# Contract-Violation Handlers


| | |
|---|---|
| Document #: | P2811R3 |
| Date: | 2023-5-04 |
| Project: | Programming Language C++ |
| Audience: | SG21 (Contracts) |


Reply-to: Joshua Berne <jberne4@bloomberg.net>



**Abstract**

Numerous use cases for contract checking in production environments depend upon handling contract violations in a consistent and locally defined way. Based on existing designs deployed at scale over many years, we present here a proposal to allow for link-time customization of contract-violation handling, along with examples of how this method can satisfy a wide variety of important, practical, and well-known usage scenarios, while being entirely forward compatible with essentially all future enhancements envisioned to date.


## Contents

## Revision History

Revision 3

- Added the *Design* section to address questions proposed by SG21
- Added `invoke_default_contract_violation_handler`
- Made `contract_violation` polymorphic and noncopyable and clarified its lifetime
- Pointed out that program termination on enforced contracts should guard against signal handlers that then violate contract checks themselves
- Clarified the purposes of the `kind` and `detection_mode` properties
- Renamed `contract_violation_detection_mode` to `detection_mode`
- Renamed `detection_mode::predicate_exception` to `evaluation_exception`
- Renamed `detection_mode::predicate_detected_undefined_behavior` to `evaluation_undefined_behavior`
- Greatly clarified intended resolution for throwing

Revision 2

- Removed the `ignore` semantic enumeration since it is currently unused in practice
- Added discussion of counting violations for observed violations
- Added usage example of safe stopping
- Added, to introduction, explicit clarification on proposed modifications

Revision 1

- Clarifications on the undefined behavior when using `longjmp`
- Reasoning for the `contract_semantic::ignore` and `detection_mode::unknown` enumerators
- An explanation for why the violation handler is in the global namespace and clarification that it should be attached to the global module
- Other minor corrections

Revision 0

- Original version of the paper for discussion during an SG21 telecon

# 1 Introduction

C++20 Contracts, prior to their removal, describe a conditionally supported mechanism for installing a user-supplied contract-violation handler. This callback function — invoked immediately upon each detected contract violation — was intended to support both (1) the handling of user-defined reporting of a contract violation, e.g., via what mechanism and in what format to report the error, and (2) managing the *semantics* of contract violations within the program, e.g., terminate (the default), `longjmp` (save and quit), throw (and what to throw), or, in some use cases, continue as if contracts were disabled (e.g., *observe*).

Handling contract violations via a user-provided callback is an established, well-tested approach that is deployed in many modern assertion facilities. In particular, this approach has evolved directly from that used by BDE[1] in `bsls_assert` and `bsls_review`. User-provided contract-violation handlers were first deployed to production at Bloomberg in 2004 and have been in continuous use ever since. In addition, this approach has had later versions make their way through both LEWG and EWG to be moved for standardization[2] only to be rejected or removed at plenary prior to Standard publication. That is, the eventual lack of acceptance of a Contracts facility for C++17 and again for C++20 had nothing whatsoever to do with the utility or design of the contract-violation handler; instead, the issue was the design as a whole, i.e., the intrinsic use of macros (C++17) and the perceived design instability (C++20).

Attempting to clarify the semantics of contract checks[3] made clear that the local aspects of what a contract check should do — i.e., determining whether control flow can continue normally after a violation and whether the contract is checked — are separate from the decision of how precisely to report a contract violation.

The current MVP[4] has no mechanism for altering the behavior on contract violation, nor does it provide much guidance on what default behavior should occur when a contract violation is detected. The only mandatory behavior dictated by the `Eval_and_abort` build mode, where all contract-checking annotations are given the *enforce*[5] semantic, is program termination.

We propose here two modifications to the Contracts MVP.

1. Add the ability to provide a function — the contract-violation handler — to be invoked as part of the contract-violation handling process. In a *checked* build mode such as `Eval_and_abort`, program termination will occur when the contract-violation handler returns normally.
2. Establish a recommended practice for the behavior of the default contract-violation handler that platforms might wish to adopt when applicable.

Concretely, the current MVP makes the behavior on contract violation in `Eval_and_abort` mode implementation defined, requiring only that the program does eventually terminate on such violations (i.e., instead of throwing or invoking `longjmp`).

[1] See [bde14].
[2] See [N4378] and [P0542R5].
[3] See [P1332R0], [P1429R3], and [P1607R1].
[4] See [P2521R3].
[5] The *enforce* semantic for contract violations was originally named `check_never_continue` in [P1332R0] and is identical, other than the name, to the semantic originally described there.

Let's consider, abstractly, the simplest form of check, an assertion (using the syntax of C++20 Contracts):

```
[[ assert : X ]];
```

The current behavior in `Eval_and_abort` mode could be imagined to be transformed into something like this:

```
if (X) {} else {
   __invoke_platform_violation_handling();
   __terminate_program();
}
```

Here the `__invoke_platform_violation_handling()` intrinsic is required to not `throw` or invoke `longjmp` but otherwise has implementation-defined behavior. The `__terminate_program()` part could be an invocation of `std::abort` but should probably include additional guards protecting against user code — such as a user-installed abort signal handler — intervening and then recursively violating another contract.

This proposal alters this implementation-defined behavior into fully specified behavior, requiring that the compiler prepare an appropriately populated `std::contracts::contract_violation` object and then pass it to the (possibly) replaceable function `::handle_contract_violation`.

We do not propose removing the program termination on normal returns from handling a contract violation, since that would introduce a new semantic, *observe*, which will be the subject of a future proposal incorporating additional build modes and meta-information with which to control the use of that contract semantic.

## 2 Motivation

Custom violation handlers will turn an overly simplified Contracts facility that is highly ineffective in many environments into a moderately flexible and practicable one. Importantly, primary control of the semantics of a contract check — i.e., whether it is checked and how control flows after the contract-checking annotation (CCA) when a violation is detected — remain governed by the choice of build mode. In the `No_eval` build mode, contracts will continue to be have the *ignore* semantic, no runtime checking will occur, no contract-violation handler will be invoked, and control will silently pass over the CCA. The `Eval_and_abort`, by contrast, is currently the only build mode that would (or could) invoke the contract-violation handler immediately following the detection of a contract violation. If, after a contract violation is detected, the contract-violation handler returns normally, program execution will be terminated.

A typical custom violation handler will log appropriately; after that, the only mechanisms to circumvent program termination are to throw an exception, block, enter a nonterminating loop, or invoke `std::longjmp`. Continuation in this build mode is simply not an option.

Throughout the standardization process, two points have become abundantly clear: (1) No one specification of violation handling is correct for all users on all platforms, and (2) having a consistent and mostly portable way to customize behavior will greatly increase the utility (and thus, in some views, the viability) of a Contracts facility for a wider range of users.

Executing any code when a contract violation is detected is often a risk because one can never be certain that the program is not already exhibiting (language) undefined behavior due to earlier defects that were not detected by an appropriate contract check. The severity of this risk, however, will vary depending on the industry and application. We must always balance this risk with the many available benefits such a (1) producing useful diagnostics (e.g., long-running financial analytics), (2) at least saving the user's data (e.g., a document editor), and sometimes (3) making attempts at recovery and just soldiering on (e.g., video games). We assert that having the ability to customize contract-violation handling, in an appropriately structured manner, beyond just logging is (a) often necessary, (b) typically useful, and (c) *never* a net negative.

On some particularly specialized platforms, the ability to configure an alternative contract-violation handler might be considered an unacceptable security risk. Therefore, the ability to provide a replacement contract-violation handler is only *conditionally supported*. We suggest that providing a candidate function for replacement on platforms that do not support such replacement be an error so as to minimize any related confusion. Alternatively, such secure platforms might still choose to disallow replacement but instead provide a suite of violation-handling implementations from a selection of well-vetted, vendor-provided routines. Again, we would expect that, on general-use platforms, developers will be able to create and supply fully custom violation-handling routines at build time.

# 3 Design

After discussion in an SG21 telecon on April 20, 2023, the SG21 consensus became clear: to have a Standard interface for a link-time replaceable contract-violation handler. During and after that telecon, copious discussion has continued on the many nuances and implications of having this handler. A number of topics have come up that have questioned and in some cases evolved this proposal. Here we present deeper reasoning and rationale for the current state of this proposal to add a Standard replaceable contract-violation handler to the MVP.

## 3.1 Separation of Concerns

Reasoning about code having contract checks might not come naturally to everyone. When analyzing a block of code that has a defect, a couple of questions regarding the contract-checking annotations in that code greatly benefit the reader's understanding of the runtime behavior of that code.

1. Will I be made aware of a violation?
2. Will execution continue after a violation?

Knowing whether a violation will be reported reliably (or at least as reliably as possible in a program with a defect that is being investigated) lets one reason based on the lack of such reports; if you didn't see a log message telling you that the particular contract check failed, then its associated contract check was quite likely not violated.

Knowing whether execution will be permitted to continue past the CCA when the contract is violated lets the reader know how broadly they must reason about the behavior of the code following the contract check.

The four possible answer combinations to this pair of questions equate exactly to the four semantics proposed for a CCA in C++20 Contracts.[6]

1. An *ignored* contract check does not inform you of violations, and control flow always continues even when the contract check would have failed.
2. An *enforced* contract check reports violations and does not allow continuation.
3. An *observed* contract check, like an *enforced* one, reports violations but, like an *ignored* one, is permitted to continue past the CCA.
4. An *assumed* contract check, similar to an *ignored* one, does not report violations, but continuing past the violation necessarily leads into (language) *undefined behavior*, thus not a program state that can be reasoned about.

That is to say, each CCA in a program is compiled to have exactly one of these four semantics based on the contract-checking build mode. In the current MVP, they would all be either *ignored* or *enforced*. In a future release of our C++ Contracts facility, any subset might be controlled independently.

Let's now imagine for a moment that we could indicate, for each CCA, exactly which of the four semantics it was allowed to be assigned, and then somehow we could control which of those was selected by the build mode.

Being able to determine locally what potential semantics a contract check might have and, for a given build, being able to determine the exact semantic each contract will have allows one to reason about the answers to these two questions and thus about the expected and observed behavior of the resulting program.

The first property, whether a violation might be reported, is determined entirely by the semantic with which the CCA is compiled. The requirement that the CCA be compiled in a particular semantic is fortified by our need to incur zero overhead when a CCA is *ignored*. If it is compiled to *ignore*, no runtime check will occur, and thus the handler will under no circumstances be invoked. If the CCA is compiled to have either the *enforce* or *observe* semantic, the runtime check will be active, and if the predicate fails to evaluate to `true`, the handler will be invoked. In general, if the CCA is compiled with the *assume* semantic, then, just like with *ignore*, no runtime check will occur and the handler is unlikely to be invoked; however, since any violation is *undefined behavior*, a compiler might, in some circumstances, decide that the optimal action for some violations is to actually invoke the contract-violation handler. Unless the CCA is compiled to a *checked* semantic in the translation unit in which it resides, the handler has little opportunity and no requirement to report the error; if the handler is called, only then does it become responsible for formatting the information and outputting it appropriately.

The second property, continuation, *could* be delegated to the violation handler. If the handler returns normally, program flow will continue; otherwise, the violation handler will have taken the responsibility for doing something else, such as throwing, terminating, or sleeping.

Compelling reasons arise, however, for not giving the violation handler the unilateral ability to choose to continue.

[6] See [P1332R0], [P1429R3], and [P1607R1].

- A highly anticipated future feature that allows for controlling the choice between *observing* and *enforcing* contract-checking annotations based on meta-information provided in the CCA itself (such as a `new` marker) will have to depend on being used with violation handlers that would respect a request to continue. A violation handler written today against the MVP will have no knowledge that it even *should* be returning normally, rendering a local *observe* semantic unusable.
- A user expecting to deliver a compiled binary that is checked — i.e., has all contracts enforced, such as with the `Eval_and_abort` build mode in the MVP — and then reason intelligently that the client will be running without unknowingly violating any contracts will be subverted if that same client can install a contract-violation handler that can simply choose to continue. The ability to deliver such checked builds could be a huge boon to library vendors that do not wish to distribute source code.
- The compiler translating a CCA that is *enforced* can, if it knows that the enforcement will never be skipped, use the truth of the contract predicate to optimize code after the check. By compiling the program termination locally, we enable one of the primary mechanisms used for the chaining of postconditions to matching preconditions of later functions in a way that has a decent chance of providing real performance improvements, even in fully runtime-checked builds.

  Though this approach would require substantial new compiler technology and investment in producing thorough precondition and postcondition annotations, it could conceivably result in a checked build of an application having *no* runtime checks yet guaranteeing that no contracts will be violated. Moreover, this design would allow a program to incrementally reach this goal, where any checks that cannot be proven from the previous (local) context will simply be evaluated at run time.

These benefits all hinge on the overall contract semantic being known by the compiler when translating a single translation unit, well before a link-time contract-violation handler is known. Even link-time optimization will not aid the binary library distribution use case.

An *assumed* CCA — should we ever choose to support it — will implicitly have the same benefit as if the *enforced* check had run and passed (independent of the contract-violation handler). A CCA that is compiled to either the *ignore* or *observe* semantic opts to forgo this benefit, allowing continuation past a potential runtime violation.

Thus, the question arises: Could we not just place the compiled semantic (e.g., either *enforced* or *observed*) into the `contract_violation` object when we pass it to the violation handler on a detected violation and *trust* that the handler (supplied by the owner of `main`) will respect that semantic?

We are left with three alternatives.

1. Refrain from assuming that the contract-violation handler will respect that property and forgo any of the above benefits that might arise from knowing that the predicate is `true`.
2. Make the contract-violation handler responsible for not returning normally when told that the contract semantic is *enforce*; if the handler does return normally, consider that to be undefined behavior.

3. Guarantee that continuation will not happen by explicitly terminating the program if the violation handler chooses to return normally to a CCA compiled with the *enforce* semantic.

Forgoing the benefits of knowing that continuation will not happen for certain builds of certain contracts hinders valid use cases and limits any mitigating performance benefits of contract checking, making the first alternative unattractive.

SG21 has repeatedly expressed its belief that introducing new sources of undefined behavior as part of the contract-checking facility is unwise. We agree. Hence, option two is a poor idea.

We therefore propose that, when compilation options and meta-information on the contract-checking annotation together have determined that the semantic of the contract check is to be *enforce*, the local code will explicitly invoke `std::terminate` should the handler return normally, which would be the typical behavior in checked builds in the Contracts MVP.

This modular design facilitates reasoning about continuation (or the lack thereof) without even knowing what contract-violation handler will eventually be installed. Moreover, this separation of concerns allows *most* such handlers to focus on just the single aspect of the violation-handling process that the default contract-violation handler would implement, which is to produce and deliver a diagnostic message indicating that a failure has occurred.

Note that the contract-violation handler is still user-defined C++ code, executed normally, so for all of the varied use cases that extend beyond those that are baked into the basic semantics provided by the Contracts facility, the violation handler remains a place to attempt to implement those semantics.

The need for this separation of concerns became apparent during the development of the initial proposals of the semantics for the C++20 Contracts facility, which came from attempts to leverage early C++20 Contracts to implement Bloomberg's already deployed contract-checking facility. Prior to that work, that such logic could all be embedded entirely in a violation handler was assumed. Attempts to implement features such as `BSLS_REVIEW` with a sufficiently advanced contract-violation handler proved unsustainable and fruitless.

- To allow any continuation, all code needed to be compiled to allow continuation. Mature checks that were trusted would be allowed to continue even past unexpected violations, leading to long latent bugs continuing to be significant business risks.
- Code that we would otherwise have *enforced* does not benefit from the runtime optimization of knowing that it either is correct or will not continue.
- Identifying those contract-checking annotations that *should* allow continuation required access to syntax added to the annotation itself or baking into the violation handler such knowledge of the source code. Neither of these were viable with C++20 Contracts prior to the addition of explicit semantics.

Once the semantics could be specified on a contract-checking annotation, however, a robust implementation of `BSLS_REVIEW` that was based on language contacts became trivial to produce.

In short, we will eventually want to further annotate our CCAs to include additional information that enables us to group and control their semantics from the contract-checking build mode. Importantly, we will want to *observe* new checks while continuing to *enforce* old ones, possibly within the same

translation unit. The MVP does not allow us to do this today, but by following the design approach suggested above, the addition of the *observe* and even the *assume* semantics is entirely backward compatible. Adding a few more basic support features will enable us to realise the full control needed for industrial-strength applications at scale.

### 3.2 ABI Compatibility

A primary goal of the design of the `contract_violation` object is to enable a fair range of ABI migration strategies to allow the use of Contracts in environments where binaries are often assembled (sometimes even at run time) from individual components built at vastly different times by potentially completely different toolchains.

ABI compatibility concerns are typically hard to resolve since their proper design requires a fair bit of speculation about future interactions. We do, however, know that we need (or do not need) to support some specific use cases.

- We should be able to expect that a platform that compiles a program's violation handler is also providing the language-support library function that will be constructing a `contract_violation` object and directly invoking the handler.
- The design must not require that all translation units in a program are built with the same settings, Standard version, or even toolchain. Eventually, ABIs will need to specify what information will be provided to the contract-violation handling process in a form that is an implementation detail of the platform rather than specified by the C++ Standard.
- The design must allow contract violations in any translation unit within the program, including those from dynamically loaded shared libraries, to result in the invocation of the single (unique) user-provided contract-violation handler linked into the program.
- The Standard specification itself must not so restrict implementations that do not benefit from such ABI migration concerns that those implementations cannot choose a potentially more optimal implementation strategy.

We thus intend to require the `contract_violation` object and its use in `::handle_contract_violation` to have a minimum of compile-time dependencies on the specific version of that type being used. The model from which this design starts is the same approach taken for `std::type_info`. Following that type, we propose a few specific properties for `contract_violation`.

- All accessors for the various properties this type exposes are nonvirtual nonstatic member functions, which allows future versions to easily add more accessors without an ABI break.
- Whether `contract_violation` is polymorphic is unspecified.[7] This freedom allows a platform that does not expect to extend `contract_violation` to use smaller objects that do not have a vtable pointer, while platforms that do wish to perform such subclassing can make the object polymorphic to enable the use of `dynamic_cast` to identify when a more specific type has been provided (see Section 5.9).
- The `contract_violation` object is not copyable and provides no `public` constructors for users to create their own instances. This restriction prevents its use in notoriously ABI-sensitive ways such as data members or automatic variables.

One alternative considered was making `contract_violation` an incomplete type and then providing all access to its properties through free functions. The idea was to insulate its definition, yet this approach, while viable, would result in minimal benefits and significantly complicate use. Also noted, by using a complete type, clients could depend on the size of the object but only in the most minimal ways (such as by declaring but never engaging an instance of `std::optional<std::contracts::contract_violation>`), which do not appear relevant.

To further maximize ABI compatibility, we disallow the dependence of `contract_violation`, in any way, on other Standard Library definitions, such as `std::string` or `std::runtime_error`, that might not have the same standards for maximizing cross-version compatibility.

### 3.3 Dynamic Libraries

Specification for the behavior of software that uses dynamically linked libraries is outside the scope of the C++ Standard itself, but we still benefit from discussing what our expectations are for the real world where applications do often link and execute additional code that was not readily available when the program started.

The contract-violation handler is a link-time construct, but since a contract violation may happen as soon as any code begins to execute at run time, the contract-violation handler must be available as soon as program execution begins. The contract-violation handler is also a global entity; there must be one and only one handler in a program. By having exactly one handler, we maximize the control the entity that owns `main`[8] has over the program's behavior in any particular deployment.

Therefore, the contract-violation handler is to be linked in such a way that it will be available at startup, and all loaded shared objects must still route to the same contract-violation handler. Performance of the violation handling process is not critical, so the cost of making this function dynamically loaded from all points of use in dynamically loaded libraries seems a reasonable expectation.

We must consider, however, the situation in which an *old* program that knows nothing of contracts loads a shared library that makes use of contracts. In such cases, we might have no violation handler outside the shared library to link to. We recommend platforms solve this problem by either (1) including a weakly linked default contract-violation handler in shared libraries, so that it may take effect when no global contract-violation handler is available, or (2) simply failing to load such libraries at run time. One mitigation strategy for this problem might also be to provide updated, compatible platform-shared libraries that include the default violation handler as a weakly linked symbol.

Application environments in which shared libraries have more bounded interfaces with their environments — such as COM, device drivers, or control panel applications and service applications on Windows — might consider instead identifying these complete components as being functionally equivalent to complete applications and thus select for each such component a single contract-violation handler. As the containers for these forms of applications evolve, they may add interfaces

[7] See [LWG2398] for the original resolution that inspired this approach.

[8] When we say "the entity that owns `main`," we generally are using that phrase as a proxy for the person responsible for choosing how an executable is built and assembled. In general, `main` exists with a one-to-one relationship to a compiled executables, hence the proxy relationship is viable. For platforms having no `main`, we recommend giving an entity having a similar property the final responsibility for picking a global violation handler.

to allow the contained components to pass contract violations to the enclosing system, thus giving the full system total control over how to handle violations centrally.

Another concern related to dynamic libraries, any static storage duration data within a dynamic library is accessible only until the library is unloaded, e.g., via a call to `dlclose()`. We cannot universally guard against this scenario (provided that the unloading may potentially happen concurrently with any part of the violation handling process, long before the contract-violation handler is even invoked), yet it offers a compelling reason to discourage users from depending on the lifetime of a `contract_violation` object extending beyond the point where the invocation of the contract-violation handler completes.

### 3.4 Reentrancy

The general problem of reentrancy for contract-violation handling is potentially non-trivial. If, during the evaluation of a contract-violation handler, a contract is again violated, the natural infinite recursion that is likely to happen can suppress the output of any diagnostics (or even suppress a prompt termination of the program itself), leading to a hard-to-analyze defect.

In practice, reentrance into a contract violation will work correctly with almost any implementation strategy. The only strategy that will prevent a recursive call to a contract-violation handler would be when a single `contract_violation` object is reused (populated differently) for every contract-violation handler invocation. We therefore do not recommend this implementation strategy.

Complexity brings with it the potential downside of failures cascading, and this proposal seeks to enable developers to intelligently make this tradeoff for themselves.

### 3.5 Enumerations

Many of the properties on a `contract_violation` object present bespoke values, something that is most naturally represented as an `enum`.

Because the values themselves are not meaningful, they are specified using `enum class` — both to scope the enumerator names and to remove the meaningless arithmetic operations that would come with a classic `enum` nested in a `struct`. To ensure sufficient space for implementation-defined values, we specify the underlying type of each of these `enum class` types to be `int`. Note that this constraint might bloat the size of a `contract_violation` object that contained these values as members, but an implementation that knows it will not use the full range can store much less, simply promoting the stored value to the full size when the corresponding accessor is invoked.

Due to the possibility of multiple compilers seeking to produce contract violations that are delivered to a single contract-violation handler as well as a desire to capture the values on a `contract_violation` object and parse them in a platform-independent manner, we have specified the values to be used explicitly for these objects.

Implementation-defined values for these `enum`s should always be considered a possibility, so we specify that a range of values are reserved for the implementations, which will thus never conflict with a future Standard. Implementations that are aware of a future Standard having more enumerators that the implementation wishes to support are encouraged to use that future Standard's values

(and so are not restricted from using values in the *Standard's* range, since they are using values the Standard is known to use in an upcoming version).

The use of opaque `enum`s having free functions to identify their values was considered. This design would allow an implementation to treat both an implementation-defined value and a standardized value as synonymous, but otherwise the idea seemed to offer little benefit and significant additional complexity. The mixing of implementation-defined and Standard-defined values seems to be a tractable enough problem to reason that this added complexity is unjustified.

The fixed values chosen for all enumerators begin at `1` to avoid different enumerators having different truthiness (i.e., different values should they be converted to `bool`).

### 3.6 Naming

The names of types, functions, and enumerators proposed in this paper will be used quite rarely in common practice. Any given program will have (at most) only a single contract-violation handler defined in it, and such handlers will often be reused by any programs that share the things the violation handler does, e.g., business needs, logging frameworks, and so on.

Therefore, minimizing the size of the names we propose here does not seem to be a priority. Instead, we sought to produce clear and self-evident names that have little to no chance of colliding with existing names. For example, we named the enumerator for contract semantics `contract_semantic`, not just `semantic`, since the latter is ambiguous about its intent and likely to be in use in other user-defined code (or even in a future Standard for some other purpose).

In the interest of protecting existing namespaces, we have, perhaps redundantly, also put all names in the `std::contracts` namespace. This choice was made deliberately to remove this rarely used set of names from what is provided when using `using namespace std`.

Overall, this naming approach leads to somewhat verbose, fully qualified names. Since users will have little need for them, we're fine with that. If, however, a consensus arises to remove that verbosity, we recommend moving the names to `namespace std`, retaining the (longer) chosen identifiers that are clearly related to the Contracts facility, even if not sequestered in the bespoke namespace `contracts` (which is sometimes but far from consistently done within the Standard Library).

No name is set in stone until the Standard ships, of course. We have proposed some alternates as well as listed some names proposed by others, and we are not especially partial to any of the names.

### 3.7 C Compatibility

SG21 certainly hopes to see, as part of a future C Standard, a contract-checking facility that is as similar as possible to and interoperates smoothly with the C++ contract-checking facility. One of the primary points of interaction between the two facilities will be a shared approach to violation handling — all reasons to have a central violation handler for all C++ code in a program apply equally well to C code in the same program. One could argue for a violation handler with C linkage, which should be fine because exactly one can be in any given program.

On the other hand, the needs of C++ contract checking will inevitably be greater than those of C — e.g., member functions, class invariants, coroutines, and use of or elevating beyond the preprocessor.

Representing all of these factors in the arguments to a contract-violation handler in C would be limited by a lack of the tools C++ provides for encapsulation.

The syntax that the C++ community is likely to find natural and acceptable is also highly likely to be distinct from whatever syntax the C Standard adopts, which might be both more conservative in some ways and more liberal in others. The syntax question — and whether it can be resolved so the same syntax works in C++ and in C, since the C feature set will inevitably be a subset of the C++ syntax — is a question that should be answered separately.

Our recommendation in that regard is to develop a syntax natural to C++ and, should C adopt a functionally equivalent syntax for the subset of features applicable to C, support that C syntax in C++ as an alternate spelling of the same functionality for the subset of features that overlap.

Independent of the syntax choice, the same concepts will apply when a contract check is evaluated and determined to have been violated. The properties specified here will be made available to a custom contract-violation handler installed according to the rules of either C or C++.

We expect, therefore, that a contract violation generated in either language should be reportable to a single contract-violation handler defined in either language; although the properties that indicate how a violation happened in a C++ program might not have values that a C contract-violation handler will be expecting, the C contract-violation handler should manage those properties gracefully.

Much of this functionality could be implemented as implementation-defined behavior. Both languages will specify a mechanism to provide a user-provided contract-violation handler, and implementations are free to require that at most one of those two mechanisms is leveraged in a single program.

A potentially more easily implemented solution is to tie the handler type to the language of the translation unit that defines `main`. If the language is C, a C++ contract-violation handler will be installed that delegates to the C violation-handling mechanism. If a C++ `main` is defined, the default C violation handler will delegate to the C++ contract-violation handling process.

As with shared libraries, the details of this process are outside the scope of either Standard to define, yet developers are free and encouraged to create implementations that make the interoperability as seamless as possible.

### 3.7.1 C Annex K

The C Standard's Annex K[9] provides a mechanism for setting a handler at run time to address violations of various constraints of the core C language. These violations are essentially all related to preconditions when using builtin functionality, and the C Standard would ideally support reporting violations of these preconditions in a way that feeds into the global contract-violation handling facility.

To integrate these preconditions, we suggest that, for a future C language implementation that supports both Annex K and Contracts (either as a new conditionally supported "Annex N" or as a core feature), the default constraint handler would invoke the contract-violation handler with appropriately populated information based on the constraint. Support for this design would benefit

[9] See [isoc11], "Annex K," p. 582.

from having a new value for the `detection_mode` and `kind` enumerations to indicate that the source of the violation was an Annex-K constraint.

### 3.8 Implementation Possibilities

Various implementation strategies can lead to different ABI evolution guarantees. One simple strategy would be to, within each translation unit when a contract check evaluation is being generated, produce a `static contract_violation` object having the appropriate contents at each point where the contract-violation object might be invoked. For example, to show the full range of how such objects might be used and initialized, consider a function, `f`, having an assertion in its body that calls another (opaque) function, `foo`, as part of its predicate:

```
// f.cpp
void foo(int* p);  // used in predicate of CCA below

void f (int* p)
{
   [[ assert : *p == 0 && foo(p)]];
}
```

The transformation of this assertion when enforced might depend on one compiler intrinsic that creates a `contract_violation` object at compile time and another that invokes the current violation handler with a specified `contract_violation` object and then proceeds to terminate the program:

```
void f(int* p)
{
  // Begin transformation of [[ assert *p == 0 && foo(p)]].
  static constexpr char*           __comment = "*p == 0 && foo(p)";
  static constexpr source_location __location { "f.cpp", 5, "void f(int*)" };
     // shared data between all local contract_violation objects

  if (p == nullptr) {
    // identified potential undefined behavior in the contract predicate
    static constexpr contract_violation __violation =
      __make_contract_violation( __comment, __location,
        contract_kind::assert, contract_semantic::enforce,
        detection_mode::evaluation_undefined_behavior);
      // initialized contract_violation object at compile time
    __enforced_contract_violated(__violation);
  }
  bool result;
  try {
    result = ( *p == 0 && foo(p) ) ? true : false;
      // Contextually convert expression result to bool.
  } catch (...) {
      static constexpr contract_violation __violation =
        __make_contract_violation( __comment, __location,
          contract_kind::assert, contract_semantic::enforce,
          detection_mode::evaluation_exception);
    __enforced_contract_violated(__violation);
  }
```

```
  if (!result) {
    // false predicate detected a violation. Invoke
    // violation handler outside of above try/catch blocks.
    static constexpr contract_violation __violation =
      __make_contract_violation( __comment, __location,
        contract_kind::assert, contract_semantic::enforce,
        detection_mode::predicate_false);
      __enforced_contract_violated(__violation);
  }
}
```

This implementation maximizes the data that can be kept in read-only memory but has the disadvantage that all translation units will need to be compiled with fully ABI-compatible versions of `contract_violation`.

An alternate approach having possibly greater (but probably insignificant) overhead is for the platform ABI to define a versioned `contract_violation_data` object for which each translation unit will create instances. Then, the `contract_violation` object will serve to adapt that versioned `struct` into the interface the compiled contract-violation handler expects. When referencing an older version of the data object, sensible defaults can be produced for new functions, and newer data objects should include all fields that any older instance of `contract_violation` might seek to use.

This design retains the ability to keep most of the objects in read-only memory, while enabling a protocol for translation units that have been built with an arbitrary range of platforms to interoperate. This same data object could also then be passed to a C linkage violation handler to be interpreted by C language free functions, thus allowing C++ contract violations to be handled by a C language violation handler, and vice versa.

### 3.9 Skipped Potential Features

*Unique Identifiers* — A unique identifier for each contract violation is potentially quite helpful to properly implement exponential backoff logging of *observed* contract violations. Various implementation strategies provide different methods in which such an identifier could be generated. An implementation that placed `contract_violation` objects prepopulated (or data objects used by the `contract_violation` objects) in read-only memory could easily use the address of that read-only data as the unique identifier for a contract violation.

Another option is to use the address of the instructions that invoke the contract-violation-handling mechanisms (i.e., the instruction pointer of the appropriate stack frame that led to the violation handler being invoked) for this unique identifier. This approach is discussed further in Section 5.8.

*Program Name* — Suggestions have been offered that the program name should be part of the `contract_violation` object. Gathering this platform-dependent information for all contract violations seems like a cost that is not universally acceptable. A contract-violation handler itself is also quite capable of invoking any API that exposes this information should doing so be needed, so we recommend that, rather than this being a property for `contract_violation`, cross-platform facilities to obtain this information be explored. Notably, the default violation handler is welcome to include the program name in its output if that is deemed appropriate for given platform.

*Nonterminating `throw` possible* — The desire that contract violations never lead to program termination has been expressed. Achieving this goal through the use of exceptions thrown from a contract-violation handler is directly in conflict with the use of `noexcept` on any function that might have a (possibly deeply nested) defect. Only unconstrained hubris would lead anyone to believe that any function can be deemed absolutely free of defects, and thus making that a requirement for the use of `noexcept` seems infeasible.

An alternative that has been suggested is to indicate, in the `contract_violation` object, that the contract-checking annotation being evaluated is a precondition or postcondition attached to a `noexcept` function. This indicator would not, however, prevent any thrown exception from a violation handler from leading directly (and swiftly) to program termination. The range of scenarios in which an exception thrown within the body of a function having a nonthrowing exception specification — from a simple assertion to any CCA on any function invoked indirectly — is much broader than can be easily identified.

On the other hand, the same logic that allows stack unwinding to be skipped when it will hit a `noexcept` boundary could be exposed through a standard interface to identify if that *would* happen should an exception be thrown.[10] This information would at least allow a violation handler to identify such cases and follow an alternate path (e.g., `std::longjmp`, or logging more verbosely) to attempt to avoid certain termination.

Addressing this concern for only preconditions and postconditions of `noexcept` functions seems likely to be insufficient for any practical purpose, since the concerns emanate from exceptions thrown within the body. The illusion that the function was somehow safe might even be considered actively harmful. In any event, this woefully incomplete solution is, in our judgment, unworthy of further consideration at this time.

# 4 Proposal

Our proposed solution to broaden the viability of a still-minimal Contracts facility starts with how to define the object that will capture the details of a contract violation, how to specify a contract-violation handler, and what the recommended default contract-violation handler should do.

## 4.1 An Extensible `contract_violation` Type

**Proposal 1: The Standard Library `std::contracts::contract_violation` Type**

A new language-support type, `std::contracts::contract_violation`, will be added to the Standard Library and will be designed for extensibility in an ABI-compatible manner.

To specify a custom violation handler, we will need to provide a type to represent the information that will be gathered and made available to a contract-violation handler when a contract violation occurs. This type might have any number of properties, but we want to ensure that it can evolve while remaining ABI compatible.

[10] See http://lists.isocpp.org/sg21/2023/04/2784.php and http://lists.isocpp.org/sg21/2023/05/3188.php, where it was pointed out that, at least with the Itanium ABI, a function such as `std::throw_finds_noexcept` could probably be implemented to indicate that stack unwinding would terminate.

We therefore recommend the following expectations and requirements for this `contract_violation` type.

- Property types will be builtin types, enumerations, or Standard-Library value types such as `std::source_location`. Strings will be null-terminated byte strings denoted by a `const char*`.[11]
- Each property accessor will return by value, not reference. Hence, a `contract_violation` object itself is never required to maintain a member for any of these properties unless it chooses to do so.
- Each property of `std::contracts::contract_violation` objects will have a recommended practice for any values supplied when a contract violation is detected but no requirements as to the specific values with which that property must be populated. Implementations may, therefore, choose to store more information for improved diagnostics or to carry less information in an executable for reduced overhead, potentially leaving that choice to the user.
- Objects of type `std::contracts::contract_violation` will be passed by `const&`, not by value, to the contract-violation handler.
- Objects of type `std::contracts::contract_violation` will not be copyable (i.e., the type will have a deleted copy constructor and move assignment operator), nor will they provide any user-facing constructors. This restriction will prevent meaningful use of these objects as user-defined variables (e.g., member variables or automatic variables) and reduces the chance of dependence on attributes of the complete type that would prevent evolution of the type.

**Proposal 1.1: The `<contract>` Header**

The type `std::contracts::contract_violation` shall be defined in a new language support header `<contract>`.

The C++20 Contracts facility has the primary declaration of this type in a new header, `<contracts>`, and we see no compelling reason to place it elsewhere. But, much like other complex systems (e.g., `std::pmr`), we now understand that this facility will inevitably evolve to require many other such supporting types; hence, we propose to give our new Contracts facility its own namespace under `std`, `std::contracts`. (This sort of logical-physical cohesion is considered by many to be an industry best practice.)

**Proposal 1.2: The `location` Property**

The type `std::contracts::contract_violation` shall provide this accessor:

```
std::source_location location() const noexcept;
```

The recommended practice for the `location` property is to provide a source location for identifying the contract-checking annotation that has been violated. When possible on a precondition, this property would ideally be the source location of the point of function invocation. When the invocation

[11] See [P1639R0] for the LEWG's reasoning behind the use of `const char*` in `std::source_location` and associated arguments in favor of using `const char*` over `std::string_view` in types necessary to the use of core language features such as `contract_violation`.

location cannot be ascertained and on other contract checks, the location provided would be the source location of the contract check itself.

Compiler flags that request that built executables not store information regarding the source code that produced the executable should not be rendered nonconforming by the need to populate this location property. To allow such options, producing a default-constructed `source_location` from this property would be permitted, although this option would be nonoptimal for those users who did not explicitly choose to have this information made unavailable.

**Proposal 1.3: The `comment` Property**

The type `std::contracts::contract_violation` shall provide this accessor:

```
const char* comment() const noexcept;
```

The recommended practice for the `comment` property is to include a textual representation of the predicate expression in the contract-checking annotation that has been violated. When storing the text of all potentially violated contract checks in a program is deemed to be too inefficient or cumbersome, returning the empty string (`""`) instead is recommended.

**Proposal 1.4: The `detection_mode` Property**

The `<contract>` header shall provide an enumeration having members contingent on which forms of violation detection are accepted:

```
namespace std::contracts {
enum class detection_mode : int {
  predicate_false               = 1,
  evaluation_exception          = 2,
  evaluation_undefined_behavior = 3
  /* to be extended with implementation-defined values and by future Standards */
  /* Implementation-defined values should have a minimum value of 1000. */
};
}
```

The type `std::contracts::contract_violation` shall provide this accessor:

```
detection_mode detection_mode() const noexcept;
```

For any given `kind` of contract-checking annotation, multiple ways might exist via which a given evaluation will end up identifying that a contract violation has occurred. Typically, a violation will be identified when a contract-checking predicate evaluates to `false`, thereby clearly indicating that the expected boolean condition failed to hold. In such cases, the value returned from the `detection_mode` accessor will be `detection_mode::predicate_false`. The various subproposals of Proposal 3 in [P2751R0] indicate that other situations, such as a predicate that throws or that has readily detectable undefined behavior, are also potential candidates for triggering a contract violation, and in such cases, indicating which mechanism led to the violation can greatly help guide the violation-handling behavior.

Other future extensions — in particular, any sources of contract violations that are not annotations with a simple boolean expression — would be associated with a distinct value for the `detection_mode` enumeration. In general, if a compelling reason justifies the ability to alter violation handling behavior based on *how* a violation was detected (such as possibly rethrowing when a predicate allowed an exception to escape), that state benefits from being identified as a distinct `detection_mode`. Purely informational distinctions are likely better served by placing that knowledge in the `comment` property in a human-readable form.

In particular, if we choose to cause an exception thrown from evaluating the predicate of a contract-checking annotation to trigger a contract violation, we must convey that information on the `contract_violation` object; otherwise, we have no way to identify that situation (an exception currently being in flight does not indicate that it was thrown by the contract predicate) and choose to give it special treatment, such as rethrowing certain classes of exceptions (see Section 5.3).

Other names that could be considered for this enumeration are

- `detection_method`
- `violation_trigger`

The name `cause` has been proposed, but in our experience, misleading any user about the potential cause of a violation is an egregious mistake. The predicate returning `false` is not the *cause*: It is an effect of the root cause, which was a defect in the program. Library developers that deploy large-scale contract-checked systems are inevitably keenly aware that often naive users blame the contract check for their own code's defects, and minimizing the chance that naming can contribute to that misunderstanding will greatly benefit all people who make use of Contracts.

**Proposal 1.5: The `semantic` Property**

The `<contract>` header shall provide this enumeration:

```
namespace std::contracts {
enum class contract_semantic : int {
  enforce = 1
  // observe = 2,  // expected in a future Standard
  // assume =  3,  // expected in a future Standard
  // ignore =  4   // expected in a future Standard
  /* to be extended with implementation-defined values and by future Standards */
  /* Implementation-defined values should have a minimum value of 1000. */
};
}
```

The type `std::contracts::contract_violation` shall provide this accessor:

```
std::contracts::contract_semantic semantic() const noexcept;
```

Contract-checking proposals, including the MVP, hinge on selecting, at build time, a runtime semantic controlling how detection and enforcement of each individual contract-checking annotation will be performed. Our research and experience tells us that at least four well-defined semantics are sound and practical to apply in appropriate circumstances. As of now, the MVP allows for contracts having two of those semantics, i.e., *ignore* and *enforce*, which are selected for all contract-checking

annotations within a program based on the build modes `No_eval` and `Eval_and_abort` respectively. In effect, this choice will result in users observing the results on only *enforced* contract-checking annotations in a contract-violation handler. We can, therefore, forgo even specifying the `ignore` enumerator as it would currently go unused. We hope that future evolution, however, will result in satisfying the use cases of those who wish for contract checks having other checked semantics, such as *observe*, since knowing which semantic is in effect for a particular violation greatly aids in programming the violation handler to behave properly.

In particular, when a contract is enforced, we know that the program will not continue normally after invoking the handler. A handler that wishes to proceed in any way other than program termination, such as by throwing or invoking `longjmp`, would thus need to do so after logging and prior to returning normally. But consider a future kind of contract-checking statement whose only two viable semantics were *observe* and *ignore*. Hard coding the throw or long-jump into the handler would interfere with the intent of always continuing but sometimes logging. In that case, we would want to do the long-jump or throw only when the contract check had the *enforce* semantic.

Moreover, since continuation might result in many violations of the same contract-checking annotation, a robust violation handler would not necessarily want to attempt to log a message on every violation. In general, the diagnostic of the first violation of each contract is quite helpful and, in many cases, only one violation might occur, so skipping that diagnostic entirely is ill advised. Diagnostics of repeated violations quickly become unhelpful, so best practice is to employ some form of exponential backoff for logging. This backoff strategy requires the violation handler to count violations of each *observed* contract check in a safe and reasonably performant manner. Providing a light stack trace to see the entire call chain is another useful technique, especially when *observing* a contract check.

In practice, a `contract_violation` whose semantic has the `ignore` value will never occur; if we are not evaluating the contract-checking annotation's predicate to determine if a violation has occurred, a contract-violation handler will never be invoked. The *ignore* semantic is still, however, one that the MVP allows contracts to have, and thus we could include it, but with no current programmatic need for it, we omit it for now.

Proposal 1.6: The `kind` Property

The `<contract>` header shall provide this enumeration:

```
namespace std::contracts {
enum class contract_kind : int {
  pre    = 1,
  post   = 2,
  assert = 3
  /* to be extended with implementation-defined values and by future Standards */
  /* Implementation-defined values should have a minimum value of 1000. */
};
}
```

The type `std::contracts::contract_violation` shall provide this accessor:

```
std::contracts::contract_kind kind() const noexcept;
```

Whether the contract annotation violated was a *precondition*, *postcondition*, or *assertion* annotation might guide the form of logging statements produced by a custom contract-violation handler.

Future extensions to Contracts might add new forms of contract checks, such as procedural interfaces[12] or class invariants, which could then be distinguished in a contract-violation handler by producing distinct values of the `contract_kind` enumeration. Extensions might also add new sources of detected contract violations, such as sanitizer checks, exceptions that attempt to escape a `noexcept` function, or C Annex K constraint violations. A contract-violation handler might benefit from distinguishing all of these cases.

The names for these enumerations should, of course, match the corresponding names used in whatever final syntax is chosen for contract-checking annotations in general. We have, for simplicity, proposed here names that match those from the C++20 attribute-like syntax.

It is important to note that the names `precondition` and `postcondition` would be uniquely bad for these enumerators because a `[[pre]]` contract-checking annotation is *not*, a priori, a *precondition*, and neither is a `[[post]]` contract-checking annotation necessarily a postcondition. A *precondition* in a contract is a responsibility that belongs to the caller, while a *postcondition* is a responsibility that belongs to the callee. The `[[pre]]` and `[[post]]` annotations are distinguished by *when* they evaluate their checks, not by who the responsible party is for those guarantees. In general, preconditions and `[[pre]]` annotations as well as postconditions and `[[post]]` annotations are highly correlated, but often this correlation fails to hold.

- Class invariants are the responsibility of the class to maintain, so generally they are not the responsibility of the caller invoking a member function of that class. On the other hand, a member function might decide to check such invariants using a `[[pre]]` annotation.
- Some guarantees a caller must make are time based and must remain `true` throughout the lifetime of a function invocation and hence might be checked by a `[[post]]` annotation, even though they are preconditions for invoking the function.
- An `[[assert]]` annotation near the start of a function is often a precondition check,[13] while one near the end of a function is often a postcondition check, but in general only the function implementer could readily identify which is the case. Some `[[assert]]`s are even better viewed as *sanity* checks — which could be considered stepping stones a function takes to verify that it is correctly on its way to guarantee its postconditions — but are not individually themselves part of the stated postconditions in a function's contract.

[12] See [P0465R0].

[13] In fact, in all existing macro-based runtime contract-checking frameworks today, *all* precondition checks are expressed as assertions within the function's body.

### 4.2 Contract-Violation Handler

**Proposal 2: Contract Violations Invoke a (Potentially) Replaceable Function**

When a contract violation is detected, prior to other specified behavior that is associated with the contract annotation, a function named `::handle_contract_violation` that is attached to the global module will be invoked. This function

- may be `noexcept`
- may be `[[noreturn]]`
- shall return `void`
- shall take a single argument of type `const std::contracts::contract_violation&`

Whether this function is replaceable is implementation defined.

Not all platforms, especially those that seek to have a thoroughly auditable security-conscious deliverable, will want to support a replaceable[14] contract-violation handler. We therefore propose that whether `handle_contract_violation` is replaceable be unspecified. For platforms where replaceability is not supported, defining the function shall lead to a link error (a clear and early indication to users who attempt to replace `handle_contract_violation` that they will not be able to take advantage of this portion of the Contracts facility). Platforms that do not allow for replacement of the violation handler are nonetheless encouraged to instead provide to users at build time a finite set of alternative behaviors that the default violation handler may have.

Notably, the language and Standard Library provide no mechanism to alter the behavior of the contract-violation handler at run time. A general feature having that purpose has been deemed by some to be too large a security risk on many platforms, though a custom violation handler can be crafted to achieve that effect.

As is generally assumed with arguments passed to a function by `const&`, accessing the `contract_violation` object after `handle_contract_violation` has returned will be undefined behavior. This enables implementations to choose to create that argument on the stack and thus allow it to be destroyed should the stack be unwound. Restricting the lifetime of this object also reduces the risk of problems if the object refers to static data in a dynamic library that might be unloaded, a likely possibility if during stack unwinding an RAII object unloads the shared library containing the contract-violation data when it is destroyed.

### 4.3 Default Violation Handler

**Proposal 3: Default Violation-Handler Behavior**

Recommended practice is that the default violation handler will output diagnostic information describing the pertinent properties of the provided `std::contracts::contract_violation` object.

**Proposal 4: Default Violation-Handler Invocation**

A new function that has behavior matching that of the *default* contract-violation handler will be added to the Standard Library :

```
namespace std::contracts {
  void invoke_default_contract_violation_handler(const contract_violation&)
}
```

When the violation handler is not replaceable or when no replacement is provided, recommended practice is to provide a violation handler that outputs useful diagnostic information (such as the contents of the `std::contracts::contract_violation` object) to a standard error-reporting channel for the platform (such as `stderr`).

[14]This proposal adds `::handle_contract_violation` to the set of replaceable functions the Standard defines, which currently includes various overloads of the global `operator new` and `operator delete`. Just like those functions, the violation-handler function is placed in the global namespace and attached to the global module.

Looking forward, should the committee adopt the ability to *observe* contract violations,[15] recommended practice would be that the default violation handler log diagnostics only infrequently, such as with some form of exponential backoff counter. Logging a diagnostic for each repeated failure of the same contract-checking annotation can quickly down a system, and observation is intended to avoid exactly that problem.

Platform capabilities, limitations, and other concerns will, of course, lead to default violation handlers that do much more or much less. This behavior can range from launching a debugger to doing nothing at all.

In addition, the simplest use case for using a custom contract-violation handler is to provide *additional* logging or diagnostics that is best layered on top of the platform default violation handler. A mechanism to, from within a user-provided contract-violation handler, invoke the platform default contract-violation handler makes this form of layering feasible without needing to reinvent wheels already provided by the platform.

# 5 Usage Examples

Providing just the hook of a custom violation handler effectively supports many use cases that would otherwise not be implementable using the MVP.

## 5.1 Custom Diagnostic Output

The most common use case for custom contract-violation handlers is to log the error to a particular output API in a particular format:

```
void ::handle_contract_violation(const std::contracts::contract_violation& violation)
{
  std::cerr << "Contract violated at:" << violation.location() << std::endl;
}
```

Taking into the account the risks and potential rewards, a handler might choose to add stack traces, the time, some subset of static program state, or other information before allowing the program to terminate.

Other programs might want to provide feedback to a user more directly:

```
#include <windows.h>`
void ::handle_contract_violation(const std::contracts::contract_violation& violation)
{
  MessageBox(NULL,
             (LPCWSTR)L"Contract violation, abort?",
             (LPCWSTR)L"Contract violation",
             MB_OK);
}
```

[15] By *observe*, we mean "determine if a contract violation is detected by a specific contract-checking annotation and then, when a violation occurs, invoke the contract-violation handler and, if the handler returns normally, continue execution immediately following the contract-checking annotation." See [P1607R1] and the earlier [P1332R0], which referred to this behavior as `check_maybe_continue`.

On other platforms, a program might send messages to `syslog`, use other central logging APIs, store an event in a diagnostic-event recording system, or perform other environment-specific actions to record the detection of a contract violation.

## 5.2 Throw on Contract Violation for Recovery

Rather than aborting, an application might instead choose to handle all contract violations as exceptions by throwing a known contract-violation-exception type from the handler:

```
void ::handle_contract_violation(const std::contracts::contract_violation& violation)
{
  throw my::contract_violation_exception(violation);
     // copies relevant fields from violation
}
```

Properly supporting this use of contract violations requires code be written with exception safety in mind, which also goes a long way toward supporting the use case of applications that do not have the choice to terminate.[16]

Since we, in general, do not want to make changes to the abstract machine itself, this use of exceptions does nothing to prevent termination related to `noexcept` functions further up the stack, but see Section 6 for more discussion on this topic.

Stack unwinding itself might, however, lead to other contract violations in the destructors of automatic variables on the stack, showing again that significant risk is involved in using exceptions as part of handling contract violations. Those who choose to do so must carefully evaluate the software they write to be sure it will handle such problems properly.

## 5.3 Propagating Predicate Exceptions

The proper response to an exception being thrown from the evaluation of a contract-checking annotation's predicate expression might arguably depend on context. Some applications might have resource recovery mechanisms to continue executing properly when, say, `std::bad_alloc` is thrown, while others might have no viable way to recover from a logical error that was expressed as a thrown exception (a common if unfortunate practice, such as through the use of `std::vector::at`).

In most cases when a predicate fails to evaluate cleanly to something contextually convertible to `true`, something is amiss and is causing the contract-violation handler to be invoked. For those tasks that are capable of recovering from a thrown exception, we can easily have a contract-violation handler propagate the exception that escaped the evaluation of the predicate:

```
void ::handle_contract_violation(const std::contracts::contract_violation& violation)
{
  if (violation.detection_mode() == detection_mode::evaluation_exception) {
    throw;  // rethrow the current exception
  }
}
```

[16] See [P2698R0].

In other cases, users can inspect the particular exception that was thrown and propagate only those that are known to represent resource acquisition failures from which the application is designed to recover:

```
void ::handle_contract_violation(const std::contracts::contract_violation& violation)
{
  // First, log appropriately.
  if (violation.detection_mode() == detection_mode::evaluation_exception) {
    std::exception_ptr exception = std::current_exception();
    try {
       std::rethrow_exception(exception);
    } catch (const std::bad_alloc&) {
       throw; // rethrow bad_alloc
    } catch (...) {
       // Return normally to abort on other exception types.
    }
  }
}
```

### 5.4 `longjmp` for Recovery

One option to avoid aborting without throwing an exception is to use `std::longjmp` to move control flow up the stack to a primary event loop and begin recovery. The hard edge on this approach is that behavior is undefined if the destructors of any non-trivial automatic variables would need to be invoked when unwinding the stack from the invocation of `longjmp` to the invocation of the corresponding `setjmp`, i.e., if the pair of calls were replaced with a `try` and a `catch`.

This undefined behavior manifests in at least two distinct ways. Many platforms simply skip any of the corresponding destructors, leading to possible resource leaks (or worse, an imbalance in the invariants that an RAII object, such as a `std::lock_guard`, was intended to enforce) but then continuing execution from the point of invocation of `setjmp`. Other platforms,[17] however, will, when `longjmp` is invoked, unwind the stack in a manner similar to what would happen when an exception is thrown. On platforms where `longjmp` will unwind the stack, all the same pitfalls that apply to attempting to recover via throwing an exception, will apply — although it can be expected that, even when crossing the boundary of a function with a nonthrowing exception specification, unwinding will continue and `std::terminate` will not be invoked.

Using this `longjmp` approach requires carefully managing threads of execution and their currently associated `jmp_buf` instances, handling the platform-specific behavior when unwinding goes past non-trivial destructors, and overall structuring an entire application to be prepared to recover in this manner. The solution proposed in [P2784R0] is similar to this approach in spirit, with similar associated benefits and potential pitfalls.

### 5.5 Performing a Safe Stop

For many purposes, program termination is the safest approach to handling software defects; users will be notified of problems through the auspices of the infrastructure that executed the program,

[17] Such as MSVC; see https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/longjmp.

actions can be taken to recover at a higher level that has not suffered from unknown defects, and normal computing activity can resume productively.

Some uses, however, do not have the surrounding infrastructure or, to make progress, must avoid making the same level of demands of a user. Naive users happily executing a graphical interactive program are unlikely to be well served by a program that simply disappears from their desktop for reasons they cannot readily identify. An embedded system navigating a car on a highway must, as a whole, continue controlling the car until a safe situation to stop execution is reached; the currently sleeping human being behind the wheel of the car will certainly prefer that termination behavior over being unexpectedly given control of a speeding car on a busy highway.

Both of these extremes — and many use cases in the middle — can benefit from a custom contract-violation handler being able to begin executing complex logic to *wind down* the system to a safe state to stop. Recovery of the original processing goals might be nonviable, but a minimal set of functionality can be launched to continue execution until a safe stopping point is reached.

For the client of software with a primary graphical interface, this safe state can be as simple as restarting the graphics-processing event loop within the violation handler to present an error dialog to the user before gracefully shutting down. This minimal runtime state can even give a user the ability to decide intelligently what information to retain or discard before finally terminating the program.

A self-driving car, on the other hand, can run much more simplified and well-tested code paths when the only goal is to bring the vehicle to a halt on the shoulder of the road, out of the way of passing traffic and ready to wait for the surprised and groggy yet still alive driver to manually take control of the vehicle once again.

All of this functionality is well defined and actionable using a custom contract-violation handler.

### 5.6 Runtime-Selectable Violation Handler

Enabling runtime selection of the mechanism for violation handling provides an attack vector to malicious actors that are capable of both (1) updating that mechanism to arbitrary functions determined by the attacker and (2) forcing a contract violation to be detected after that. Many of those who deploy C++ software, however, might determine that an attacker capable of those two steps is likely capable of many other malicious acts, so the risk of enabling runtime selection is acceptable. Put another way, locking the door to the garage doesn't help if the attacker is already in your house; the locked door just makes unloading your groceries from your car more difficult.

A custom violation handler that simply delegates to a user-specifiable violation handler that can be altered at run time is straightforward to implement by maintaining a pointer to a violation-handling function with static storage duration that can be updated by clients as well as accessed by the violation handler:

```
class RuntimeViolationHandler
{
public:
  using Handler = void(*)(const std::contracts::contract_violation& violation);

private:
```

```
  static std::atomic<Handler> s_handler

public:
  static void invokeViolationHandler(
      const std::contracts::contract_violation& violation)
  {
     Handler handler = s_handler.load();
     handler(violation);
  }
  static void setViolationHandler(Handler handler)
  {
     s_handler.store(handler);
  }
  static void defaultViolationHandler(
      const std::contracts::contract_violation& violation)
      // invoke platform-default contract-violation handler
  {
     std::contracts::invoke_default_contract_violation_handler(violation);
  }
};

std::atomic<RuntimeViolationHandler::Handler>
  RuntimeViolationHandler::s_handler = &RuntimeViolationHandler.defaultViolationHandler;

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

Runtime selection of the contract-violation handling behavior enables several common use cases whose benefits depend on the environment in which they might be used.

- Choosing violation handling behavior based on configuration options not identified until after `main` has begun
- Dynamically altering what information is gathered by a violation based on what phase a program is in during its execution, i.e., storing and attempting to save user information only after the user login process has completed and normal program usage is underway
- Dynamically altering the form of recovery that might be attempted on violation, such as attempting to recover via thrown exception only after the main program execution loop is underway

Any custom violation handler could certainly modify its behavior based on program state. A generic runtime-replaceable facility such as this one allows for a library solution to leave the choice of specific violation-handling mechanics to a higher-level component.

### 5.7 Negative Testing of Non-`noexcept` Functions

For non-`noexcept` functions (though see Section 6), a runtime-selectable violation handler (such as the one implemented above) allows for effective, practical *negative testing* of contract checks in a

fully well-defined manner. Negative testing is a tool to verify that contract checks have themselves been written to properly detect inputs outside of the domain of a function. The negative-testing algorithm consists of a few steps.

1. Execute negative tests only in a build mode where the contract checks under test will be evaluated and invoke the custom violation handler, such as `Eval_and_abort` mode on a platform that allows replacing the violation handler.
2. Prior to beginning a negative test, install a violation handler that will throw a known testing-only exception containing the violation details.
3. Inside a `try...catch` block, execute the function with inputs that are out of bounds.
4. The precondition checks on the function should detect the out-of-bounds input and lead to an immediate exception, unwinding the construction of the function arguments and being caught by the caller. Importantly, this method is the only one that doesn't leak when resource-allocating objects are passed by value.
5. The caller then checks that control flow passed through the `catch` and the function under test did not return normally. Returning normally from the function or throwing any other exception type is considered a test failure.
6. The caller also checks that the violation handler was invoked from a contract-checking annotation in an appropriate-seeming `location`. A violation in a different function called from within the function under test would indicate that the precondition was not properly checked.
7. Once all tests are run, restore the runtime-selectable violation handler to the value it had before the negative test began.

### 5.8 Counting Repeated Violations

Should the Contracts facility evolve to enable contracts having the *observe* semantic, it will quickly become possible for a single contract to be violated a significant number of times in a single program. Allowing violation handling to throw, and thus potentially return again to the same point of violation at a later point in execution, also opens up this possibility.

The overhead of producing a diagnostic for each violation, when attempting to simply observe the violations that are otherwise benign, can be too large to enable normal business operations. On the other hand, accidentally skipping diagnostics for all occurrences of a particular contract check failing might hide real program defects, throwing the proverbial baby out with the bath water.

To fix these issues, a violation handler can keep count of how many times a particular contract check has been violated and emit a diagnostic for only progressively fewer and fewer of them, usually with a strategy such as logging only when the count is a power of 2, an approach known as *exponential backoff*.

The hard part of performing this type of backoff hinges on identifying individual contract checks. The `source_location` provided on the `contract_violation` object is a rough approximation but is somewhat lacking.

- For contracts on a templated entity, the distinct instantiations of the template *might* be captured by the `function_name` property on the `source_location`, but many compilers choose not to keep the full text of the function name available to runtime violation handling, so this solution might be limited.
- Any particular precondition when violated from distinct call sites should arguably be considered a distinct defect, so counting is benefited by counting distinct caller and callee pairings separately.
- Multiple checks of the same precondition, such as when invoking the same function in a complex expression, can occur on the same line from within the same function, leading to `source_location` not uniquely identifying the defect.

Two potential fallouts arise when a defect is not uniquely identified.

1. Diagnosing and fixing the problem can be greatly hindered by needing to narrow down the specific control flow that led to the defect or, worse, not even realizing that the defect occurred on a path distinct from the one where a fix was applied.
2. When defects occur on multiple paths, exponential backoff logging can suppress some of them such that an improper conclusion is reached that no defect exists along the path where diagnostics were not emitted.

Macro-based contract-checking facilities often use a `static` local variable to track the count where the macro is placed, which solves some of these problems but, as with the rest of the macro-based facility's uses, does not allow for capturing the caller location on precondition checks. A contract-violation handler can use the `source_location` object's properties as a key in an associative container to achieve the same level of tracking as this proposal.

A future proposal that makes *observe* a usable contract-checking semantic in the language would be well served to also include the ability to, on a `contract_violation` object, expose a unique identifier for the contract-checking annotation that was violated. The type of this property should be something suitable to use as the key in an associative container while also facilitating easy generation of a unique identifier by the platform. Generating such a unique value can be done by taking the address of the *instructions* that execute the contract violation handling process. This would allow for distinct counting of separate inlined versions of the same function in different contexts, which are invariably distinct defects. To facilitate this uniqueness, the type of this property must thus be an integral type at least as large as `std::uintptr_t`.

Keeping an opaque identifier would leave the compiler *quality of implementation* (QoI) to determine, like many of the other properties of the `contract_violation` object, how accurate and useful the identifier is. Careful wording would be needed to at least guarantee uniqueness as effective as the `source_location` but leave more granularity up to the compiler.

### 5.9 Platform-Specific `contract_violation` Subclasses

Platforms might wish to expose vendor-specific extensions to the `contract_violation` object. Consider, for example, a scenario in which GCC decided to track the number of violations of a contract for the user to thus ease the implementation of exponential backoff logging.

GCC might provide that information by making its `contract_violation` object polymorphic (through choosing to have its destructor be `virtual`) and defining a GCC-specific subclass:

```
// in gcc/gcc_contract_violation.h:
namespace gcc {
class gcc_contract_violation : public contract_violation {
public:
  std::int32_t count() const noexcept;
};
```

A client wishing to access this information when available yet still have a cross-platform contract-violation handler might identify this subclass when it is present through the use of `dynamic_cast`:

```
void ::handle_contract_violation(
    const std::contracts::contract_violation& violation)
{
  if constexpr (std::is_polymorphic_v<std::contracts::contract_violation>) {
    if (auto* gcc_violation =
          dynamic_cast<const gcc::gcc_contract_violation*>(&violation)) {
      std::int32_t count = gcc_violation->count();/
      // Do stuff with count.
    }
    else {
      // This particular contract_violation was not generated with GCC extensions.
    }
  }
  else {
    // This platform does not support subclassing contract_violation.
  }
}
```

# 6 Throwing

Exception propagation from the evaluation of a contract-checking annotation is, unsurprisingly, in strong conflict with the use of `noexcept`. The Lakos Rule,[18] which proscribed the use of `noexcept` on functions with narrow contracts, i.e., those having one or more preconditions, was invented and applied to all of the C++11 Standard Library precisely to address this issue. Once such an exception escaping from a violation handler begins to propagate up the call chain, any intervening `noexcept` function that fails to catch the exception will force termination.

Allowing the `noexcept` property to either report an incorrect value (i.e., `true` for an expression that will actually throw) or vary across build modes would be a fundamental flaw in a Contracts facility for C++, opening the door to major problems being undiagnosable or even caused by the contract-checking facility.[19] Therefore, we recommend that contract checks on a function, either as preconditions or postconditions on the declaration or those evaluated by the function body or other functions it invokes, should all be subject to the normal rules of exception propagation and

[18] See [N3279].

`noexcept` function boundaries.[20]

We would break down our recommendations for exception-related decisions for contract-checking annotations as follows:

- Any exception emanating from the evaluation of a precondition or postcondition should be treated as though it originated within the function's body.
- An exception escaping from the evaluation of a contract-checking annotation's predicate should invoke the contract-violation handler (with the `detection_mode` of `evaluation_exception`). Users who wish to propagate this exception to the caller can implement a custom violation handler that does this (see Section 5.3).
- An exception may escape from the invocation of the contract-violation handler. Any such exception will perform stack unwinding with the same behavior as an exception that escaped any other function.

# 7 Wording Changes

The current MVP[21] does not contain suggested wording, somewhat by design. A previous paper, [P2388R4], contains Standard wording for an earlier iteration of the MVP, and the final wording for the MVP can be expected to evolve from that.

In [dcl.correct.test], introduced in [P2388R4], add a new paragraph after paragraph 2:

> The *contract-violation handler* of a program is a function of type "*opt* `[[noreturn]]` *opt* `noexcept` function of (lvalue reference to `const std::contracts::contract_violation`) returning `void`" named `::handle_contract_violation`. Whether the contract-violation handler is replaceable is implementation defined. (A C++ program may define a function with this name and signature and thereby displace the default version defined by the implementation.) [ *Note:* The definition of a contract-violation handler on an implementation where the contract-violation handler is not replaceable will result in multiple definitions of the contract-violation handler and thus be ill formed. — *end note* ]
>
> *Recommended practice:* The default contract-violation handler provided by the implementation should produce diagnostic output that suitably formats the most relevant contents of the `std::contracts::contract_violation` object and then return normally.

Note: The existing wording in [P2388R4] already references the contract-violation handler without further detail, specifying that it is invoked for an unsuccessful *enforced correctness annotation test*.

Add a new section, [support.contract], after section [support.coroutine]:

[19] See [P2834R0].

[20] Previous versions of this paper (see [P2811R2]) have explored this topic further. Here we have simplified to present only our recommended decision.

[21] See [P2521R3].

## Contract-violation handling [support.contract]

### Header <contract> synopsis [contract.syn]

The header <contract> defines a type for reporting information about contract violations generated by the implementation.

```
namespace std::contracts {
  enum class detection_mode : int;
  enum class contract_semantic : int;
  enum class contract_kind : int;
  class contract_violation;
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

### Enum class detection_mode [support.contract.violation.detection.mode]

Enum class detection_mode [tab:support.contract.violation.detection.mode]

| Name | Value | Meaning |
|---|---|---|
| predicate_false | 1 | Contract predicate returned false |
| evaluation_exception | 2 | Unhandled exception evaluating contract predicate |
| evaluation_undefined_behavior | 3 | Contract predicate would have undefined behavior when evaluated |

### Enum class contract_semantic [support.contract.semantic]

Enum class contract_semantic [tab:support.contract.semantic]

| Name | Value | Meaning |
|---|---|---|
| enforce | 1 | End program on violation |

### Enum class contract_kind [support.contract.kind]

Enum class contract_kind [tab:support.contract.kind]

| Name | Value | Meaning |
|---|---|---|
| pre | 1 | A [[pre]] contract annotation |
| post | 2 | A [[post]] contract annotation |
| assert | 3 | An [[assert]] contract annotation |

### Class contract_violation [support.contract.cviol]

```
namespace std::contracts {
  class contract_violation {
  public:
    @\seebelow@ ~contract_violation();
    contract_violation(const contract_violation&) = delete;
```

```
      // cannot be copied
    contract_violation& operator=(const contract_violation&) = delete;
      // cannot be copied

    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    contract_kind kind() const noexcept;
    source_location location() const noexcept;
    contract_semantic semantic() const noexcept;
  };
}
```

The class `contract_violation` describes information about a contract violation generated by the implementation. Whether `~contract_violation()` is virtual is implementation defined.

```
const char* comment() const noexcept;
```

*Returns:* Implementation-defined text describing the predicate of the violated contract.

```
detection_mode detection_mode() const noexcept;
```

*Returns:* The manner in which this contract violation was detected.

```
contract_kind kind() const noexcept;
```

*Returns:* The kind of contract annotation whose check detected this contract violation.

```
source_location location() const noexcept;
```

*Returns:* The implementation-defined source code location where this contract violation was detected.

```
contract_semantic semantic() const noexcept;
```

*Returns:* The runtime semantic chosen (at build time) for the contract annotation that has been violated.

### `invoke_default_contract_violation_handler` [support.contract.invdef]

```
void invoke_default_contract_violation_handler(const contract_violation&);
```

*Effects:* Equivalent to `::handle_contract_violation` if it is not replaced with a user-provided function (see [dcl.correct.test]).

# 8 Conclusion

With growing concerns over the MVP's severely limited ability to meet the needs of many existing C++ users,[22] SG21 will inevitably be compelled to consider various proposals to address each of those individual concerns. The well-proven approach of supporting a user-defined contract-violation handler has been shown to address these use cases clearly, effectively, and without the need for excessive core-language specification efforts. Although a contract-violation handler does not in and of itself solve all problems, the new information about the expectations of the MVP clearly indicates that we should reconsider this flexible solution, which could in turn immediately unleash real-world use of the language feature SG21 is striving to produce. Importantly, what is being proposed here strives to be backward compatible with many other highly anticipated and sorely needed features that can be quickly enabled to solve still more practical problems that reoccur in industry, especially at scale.

# Acknowledgements

Thanks to John Lakos, Bjarne Stroustrup, Tom Honermann, Andrzej Krzemieński, Ville Voutilainen, Gašper Ažman, Aaron Ballman, and Mungo Gill for feedback on the earlier revisions of this paper. Special thanks to Ville Voutilainen for producing [P2838R0] and Tom Honermann for [D2852R0], both of which have supported and evolved the ideas in this paper.

Lori Hughes made great contributions to the linguistic quality of this paper, as always. Any lingering failures to properly use the English language are the fault of the author.

# Bibliography

[bde14] "Basic Development Environment". Bloomberg
https://github.com/bloomberg/bde/

[D2852R0] Tom Honermann, "Contract violation handling semantics for the contracts MVP", 2023
http://wg21.link/D2852R0

[isoc11] *ISO/IEC 9899:2011 Information Technology — Programming Languages — C* (Geneva, Switzerland: International Standards Organization, 2011)
https://www.iso.org/standard/57853.html

[LWG2398] Stephan T. Lavavej, "type_info's destructor shouldn't be required to be virtual"
https://wg21.link/lwg2398

[N3279] A. Meredith and J. Lakos, "Conservative use of noexcept in the Library", 2011
http://wg21.link/N3279

[N4378] John Lakos, Nathan Myers, Alexei Zakharov, and Alexander Beels, "Language Support for Contract Assertions", 2015
http://wg21.link/N4378

[22] See [P2698R0].

[P0465R0] Lisa Lippincott, “Procedural Function Interfaces”, 2016
http://wg21.link/P0465R0

[P0542R5] J. Daniel Garcia, “Support for contract based programming in C++”, 2018
http://wg21.link/P0542R5

[P1332R0] Joshua Berne, Nathan Burgers, Hyman Rosen, and John Lakos, “Contract Checking in C++: A (long-term) Road Map”, 2018
http://wg21.link/P1332R0

[P1429R3] Joshua Berne and John Lakos, “Contracts That Work”, 2019
http://wg21.link/P1429R3

[P1607R1] Joshua Berne, Jeff Snyder, and Ryan McDougall, “Minimizing Contracts”, 2019
http://wg21.link/P1607R1

[P1639R0] Corentin Jabot, “Unifying source_location and contract_violation”, 2019
http://wg21.link/P1639R0

[P2388R4] Andrzej Krzemieński and Gašper Ažman, “Minimum Contract Support: either No_-eval or Eval_and_abort”, 2021
http://wg21.link/P2388R4

[P2521R3] Andrzej Krzemieński, Gašper Ažman, Joshua Berne, Bronek Kozicki, Ryan McDougall, and Caleb Sunstrum, “Contract support – Record of SG21 consensus”, 2023
http://wg21.link/P2521R3

[P2698R0] Bjarne Stroustrup, “Unconditional termination is a serious problem”, 2022
http://wg21.link/P2698R0

[P2751R0] Joshua Berne, “Evaluation of Checked Contracts”, 2023
http://wg21.link/P2751R0

[P2784R0] Andrzej Krzemieński, “Not halting the program after detected contract violation”, 2023
http://wg21.link/P2784R0

[P2811R2] Joshua Berne, “Contract Violation Handlers”, 2023
http://wg21.link/P2811R2

[P2834R0] Joshua Berne and John Lakos, “Semantic Stability Across Contract-Checking Build Modes”, 2023
http://wg21.link/P2834R0

[P2838R0] Ville Voutilainen, “Unconditional contract violation handling of any kind is a serious problem”, 2023
http://wg21.link/P2838R0