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

Joshua Berne
Timur Doumler
Andrzej Krzemieński

Document #: P3189R0
Date: 2024-03-18
Audience: LEWG

Overview

- What are Contracts and what are they for?
- History and context
- Proposal summary
- 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
- Proposal summary
- Design principles
- Language specification
  - Syntax
  - Semantic rules and restrictions
  - Evaluation and contract-violation handling
  - Noteworthy design consequences
- Library API specification

What are contracts and what are they for?

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

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

P2900R6 language proposal summary

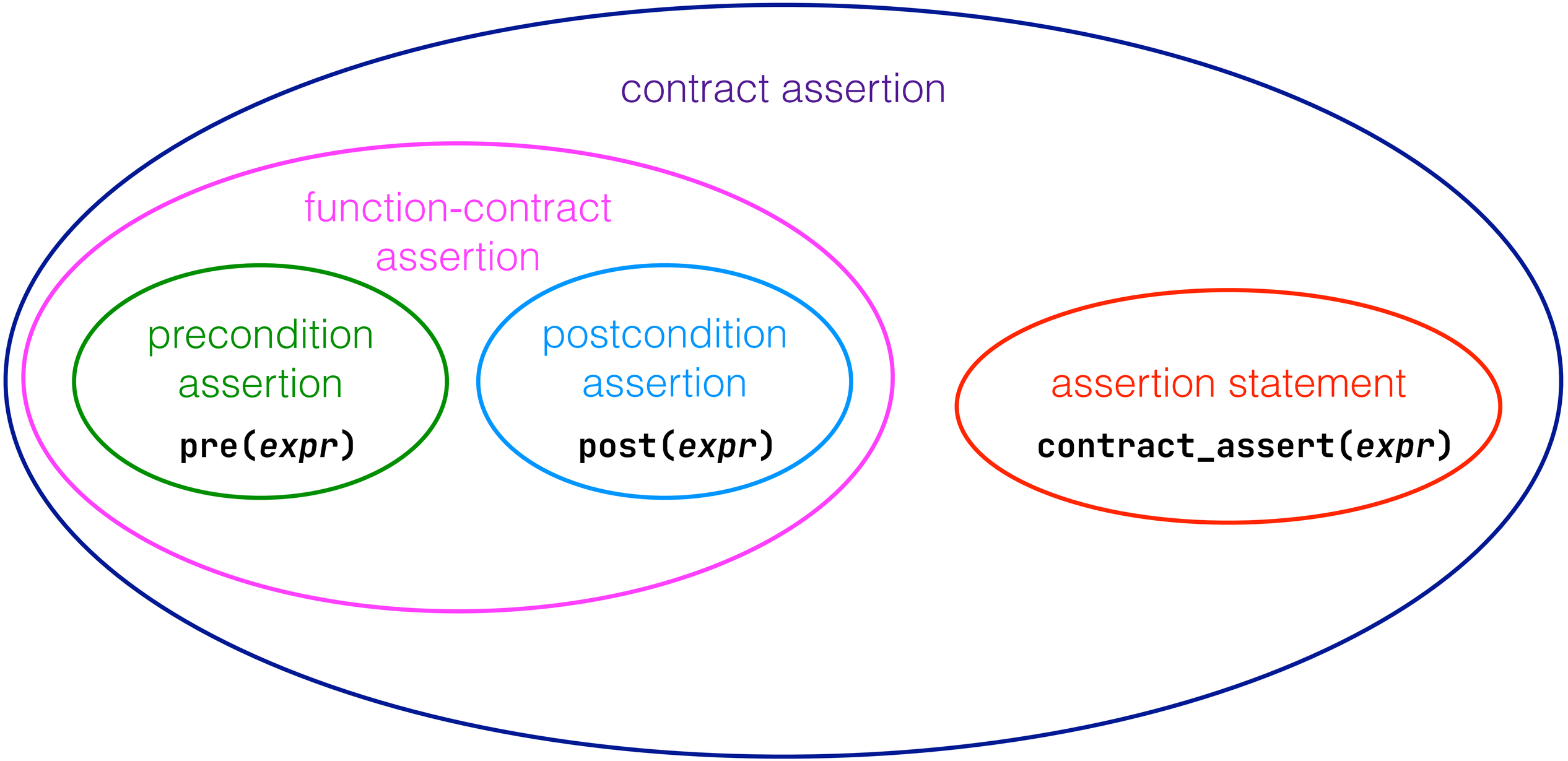
```
int f(int x)
  pre (x != 1);       // precondition assertion
```

```
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

// 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 assertion
  post (r: r != 2)    // postcondition assertion; `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 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)

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

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
}
```

Evaluating a contract assertion:

- check the contract predicate (evaluate the boolean expression), or
- do not check the contract predicate (just parse and odr-use the expression)

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 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 (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()));
}
```

User-defined contract-violation handler

```
void ::handle_contract_violation
(const std::contracts::contract_violation& violation)
{
  phone_home(violation);
  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 violations 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 violations 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!
    → policy discussion on Tuesday

Contract semantics

- When is a contract assertion checked or unchecked?
- What happens after the contract-violation handler returns?

Contract semantics proposed in P2900R6

	Evaluate the predicate ("check the assertion")	After contract-violation handler returns:
ignore	no	–
enforce	yes	call std::abort
observe	yes	continue execution

Contract semantics

- P2900R6 proposes three standard contract semantics:
  ignore, ***enforce***, ***observe***
  - *ignore* is an **unchecked semantic** (predicate is only parsed & odr-used)
  - *enforce* and *observe* are **checked semantics** (predicate is evaluated)
- 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:

1. an implementation provide a mode where all contract assertions have the *ignore* semantic;
2. an implementation provide a mode where all contract assertions have the *enforce* semantic;
3. when nothing else has been specified by the user, the default is 2.

Proposed Standard Library API in P2900R6

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.