Contracts Interaction With Tooling

Document #: P3321R0
Date: 2024-07-12
Project: Programming Language C++
Audience: EWG (Evolution), SG15 (Tooling), SG21 (Contracts)
Reply-to: Joshua Berne <jberne4@bloomberg.net>

Abstract

The Contracts MVP ([P2900R7]) leaves up to implementations a number of aspects about how contract assertions behave. These choices do not impact the semantics of correct programs but provide significant flexibility in how users will get to build and deploy software that makes use of contract assertions. This paper will explore the ranges of possible implementation strategies, showcasing how they will enable powerful use cases while allowing working engineers to focus on writing portable and correct code. For each part of [P2900R7]'s specification that allows a fair bit of freedom, we will offer a potential SG21 recommendation of the appropriate baseline functionality that a healthy C++ ecosystem, including potentially multiple compilers and linkers, should support.

Contents

Revision History

Revision 0

- Original version of the paper

1 Introduction

The C++ Contracts MVP, [P2900R7], is sometimes criticized as leaving up to implementations too many important decisions about the behavior of contract assertions. In truth, however, two primary loci of implementation-defined behavior in the Contracts MVP are fundamental to our expectations of how the Contracts facility should behave.

1. The specifics of if, when, and how a contract assertion is evaluated are controlled by the implementation and thus the user.
   - Evaluations may be repeated.
   - Predicates may be elided when their results can be determined.
   - Each evaluation may happen with the *ignore*, *observe*, *enforce*, or *quick_enforce* semantic.
2. When a violation is detected by a contract assertion evaluated with the *enforce* or *observe* semantic, a user-defined contract-violation handler is invoked.
   - This handler is conditionally replaceable and has a default implementation provided by the platform.
   - The platform is responsible for creating and populating `contract_violation` objects as well as other parts of the semantic, such as the fashion in which a program is terminated when a contract assertion is *enforced*.

Having these behaviors be implementation defined achieves a variety of goals.

- Platforms have the freedom to choose from a number of potential implementation strategies and can still provide for well-formed and compliant programs even when mixing those implementation strategies.
- Implementations can provide as much or as little flexibility in configuring these behaviors as their customers want and need. This range includes implementations that choose to provide little or no flexibility to thus minimize the range of possible builds their customers might need to support.
- Strategies that require absolutely no ABI changes and that are thus compatible with the widest range of prebuilt binaries already built by existing or unrelated compilers are free to provide weaker guarantees than those that can provide improved user experiences by deploying more widespread ABI changes.

Altogether, the specification of [P2900R7] has been crafted, from the start, with the end goal of allowing the full range of expected implementation approaches to be conforming implementations.

2 Implementation Questions

The various groups of implementation-defined behaviors in [P2900R7] can be characterized by the questions an implementation must answer, generally through a combination of configuration options, documentation, and actual behavior.

2.1 Where are contract assertions evaluated?

The first question a compiler must address when implementing the behavior specified in [P2900R7] is where and when the compiler will generate the instructions that perform the contract-assertion evaluation. Many of these possibilities are also explored from a slightly different approach in [P3267R1].

For an assertion statement using `contract_assert`, the answer is fairly straightforward: The instructions will be emitted as part of the function body in which the assertion statement appears.

For function contract assertions (i.e., preconditions and postconditions), the answer is more involved because these assertions are evaluated on the boundary between the caller and the callee. A number of considerations impact where the code for a contract assertion will be generated.

- In some ABIs, the callee destroys function parameters, and thus postconditions *cannot* be evaluated by the caller since they will be using objects that are already past the end of their lifetimes.
- On any indirect function invocation (e.g., through a pointer or with virtual dispatch), the set of function contract assertions known to the caller and callee may be different. When doing virtual dispatch,[1] the caller-side contract assertions will be those of the function on the static type of the object through which virtual dispatch is occurring. For function pointers,[2] an empty set of function contract assertions will be on the caller side. In both cases, the callee-side list of contract assertions will be the contract assertions of the function definition being invoked — either the selected final overrider or the function denoted by the function pointer.

  In many cases, the caller cannot know, without ABI changes, the specific function being invoked through indirection, and similarly any generated function definition will be invocable in a variety of ways with different contract assertions applied by the caller. Therefore, for virtual functions, contract assertions from the static type will likely be generated at the call site while those of the final overrider will always be placed in the function definition. For invocation through a function pointer, no contract-assertion evaluations will be generated at the call site, and a version of the function that checks the contract assertions in the body will always be entered through indirection.
- Even when the caller and callee function contract assertions are different, they form a contract assertion sequence, and the assertions may be repeatedly evaluated. This design choice is

[1]The use of function contract assertions on virtual functions has achieved consensus in EWG with the design described in [P3097R0], which will be incorporated into [P2900R8].

[2]Attaching function contract assertions to function pointers is currently not possible. While introducing contract assertions on existing function pointer types might not be feasible, proposals such as [P3271R0] might reach maturity and be adopted to better satisfy the indirect invocation use case.

intentional to allow platforms to choose to emit evaluations of all the contract assertions at the call site (in the invoking translation unit, with the invoking translation unit's configuration) or in the function body with the configuration that governed how that function body was compiled.

In practice, most final overrider bodies will now know which specific virtual function was used to invoke them, and most call sites doing indirect invocation will not know the specific function definition to which they are passing control. For this reason, caller-side contract assertions will often be evaluated in the caller's translation unit, and callee-side contract assertions will be evaluated in the function body.

- When we have no viable way to check the contract assertions that must be evaluated, a platform must document that, in such a situation, the contract assertions will be evaluated with the *ignore* semantic. This scenario might occur, for example, for caller-side postcondition assertions when doing virtual dispatch if the platform ABI does not allow for caller-side checking of postconditions.
- A function contract assertion that is emitted in the caller may do so at any and all call sites for that function. If a chosen location for the generated code (which is always within some function) is within an inline function, then the code for that contract assertion will potentially be generated in many different translations of that same inline function. In both cases, the same function contract assertion may have differently generated code depending on how it is configured, but that distinction will be based on how the selection of semantics gets optimized differently in different translation units, which we will discussion in Section 2.2.
- Choosing different entry points for indirect function invocation can result in invoking, in the callee, the evaluation of the caller-side contract assertions. Changing the exact function invoked for different types of invocation, however, will often require an ABI change that most platforms will be reluctant to make. See [P3267R1] for some discussion of different implementation strategies that might be achieved with different amounts of changes to the ABI.

Considering all these points, we can offer SG15 the following recommendations regarding when contract assertions are evaluated.

- The caller-side checks for a function invocation should be generated in each caller. The callee-side checks for a function should be generated in the caller. When these are the same (such as for any direct invocation), the caller-side checks should default to evaluating with the *ignore* semantic (and thus generate no code).
- For virtual dispatch, where the caller and callee side checks might differ, emit evaluations with the configured semantics at both the call site and in the function body.
- Provide a configuration parameter to switch the choice of callee-side checking for direction function invocations to caller-side checking, essentially changing the semantic for callee-side checks to *ignore* and applying the configured semantics to the caller-side evaluations.

2.2 How are semantics chosen for the evaluation of contract assertions?

For any contract-assertion evaluation, a wide range of possible inputs will determine the specifics of how that evaluation will take place:

- The configuration of the translation unit (TU) where the contract assertion is being emitted
  - For inline functions, the applicable configuration may be the configuration of the TU into which the function is being emitted, or it may be the configuration from a different TU that translated the same function as a weakly linked, non-inlined function.
- For declarations attached to a module, the configuration of the module translation unit where the function is defined
- Configurations passed to the linker can control the eventual runtime semantic of contract assertions in two concrete ways.
  1. Any earlier choices might defer the selection of contract semantic to the link-time configuration.
  2. Inline functions can be generated with additional information to allow them to be distinguished based on contract-assertion configuration, and a linker might be configured to give preference to certain configurations instead of picking a generated inline function in a fundamentally random fashion.

In spirit, we can recall from [P2900R7] that the process of evaluating an assertion conceptually begins by determining the contract semantic with which that evaluation will take place:

```
evaluation_semantic _semantic = __current_semantic();
if (evaluation_semantic::ignore == _semantic) {
  // Do nothing.
}
else if (evaluation_semantic::observe == _semantic
      || evaluation_semantic::enforce == _semantic
      || evaluation_semantic::quick_enforce == _semantic)
{
  // Evaluate with checking semantic.
  // ...
}
```

The behavior of this intrinsic, `__current_semantic()`, is arguably the largest and most important piece of implementation-defined behavior in [P2900R7]. Conceptually, one can think of this intrinsic as applying an algorithm to inspect the appropriate compiler configuration and to determine a contract semantic for this evaluation.

- Thanks to the C++ model of compiling translation units into generated code, the first decision-making factor is always the configuration of the TU that compiled the generated code.
- For every function invocation, the function contract assertions on that function can be generated in two different places: in the function definition of the caller or in the definition of the function itself. Since these two places might be compiled in different TUs, they might be compiled with different configurations, but in the end, each individual contract-assertion evaluation will be within code generated for a specific function when translating a specific TU.
- When compiling an inline function, the same function may be compiled by many different translation units. With current linker technology, two possible scenarios result.

1. Where the applicable function is actually inlined, the configuration of the TU that translated the containing function will be applicable.
2. For all cases within a program where the applicable function is *not* inlined, a single copy of the generated function will be selected, and the configuration of the TU that translated that copy will be applicable.

Depending on what functionality the platform is seeking to provide, this configuration may include a number of different things.

- A single choice of concrete semantic — *ignore*, *observe*, *quick_enforce*, or *enforce* — can be configured, producing a result for the algorithm as soon as the semantic is found:

  ```
  gcc -std=C++26 -fcontract-semantics=enforce
  ```

- Other properties of source code — expressed when building it — might impact which semantic is appropriate for individual contract assertions in that context.
  - In *trusted* code that has no bugs, we can often safely *ignore* the contract assertions that follow the code. Therefore, if a translation unit contains functions that are trusted sufficiently (often due to the existence of thorough testing), postconditions of that code and preconditions of functions to which that code delegates may be ignored. Conversely, when a translation unit is *suspect*, those same contract assertions should be checked by the builds running the suspect code, and the *enforce*, *quick_enforce*, or *observe* semantic are the appropriate semantics to apply.
  - With *sensitive* code and where the risk of running outside of contract is exceptionally high, we will want to check contract assertions wherever control flow might cross a boundary into the sensitive code. In this case, preconditions of sensitive functions and postconditions of functions to which they delegate will benefit from preferring a checking semantic.
  - *Suspect* contract assertions — those that are newly introduced or those in translation units that never previously had contract assertions checked — benefit greatly from preferring the *observe* semantic over the *ignore* semantic.

  All the above flags might be part of a translation unit's configuration and could result in different semantics based on the properties a user has specified for their contract assertions.

- An implementation could allow for a configuration file, perhaps encoded as `json` or `yaml`, that allows a user to select contract semantics based on file location, contract kind, or even expression patterns:

  ```
  [
    { path : "my_file.cpp",  line: 17, semantic : "ignore" },
    { kind : "post",                   semantic : "ignore" },
    { expression : ".*",               semantic : "enforce" }
  ]
  ```

- For functions attached to a module, an implementation could defer to the configuration with which the module was compiled for any function not defined in the module of the applicable translation unit. This approach would have the advantage of always, in practice, using the same configuration for a contract assertion regardless of how individual translation units were compiled — a property that some environments will prefer.

- The configuration can indicate, either for all contract assertions or for specifically selected assertions, that a choice of configuration should be delayed until link time or run time.

Note that, until this point, the algorithm above can be evaluated during the translation of a function, i.e., at compile time. If this evaluation results in a choice of evaluation semantic, then this seemingly complex runtime algorithm reduces to no need to jump based on the semantic or any unneeded runtime overhead.

- Conceptually, as far as the abstract machine is concerned, the same algorithm applies in these cases as would apply if the decision on contract semantic were deferred until link time or run time.
- If the chosen semantic is *ignore*, the entire contract assertion will result in no generated code, and no optimizations should be hindered by the presence of the contract assertion.
- If the chosen semantic is *enforce* or *quick_enforce*, any code that follows the contract assertion might be optimized with knowledge that the predicate evaluated to `true`.
- If the chosen semantic is *observe*, the code following the contract assertion might be duplicated onto both branches to thus have one version optimized based on the contract predicate's truth and another without such optimizations should the contract-violation handler return normally.

While conceptually an algorithm is always performed that *might* result in deferring the decision until run time, that algorithm never has to result in generating a function call to determine the semantic unless the user has actually built a translation unit in a mode in which that functionality is requested.

In practice, most default compilation strategies will stop here, never needing to consider any more-complicated code generation that might ensue from supporting link-time or runtime selection of contract semantic. Just like with a macro-based facility, when contract assertions are *ignored*, they will have no overhead, and when enforced, they will have the overhead of predicate evaluation but will provide improved code generation following the assertion. Nothing in the specification of [P2900R7] requires that a platform even support configuration options that are not completely determined during the compilation of a translation unit.

On the other hand, flexibility for controlling semantics at fairly arbitrary granularities at compile time requires that code must always be written to work correctly regardless of which evaluation semantic will be used to evaluate any individual contract assertion. Thus, correct code can also, without needing to be rewritten, take advantage of builds in which the evaluation semantic is *not* determined at compile time; this choice of evaluation semantic is in the hands of those who assemble a program, not those who write each individual line of code that goes into the program.

When the semantic is *not* determined at compile time, the compiler must instead emit the full switch statement and an appropriate function invocation to determine which branch of the switch statement to take.

To determine the contract semantic at link time, a linker script might be provided that will do so based on linker configuration similar to the compiler configuration that determined the configuration for a translation unit.

- The linker configuration obtained by a linker script might be retrieved from flags passed to the linker, such as through the `-Xlinker` arguments of `gcc`.
- The linker configuration might, alternatively, be retrieved from environment variables at link time; many different such conventions already exist for controlling link-time behaviors.
- When the linker configuration results in a semantic being chosen, this choice can be stored for run time in any of a number of ways. With suitably structured generated code, all function invocations and conditionals can be replaced by a single inserted unconditional jump instruction — in practice, to the branch of the contract-assertion evaluation `switch` statement for that chosen semantic. With properly organized code generation, this approach could even reduce the runtime cost of an *ignored* contract assertion to a single `no-op` instruction, assuming the compiler is aware enough to place the branch for *ignore* directly inline with the switch statement and generate the code for other semantics in separate blocks that jump back when finished.

Finally, if either of the previous steps conclude that the contract-evaluation semantic should be chosen at run time, the function invocation to determine that semantic and then the branching on the result can be left in the generated code. The runtime semantic may then be chosen with a wide variety of different algorithms.

- The semantic to use might be stored in a central location that can be modified using a platform-provided API, allowing for dynamic switching of the contract semantic based on user-decided parameters, such as time of day, system load, or whether active debugging of a suspect process is needed.
- The semantic might be randomly chosen, with a user-defined distribution, between *ignore* and *enforce* or between *ignore* and *observe*. This approach would allow for heuristic contract-assertion checking while balancing the cost of checking with the safety provided.

How much or little of the above algorithm is provided by a platform is not specified in the Standard and is intentionally left up to platforms to design and provide based on the needs of their users.

To allow for repetitions of contract assertions, the same configuration steps can be used to specify the number of repetitions that the code generated for an individual contract-assertion evaluation will perform.

Again, SG21 should recommend that platforms provide a certain baseline of functionality.

- Compilers should support a standard set of flags that allow configuration of contract assertions at both the per-TU and per-contract assertion level.
    - Translation unit configuration should include, at a minimum, a specification regarding explicit evaluation semantics but should be extensible to more meaningful configuration options, such as *sensitive* or *untrusted* code.

  - Configuration of individual contract assertions should be performed through a commonly defined external file format that allows for setting the same parameters that can be set globally. In practice, this form of configuration can be essential when, for example, a user must work around a compiler bug that occurs if a particular contract assertion is evaluated with a particular evaluation semantic.

- Linkers should be updated to support configuration that selects inline function definitions based on how their translation units were configured. When nothing is explicitly specified to a linker, the linker should select the definition with the strictest checking to thereby avoid users asking for enforcing in a particular translation unit and ending up *not* enforcing contract assertions simply because their functions did not get inlined and the linker chose a more lenient definition.

  Configurations to instruct linkers to select the semantics with the lowest runtime overhead should also be provided since some builds will have a strong preference for avoiding the introduction of overhead into code that is on the hot path of an application.

- Link-time selection of semantics should be an available option to enable distribution of a single binary library that can be used with multiple different contract configurations.

- Compile-time and link-time configurations should be consistent because all the considerations that apply to compile-time configurations might apply equally when integrating a binary delivered by a third party.

- Runtime selection of semantics, when chosen, should go through some form of user-configurable function, but that can easily be compiler specific.

2.3 How does the contract-violation handler get invoked?

To invoke the contract-violation handler during the evaluation of a contract assertion, the generated code needs to collect certain information to populate the `std::contracts::contract_violation` object:

- The `source_location` and `comment` that will be used to identify the violation contract assertion
- The semantic, kind, and detection mode that led to the contract-violation handler being invoked

Inevitably, the shared ABI of a platform — such as the Itanium[3] or MSVC ABIs — will need to provide functions that can be referenced and that take the above information and, internally, construct a `contract_violation` object, find the contract-violation handler, and invoke it with the constructed object.

Consider, for example, how the Itanium ABI might specify the shared sets of functions for handling contract violations. These functions will be used from two ends.

1. At each location where a contract assertion is evaluated, one (or more) invocations of functions will be present and will end up calling the contract-violation handler.

[3]See [itanium].

2. Within the definition of the contract-violation handler, all member functions of `std::contracts::contract_violation` will need to access data produced by the functions above.

Our suggestion would be to provide, at the ABI level, functions that insulate both the above processes from the specific layouts and versions provided by the linked-in runtime support library. Note that future revisions might add more semantics or more properties to `contract_violation`, and clients and violation handlers built with both new and old Standards must be able to link together through the common ABI.

2.3.1 Potential `contract_violation` and handler ABI

For the contract-violation handler and explicitly for the implementation any compiler provides for `contract_violation`, the ABI should provide a data type and accessors to extract information from that datatype. In a fashion similar to the exception-handling ABI in Itanium,[4] we will assume that such functions are provided with a prefix of `_Contract`, behave as if defined `extern "C"`, and have names that are never mangled.

- As a common vocabulary for all these functions, the `_Contract_Violation` datatype will be used as an opaque type referring to a system-provided data structure that is always created and destroyed by the system.
  - This object should include sufficient information to construct the `source_location` that will be made available in the `contract_violation` object passed to the contract-violation handler.
  - The contents of this struct should include a dynamically sized tail that can be expanded with arbitrary additional information that might be used in future extensions, such as specifying labels or custom messages that might have been specified on the contract assertion that was violated.
- ABI-specific enumerations for the various return values needed will be provided and will include both the standard-defined values and any additional values agreed upon as extensions provided by the ABI:

```
typedef enum {
  _CAK_PRE = 1,
  _CAK_POST = 2,
  _CAK_ASSERT = 3
} _Contract_Assertion_Kind;

typedef enum {
  _CES_ENFORCE = 1,
  _CES_OBSERVE = 2
} _Contract_Evaluation_Semantic;

typedef enum {
  _CDM_PREDICATE_FALSE = 1,
  _CDM_EVALUATION_EXCEPTION = 2
```

[4]See Itanium Exception Handling ABI.

```
    } _Contract_Detection_Mode;
```

- The platform runtime library should provide functions to access the state of a `_Contract_Violation` object:

  ```
  const char *_Contract_Get_Comment(const struct _Contract_Violation *violation);

  _Contract_Detection_Mode _Contract_Get_Detection_Mode
                                   (const struct _Contract_Violation *violation);

  _Contract_Assertion_Kind _Contract_Get_Assertion_Kind
                                   (const struct _Contract_Violation *violation);

  _Contract_Evaluation_Semantic _Contract_Get_Evaluation_Semantic
                                   (const struct _Contract_Violation *violation);
  ```

- Accessing the source location from a `_Contract_Violation` object will need special treatment that depends on whether the ABI specification itself includes a specification for `std::source_location` or not. If it does, a single function that populates a `source_location` object could be provided. If it does not, individual functions that return the data members needed to populate a `source_location` — line, column, file name, and function name — would need to provided instead.

Given the above properties, each Standard Library could implement its own version of `std::contracts::contract_violation` that simply wrapped a `const _Contract_Violation*`. Any new functions added in future revisions to the above lists could have defaulted implementations output as weak symbols by the Standard Library to thereby allow linking against older runtime support libraries that do not provide that information. These defaulted implementations would produce values that indicate to the violation handler that the data being requested is unavailable.

The next part of the ABI that needs to be agreed upon is the entry point that will be used by the runtime support library for the contract-violation handler itself. For this purpose, instead of hooking directly into `::handle_contract_violation`, the runtime support library should invoke a different function, which is provided with weak linkage by the runtime library:

```
void _Contract_Handle_Contract_Violation
                                   (const struct _Contract_Violation *violation);
```

When a C++ program compiles a contract-violation handler named `::handle_contract_violation`, an implementation of `_Contract_Handle_Contract_Violation` will be emitted that creates that Standard Library's version of `std::contracts::contract_violation` from the provided pointer and delegates to the C++ contract-violation handler. Only one translation unit within a program should be able to do this, so that function will not have weak linkage.

Note that all the above properties also allow for non-C++ violation handlers to be installed as well, and these handlers will be able to access all the same data that is provided by the contract-violation detection process and that is available to the C++ contract-violation handler.

Finally, for platforms that do not intend to support the interoperation of multiple vendors (such as an embedded platform), the above shared ABI and runtime support library is not necessary needed.

While minimizing the code size for each contract-assertion evaluation is still of great benefit, the insulation of the contract-violation handler from the runtime support library might not be needed on such platforms.

2.3.2 Replaceability of the contract-violation handler

Whether the contract-violation handler is replaceable is implementation defined. A platform must, therefore, make one of two choices and inform users of that choice.

1. When a function with the appropriate name (`::handle_contract_violation`) and linkage is linked into a program, that function will replace the default contract-violation handler provided by the platform.
2. The platform-provided contract-violation handler will *always* be the contract-violation handler, and a function with the name `::handle_contract_violation` will likely be ignored.

A platform that selects the second choice provides the most flexibility to users in terms of how they wish to mitigate violations in their environments. The second choice is suited for platforms that target very restricted environments or that seek to make highly restrictive guarantees on how the programs they build will behave.

A platform that does not provide for replaceability should, ideally, produce a warning or error if a user does attempt to build a translation unit containing what would otherwise be a valid replacement for the contract violation handler.

2.3.3 Potential violation-detection ABI

Contract-violation detection is the other side of the process for which an ABI needs to provide a specification. Importantly, the generated code for every single potentially violated contract-assertion evaluation will be invoking this process on at least one branch and possibly more if the predicate is one that might throw. Because we expect to support large amounts of contract-checking, the ABI must minimize the required overhead of contract detection.

To minimize the object-size overhead of each individual contract assertion, the ABI can provide functions that encapsulate as much information as possible into the function chosen. In particular, the various enumerations that will populate the `_Contract_Violation` data structure can all be determined from control flow and can thus be embedded in the choice of function. Given three values of `_Contract_Assertion_Kind`, two values of `_Contract_Evaluation_Semantic`, and two values of `_Contract_Detection_Mode`, we need $3 * 2 * 2 = 12$ functions:

```
[[noreturn]] void _Contract_Handle_Violation_Pre_Enforced(...)
[[noreturn]] void _Contract_Handle_Violation_Post_Enforced(...)
[[noreturn]] void _Contract_Handle_Violation_Assert_Enforced(...)
void _Contract_Handle_Violation_Pre_Observed(...)
void _Contract_Handle_Violation_Post_Observed(...)
void _Contract_Handle_Violation_Assert_Observed(...)
[[noreturn]] void _Contract_Handle_Violation_Exception_Pre_Enforced(...)
[[noreturn]] void _Contract_Handle_Violation_Exception_Post_Enforced(...)
[[noreturn]] void _Contract_Handle_Violation_Exception_Assert_Enforced(...)
_Contract_Handle_Violation_Exception_Pre_Observed(...)
```

```
_Contract_Handle_Violation_Exception_Post_Observed(...)
_Contract_Handle_Violation_Exception_Assert_Observed(...)
```

For function parameters, a few possible options could be explored for an ABI.

- The source location and comment could be encoded in a separate table in the object file and thus looked up when needed instead of being embedded at the call sight. This approach would also minimize the complexity of removing this information for security-conscious users who do not want to allow source information to be included with their distributed binaries.
- This information could be stored using the same mechanisms used on a platform to store stack-trace information.
- If the ABI provides a suitable specification for the contents of `std::source_location`, that structure could be one parameter alongside a `const char*` comment parameter.
- Finally, the comment and all the data members of `std::source_location` could be individual parameters of these functions. This approach, however, would have the highest overhead in terms of the number of instructions emitted when generating the code for a contract-assertion evaluation.

A single entry point that took all the various properties as function parameters could also be provided, though this option has the highest overhead in terms of the number of instructions needed to perform the function invocation.

2.3.4 Recommended baseline

Overall, for shared ABIs (such as the Itanium ABI), the Committee, through SG15, should recommend supporting a replaceable contract-violation handler and amend the shared ABI to include the adoption of extensions similar to what we described above.

Any large-scale shared ABI should also mandate support for users providing their own contract-violation handler since that flexibility is, for many users, a central component to benefiting from the use of contract assertions.

2.4 What strings get put in a contract-violation object?

The specific contents of a number of properties of the `contract_violation` object are not explicitly specified in [P2900R7] but are implied via a recommended practice.

- One purpose of this flexibility is to allow implementations to remain conforming even when building with options where all debug information — such as file names, function names, and line numbers — is stripped from the delivered executable.

  As described in [P2947R0], this ability to exclude sensitive information yet invoke the contract-violation handler is essential for certain use cases.
- Another goal of this flexibility to exclude filename, function, and comment is to avoid the potentially great cost in executable size if large filenames and strings must be included in the executable. These strings might, therefore, be truncated or summarized in arbitrary ways to retain the essential information while reducing the bytes wasted to store that information.

- Flexibility in the values that populate the `contract_violation` object also allows for providing the source location of the caller instead of the callee on precondition violations, an essential feature for many who want that location to navigate users as swiftly as possible to the location closest to where a bug exists in the source code, thereby enhancing user efficiency.

For this aspect of implementation-defined behavior, SG15 should recommend that the above use cases be satisfied on demand.

- Compilers should provide a single flag to request stripping from an executable all unnecessary and potentially sensitive information, including any information that would otherwise populate the `location` or `comment` properties of a `contact_violation` object.
- In other cases and whenever reasonably simple to do so, the location where the contract-assertion evaluation is generated should be used for `location` to thus capture call-site locations, and the start of the function body should be captured in other situations.

2.5 What should the default contract-violation handler do?

In [P2900R7], the behavior of the implementation-provided default contract-violation handler is described as follows:

> *Recommended practice*: The default contract-violation handler should produce diagnostic output that suitably formats the most relevant contents of the `std::contracts::contract_violation` object, rate-limited for potentially repeated violations of observed contract assertions, and then return normally.

Just as with `assert`, the specific format of the output string is left up to the implementation to optimize for its users, so the format can be as bare-bones or attractive as needed. A robust default contract-violation handler should, however, consider a few additional points.

- The code path taken on an enforced contract violation should be as direct and robust as possible, avoiding the use of any facilities such as allocation or synchronization that might not be stable in a program already in a corrupted state.
- For observed contract violations, excessive logging, in the presence of an incorrectly introduced contract assertion, might bring down a system that would otherwise be healthy. Therefore, violations of the same contract assertion need not be logged on every single occurrence. Conversely, if one assertion is being violated many times, logging of other observed violations should not be dropped completely.

  Thus, we recommend that a concurrent associative data structure be used to track the number of contract violations occurring keyed by the contents of the source location provided to the violation handler. Note that, because the `contract_violation` object's contents should be considered ephemeral, copies of the strings and data used for this key should be kept internally. Based on these counts, a log message should be generated only when they reach new powers of 2, preferably with an indicator relaying how many violations were *not* logged when this happens.
- When the `detection_mode` is `evaluation_exception` (indicating that the evaluation threw an exception), information from that exception should be included in the diagnostic message

output. For example, the exception can be rethrown locally to determine if it is a subclass of `std::exception` and, if so, the value of `what()` can be included in the diagnostic output.

Overall, the contract-violation handler should be written to work as correctly as possible even when running in a corrupted member space, minimizing all use of blocking, allocation, and higher-level APIs. For example, any data structure used to track the counts of observed contract violation should preallocate storage of a fixed size.

Note that one other consideration might be worth taking into account: the possibility of a contract violation occurring during a signal handler. For any contract violation handler to be signal safe, it would have to refrain from any output, a constraint that is not viable for a default implementation. On the other hand, being as signal safe as possible for enforced contract violations is a worthy design consideration for any parts of the contract-violation handler that *can* be signal safe.

2.6 In what fashion is program termination accomplished?

Also implementation defined in the Contracts MVP is the specific manner in which program termination happens in two scenarios:

1. When the evaluation of a contract predicate throws an exception or evaluates to `false` during the evaluation of a contract assertion having the *quick_enforce* semantic
2. When the contract-violation handler returns normally during the evaluation of a contract assertion with the *enforce* semantic

In both cases, the specific mechanism of program termination is implementation defined.

In general, we should expect that mechanism to either invoke `std::abort` or to be the equivalent of doing so.

- The specification has deliberately avoided making that statement because implementation providers have expressed serious concerns about a requirement to emit a call from the core language to the actual Standard Library function `std::abort`.
- For cases related to the *quick_enforce* semantic, potentially better mechanisms are available for program termination that will require significantly less additional object code — perhaps as little as a single instruction — than a function invocation.

  In particular, a mechanism like that used by `libc++` in which a compiler built-in that traps, such as `__builtin_verbose_trap`, can be used to accomplish two goals: recording diagnostic information that will be available in a debugger and terminating the program using a single trap instruction.
- Again, with the *quick_enforce* semantic and when a contract-assertion predicate is potentially throwing, the exception-handling mechanism on some platforms can possibly be instructed to invoke `std::terminate` if an exception is thrown by simply setting a single bit or by not providing a handler for a specific stack frame.

2.7 When can predicates be elided?

Predicate elision is largely a question of quality of implementation and is never required by the specification in [P2900R7].

The simplest case to detect is that in which a contract predicate's definition, including any functions it might invoke, is visible to the compiler and can be determined (at compile time) to have no side effects other than possibly modifying trivially copyable objects:

```
int y;
void f(int x)
  pre ( ++ const_cast<int&>(x) > ++y );
```

In such cases, a compiler can make copies of all the trivially copyable objects and evaluate the predicate against those copies, using the result as needed.

Predicate elision is more interesting for future possible hardware and evaluation contexts for which alternatives such as evaluating a predicate in a transactional state and then unrolling that transaction but using the result can be explored.

3 Conclusion

From afar, a casual observer might think the C++ Contracts MVP specification [P2900R7] does not support writing fully portable code due to the admittedly large degree of implementation-defined behavior upon which the specification relies. In this paper, we have listed all the implementation-defined behavior, explained exactly what freedoms it allows, and demonstrated that the *only* allowed difference between one implementation and the next is the degree of granularity and flexibility with which individual contract assertions can be controlled — nothing more.

The limited flexibility afforded to vendors is carefully designed to allow them to provide distinct physical trade-offs while remaining entirely compliant with the specified requirements on which all users rely. Programmers can safely write software that will work on all conforming implementations, and vendors can start with simple implementations and evolve to support more robust needs as they arise, all of which will be enabled, *not hindered*, by the specification of Contracts in the Standard.

Acknowledgments

Thanks to John Lakos, Timur Doumler, Daniel Ruoso, and Eric Fiselier for feedback on this paper. More thanks to Lori Hughes for helping us keep this paper readable and clear.

Bibliography

[itanium] "Itanium C++ ABI". Github
https://itanium-cxx-abi.github.io/cxx-abi/

[P2900R7] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, "Contracts for C++", 2024
http://wg21.link/P2900R7

[P2900R8] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, “Contracts for C++”, 2024
http://wg21.link/P2900R8

[P2947R0] Andrei Zissu, Ran Regev, Gal Zaban, and Inbal Levi, “Contracts must avoid disclosing sensitive information”, 2023
http://wg21.link/P2947R0

[P3097R0] Timur Doumler, Joshua Berne, and Gašper Ažman, “Contracts for C++: Support for virtual functions”, 2024
http://wg21.link/P3097R0

[P3267R1] Peter Bindels and Tom Honermann, “Approaches to C++ Contracts”, 2024
http://wg21.link/P3267R1

[P3271R0] Lisa Lippincott, “Function Usage Types (Contracts for Function Pointers)”, 2024
http://wg21.link/P3271R0