Differentiating potentially throwing and nonthrowing violation handlers

Abstract

We propose a clarification of the specification of the noexcept specifier of the `::handle_contract_violation(const std::contracts::contract_violation&)` function.

The current contract MVP proposal, [P2900R4, 3.5.7] states:

The Contract-Violation handler is a function named `::handle_contract_violation` that is attached to the global module. This function will be invoked when a contract violation is detected at runtime. This function
– shall take a single argument of type `const std::contracts::contract_violation&`,
– shall return `void`,
– may or may not be noexcept [emphasis ours].

This document clarifies the meaning of the last point.

Proposal

This proposal clarifies the meaning of the value of the boolean expression

```cpp
    noexcept(
        ::handle_contract_violation(
            std::declval<const std::contracts::contract_violation&>()
        )
    )
```

If the value is `true`, installing a throwing violation handler is ill-formed, and the above is the recommended way for code to detect this implementation-defined property of the abstract machine.
A compiler might choose to control the noexcept-ness of the violation handler with a compiler flag, for example `-fthrowing-violation-handler`.

We feel the final point in P2900R4 is insufficiently clear; we propose to change it from

- `may or may not be noexcept`

To

- It is implementation defined whether ::handle_contract_violation(const std::contracts::contract_violation&) is marked noexcept. [Note: this is the primary means for an implementation to expose the possibility a throwing violation handler to user code -- end note]

**Motivation**

A throwing contract handler may be useful and even necessary in known situations already discussed in P2900.

Conversely, a piece of code might not be designed to work with exceptions at all. Such code might want to `static_assert()` that the contract violation handler cannot throw; other code might be able to optimize based on the knowledge of that fact (if constexpr-gated RAII cleanup, for instance).

This paper details why we believe that giving the programmer the ability to reason about exceptions being thrown from the handler at constexpr-time is crucial, and why.

We also want to encourage implementation to default to marking the handler function `noexcept`, and specify how they will handle linking units with differing choices for this option (this is out of the scope of the standard). One would hope that making the `noexcept` and non-`noexcept` symbols conflict at link-time would help with ODR-violations arising from incongruent compilation configuration.

**Examples**

**Code relying on non-throwing sections**

Consider a piece of code of the form

```cpp
auto resource = acquire_resource(); // non-RAII resource handle
f(resource); // known not to throw
release(resource);
```
This code is correct, but not exception-safe.

If we add a precondition on \( f \), this code becomes incorrect in the presence of a potentially-throwing handler, if we ever want to continue with program execution after catching the exception thrown by the handler at a higher level (this is the motivation for throwing handlers, after all).

This code can be made correct by the inclusion of

```cpp
static_assert(
    noexcept(
        ::handle_contract_violation(
            declval<const contract_violation&>()
        )
    )
);
```

**Adaptive library code**

Code might want to selectively adapt to cleanup-upon-exception.

```cpp
auto guard = [&] {
    auto const handler_may_throw =
        not noexcept(
            ::handle_contract_violation(
                declval<const contract_violation&>()
            )
        );
    if constexpr (handler_may_throw)
    {
        return on_scope_error([&] {
            release(resource);
        });
    }
    else
    {
        return 0;
    }
}();
```

The above code only installs a cleanup handler if a violation handler can throw, otherwise it leaves a clean instruction stream, because that is the only possible source of an exceptional exit from the scope.
Code that expects a closed set of exceptions

Code that expects a closed set of exception types becomes incorrect in the presence of a throwing violation handler (it opens the set of possible exceptions).

```cpp
try {
    auto resource = acquire_resource();
    try {
        send_to_queue(resource);
    }
    catch(queue_full const&)
    { 
        release(resource);
    } // all possible exceptions are handled... or are they?
} 
catch (...) {
    // handle acquire_resource() errors
    // swallows send_to_queue contract violation exception by accident
}
```

Interfaces that want to be violation-tolerant in `noexcept` specifications

Code that advertises that signals failure through non-exceptional means through `noexcept` might want to be extra truthful for the benefit of code composition.

```cpp
std::expected<...> f(args) noexcept;
```

 Might want to advertise its interface as

```cpp
std::expected<...> f(args) noexcept(handler_may_throw);
```

instead, to prevent `f` being used as a callback in APIs that enforce `noexcept` function pointers because they cannot deal with throwing callbacks (example: threadpool submission queues).

References