Doc. No.: P2853R0
Project: Programming Language - C++ (WG21)
Audience: SG21 Contracts
Author: Andrew Tomazos <andrewtomazos@gmail.com>
Date: 2023-04-24

Proposal of std::contract_violation

1. Overview

On 2023-04-20 SG21 voted to add a standard link-time replaceable contract violation handling interface to its upcoming C++26 contracts proposal (MVP), to enable link-time user configuration of program behavior upon a contract violation.

We propose such an interface. We assume condition-centric syntax (P2737R0), but the interface requires only minor adaptation (replace enumerator `incondition` with enumerator `assertion`) for use with attribute-like or closure-based syntax, should they be selected instead.

The standard library header is `<contract_violation>`, and it declares a subclass of `std::exception` called `std::contract_violation` that has public member functions (`.kind(), .source_code(), .source_location()`) for accessing information about a contract violation. There are two associated enumerations, `std::contract_kind` (`empty`, `precondition`, `incondition` or `postcondition`) and `std::contract_resolution` (only one enumerator: `abort_program`).

`std::contract_violation` is a regular C++ type and is default constructible to an empty value, copy constructible, move constructible, copy assignable and move assignable. Furthermore, `std::contract_violation` uses an implementation technique we call an “inline pimpl idiom”, which means that it is ABI-extensible to add public member functions and data members in future (either non-standard extensions or standard extensions), and it does not use the heap at all. (Many felt it was important not to use the heap in a contract violation handling context for a variety of reasons). It has a large fixed size storage buffer (implementation-defined size, but the reference implementation uses 512 bytes), and then the private members are allocated at the start of that buffer. The remainder of the buffer is used to lazily cache the .what() message on its first call (if it is used).

<contract_violation> also declares the global contract violation handler function:

```
std::contract_resolution
::handle_contract_violation(const std::contract_violation&);
```

which has a default implementation provided by the implementation that logs the contract_violation `.what()` to standard error and returns `std::contract_resolution::abort_program`. It is recommended that this function is replaceable (ie weak-linked) at link-time.

2. Header <contract_violation>

Reference Implementation: https://github.com/tomazos/contract_violation

Highlighted in green are new entities relative to P2811
~~Struckthrough in red are entities removed relative to P2811~~
The diff does not highlight differences with P2811 that are in name-only.

```
namespace std {

enum class contract_kind { empty, precondition, incondition, postcondition };
enum class contract_resolution { abort_program };
~~enum class contract_semantic : *unspecified*;~~
~~enum class contract_violation_detection_mode : *unspecified*;~~

class contract_violation : public std::exception {
 public:
  contract_violation() noexcept;
  contract_violation(const contract_violation &) noexcept;
  contract_violation(contract_violation &&) noexcept;
  ~contract_violation();
  contract_violation &operator=(const contract_violation &) noexcept;
  contract_violation &operator=(contract_violation &&) noexcept;

  const char *what() const noexcept override;

  std::contract_kind kind() const noexcept;
~~  contract_violation_detection_mode detection_mode() const noexcept;~~
~~  contract_semantic semantic() const noexcept;~~

  // Returns either the source code/text of the contract expression
  // or the empty string if it is not available.
  const char *source_code() const noexcept;

  const std::source_location &source_location() const noexcept;

 private:
  /*exposition-only*/
```

```
  static constexpr size_t size = 512;
  alignas(std::max_align_t) mutable char storage[size];
};

}  // namespace std

~~void~~
std::contract_resolution handle_contract_violation(
    const std::contract_violation &);
```

3. Use Cases

3.1 Eval_and_abort

```
std::contract_resolution
handle_contract_violation(const std::contract_violation& v) {
    std::cerr << v.what() << std::endl;
    return std::contract_resolution::abort_program;
}
```

3.2 Eval_and_throw

```
std::contract_resolution
handle_contract_violation(const std::contract_violation& v) {
    throw v;
}
```

3.3 Detect and/or rethrow exception

```
std::contract_resolution
handle_contract_violation(const std::contract_violation& v) {
    if (std::exception_ptr e = std::current_exception())
        std::rethrow_exception(e);
    else
        /*...*/;
}
```

3.4 Custom error message / Eval_and_spin

```
std::contract_resolution
handle_contract_violation(const std::contract_violation& v) {
    std::contract_kind kind = v.kind();
    const char* code = v.source_code();
    std::source_location location = v.source_location();
    ErrorMessage msg = FormatErrorMessage(kind, code, location);
    DisplayErrorMessageAndWait(msg);
    return std::contract_resolution::abort_program;
}
```

3.5 Others

For others (longjmp, runtime error handler, etc) see P2811.

4 FAQ

4.1 Why no .semantic() accessor like P2811 ?

There is only one so-called “semantic” for the current MVP, so such an accessor is pointless. We can always add such an accessor in a later version if/when there is a second.

4.2 Why no .detection_mode() accessor like P2811 ?

Whether or not an exception has been thrown can be detected with std::current_exception, so the “exception” enumerator isn’t needed.

It is unknown whether implementations will provide “undefined behavior” detection or “unknown/other” methods of detection beyond false evaluation, so we don’t want to standardize an interface for something that may turn out to be never implemented and vestigial. If an implementation does choose to provide “undefined behavior” or “other” methods of detection they can provide non-standard accessors to discern this, and we can then later standardize these based on that existing practice.

4.3 Why subclass std::exception ?

It makes Eval_and_throw easier to implement (see 3.2), and there is an interoperability advantage to having a standard class type to represent a contract violation exception (as per existing practice in other languages) - rather than each project inventing their own.

Here's a list of assertion statements and the exceptions they throw from some of the most common programming languages:

Python: assert <condition>, <error message>: Throws an AssertionError if the condition is false.

Java: assert <condition>;: Throws an AssertionError if the condition is false

JavaScript: console.assert(<condition>, <error message>): Throws an AssertionError if the condition is false.

Ruby: assert <condition>: Throws an AssertionError if the condition is false.

PHP: assert(<condition>, <error message>): Throws an AssertionError if the condition is false.

4.4 Why the different names for things than P2811 ?

We think our names are better, more readable, more consistent, less redundant, clearer, etc, etc.

4.5 Why return std::contract_resolution instead of void?

The current design of the MVP has that what is called “Eval_and_warn” (aka “Eval_and_continue” aka “Observe” aka “Return To Contract”) resolution will NOT be supported. Such a resolution would mean that once the handler is complete, control returns to immediately after the violated contract, and the program continues.

It is foreseen that we might add this (and possible other) resolutions to later versions of contracts.

So that if/when we do add new resolutions in later contract versions, existing contract violation handlers don’t change their semantics on a compiler upgrade, we add this return type and return statement so that the abort semantic is baked into the first version of contract violation handlers.

If and when we do add additional resolutions, we will add additional enumerators to std::contract_resolution - and then people can migrate their contract violation handlers to them with an opt-in code change to their return statement (rather than the semantics silently changing on compiler upgrade).

5. References

[P2737] Proposal of Condition-Centric Contract Syntax - Andrew Tomazos
http://wg21.link/P2737

[P2811] Contract Violation Handlers - Joshua Berne
http://wg21.link/P2811