: C++20 Contracts – P0542
  (Gabriel Dos Reis, J. Daniel Garcia, John Lakos, Alisdair Meredith, Nathan Myers, Bjarne Stroustrup)
- 2019-today: Contracts MVP – P2900
  (Joshua Berne, Timur Doumler, Andrzej Krzemieński, Gašper Ažman, Tom Honermann, Lisa Lippincott, Jens Maurer, Jason Merrill, Ville Voutilainen)

The Contracts MVP

- Minimal viable product
  - Does not yet support all use cases!
    - However, explicitly designed for extensibility
  - Provides immediate value for a significant fraction of C++ users

Document no: P1995R1

Date: 2020-03-02

Authors: Joshua Berne, Timur Doumler, Andrzej Krzemieński, Ryan McDougall, Herb Sutter

Reply-to: jberne4@bloomberg.net

Audience: SG21

Contracts — Use Cases

Introduction

SG21 has gathered a large number of use cases for contracts between teh WG21 Cologne and Belfast meetings. This paper presents those use cases, along with some initial results from polling done of SG21 members to identify some level of important to the community for each individual use case.

Each use case has been assigned an identifier that can be used to reference these use cases in other papers, which will hopefully be stable. We expect this content to evolve in a number of ways:

Contracts – Use Cases

- Documenting contracts in code
  (consumable by both human readers and tooling)
- Runtime checking of contract assertions
- Static analysis
- Formal verification
- Guiding optimization to improve performance

Contracts – Use Cases

✅ Documenting contracts in code
(consumable by both human readers and tooling)

- Runtime checking of contract assertions
- Static analysis
- Formal verification
- Guiding optimization to improve performance

Contracts – Use Cases

✅ Documenting contracts in code
(consumable by both human readers and tooling)

✅ Runtime checking of contract assertions

- Static analysis
- Formal verification
- Guiding optimization to improve performance

Runtime checking of contract assertions in P2900R6

- replacement for `<cassert>`
- replacement for custom assertion macros
- can be placed on function declarations
- customisable behaviour
- information about the contract violation is available programmatically
- no macros :)

Contracts – Use Cases

✅ Documenting contracts in code
(consumable by both human readers and tooling)

✅ Runtime checking of contract assertions

- Static analysis
- Formal verification
- Guiding optimization to improve performance

Contracts – Use Cases

✅ Documenting contracts in code
(consumable by both human readers and tooling)

✅ Runtime checking of contract assertions

🤷‍♂️ Static analysis

🤷‍♂️ Formal verification

🤷‍♂️ Guiding optimization to improve performance

Overview

- What are Contracts and what are they for?
- History and context
- Scope: what P2900 proposes and what it doesn't
- Design principles
- Language specification
  - Syntax
  - Semantic rules and restrictions
  - Evaluation and contract-violation handling
  - Noteworthy design consequences
- Library API specification

```
int f(int x)
  pre (x != 1);       // precondition assertion
```

```
int f(int x)
  pre (x != 1)        // precondition assertion
  post (r: r != 2);   // postcondition assertion; `r` names return value
```

```
int f(int x)
  pre (x != 1)         // precondition assertion
  post (r: r != 2)     // postcondition assertion; `r` names return value
{
  contract_assert (x != 3); // assertion statement
  return x;
}
```

```
int f(int x)
  pre (x != 1)       // precondition assertion
  post (r: r != 2)   // postcondition assertion; `r` names return value
{
  contract_assert (x != 3); // assertion statement
  return x;
}

void g() {
  f(0); // no contract violation
  f(1); // violates precondition assertion of f
  f(2); // violates postcondition assertion of f
  f(3); // violates assertion statement within f
  f(4); // no contract violation
}
```


contract assertion
function-contract assertion
precondition assertion
pre(expr)
postcondition assertion
post(expr)
assertion statement
contract_assert(expr)

Function-contract assertions

- A precondition is usually, **but not always,** expressed by a precondition assertion.
- Preconditions and postconditions are categorised by **who is responsible** for ensuring that they are true (caller vs. callee)
- Precondition assertions, postcondition assertions, and assertion statements are categorised by **the time when they are evaluated**.
- Example: using a postcondition assertion to check a precondition:

```
T& select(vector<T> & elems)
  // Precondition: for every e in elems, pred(e) is true
  post (r : pred(r));
```

```
int f(int x)
  pre (x != 1)         // precondition assertion
  post (r: r != 2)   // postcondition assertion; `r` names return value
{
  contract_assert (x != 3); // assertion statement
  return x;
}

void g() {
  f(0); // no contract violation
  f(1); // violates precondition assertion of f
  f(2); // violates postcondition assertion of f
  f(3); // violates assertion statement within f
  f(4); // no contract violation
}
```

A contract assertion can be evaluated with one of the following three contract semantics:

- ***ignore***: do not check the predicate
- ***enforce***: check the predicate, if the check fails call the contract-violation handler, then std::abort
- ***observe***: check the predicate, if the check fails call the contract-violation handler, then continue

The contract-violation handler

- Function named `::handle_contract_violation`
  - Attached to the global module
  - Takes a single argument `const std::contracts::contract_violation&`
  - Returns `void`
  - May be `noexcept(true)` or `noexcept(false)`
- Implementation provides a default definition: **default contract-violation handler**
  - semantics implementation-defined, recommendation: print info about contract violation
- Implementation-defined whether it is **replaceable** (like `operator new/delete`)
  - You can provide your own **user-defined contract-violation handler** by implementing a function with a matching name and signature, and linking it in

User-defined contract-violation handler

```
void ::handle_contract_violation
(const std::contracts::contract_violation& violation)
{
  LOG(std::format("Contract violated at: {}\n", violation.location()));
}
```

What is <u>not</u> included in the Contracts MVP

- Precondition and postcondition assertions on virtual functions
- Precondition and postcondition assertions on coroutines
- Ability to refer to "old" values (at the time of call) inside a postcondition predicate
- Optimise based on assumption that predicate evaluates to true; otherwise, the behaviour is undefined (*assume* semantic)
- Contract levels ("audit", etc), explicit contract semantics, or other labels or meta-annotations that control the meaning of a contract assertion
- Expressing postconditions expected to hold when a function exits via an exception
- Contract assertions that cannot be expressed by boolean predicates (procedural interfaces)
- Predicates that cannot be evaluated at runtime
- Class invariants

What is not included in the Contracts MVP

- Precondition and postcondition assertions on virtual functions
- Precondition and postcondition assertions on coroutines
- Ability to refer to "old" values (at the time of call) inside a postcondition predicate
- Optimise based on assumption that predicate evaluates to true; otherwise, the behaviour is undefined (*assume* semantic)
- Contract levels ("audit", etc), explicit contract semantics, or other labels or meta-annotations that control the meaning of a contract assertion
- Expressing postconditions expected to hold when a function exits via an exception
- Contract assertions that cannot be expressed by boolean predicates (procedural interfaces)
- Predicates that cannot be evaluated at runtime
- Class invariants

Overview

- What are Contracts and what are they for?
- History and context
- Scope: what P2900 proposes and what it doesn't
- Design principles
- Language specification
  - Syntax
  - Semantic rules and restrictions
  - Evaluation and contract-violation handling
  - Noteworthy design consequences
- Library API specification

Contract assertions help identify bugs in existing programs

- Adding a contract annotation to an existing program, or changing the contract semantics of an existing annotation, **should not change the compile-time semantics** of that program.
  - **Concepts do not see Contracts:** Contract annotations should not be seen by Concepts, affect overload resolution, type traits the result of the `noexcept` operator, which branch of an `if constexpr` is taken, should not be SFINAEable on, etc.
  - **Zero Overhead:** An *ignore*d contract annotation should not cause additional copies or destructions of objects
  - **Semantic Independence:** Which contract semantic will be used for any given evaluation of a contract assertion, and whether it is a checked semantic, must not be detectable at compile time.

Relationship between contract annotations and plain-language contracts

- The function contract specifiers on a function declaration should specify a sub-set of the plain-language contract of **that function** and not some other function.
- Function contract assertions serve both caller and callee and are therefore both **part of the interface and part of the implementation:**
  - Callers promise to satisfy a function’s preconditions, resulting in callees being able to rely upon those preconditions being true.
  - Callees (i.e., function implementers) promise to satisfy a function’s postconditions when invoked properly, resulting in a caller’s ability to rely upon those postconditions.
- Contract assertions are not flow control
  - **Contract assertions are not error handling**

Addressing open design questions

- The Contracts MVP is a starting point **designed for extensibility**.
- The Contracts MVP **does not intentionally introduce new undefined behaviour** to the C++ language.
- Whenever there is no consensus on what the correct design choice for a given problem is, and/or how the other design principles can be satisfied, we leave the relevant construct **ill-formed** rather than giving it unspecified or undefined behaviour.

Overview

- What are Contracts and what are they for?
- History and context
- Scope: what P2900 proposes and what it doesn't
- Design principles
- Language specification
    - Syntax
    - Semantic rules and restrictions
    - Evaluation and contract-violation handling
    - Noteworthy design consequences
- Library API specification

```
int f(int x)
  pre (x != 1);        // precondition specifier -
                       // introducing a precondition assertion
```

```
int f(int x)
  pre (x != 1);      // precondition specifier -
                     // introducing a precondition assertion

// pre/postcondition specifier ~ noexcept-specifier (syntactic construct)
// pre/postcondition assertion ~ exception specification (semantic property)
```

```
int f(int x)
  pre (x != 1)        // precondition specifier
  post (r: r != 2);  // postcondition specifier; `r` names return value

// return value name is optional
// `pre` and `post` are contextual keywords
// pre(...) and post(...) appear at the end of the declaration
```

```
int f(int x)
  pre (x != 1)         // precondition specifier
  post (r: r != 2)     // postcondition specifier; `r` names return value
{
  contract_assert (x != 3); // assertion statement
  return x;
}

// `contract_assert` is full keyword
// we did not use `assert` because of clash with assert macro
```

```
int f(int x)
  pre (x != 1)        // precondition specifier
  post (r: r != 2)    // postcondition specifier; `r` names return value
{
  contract_assert (x != 3); // assertion statement
  return x;
}

// `contract_assert` is full keyword
// we did not use `assert` because of clash with assert macro

// unlike assert macro, assertion statement is not an expression:
const int j = (contract_assert(i > 0), i); // syntax error
```

init-declarator :

declarator initializer$_{opt}$

declarator requires-clause$_{opt}$ function-contract-specifier-seq$_{opt}$

function-definition :

attribute-specifier-seq$_{opt}$ decl-specifier-seq$_{opt}$ declarator virt-specifier-seq$_{opt}$
function-contract-specifier-seq$_{opt}$ function-body

attribute-specifier-seq$_{opt}$ decl-specifier-seq$_{opt}$ declarator requires-clause
function-contract-specifier-seq$_{opt}$ function-body

member-declarator :

declarator virt-specifier$_{opt}$ function-contract-specifier-seq$_{opt}$ pure-specifier$_{opt}$

declarator requires-clause

declarator requires-clause$_{opt}$ function-contract-specifier-seq$_{opt}$

declarator brace-or-equals-initializer$_{opt}$

identifier$_{opt}$ attribute-specifier-seq$_{opt}$: brace-or-equals-initializer$_{opt}$

statement :

attribute-specifier-seq$_{opt}$ expression-statement

attribute-specifier-seq$_{opt}$ compound-statement

attribute-specifier-seq$_{opt}$ selection-statement

attribute-specifier-seq$_{opt}$ iteration-statement

attribute-specifier-seq$_{opt}$ jump-statement

attribute-specifier-seq$_{opt}$ assertion-statement

declaration-statement

attribute-specifier-seq$_{opt}$ try-block *function-contract-specifier-seq :*

  function-contract-specifier function-contract-specifier-seq

function-contract-specifier :

  precondition-specifier

  postcondition-specifier

precondition-specifier :

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

postcondition-specifier :

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

result-name-introducer :

  identifier attribute-specifier-seq$_{opt}$ `:`

assertion-statement :

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

```
bool binary_search(Range r, const T& value)
  pre [[vendor::message("Nonsorted range provided")]] (is_sorted(r));

void f() {
  int i = get_i();
  contract_assert [[analyzer::prove_this]] (i > 0);
  // ...
}
```

```
void g(int x) {
  if (x >= 0) {
    [[likely]] contract_assert(x <= 100);
    // ...
  }
  else {
    [[unlikely]] contract_assert(x >= -100);
    // ...
  }
}
```

```
int g()
  post (r [[maybe_unused]]: r > 0);
```

Where you can place a contract annotation

- `pre`, `post`:
  - on declarations of functions and function templates
    - obligatory on first declarations*, optional on redeclarations
    - if deduced (auto) return type, first declaration has to be a definition
  - on lambda expressions
- `contract_assert`:
  - Anywhere you can place a statement

---

*first declaration = declaration from which no other declaration is reachable

Where you *cannot* place a contract annotation

- `pre`, `post`:
  - not on `=deleted` functions
  - not on functions `=defaulted` on their first declaration
  - not on virtual functions (coming soon → P3097R0, P3165R0, D3169R0)
  - not on function pointers
    (`pre`, `post` are still evaluated when calling through a function pointer!)
  - not on coroutines (`contract_assert` is allowed inside a coroutine)

No co_yield or co_await inside a contract_assert

```
std::generator<int> f() {
  contract_assert(((co_yield 1), true));           // error
}

stdex::task<void> g() {
  contract_assert((co_await query_database()) > 0);  // error
}
```

No co_yield or co_await inside a contract_assert

```
std::generator<int> f() {
  contract_assert(((co_yield 1), true));           // error
}

stdex::task<void> g() {
  contract_assert((co_await query_database()) > 0);  // error
}

auto h() {
  contract_assert(([]() -> std::generator<int> {
    co_yield 1;        // OK: not suspending the function h()
  }(), true));
}
```

Overview

- What are Contracts and what are they for?
- History and context
- Scope: what P2900 proposes and what it doesn't
- Design principles
- Language specification
  - Syntax
  - Semantic rules and restrictions
  - Evaluation and contract-violation handling
  - Noteworthy design consequences
- Library API specification

Name lookup and access control

```
struct X {
  void f(int j)
    pre (j != i);  // name lookup & access as-if first statement in body
private:
  int i = 0;
};
```

Local variables are implicitly `const`

```
int global = 0;

void f(int x, int y, char *p, int& ref)
  pre((x = 0) == 0)            // error: assignment to const lvalue
  pre((*p = 5))                // OK
  pre((ref = 5))               // error: assignment to const lvalue
  pre((global = 2))            // OK
{
  contract_assert((x = 0));    // error: assignment to const lvalue
  int var = 42;
  contract_assert((var = 42)); // error: assignment to const lvalue

  static int svar = 1;
  contract_assert((svar = 1)); // OK
}
```

Local variables are implicitly `const`

```
void f() {
  int var = 42;
  contract_assert(++const_cast<int&>(var), true);  // OK (but evil)
}
```

Referring to result value in post

```
int f()
  post(r: r > 0);

// r is an lvalue of type `const T` referring to result object
```

Referring to result value in post

```
int f()
  post(r: r > 0);

// r is an lvalue of type `const T` referring to result object

int f2()
  post(r: ++r);  // error

int f3()
  post(r: ++const_cast<int&>(r));  // OK (but evil)
```

Referring to result value in post

```
int f()
  post(r: r > 0);

// r is an lvalue of type `const T` referring to result object
// `decltype(r)` is `T` (not `const T`!)
// `decltype((r))` is `const T&`
```

Referring to result value in post

```
struct S {
  S();
  S(const S&) = delete;  // non-copyable, non-movable
  int i = 0;
  bool foo() const;
};

const S f()
  post(r: (const_cast<S&>(r).i = 1))  // OK (but evil)
{
  return S{};
}

const S y = f();     // well-defined behavior
bool b = f().foo();  // well-defined behavior
```

Referring to result value in post

```
X f(X* ptr)
  post(r: &r == ptr)  // guaranteed to pass (for the call from `main` below)
{                     // if `X` is not trivially copyable
  return X{};
}

int main() {
  X x = f(&x);
}
```

Referring to result value in post

```
auto f1() post (r : r > 0);  // error, type of `r` is not readily available.

auto f2() post (r : r > 0)   // OK, type of `r` is deduced below.
{ return 5; }

template <typename T>
auto f3() post (r : r > 0);  // OK, postcondition instantiated with template

auto f4() post (true);       // OK, return value not named
```

Referring to non-reference parameters in `post`

```
int clamp(int v, int min, int max)
  post (r: val < min ? r == min : r == val)
  post (r: val > max ? r == max : r == val);
```

Referring to non-reference parameters in post

```
int clamp(const int v, const int min, const int max) // on all declarations
  post (r: val < min ? r == min : r == val)
  post (r: val > max ? r == max : r == val);
```

Referring to non-reference parameters in post

```
int clamp(int v, int min, int max)
  post (r: val < min ? r == min : r == val)
  post (r: val > max ? r == max : r == val)
{
  min = max = value = 0;
  return 0;
}
```

Referring to non-reference parameters in post

```
int clamp(const int v, const int min, const int max) // on all declarations
  post (r: val < min ? r == min : r == val)
  post (r: val > max ? r == max : r == val);
```

Not part of the immediate context

```
template <std::regular T>
void f(T v, T u)
  pre ( v < u );    // not part of `std::regular`

template <typename T>
constexpr bool has_f = std::regular<T> && requires(T v, T u) { f(v, u); };

static_assert( has_f<std::string>);          // OK, `has_f` returns `true`.
static_assert(!has_f<std::complex<float>>); // error, `has_f` causes hard
                                            // instantiation error.
```

Function template specialisations are independent from the primary template

```
bool a = true;
bool b = false;

template <typename T>
void f() pre(a) {}      // primary template with precondition assertion

template<>
void f<int>() pre(b) {}  // OK, precondition assertion different from that of
                         // primary template

template<>
void f<bool>() {}        // OK, no precondition assertion
```

No implicit lambda captures in predicates

```
int main() {
  int i = 1;
  auto f = [=] pre(i > 0) // error: cannot implicitly capture `i` here
  {};
}
```

No implicit lambda captures in predicates

```
int main() {
  int i = 1;
  auto f = [=] {
    contract_assert(i > 0); // error: cannot implicitly capture `i` here
  };
}
```

No implicit lambda captures in predicates

```
int main() {
  int i = 1;
  auto f = [=] {
    contract_assert(i > 0); // OK (`i` captured below)
    (void)i;                // `i` captured here
  };
}
```

No implicit lambda captures in predicates

```
int main() {
  int i = 1;
  auto f = [i] {
    contract_assert(i > 0); // OK (`i` captured explicitly above)
  };
}
```

No implicit lambda captures in predicates

```
static int i = 1;

int main() {
  auto f = [=] {
    contract_assert(i > 0);  // OK (`i` does not need to be captured)
  };
}
```

Overview

- What are Contracts and what are they for?
- History and context
- Scope: what P2900 proposes and what it doesn't
- Design principles
- Language specification
  - Syntax
  - Semantic rules and restrictions
  - Evaluation and contract-violation handling
  - Noteworthy design consequences
- Library API specification

Point of evaluation

- **Precondition assertions:**
  after the initialisation of function parameters,
  before the evaluation of the function body
- **Postcondition assertions:**
  after the result object value has been initialised and local automatic variables have been destroyed, but prior to the destruction of function parameters
- **Assertion statements:**
  when the statement is executed

Contract semantics

- When is a contract assertion checked or unchecked?
- When it is checked and the check fails, what happens after the contract-violation handler returns?

Possible contract semantics (P1429R3)

	Evaluate the predicate ("check the assertion")	After contract-violation handler returns:	Compiler is allowed to assume (otherwise UB):
ignore	no	–	–
assume	no	–	that the predicate would always evaluate to true
check_never_continue	yes	std::abort	that the predicate evaluated to true
check_maybe_continue	yes	continue execution	–
check_always_continue	yes	continue execution	that the contract-violation handler always returns

Possible contract semantics (P1429R3)

	Evaluate the predicate ("check the assertion")	After contract-violation handler returns:	Compiler is allowed to assume (otherwise UB):
ignore	no	–	–
assume	no	–	that the predicate would always evaluate to true
check_never_continue	yes	std::abort	that the predicate evaluated to true
check_maybe_continue	yes	continue execution	–
~~**check_always_continue**~~	~~yes~~	~~continue execution~~	~~that the contract violation handler always returns~~

Possible contract semantics (P1429R3)

	Evaluate the predicate ("check the assertion")	After contract-violation handler returns:	Compiler is allowed to assume (otherwise UB):
ignore	no	–	–
assume	no	–	that the predicate would always evaluate to true
enforce	yes	std::abort	that the predicate evaluated to true
observe	yes	continue execution	–

Possible contract semantics (P1429R3)

	Evaluate the predicate ("check the assertion")	After contract-violation handler returns:	Compiler is allowed to assume (otherwise UB):
ignore	no	–	–
~~*assume*~~	~~no~~		~~that the predicate would always evaluate to true~~
enforce	yes	std::abort	that the predicate evaluated to true
observe	yes	continue execution	–

Contract semantics proposed in P2900R6

	Evaluate the predicate ("check the assertion")	After contract-violation handler returns:	Compiler is allowed to assume (otherwise UB):
ignore	no	–	–
enforce	yes	std::abort	that the predicate evaluated to true
observe	yes	continue execution	–

Contract semantics

- P2900R6 proposes three standard contract semantics:
  ignore, ***enforce***, ***observe***
  - *ignore* is an **unchecked semantic**
  - *enforce* and *observe* are **checked semantics**
- The mechanism of choosing a contract semantic is
  implementation-defined
  - Contract semantic can be different for each contract annotation,
    or even for each evaluation of the same contract annotation
  - Contract semantic can be chosen at compile time, link time, or runtime

Recommended practice

- It is recommended that an implementation provide a mode where all contract assertions have the *ignore* semantic;
- It is recommended that an implementation provide a mode where all contract assertions have the *enforce* semantic;
- When nothing else has been specified by the user, it is recommended that a contract assertion have the *enforce* semantic.

Checking the contract predicate

- The predicate evaluates to true → no contract violation, execution continues
- The predicate evaluates to false → contract violation
- Evaluation of the predicate does not finish, but control remains in the purview of the contract-checking process → contract violation
  - Evaluation exits via an exception
  - Evaluation occurs during constant evaluation, and predicate is not a core constant expression
- Evaluation of the predicate does not finish, control never returns to the purview of the contract-checking process → "you get what you get"
  - longjmp, terminate, infinite loop, suspend current thread forever, etc.

Checking the contract predicate

- When a contract violation has been identified:
  - An object of type `std::contracts::contract_violation` will be produced through implementation-defined means,
  - the **contract-violation handler** will be called,
  - the `std::contracts::contract_violation` object will be passed to the contract-violation handler (by `const&`)
  - If the contract violation occurred because evaluation of the predicate exited via an exception, the contract-violation handler acts as a handler for that exception (i..e the exception can be acccessed from within the contract-violation handler via `std::current_exception()`).

The contract-violation handler

- Function named `::handle_contract_violation`
  - Attached to the global module
  - Takes a single argument `const std::contracts::contract_violation&`
  - Returns `void`
  - May be `noexcept(true)` or `noexcept(false)`
- No declaration of `::handle_contract_violation` provided in any standard library header
- Implementation provides a default definition: **default contract-violation handler**
  - semantics implementation-defined, recommendation: print info about contract violation
- Implementation-defined whether it is **replaceable** (at link time, like `operator new/delete`)
  - You can provide your own **user-defined contract-violation handler** by implementing a function with a matching name and signature, and linking it into your program

User-defined contract-violation handler

```
void ::handle_contract_violation
(const std::contracts::contract_violation& violation)
{
  LOG(std::format("Contract violated at: {}\n", violation.location()));
}
```

User-defined contract-violation handler

```
void ::handle_contract_violation
(const std::contracts::contract_violation& violation)
{
  LOG(std::format("Contract violated at: {}\n", violation.location()));
  std::contracts::invoke_default_contract_violation_handler(violation);
}
```

User-defined contract-violation handler

```
void ::handle_contract_violation
(const std::contracts::contract_violation& violation)
{
  std::breakpoint();
}
```

User-defined contract-violation handler

```
void ::handle_contract_violation
(const std::contracts::contract_violation& violation)
{
  throw my::contract_violation_exception(violation);
}
```

Throwing contract-violation handlers

- Use cases:
  - Portably handle contract violation without terminating the program and without continuing into buggy code
  - Write unit tests for contract assertions ("negative testing")

Throwing contract-violation handlers

- Use cases:
  - Portably handle contract violation without terminating the program and without continuing into buggy code
  - Write unit tests for contract assertions ("negative testing")
- Requires following the **Lakos Rule**:
  - A function with a narrow contract shall not be `noexcept`
  - Even if it never throws an exception when called in-contract!

The Lakos Rule is foundational for Contracts

```
int f(int i) noexcept
  pre(i > 0);  // `pre` and `post` cannot throw through noexcept!
               // instead, you get std::terminate
```

Consecutive and repeated evaluations

```
void f(int *p)
  pre( p != nullptr ) // precondition #1
  pre( *p > 0 );      // precondition #2

// typical sequence: 1-2 or 1-2-1-2
// also allowed: 1-2-1, 1-2-2, 1-2-2-1, etc.
// *not* allowed: 1, 1-1, 2-1, 2-2, etc.
```

Predicate side effects

- Predicates with side effects allowed (use cases: alloc, lock/unlock mutex...)
- Side effects can occur multiple times (see rules on previous slide)
- Side effects can be elided if the compiler can prove that the predicate would evaluate to true or false (and never throw, longjmp, terminate, spin/sleep indefinitely...)
  - thrown exception must be available in contract-violation handler via `std::current_exception`
  - longjmp, terminate, etc. are guaranteed to occur ("you get what you get")
- Side effect-free boolean expression behaves as-if evaluated once

Predicate side effects

```
int i = 0;
void f()
  pre ((++i, true));

void g() {
  f();  // `i` may be 0, 1, 17, etc.
}
```

Predicate side effects

```
int i = 0;
void f()
  pre ((++i, false));

void g() {
  f();  // `i` may be any value; the contract-violation handler
}       // will be invoked at most that number of times
```

Predicate side effects

```
int i = 0;
void f()
  pre ((++i, throw 666));

void g() {
  f();  // `i` may be 1, 2, 17, etc. but not 0
}
```

Contract assertions during constant evaluation

```
constexpr int f(int i)
  pre(i > 0)  // it's a bug to call this function with nonpositive arg!
{
  return i * i;
}

int main() {
  std::cout << f(0);        // contract violation at runtime
  std::array<int, f(0)> a;  // contract violation at compile time
}
```

Contract assertions during constant evaluation

- When checking a contract predicate during constant evaluation, only three things can happen:
  - Evaluates to true
  - Evaluates to false
  - Not a core constant expression

Contract assertions during constant evaluation

- When checking a contract predicate during constant evaluation, only three things can happen:
  - Evaluates to true → no contract violation, constant evaluation continues
  - Evaluates to false → contract violation
  - Not a core constant expression → contract violation
    (contract assertion is always a core constant expression,
    even if predicate is not → Concepts do not see Contracts principle)

Contract assertions during constant evaluation

- In a manifestly constant evaluated context, a contract assertion can be evaluated with one of the three semantics: *ignore, observe, enforce*
- choice of semantic is implementation-defined (for every evaluation)

Contract assertions during constant evaluation

- In a manifestly constant evaluated context, a contract assertion can be evaluated with one of the three semantics: *ignore, observe, enforce*
- choice of semantic is implementation-defined (for every evaluation)
- *ignore* does nothing (except parsing and odr-using)
- *observe* and *enforce* perform constant evaluation of the predicate
    - `true` → no effect
    - `false` or not a core constant expression → contract violation
        - *observe* → diagnostic (compiler warning)
        - *enforce* → program is ill-formed (hard compiler error)

Trial constant evaluation

Trial constant evaluation

```
int compute_at_runtime(int n);  // not `constexpr`

constexpr int compute(int n) {
  return n == 0 ? 42: compute_at_runtime(n);
}

void f() {
  const int i = compute(0); // constant initialization
  const int j = compute(1); // dynamic initialization
}
```

Trial constant evaluation

- The addition of `pre`, `post`, or `contract_assert` should:
  - <u>not</u> silently change static initialisation to dynamic initialisation
  - <u>not</u> trigger a compile-time contract violation if we would otherwise get well-formed dynamic initialisation

Trial constant evaluation - case 1

```
constexpr int f()
{
  return 42;
}

static int i = f();  // static initialisation
```

Trial constant evaluation - case 1

```
bool whatever();  // not constexpr

constexpr int f() pre(whatever())  // pre not checkable at compile time
{
  return 42;
}

static int i = f();
```

Trial constant evaluation - case 1

```
bool whatever();  // not constexpr

constexpr int f() pre(whatever())  // pre not checkable at compile time
{
  return 42;
}

static int i = f();  // must not be dynamic initialisation!
```

Trial constant evaluation - case 1

```
bool whatever();  // not constexpr

constexpr int f() pre(whatever())  // -> compile-time contract violation
{
  return 42;
}

static int i = f();  // must not be dynamic initialisation!
```

Trial constant evaluation - case 2

```
constexpr int f()
{
  if (i == 0)
    return runtime_thingy::get_value();  // not constexpr

  return i;
}

static int i = f(0);  // dynamic initialisation
```

Trial constant evaluation - case 2

```
bool whatever();  // not constexpr

constexpr int f() pre(whatever())  // not constexpr
{
  if (i == 0)
    return runtime_thingy::get_value();  // not constexpr

  return i;
}

static int i = f(0);  // dynamic initialisation
```

Trial constant evaluation - case 2

```
bool whatever();  // not constexpr

constexpr int f() pre(whatever())  // don't try evaluate at compile time!
{
  if (i == 0)
    return runtime_thingy::get_value();  // not constexpr

  return i;
}

static int i = f(0);  // must still be dynamic initialisation!
```

Trial constant evaluation - case 2

```
bool whatever();  // not constexpr

constexpr int f() pre(whatever())  // -> evaluate at runtime
{
  if (i == 0)
    return runtime_thingy::get_value();  // not constexpr

  return i;
}

static int i = f(0);  // must still be dynamic initialisation!
```

Contract assertions during constant evaluation (part II)

- When determining whether an expression E is a core constant expression ("trial evaluation"), ignore all contract annotations
- If E is a core constant expression, or if E is not a core constant expression but it is in a manifestly constant-evaluated context, re-evaluate E with every contract annotation having one of three semantics *(ignore, observe, enforce*) chosen in a implementation-defined manner
- Semantic is not *ignore* and predicate evaluates to `false` or is not a core constant expression → a compile-time contract violation occurs
  - *observe*: diagnostic
  - *enforce*: diagnostic, program is ill-formed

Overview

- What are Contracts and what are they for?
- History and context
- Scope: what P2900 proposes and what it doesn't
- Design principles
- Language specification
  - Syntax
  - Semantic rules and restrictions
  - Evaluation and contract-violation handling
  - Noteworthy design consequences
- Library API specification

Constructors and destructors

- `pre` and `post` on constructors and destructors follow same rules as for regular function declarations

Constructors and destructors

- `pre` and `post` on constructors and destructors follow same rules as for regular function declarations:
  - `pre` on a constructor are evaluated before the complete function body (which includes the function-try block and member initializer list)
  - post on a destructor are evaluated before returning to the caller (and therefore after the destruction of all members and base classes)

Constructors and destructors

- `pre` and `post` on constructors and destructors follow same rules as for regular function declarations:
  - `pre` on a constructor are evaluated before the complete function body (which includes the function-try block and member initializer list)
  - post on a destructor are evaluated before returning to the caller (and therefore after the destruction of all members and base classes)
    - Accessing members, base classes, invoking virtual functions, etc. in the predicate of a `pre` or `post` in the above situations is **undefined behaviour.**

Constructors and destructors

- `pre` and `post` on constructors and destructors follow same rules as for regular function declarations:
  - `pre` on a constructor are evaluated before the complete function body (which includes the function-try block and member initializer list)
  - post on a destructor are evaluated before returning to the caller (and therefore after the destruction of all members and base classes)
    - Accessing members, base classes, invoking virtual functions, etc. in the predicate of a `pre` or `post` in the above situations is **undefined behaviour.**
  - `post` on a constructor and `pre` on a destructor do not know the dynamic type of `this`.

Constructors and destructors

```
struct B { virtual ~B(); }  // polymorphic base

template <typename Base>
struct D : public Base {};  // generic derived class

struct C : public B {
  C()
    post( typeid(*this) == typeid(C) )           // Type is always `C` for now.
    post( dynamic_cast<C*   >(this) == this )      // `dynamic_cast` works.
    post( dynamic_cast<D<C>*>(this) == nullptr );  // never derived class yet.

  ~C()
    pre( typeid(*this) == typeid(C) )  //
    pre( dynamic_cast<C*   >(this) == this )
    pre( dynamic_cast<D<C>*>(this) == nullptr );
};
```

Friend declarations inside templates

- `pre` and `post` are required on any first declaration (declaration from which no other declaration is reachable)
- but optional on redeclaration
- Each TU has a first declaration
- All first declarations must have same sequence of `pre` and `post` (IFNDR)
- It is not always obvious which declaration is a first declaration:
  - a friend declaration of a function inside a template is only reachable from the point when that template is instantiated

Friend declarations inside templates

```
// x.h
template <typename T>
struct X {
  friend void f() pre (x);  // #1
};

// y.h
template <typename T>
struct Y {
  friend void f() pre (x);  // #2
};

// f.h
void f() pre (x);           // #3
```

Friend declarations inside templates

```
// x.h
template <typename T>
struct X {
  friend void f() pre (x);  // #1
};

// y.h
template <typename T>
struct Y {
  friend void f() pre (x);  // #2
};

// f.h
void f() pre (x);           // #3
```

```
// g.cpp
#include <x.h>
#include <y.h>
int g() {
  Y<int>   y1;  // #4
  Y<long>  y2;  // #5
  X<int>   x;   // #6
}
#include <f.h>
```

Friend declarations inside templates

- When using a friend declaration of a function with function contract assertions inside a template, we recommend to always do one of the following:
  - Befriend functions that have reachable declarations, such that the friend declaration will always be a redeclaration.
  - Duplicate the function contract specifiers on each friend declaration.
  - Make the function a hidden friend; i.e., the friend declaration is the only declaration of the function and is also a definition.

Recursive contract violations

Recursive contract violations

- "you get what you get"

Undefined behaviour

```
int f(int a) {
  return a + 100;
}

int g(int a)
  pre (f(a) < a);
```

Undefined behaviour

```
int f(int a) {
  return a + 100;  // compiler can assume this never overflows
}

int g(int a)
  pre (f(a) < a);  // compiler can replace this with `pre (false)`
```

Undefined behaviour

```
int f(int* p)
  pre ( p != nullptr ) {
  std::cout << *p;          // undefined behaviour!
}

int main() {
  f(nullptr);
}
```

Undefined behaviour

```
int f(int* p)
  pre ( p != nullptr ) {  // ignore: precondition not checked
  std::cout << *p;        // undefined behaviour!
}

int main() {
  f(nullptr);
}
```

Undefined behaviour

```
int f(int* p)
  pre ( p != nullptr ) {  // enforce: terminate here
  std::cout << *p;        // cannot get here!
}

int main() {
  f(nullptr);
}
```

Undefined behaviour

```
int f(int* p)
  pre ( p != nullptr ) {  // observe: compiler can elide check
  std::cout << *p;        // undefined behaviour!
}

int main() {
  f(nullptr);
}
```

Overview

- What are Contracts and what are they for?
- History and context
- Scope: what P2900 proposes and what it doesn't
- Design principles
- Language specification
  - Syntax
  - Semantic rules and restrictions
  - Evaluation and contract-violation handling
  - Noteworthy design consequences
- Library API specification

Standard Library API

- Only needed to implement a user-defined violation handler, not needed to add contract assertions to your code!
- Everything is in header `<contracts>`
- Everything is in namespace `std::contracts`
- One class `contract_violation` (passed into the contract-violation handler)
- Three enums to express the return values of some of its member functions
- One free function
- That's it!

Standard Library API

```
namespace std::contracts {
  class contract_violation {
    // No user-accessible constructor, not copyable/movable/assignable
  public:
    std::source_location location() const noexcept;
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    contract_semantic semantic() const noexcept;
    contract_kind kind() const noexcept;
  };
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

Standard Library API

```
namespace std::contracts {
  class contract_violation {
    // No user-accessible constructor, not copyable/movable/assignable
public:
    std::source_location location() const noexcept;
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    contract_semantic semantic() const noexcept;
    contract_kind kind() const noexcept;
  };
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

Standard Library API

```
namespace std::contracts {
  class contract_violation {
    // No user-accessible constructor, not copyable/movable/assignable
public:
    std::source_location location() const noexcept;
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    contract_semantic semantic() const noexcept;
    contract_kind kind() const noexcept;
  };
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

Standard Library API

```
namespace std::contracts {
  class contract_violation {
    // No user-accessible constructor, not copyable/movable/assignable
public:
    std::source_location location() const noexcept;
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    contract_semantic semantic() const noexcept;
    contract_kind kind() const noexcept;
  };
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

```
namespace std::contracts {
  enum class detection_mode : int {
    predicate_false = 1,
    evaluation_exception = 2,
    // implementation-defined additional values allowed, must be >= 1000
  };
}
```

Standard Library API

```
namespace std::contracts {
  class contract_violation {
    // No user-accessible constructor, not copyable/movable/assignable
public:
    std::source_location location() const noexcept;
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    contract_semantic semantic() const noexcept;
    contract_kind kind() const noexcept;
  };
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

```
namespace std::contracts {
  enum class contract_semantic : int {
    enforce = 1,
    observe = 2,
    // implementation-defined additional values allowed, must be >= 1000
  };
}
```

Standard Library API

```
namespace std::contracts {
  class contract_violation {
    No user-accessible constructor, not copyable/movable/assignable
public:
    std::source_location location() const noexcept;
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    contract_semantic semantic() const noexcept;
    contract_kind kind() const noexcept;
  };
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

```
namespace std::contracts {
  enum class contract_kind : int {
    pre = 1,
    post = 2,
    assert = 3,
    // implementation-defined additional values allowed, must be >= 1000
  };
}
```

Standard Library API

```
namespace std::contracts {
  class contract_violation {
    No user-accessible constructor, not copyable/movable/assignable
public:
    std::source_location location() const noexcept;
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    contract_semantic semantic() const noexcept;
    contract_kind kind() const noexcept;
  };
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

Impact on existing library facilities

Unless specified otherwise, an implementation is **allowed but not required** to check a subset of the preconditions and postconditions specified in the C++ standard library using contract assertions.

Slides for EWG presentation of P2900R6: Contracts for C++

Joshua Berne

Timur Doumler

Andrzej Krzemieński

Document #: P3190R0

Date: 2024-03-20

Audience: LEWG