Contract annotations are potentially-throwing

Timur Doumler (papers@timur.audio)
Ville Voutilainen (ville.voutilainen@gmail.com)
Tom Honermann (tom@honermann.net)

Document #: P2969R0
Date: 2023-12-04
Project: Programming Language C++
Audience: SG21

Abstract

Contract-violation handlers in the Contracts MVP can throw; contract annotations are therefore potentially-throwing constructs. This has interesting consequences for the interaction of contract annotations with other C++ features, in particular the `noexcept` operator and deduced exception specifications. It also reveals some unanswered design questions in those areas in the Contracts MVP. In this paper, we mention different alternatives for answering these design questions and discuss the associated tradeoffs.

1 Status quo

1.1 Throwing contract-violation handlers

Bjarne Stroustrup's paper "Unconditional termination is a serious problem" [P2698R0] led to the adoption of throwing violation handlers to the Contracts MVP [P2900R1]. Without the ability to throw an exception from a contract-violation handler, there are two options after a contract check has failed: either to terminate the program, or to continue execution, which is likely to run into undefined behaviour and either crash the program or invoke other unpredictable and undesirable behaviour. Throwing violation handlers add a third option: to unwind the stack and to restore the program to some state from which it can continue.

When a contract check fails, we know that the program is incorrect, i.e. there is a bug in the code rather than just a runtime error. In many cases, termination is appropriate in this case, for example in financial applications where an incorrect result can lead to the loss of huge sums of money, or in safety-critical applications where an incorrect result can threaten lives or property. In other cases however, producing incorrect results is much preferable to terminating the program, for example in text or image editors, computer games, music production software, and countless other consumer-facing applications. Often, such applications will want to diagnose a failing contract check in production and silently send an appropriate bug report, which greatly helps improving code quality, while at the same time continuing execution. In such applications, occasional "glitches" can be relatively common and are much less likely to upset the customer than terminating the entire app (and in the case of a plug-in, such as those used in office software, graphics editors, IDEs and digital audio workstations, potentially taking the host application down as well).

1.2 Contract checks can throw

SG21 agreed on the following set of semantics for throwing contract-violation handlers, proposed in [P2811R7] and now part of the Contracts MVP [P2900R1]:

— Throwing an exception from the evaluation of a contract-checking predicate shall be treated as a contract violation, and invoke the violation handler, regardless whether the function is declared `noexcept`;

— Throwing an exception from a contract violation handler shall invoke the usual exception semantics: stack unwinding occurs, and if a `noexcept` barrier is encountered during unwinding, `std::terminate` is called;

— Whether a contract-checking annotation is checked or unchecked shall never change the result of the `noexcept` operator or the type of a function pointer to which the contract annotation is attached;

— Preconditions, postconditions, and assertions shall all behave the same way with respect to throwing an exception from the predicate or the violation handler.

From this follows that there are well-formed C++ programs with well-defined behaviour where a contract check (whether it is a precondition, a postcondition, or an assertion) throws an exception, such as:

```
#include <contracts>
using namespace std::contracts;

handle_contract_violation(const contract_violation&) {
  throw 666;
}

int main() {
  contract_assert(false);   // this statement throws an exception
}
```

Unless we know at compile time that the contract-violation handler is `noexcept` (and in general this is *not* known at compile time), contract annotations are therefore potentially-throwing constructs, regardless of whether the violation handler actually throws an exception.

2 Open questions

2.1 Interaction with existing C++ features

There are currently several situations in the C++ language where the compile-time semantics of a program depend on whether a particular expression is potentially-throwing (see [except.spec] p3): when determining the result of the `noexcept` operator, and when deducing an exception specification for a special member function defaulted on its first declaration.

At the autumn 2023 meeting in Kona, SG21 adopted [P2932R2] proposal 1.C, making it ill-formed to add contract annotations to functions defaulted on its first declaration. Nevertheless, there are two situations left where the presence or absence of a contract annotation may change a deduced exception specification. The Contracts MVP does not currently define how contract annotations are supposed to behave in these situations, so we need to define it. There are several options for how to do that which we will discuss in this paper.

The first such case is the result of the `noexcept` operator when applied directly to a contract annotation, or an expression that contains a contract annotation as a subexpression. Note that this

only affects assertions, not preconditions or postconditions, as only assertions can be used directly as expressions:

```
noexcept(contract_assert(false));    // Is this true or false?
```

The second case concerns special member functions (such as constructors) that are defaulted on their first declaration and have a deduced exception specification. Unlike elsewhere in the language, in this case the exception specification is deduced as `noexcept(true)` by default, unless the constructor involves some potentially-throwing construct, in which case it is deduced as `noexcept(false)`. Now, we cannot put contract annotations on such functions directly, but they may appear in other places where they can change the outcome of the deduction:

```
struct B {
  int i = (contract_assert(false), 42);
};
struct S : B {
  // Are the following constructors noexcept or not?
  S() = default;
  S(S&&) = default;
};
```

Note that special member functions auto-compose recursively and a deduced exception specification impact can be viral. Changing the deduced exception specification of a move constructor of one type might change the deduced exception specification for the move constructors of all classes that contain a data member of that type.

Note further that regardless of what we decide should happen for deduced exception specification, the addition of a contract annotation can never change an explicitly specified exception specification. If a function is explicitly declared `noexcept`, throwing from a violation handler will run into the `noexcept` barrier and call `std::terminate`.

2.2 Interaction with potential future C++ features

While the result of the `noexcept` operator and deduced exception specifications on defaulted special member functions are the only two features in the language today where the potentially-throwing nature of contract annotations manifests itself, it is possible that more such features will be proposed for a future C++ Standard. In particular, we could get a feature like `noexcept(auto)` (see [N4473] and [P0133R0]) that can deduce the exception specification for any user-defined function without an explicit exception specification, depending on whether the function contains any potentially-throwing constructs. We need to consider how such a feature would interact with contract annotations. Since contract annotations are potentially-throwing, does this mean that `noexcept(auto)` would deduce to `noexcept(false)` if the function in question contains any contract annotations, even though it might deduce to `noexcept(true)` otherwise?

3 Possible solutions

3.1 Treat contract annotations as potentially-throwing?

The most straightforward approach to this problem is to say that, since contract checks can throw an exception, we should treat them as potentially-throwing in the language. This would mean that the `noexcept` operator, when applied directly to an assertion expression, or an expression that contains an assertion expression as a subexpression, would return `false`. It would also mean that if a special member function defaulted on its first declaration encounters a contract annotation during the deduction of its exception specification, it would be deduced as `noexcept(false)`, whereas without the contract annotation it might be deduced as `noexcept(true)`. Finally, it would mean that if we ever get a feature like `noexcept(auto)` that lets us deduce the exception specification of

a function in more cases, adding a contract annotation to such a function would also lead to the exception specification of the function to be deduced as `noexcept(false)`.

The above semantics seem straightforward; contract annotations behave the same way as any other potentially-throwing constructs. However, they run afoul of the so-called zero overhead principle, formulated in [P2932R1]: the addition of a contract annotation causes algorithms that branch on the exception specification a function (e.g. a move constructor), such as `std::vector::push_back`, to take the other branch. This is undesirable for two reasons.

First, it can lead to "heisenbugs": a program contains a bug, we add a contract annotation to find the bug, but the contract annotation causes the program to take another branch where the bug does not exist. The contract check therefore fail to prove the correctness of the original program that they have been added to; instead, their addition altered the program and now prove the correctness of another program. This arguably defeats the purpose of contract checks.

Second, adding a contract annotation can in itself cause a significant performance degradation, even if the contract check has the *ignore* semantic and is not evaluated, because the `noexcept(false)` path of such algorithms is typically significantly slower. Consider the following case where the presence of a postcondition check renders the move constructor potentially-thowing, causing `push_back` to call the more expensive copy constructor instead:

```
class S_impl { /*...*/ };
class S {
  std::unique_ptr<S_impl> d_pimpl;

public:
  bool empty() const noexcept { return d_pimpl == nullptr; }
  S(const S& orig)
    : d_pimpl( new S_impl(*orig.d_pimpl) )  // make expensive copy
  {}

 S(S&& orig)
   post(orig.empty()) // guarantee our moved-from state is empty
                      // and we need to be reinitialised
   = default;
```

It has been suggested that if contract checks can introduce such performance degradations, it would be a major disincentive for the adoption of Contracts in C++.

3.2 Treat contract annotations as non-throwing?

One alternative is to treat contract annotations as non-throwing constructs in the language, even though they are in fact potentially-throwing because the user can install a throwing violation handler. This would make sure that adding a contract annotation can never lead to a different branch being taken at compile time.

This would mean that if the `noexcept` operator answers the question "can this construct throw?", it gives us a factually incorrect answer to this question in the presence of contract annotations: `noexcept(contract_assert(false))` would report that the operand cannot throw whereas in reality it can, and deduced exception specifications of special member functions defaulted on their first declaration would report that the function cannot throw whereas in reality it can. This seems rather confusing.

One way to mitigate the confusion would be to change the meaning of the `noexcept` operator. Instead of teaching that the `noexcept` operator answers the question "can this construct throw?", we would have to teach that it answers the question "can this construct throw *if no contracts are violated*?". Exceptions escaping a throwing violation handler would be treated differently from exceptions thrown elsewhere in the language: we would consider them *erroneously thrown*

exceptions, i.e. exceptions that are only thrown if the program is incorrect, and we would define the `noexcept` operator and deduced exception specifications to ignore the possibility of erroneously thrown exceptions.

One downside of such an approach is that it would make it somewhat more difficult to work with throwing violation handlers: they are meant to enable violation handling without termination, but if contract annotations are treated as non-throwing, and do not deduce to `noexcept(false)`, throwing from such a violation handler would run into an unintended `noexcept` barrier and call `std::terminate` — the exact thing that throwing violation handlers are designed to avoid. In order to use throwing violation handlers, one would have to not only *not* declare any functions between the point where the exception is thrown and the point where it is caught `noexcept` (which you have to do anyway), but also declare some of those functions explicitly `noexcept(false)` if they previously had a deduced exception specification.

Furthermore, when handling an exception thrown from a violation handler, you actually *have* to be in the slow path of an algorithm like `push_back`, because the fast path assumes that there can be no exceptions in flight. The change in compile-time semantics is therefore desired in this case. If the author of the code in question is not interested in supporting throwing violation handlers, forcing the fast path and ensuring the zero overhead principle is always possible by explicitly adding a non-throwing exception specifier (`noexcept`) when the contract annotation is added.

For certain kinds of asynchronous code in particular, this is not just a "fast path" vs. "slow path" optimisation. In code like the generic algorithm code seen at https://github.com/NVIDIA/stdexec, a generic algorithm will need to catch exceptions that may be thrown, and transform or translate those onto an error channel. The code using such exception-neutral generic algorithms will need to do that translation, because exceptions cannot be allowed to just escape into the calling context, which is not a thread of execution that the user can write `catch`-clauses in — but the code cannot terminate either, because that would fail to allow the ultimate user to decide how to handle errors. This is a fundamental property of such asynchronous frameworks: silently swallowing exceptions loses information, and disallows custom error handling. Terminating does the same thing. The framework is written to be generic, and applicable to multiple different domains. That is why it thus translates exceptions onto an error channel of a completion mechanism, so that e.g. termination is not just imposed on users, they get to choose. The costs of that can be avoided when exceptions can be guaranteed not to be thrown, but that requires that `noexcept` operators give the correct answer.

And this is where future extensions with new and better rationale for something like `noexcept(auto)` come in. It is much simpler to write such code if exception specifications do not have to be repeated transformations of what is in the function body, for functions that need to be exception-neutral. And being exception-neutral in such code means that if a contract check throws, and the user writing the whole program wants to deal with contract violations as exceptions, the framework works together with that, being fully exception-neutral. The violation handler throws, the framework catches such exceptions, and translates them onto the error completion channel — even in code that runs in different contexts, like different threads or different schedulers, like event loops. Making the `noexcept` operator pretend that contract checks do not throw where in reality they can make this approach impossible.

3.3 Allow both options?

We could contemplate allowing both options: giving the user the choice between throwing and non-throwing contract checks. This could happen either in code:

```
int f(int i)
  pre (i > 0);  // potentially-throwing contract check
int g(int i)
  pre noexcept (i > 0);  // non-throwing contract check
```

or as a build mode (`-fnoexcept-contracts`) which switches all contract checks to non-throwing or potentially-throwing, either as a standardised build mode or as a vendor extension. We could also have both the meta-annotation and the build mode.

However, this does not seem like a viable direction. If the user needs to meta-annotate a contract annotation with a `noexcept` label to make it non-throwing and thus to satisfy the zero overhead principle, they might as well simply annotate the whole function with `noexcept`, which works today and does not require the introduction of a new feature. The idea of the zero overhead principle is that the addition of a contract annotation should *never* cause a different code path to be taken, not just in cases where the user remembered to add an extra annotation. Introducing a build mode would give users the option to globally opt into the zero overhead behaviour if they wish, but it would also introduce a language dialect: the same C++ code would have different meaning depending on the build mode. Such language dialects are widely considered undesirable and bad for the C++ ecosystem.

3.4 Allow erroneously thrown exceptions to escape deduced non-throwing exception specifications?

Another possible solution is to say that contract annotations are treated as non-throwing constructs in the language, but exceptions that have been thrown from a contract-violation handler (i.e. *erroneously* thrown exceptions) are allowed to throw through a deduced non-throwing exception specification without causing `std::terminate` to be called.

This approach would partially mitigate the negative impact on throwing violation handlers that treating contract annotations as non-throwing would otherwise have, while simultaneously not violating the zero overhead principle. However, it would not completely solve the problem for throwing violation handlers: there would still be cases where control flow ends up in the wrong branch of a `noexcept`-sensitive algorithm (the one that assumes that no exceptions are being thrown), preventing the program from recovering to some reasonable state after catching the exception. It would also greatly complicate our mental model of exception specifications, add a highly unintuitive inconsistency to the language, and add specification and implementation burden. We therefore do not consider this a viable direction either.

3.5 Make the question ill-formed?

If the straightforward and intuitive behaviour of treating contract annotations as potentially-throwing is not acceptable because we must adhere to the zero overhead principle in all possible cases, one possible compromise is to make any code ill-formed where a deduced exception specification or the result of the `noexcept` operator may depend on the presence of a contract annotation.

There are three ways to achieve this:

1. Make `contract_assert` a statement instead of an expression, so it can not appear in the operand of the `noexcept` operator or in a context where it could influence a deduced exception specification. This is the simplest option, but also the one that makes the most code unnecessarily ill-formed.

2. Make it ill-formed to apply the `noexcept` operator directly to an assertion expression, or to any expression that contains an assertion expression as a subexpression; further, make it ill-formed for an assertion expression to appear in a context where it could influence a deduced exception specification (for example, in a member initialiser of a class without a user-provided constructor). This option makes less code unnecessarily ill-formed, but is somewhat more complex to specify and implement.

3. Make it ill-formed to apply the `noexcept` operator directly to an assertion expression, or to any expression that contains an assertion expression as a subexpression; only make it ill-formed for an assertion expression to appear in a context where it could influence a deduced exception specification if its presence would actually change the result of that deduction. In other words, such code is only ill-formed if a non-throwing exception specification would have been deduced without the contract check, but adding the contract check is allowed if a potentially-throwing construct is already present. This is the option that makes the least amount of code ill-formed, but has the highest specification and implementation complexity.

The first option seems too drastic, but either of the other two seems like a reasonable compromise. It is also conceptually sound: whether or not the violation handler is `noexcept` is not known until link time, therefore the question whether a contract check can throw cannot be answered at compile time and will instead result in a compile error. The user is forced to state their intent — whether they want to allow their contract checks to throw — by explicitly specifying the exception specification for the function in question as `noexcept(true)` or `noexcept(false)`. This is more work for the user, but it avoids having a `noexcept` operator that gives a factually incorrect answer — or having to change the `noexcept` operator to only consider non-erroneously thrown exceptions — while simultaneously satisfying the zero overhead principle in all cases.

If we go down this route, the same logic must apply to all contexts where an exception specification is being deduced. This means that if we ever standardise `noexcept(auto)`, such a feature would be essentially mutually exclusive with contract annotations.

There is yet another downside to this approach. It would mean that `contract_assert` can no longer act as a drop-in replacement for existing `assert` macros in all cases, because there would now be contexts where the latter is allowed but the former is ill-formed. This can be a significant obstacle for transitioning large codebases with a high amount of assertions from macros to Contracts.

3.6 Rely on guidelines and diagnostics?

Rather than making code ill-formed where a potentially-throwing contract annotation changes the exception specification, a possible alternative is to rely on guidelines and diagnostics. This is what we do in other parts of the language where using a certain feature in the wrong way can unintentionally change the program semantics and introduce performance regressions. Consider:

```
std::map<int, Widget> map = { /* ... */ };
for (const std::pair<int, Widget>& elem : map)
  // do something with elem
```

In this case, the user got the element type of `std::map` wrong (which is `std::pair<const int, Widget>` rather than `std::pair<int, Widget>`); this generates an unintended implicit conversion, which in turn yields a temporary object that is lifetime-extended by the `const&`. This code compiles and works, but has a silent performance degradation due to the unnecessary conversion and object creation on every iteration of the loop. Such inefficiency is unfortunate; however, we do not add special cases to basic language rules such as range-based `for` loops, implicit conversions, or reference semantics to make these cases ill-formed. Instead, the user gets what they get, a quality compiler or static analysis tool will issue a warning, and several straightforward fixes are available (use the correct type, or just use `auto`).

In the same spirit, we could teach that one should not use `contract_assert`s in subexpressions when the result of a `noexcept` operator query may depend on it. Compilers can issue diagnostics whenever the user write such code. These diagnostics could be combined with either semantics: treating contract annotations as throwing or non-throwing. The diagnostic could be issued either as a matter of QoI, or even mandated by the standard. We created a precedent for having well-formed programs that nevertheless require a diagnostic by standardising `#warning`, and we might

standardise another such case if we adopt the compile-time-*observe* semantic from [**?**], so we could do the same here.

However, if we really want to normatively add a warning to discourage the user from writing such code, because we think it is bad practice to do so, it seems like the better choice is to just make it ill-formed.

3.7 Remove support for throwing violation handlers?

Throwing violation handlers have limitations in practice. In order to use them, the codebase must:

— not have any functions marked `noexcept` between the contract check and the `catch` clause handling the exception,

— be written in an exception-safe way.

It has therefore been suggested that it may be impractical to use throwing violation handlers; that we should instead specify that violation handlers shall be `noexcept`; and that we should replace throwing violation handlers by some other not-yet-specified mechanism to enable non-terminating contract violation handling in a way not based on exceptions.

On the one hand, removing throwing violation handlers from the language would also remove all the issues discussed here, as contract checks would then become unambiguously non-throwing. On the other hand, once we ship a Contracts facility specified in this way, it might turn out to be very difficult to retrofit throwing violation handlers afterwards if we change our mind. Throwing violation handlers might not be a panacea, but when the codebase is written accordingly they can be used very effectively, we do have implementation experience with this approach (see [P2698R0]), and we currently do not have an established alternative mechanism to enable non-terminating contract violation handling. Therefore, it does not seem wise to remove violation handers from the Contracts MVP unless a convincing rationale for a concrete alternative mechanism materialises in time. We can however encourage work on developing such an alternative mechanism.

4 Summary

Contract-violation handlers in the Contracts MVP can throw; contract checks are therefore potentially-throwing constructs. This means that they interact with the `noexcept` operator and with deduced exception specifications. Treating them as what they are — potentially-throwing — for these interactions is straightforward and intuitive, but violates the zero overhead principle: the addition of a contract check to a program can lead to a different branch being taken at compile time. This is considered an undesirable property because it means that the addition of contract checks can cause "heisenbugs" and performance degradations even if the contract check has the *ignore* semantic.

The alternative, treating contract checks as non-throwing in the language even though they are actually potentially-throwing, avoids violations of the zero overhead principle, but would require us to change the meaning of the `noexcept` operator and deduced exception specifications: instead of answering the question "can this construct throw?", they would answer the question "can this construct throw *if no contracts are violated*?". This approach would make these features arguably less intuitive, and is hostile to throwing violation handlers and exception-neutral generic algorithms such as sender/receiver.

There are other possible directions but they all have negative tradeoffs as well. We could give the user the option to switch between the two behaviours (contract checks being potentially-throwing or nonthrowing), either through code annotations (which would still allow for contract checks that violate the zero overhead principle) or through different build modes (which would effectively create

a language dialect). We could allow exceptions thrown from contract-violation handlers to escape deduced non-throwing exception specifications (which would significantly complicate the mental model while only partially mitigate the negative effect on violation handlers). We could make any code ill-formed where the result of the `noexcept` operator or a deduced exception specification may depend on the presence of a contract check (which would mean that `contract_assert` can no longer be a full drop-in replacement for `assert` macros as there would be contexts where the latter are allowed but the former are ill-formed), and there are at least three different ways in which we could specify such code as ill-formed. We could rely on guidelines and diagnostics instead of hard compiler errors (which would put more burden on the user), either as a matter of QoI or normatively. Finally, we could remove support for throwing violation handlers (which would leave us without any mechanism for non-terminating contract-violation handling until we can develop an alternative mechanism for this purpose, a highly non-trivial task).

Acknowledgements

We would like to thank Lewis Baker for starting the discussion that led to this paper, as well as Joshua Berne, Jens Maurer, and Lisa Lippincott for their insightful comments.

References

[N4473] Ville Voutilainen. `noexcept(auto)`, again. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4473.html, 2015-04-20.

[P0133R0] Ville Voutilainen. Putting `noexcept(auto)` on hold, again. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0133r0.html, 2015-09-27.

[P2698R0] Bjarne Stroustrup. Unconditional termination is a serious problem. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2698r0.pdf, 2022-11-18.

[P2811R7] Joshua Berne. Contract-violation handlers. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2811r7.pdf, 2023-06-27.

[P2900R1] Joshua Berne, Timur Doumler, and Andrzej Krzemieński. Contracts for C++. https://wg21.link/p2900r1, 2023-10-09.

[P2932R1] Joshua Berne. A Principled Approach to Open Design Questions for Contracts. https://wg21.link/p2932r1, 2023-10-04.

[P2932R2] Joshua Berne. A Principled Approach to Open Design Questions for Contracts. https://wg21.link/p2932r2, 2023-11-14.