Contract Build Modes, Semantics, and Implementation Strategies

Document #:	P2877R0
Date:	2023-06-13
Project:	Programming Language C++
Audience:	SG21 (Contracts)
Reply-to:	Joshua Berne <jberne4@bloomberg.net>
	Tom Honermann <tom@honermann.net>

Abstract

To see Contracts become a fruitful addition to the C++ ecosystem, SG21 must consider the various use cases in which Contracts have historically been deployed and will be in the future. The Contracts facility we include in C++ must be suitably loosely defined to enable implementation strategies that properly consider and support these uses; otherwise, the Contracts facility will result in nothing more than an educational toy. To allow the necessary level of experimentation and to open possible paths for evolution, we propose significant relaxations to the semantics the evaluation of a contract-checking annotation might have. Along with that proposal, we describe how various implementation strategies can achieve different goals and tradeoffs once given the freedom we envision.

Contents

Revision History

Revision 0 (Prior to 2023-06 Varna Meeting)

- Original version of the paper for discussion during an SG21 face-to-face meeting

1 Introduction

The Contracts MVP[1] is highly restrictive in terms of the possible semantics the evaluation of a contract-checking annotation (CCA) may have.

- Exactly two build modes, `No_eval` and `Eval_and_abort`, are provided.
- The mixing of build modes across translation units (TUs) is conditionally supported with implementation-defined semantics.
- In the `No_eval` build mode, all CCAs have the *ignore* semantic, i.e., the predicate of the CCA is not evaluated, no violations will be detected, and no other program semantics are altered.
- In the `Eval_and_abort` build mode, all CCAs have the *enforce* semantic, i.e., the result of the CCA's predicate is determined and, if it is not `true`, the contract-violation handling process occurs followed by program termination.

As specified, nonobvious limitations are put on C++ implementations. Properties of the only two build modes provided by the Standard will be quickly assumed to be properties of Contracts in C++, severely limiting the ability to contradict those assumptions with either platform-specific build modes or future evolution. These properties, importantly, include that all contracts in a TU have the same semantic and that all evaluations of a specific contract in a program will have the same semantic.

We propose updating SG21's MVP such that, instead of having two bespoke build modes combined with the hope for future evolution and platform-specific extensions, we take the more flexible approach to specification and keep implementation freedom maximized. This can be accomplished by simply stating that, on each evaluation of a CCA, the semantic with which the CCA is evaluated is implementation defined.

By providing this level of freedom to implementation, we can begin to see useful experimentation with contract semantics to help solve fundamental deployment and usage needs that any C++ feature must be able to meet.

Implementations remain free to follow the current approach of permitting only two build modes and requiring that all translation units be compiled with matched build modes. Compilers may also, to meet the needs of different audiences, provide build options that enable link-time or even runtime selection of contract semantics. We will explore various implementation strategies that achieve these goals in Section 5 and will discuss how these choices might even live side by side in the same program.

Finally, the set of possible semantics available for implementations to choose should include *ignore* and *enforce* to match the currently proposed functionality of the MVP. In addition, the *observe*

[1]See [P2521R3].

semantic should be available given the long-standing industry experience with it and the range of needed implementation and deployment choices that it enables.

2 Use Cases

Numerous situations must be considered regarding how Contracts will be integrated into the C++ ecosystem once they are an available part of the language. An overly minimal MVP that rejects the needs of these use cases will actively inhibit the deployment and experimentation that should be possible to see Contracts evolve into a facility that truly meets the needs of modern C++ developers and our ever increasing safety- and correctness-conscious global environment.

2.1 Package Managers

Linux and UNIX systems have long used package managers like Red Hat's RPM and Debian's `dpkg` to provide prebuilt software. More recently, package managers like VCPkg and Conan have started offering similar functionality for multiple operating systems, including Windows. These package managers generally provide a single build of each version of a project; e.g., they do not tend to offer both debug and release builds because doing so would introduce considerable additional build and distribution cost as well as dependency management complexity. A Contracts design that requires build-time decisions regarding whether contracts are evaluated and what the consequences of contract violation are creates a significant burden for package managers. They would have to choose whether to consistently build their packages for a specific build mode or whether to evaluate each package independently. These package providers are not typically the authors or maintainers of the source code that they build, but their decisions could impact project authors and the adoption of contracts within the ecosystem. If they were to choose to unilaterally enable contract evaluation with enforce semantics, that decision could disincentivize project authors from using Contracts for fear of performance overhead and support burden. Likewise, if package providers were to choose to unilaterally disable contract evaluation, that could disincentivize program authors from using Contracts since they wouldn't be active in deployment anyway. Most likely, package managers would defer contract enablement to the *release* build mode for each project with the result that contract enablement within the ecosystem would be inconsistent, a situation that would be fatal to mixing libraries if the platform does not support at least objects built with inconsistent build modes linking together. None of these outcomes would bode well for the ubiquitous use and success of the Contracts facility within the ecosystem.

2.2 Packaged Software

Providers of packaged software frequently distribute and provide support for prebuilt software. Historically, some providers have chosen to distribute packages with assertion facilities (of some form) enabled, and others have elected to disable their assertion facilities in packages distributed to their users on the basis that testing (with assertions enabled) has enabled sufficient confidence that the overhead associated with assertions is unnecessary to achieve their quality and support goals. C++ contracts *must* continue to provide these choices: A design that does not allow contract annotations to be unevaluated would not be viable; this is not controversial.

Traditional implementations of the C `<assert.h>` and C++ `<cassert>` facilities have required that asserts be enabled or disabled at compile time on a per-TU basis. This creates an incentive against distributing prebuilt software that has assertions enabled because an incorrect assertion (presumably one that was not adequately tested or that was added without full knowledge of valid use cases) could create a user support incident by triggering a runtime abort. When such incidents occur, the only options available to the software provider are to provide a new build of the software that either corrects or avoids the issue or to identify a workaround that avoids the issue without requiring a rebuild and that is acceptable to the user (e.g., a configuration or workflow change). The first option is often slow and expensive due to the time required to reproduce the issue, diagnose the root cause, identify an appropriate source-code change, rebuild, validate the fix, recertify, repackage, and redistribute the new build, all of which might occur under significant pressure from users desperate for a solution. Such updates can also be expensive or risky for users, particularly for consumers of safety-critical software. The second option, identification of a workaround, is almost always preferred, assuming an acceptable solution is identified. Unfortunately, suitable workarounds are not always possible.

Assertion facilities exist that do not require that a binary choice to enable or disable assertions be made at compile-time. For example, Coverity implements a conditional assertion facility that is enabled in the builds distributed to its users but that also allows each individual conditional assertion statement to be disabled at runtime by setting an environment variable with a value that is keyed to the source location of the statement. When a conditional assertion fails, a suitable message is produced that includes the source location information needed to disable the conditional assertion statement. This approach therefore provides a workaround for cases where the assertion statement itself is the problematic code and a potential workaround for cases where defensive code follows the assertion statement. The availability of such an option significantly reduces disincentives against liberal use of the conditional assertion facility.

Coverity consists of a large set of short duration programs that can be aborted if an assertion fails and a few long duration programs that, if aborted, risk losing analysis results produced over many hours or days. Coverity can't afford a check-pointing process for performance reasons. Since Coverity is not itself a safety-critical application, graceful degradation of analysis results is strongly preferred over the loss of all analysis results. For those reasons, these long-duration programs operate in a mode in which assertion failures are logged but do not result in program termination; execution continues following the conditional-assertion statement. The short-duration and long-duration programs share libraries built from common source code; thus the behavior exhibited when an assertion fails must be load-time or runtime configurable.

2.3 Releasing Contracts

One of the hardest parts of making use of Contracts in a production environment is the introduction of new contract checks into existing code. Bloomberg has extensive experience with doing this with the `bsls_review` portion of the `bsls_assert` contract-checking facility.

The fundamental problem is that, given an existing program that does *not* have contract checks in it or that is missing a contract check that should have been there before, the cost of program termination when contract checks are violated is too high. What is known about the program is that it runs on production inputs fine with the existing version, and so any bugs it might have are probably at least tolerable. Ideally, software is tested on production-like inputs sufficiently to encounter such failures only in noncritical environments. Sadly, a single program termination in a production system can quickly dissuade an entire enterprise from increasing production correctness by enabling contract checks in production.

The semantics available for new checks in production environments strongly impact the likelihood of actually finding and fixing bugs that might occur only in production.

- Enforced checks cause massive production outages when they are violated, and should they occur repeatedly during the deployment of Contracts in an organization, the needed fortitude to continue to reduce bugs by suffering crashes will quickly erode.
- Ignored checks in turn ignore contract violations, allowing potentially business-risking bugs to persist almost indefinitely. If testing fails to find the bug today, rarely will an existing system actively hunt down and find more production-only bugs tomorrow.
- The observe semantic, which allows a system to identify violations while continuing with the pre-existing control flow after the violation is detected, enables an organization to quickly reduce the number of bugs in their production environments without risking the stability of their already-existing business interests.

The *observe* semantic is clearly the best choice for identifying production bugs without risking the introduction of *new* production failures that will be blamed on the use of Contracts.

Whether a CCA is actually *new* depends on the answer to two questions.

1. Was a previous version of the same function deployed lacking the CCA completely?
2. Was a previous version of the same function deployed and the CCA not evaluated, due to either being built with a pre-Contracts version of C++ or being deployed using the *ignore* semantic?

In both cases, an organization benefits greatly from its ability to choose the *observe* semantic until such a time as these questions answers are no longer “yes” and the *enforce* semantic can be safely chosen.

The first case is truly identifiable only by marking up a CCA as `new` in the source code itself, and such local control of contract-checking semantics has a large design space that needs to be explored after consensus is reached on a Contracts MVP.[2] On the other hand, the second case is often one that can be readily identified at a wider scale, such as when first upgrading to a version of C++ with Contracts, when upgrading a library to a new version that has added CCAs, or when choosing to enable Contracts in an environment where they were previously ignored.

2.4 REPL

Read-Evaluate-Print Loop (REPL) interpreters are useful for rapid prototyping or language exploration. In a REPL environment, program source code is not compiled in a traditional sense; there are no build modes. Whether contract evaluation is enabled and what the consequences are for contract violations is, therefore, a dynamic configuration property that a user might want to change at any time.

[2]The upcoming [P2755R0] will provide a robust exploration of this design space.

2.5 Debugging

The expressions contained in contract annotations must not cause side effects that influence the essential behavior of the program. Absent a mechanism to formally prevent programmers from authoring contract predicates that exhibit such side effects, it must be accepted that, occasionally, programmers will inadvertently do so. It can be difficult to identify such mistakes; a side effect might occur within the evaluation of a series of indirect or conditional function calls. A useful technique for determining if such side effects are occurring is to compare the behavior of a program run with contract evaluation disabled and the same program run with contract evaluation enabled with the *observe* semantic. If any deviation in the program behavior is observed, then the program likely contains undefined behavior or behavior contingent on the evaluation of contract predicates, which is useful information in either case.

2.6 Summary of Requirements

All of the above use cases hinge on certain attributes of a contract-checking facility that the current MVP proposal does not provide. In order to have a robust Contracts facility that *can* satisfy the above use cases, we must enable a platform to satisfy the following requirements:

- It must be possible for a conforming Contracts implementation to allow/facilitate enabling and disabling contract evaluation without requiring a rebuild.

  Lacking this ability, the ecosystem of package managers, debugging, and deployment will need not just a single (i.e., checked vs. unchecked) distribution of every build but an ever-increasing number as the scope of contract checking options increases.

- It must be possible for a conforming implementation to allow users to choose, potentially on a per-CCA basis or via violation handler behavior, to continue execution following the detection of a violated contract.

  Choosing to not enable the *observe* semantic results in a failure to enable the deployment of contracts in existing systems, as well as an inability to control continuation more flexibly from within a contract-violation handler.

Neither of these requirements are universal - platforms should be free to choose to *not* support these requirements when it is not appropriate for their user base. Our intent is not to have embedded systems, hardened compilers, or maximally optimized deployments all need to support the same feature sets - it is to keep all such implementations as well-defined, standards-conforming, and subject to the same training and reasoning about behavior that is applicable to other C++ platforms.

3 Proposal

To allow Contracts to meet the requirements specified above, we begin by proposing the possible semantics a CCA may have.

Proposal 1: Semantics

When evaluated, a contract-checking annotation may have a semantic of *ignore*, *observe*, or *enforce*.

Proposal 1.1: Ignore Semantic

A CCA evaluated with the semantic of *ignore* does not evaluate its predicate and does not result in a contract violation.

The *ignore* semantic is the simplest (we hope) to understand and implement; an ignored semantic is functionally equivalent to having the predicate as an unevaluated operand, such as being the argument to the `sizeof` operator.

Proposal 1.2: Observe Semantic

A CCA evaluated with the *observe* semantic may evaluate its predicate and, if it would not return `true` the contract violation handling process will be invoked.

For CCAs with the *observe* semantic, if control flow returns normally from the contract violation handling process then the evaluation of the CCA will be complete and control flow will continue normally after the point of evaluation of the CCA.

Proposal 1.3: Enforce Semantic

A CCA evaluated with the *enforce* semantic may evaluate its predicate and, if it would not return `true`, the contract violation process will be invoked. If control returns normally from the contract violation handling process, the program will be terminated in an implementation-defined manner.

For CCAs evaluated with the *enforce* semantic, control flow will never continue normally from the point of evaluation of the CCA if there is a violation — either the violation handling process will not return normally or the program will be terminated.

For a contract evaluated with either the *observe* or *enforce* semantic, the evaluation of a predicate may happen zero or more times, following the SG21 consensus to adopt the proposals in [P2751R1]. If the predicate would not evaluate to `true` for any of this (including throwing an exception), the violation handling process will follow.

Should SG21 adopt a user-provided contract-violation handler as in [P2811R5], the `contract_violation` object's `semantic` property will reflect the semantic with which the CCA was evaluated, i.e., `observe` or `enforce`. Other properties will similarly be populated based on how the evaluation determined that a contract violation had occurred.

The current MVP already proposes the *enforce* and *ignore* semantics, this proposal simply adds in the possibility of the *observe* semantic on top of the status quo.

Proposal 2: Chosen Semantic

For each evaluation of a CCA, what semantic that CCA will have is implementation defined.

This approach to the semantic a contract check may have gives implementations significant freedom to meet many needs.

- All checks in a program might be forced to have the same semantic.
- Having different semantics upon different evaluations makes having different fixed semantics in different inlined versions of the same function *not* an ODR violation.
- Implementations have the ability to allow the selection of the semantic for a CCA to happen at compile time, link time, or run time as long as the implementation defines for its users the mechanics of how to make that selection.

It is important to understand that this proposal makes the semantic no longer a property of the CCA — it is a property of each individual evaluation of a CCA, and may vary from one evaluation to the next. Because this is no longer a property of the CCA, compile-time attributes of anything containing a CCA cannot depend on the semantic that CCA will have. Therefore, the first principle in [P2834R1][3] follows from this proposal as well. This is a useful synergy, as the arguments in [P2834R1] show the fundamental correctness problems that would fall out from not following its primary principle.

The current MVP only provides two bespoke build modes where all contract semantics for a single translation unit are either *enforce* or *ignore*. Mixing modes across TUs is conditionally-supported with implementation-defined semantics, an approach that was also take with C++20 contracts. This proposal effectively makes the semantics and all mixing implementation-defined, maximizing conforming implementation flexibility.

4 Recommended Practices

The current Contracts MVP (which is most formally reprsented in [P2388R4]) requires two *build modes* and, when none is specified, notes that the default mode should be `Eval_and_abort`. Should SG21 desire to continue to have these requirements on top of our basic proposal, these could be standardized as recommended practices.

Proposal 3: Default Ignore — The `No_eval` Build Mode

Recommended Practice: An implementation should provide to users the ability to translate (all translation units of) a program such that all CCAs have a semantic of *ignore*.

[3]The build mode governing the semantics of a CCA must not affect the *proximate* compile-time semantics surrounding that annotation.

Proposal 4: Default Enforce — The `Eval_and_abort` Build Mode

Recommended Practice: An implementation should provide to users the ability to translate (all translation units of) a program such that all CCAs have a semantic of *enforce*.

These recommended practices reproduce the `No_eval` and `Eval_and_abort` build modes of the MVP. It is entirely reasonable for a platform to provide additional options to select, for example, the *observe* semantic for all contracts or to *enforce* preconditions and *ignore* postconditions, or any number of a myriad of other possibilities. While SG21 could include these as recommended practices, we do not see the need to overly penalize those platforms that intend to provide more limitted choices, or to over-proscribe the available possibilities and inadvertently imply to users that those are the only potential options that should be available to them.

Proposal 5: Default Semantic

Recommended Practice: When nothing else has been specified by a user, a CCA will have the *enforce* semantic.

Many advocates of contract-checking see this recommended practice as essential to having C++ be maximally safe when used out of the box. This is also the effective default behavior of `<cassert>` when `NDEBUG` is not defined. It has been noted[4] that this requirement to have a default semantic has no meaningful enforcability as a normative requirement, but that does not preclude it being stated as a recommended practice.

On the other hand, there is an incentive for compilers to respect the choices that their users make and apply those choices intelligently to how CCAs will be evaluated. For example, a user requesting an optimized build is choosing to prefer performance over other considerations, and thus it would be reasonable for this to be considered "saying something" about performance and thus how much energy should be expended on correctness checking, and therefore choose to *ignore* CCAs in optimized builds (when no more specific contract configuration has been requested). The "nothing else specified" in the above proposal is largely intended for vanilla invocations of a compiler by a novice user — giving such users the most protection as they are learning, and making it clear to users how to begin to make additional choices in more advanced builds in order to refine the balance between performance and safety.

All recommended practices, being non-normative, could freely be included or dropped from the Contracts feature proposed by SG21 with no normative effect. We recommend that SG21 discuss these options individually and included the practices that have the most consensus - with no overt requirements on implementations either way.

5 Implementation Strategies

We will now explore some possibilities that exist for implementation strategies that are enabled by the MVP and our proposals.

[4]See [P1769R0].

The clearest way to explore what the implementation of evaluation of a CCA might look like is to show how a CCA might be transformed into equivalent C++ code. To facilitate that, we will consider how the CCA `[[assert : X ]]` will be transformed into equivalent code.

The evaluation of the predicate of a CCA is allowed to happen zero or more times,[5] and enabling elision would require special treatment of the predicate. Without loss of generality, we will show implementations that evaluate the predicate exactly once using an `if` statement or ternary operator.

These examples will assume that the specifics of a user-provided contract violation handler match those proposed in [P2811R5], but adaptation to changes in whatever SG21 adopts for violation handling should be obvious.

5.1 Fixed Semantics

The first approach to implementation we will explore is translation of a CCA with fixed semantics. Here, for each CCA evaluation, a semantic is chosen at compile time, and the CCA is transformed into suitable instructions to implement that semantic. This strategy is the most straightforward translation of a CCA and comes with the least flexibility.

We expect this or a similar approach to implementing CCA evaluation to be employed by systems with the tightest resource constraints and least need for ongoing deployment flexibility, such as embedded systems, or those that seek to minimize the runtime flexibility of their generated code.

To perform the direct invocation, let us assume a few implementation-defined functions are available whose details should be self-evident (and which might vary significantly based on other pending SG21 decisions).

- A `consteval` function to initialize an object of the implementation-defined type `__contract_info` that identifies the details of the currently checked contract:

  ```
  struct __contract_info;
  consteval __contract_info __current_contract_info();
  ```

- A function whose purpose is to take a `__contract_info` object along with a `semantic` and a `detection_mode` and then invoke the replaceable contract violation handler:

  ```
  void __invoke_violation_handler(
      const __contract_info&,
      semantic,
      detection_mode = detection_mode::predicate_false);
  ```

- A function to terminate the program in an implementation-defined manner when an enforced contract has been violated:

  ```
  [[noreturn]] void __terminate_on_enforced_violation() noexcept;
  ```

The invocation of a CCA with a fixed semantic translates differently based on the semantic.

[5] See [P2751R1].

Semantic	As-If Code
Ignore	`sizeof( (X) ? true : false );`
Observe	`if (X) {} else {`
	`  static constexpr __contract_info contract_info = __current_contract_info();`
	`  __invoke_violation_handler(contract_info, semantic::observe);`
	`}`
Enforce	`if (X) {} else {`
	`  static constexpr __contract_info contract_info = __current_contract_info();`
	`  __invoke_violation_handler(contract_info, semantic::enforce);`
	`  __terminate_on_enforced_violation();`
	`}`

Embedding the full contract semantics where the CCA is evaluated maximizes the number of beneficial code transformations that a program may perform based on the chosen semantic but requires a rebuild to choose a different semantic for a CCA.

5.2 Dynamic Semantics

Using the same interface to the contract-violation handling process, we can generate code that, based on some determination of the appropriate semantic, will use that semantic for the evaluation of a CCA. Consider another built-in function to obtain the semantic that should be used for the currently evaluated CCA:

```
semantic __current_contract_semantic();
```

Based on this function, a CCA might be translated in this manner:

```
static constexpr __contract_info contract_info = __current_contract_info();
switch (__current_contract_semantic()) {
case semantic::ignore:
    sizeof( (X) ? true : false );
    break;
case semantic::observe:
    if (X) {} else {
        __invoke_violation_handler(contract_info, semantic::observe);
    }
    break;
case semantic::enforce:
    if (X) {} else {
        __invoke_violation_handler(contract_info, semantic::enforce);
        __terminate_on_enforced_violation();
    }
    break;
// case semantic::assume:
//     [[ assume : X ]];
//     break;
}
```

Enabling this approach has certain significant advantages.

- Only a single binary needs to be produced to support all contract-checking variations that a platform might support.
- The `__current_contract_semantic()` invocation may be, depending on the implementation, a function that produces a fixed result for an entire task, is evaluated at runtime, or takes any number of other factors into account.
- At link or load time, enough information might be preserved to find all invocations of `__current_contract_info()` and replace their invocation and the immediately following branch based on the returned semantic by a single jump to the appropriate location, removing any potential runtime overhead associated with the contract violation.
- Support for an additional potential semantic, such as *assume*, is simple and nonintrusive to add.

On first glance, this approach might seem to lose the ability to leverage optimizations that could be possible after an *enforced* CCA has been evaluated. In this case, remember that a compiler has the ability (which is often taken advantage of) to perform some truly heroic transformations on the code it generates. Assuming our CCA is followed by a function body (`BODY;`), we might see the above code transform into something equivalent:

```
static constexpr __contract_info contract_info = __current_contract_info();
semantic current_semantic = __current_contract_semantic();
if (semantic == semantic::ignore || semantic == semantic::observe) {
    if (semantic == semantic::observe) {
      observe_label:
        if (X) {} else {
            __invoke_violation_handler(contract_info, semantic::observe);
        }
    }
  ignore_label:
    BODY; // #1
}
else {

    if (semantic == semantic::enforce) {
      enforce_label:
        if (X) {} else {
            __invoke_violation_handler(contract_info, semantic::enforce);
            __terminate_on_enforced_violation();
        }
    }
  // assume_label:
  //  [[ assume(X) ]]; // redundant for enforced checks

    BODY; // #2
}
```

[6]You might be alarmed to see `goto`, but we're discussing the code a compiler will transform user code into, which in practice already consists of vast numbers of moral equivalents to `goto`. We recommend you minimize writing `goto` in your code, but feel free to benefit from all the wonders that result from your compiler's careful and structured use of jumps for control flow.

Each potential semantic value could also be used with a `goto`[6] to jump to the labels we have indicated in the above code.

For an *observed* or *ignored* CCA, the `BODY` at `#1` that gets executed has been compiled with no ability to draw conclusions about the truth or falsehood of the CCA's predicate. This is exactly what we want: Neither of these semantics introduce any situations in which the code that follows them will be unreachable.

For an *enforced* CCA, however, the `BODY` at `#2` is compiled with the assumption that the contract predicate has not been violated. That `__terminate_on_enforced_violation` is `[[noreturn]]` guarantees that control flow never reaches `BODY` (`#2`) when `X` is `false`. Should we introduce the *assume* contract semantic in some future evolution, dynamic contract semantics could still take advantage of the resulting forward-looking optimization opportunities by jumping to the `assume_label` above and using the portable assumption attribute, `[[assume]]`.[7] Note that, along the branches taken for the *enforce* semantic, this use of `[[assume]]` is redundant; control-flow analysis already tells the compiler that `X` is a usable fact at this point in the code.

5.3 User Interface

The natural question comes up as to how a user might interact with a library built with dynamic contract semantics.

A possible approach for dynamically linked programs on Linux would be to allow a link-time selection of the default semantic that can then be overridden at load-time via an environment variable recognized by ld.so; see the ld.so man page for examples of environment variables that are currently recognized. For example, an `LD_CONTRACT_SEMANTIC` environment variable could be recognized. An end user wishing to override the default behavior would then set the environment variable appropriately when running the program.

There are several existing mechanisms that could be leveraged for Windows programs. An environment variable could be recognized of course, but that isn't how this kind of configuration is typically managed on Windows. one mechanism typically employed involves program image specific registry settings managed by the `gflags.exe` utility. Thus, an end user wishing to override the default behavior might be expected to change a registry setting or run `gflags.exe`. Windows application manifests could be another mechanism used to override the default contract semantic.

In REPL scenarios, the end user would presumably issue a REPL specific command.

Perhaps debuggers would override the default semantic (e.g., to select observe) and automatically react to contract violations (e.g., by automatically setting a break point on the violation handler).

Our expectation is that changing the default semantic at link-time, load-time, or run-time would be rare and generally done to either workaround a problem (e.g., to select ignore or observe to override a default of enforce in order to suppress program termination following a contract violation) or to enable debugging (e.g., to select enforce to override a default of ignore or observe in order to force the program to detect and abort on contract violation; perhaps while running in a debugger). We don't think the user experience has to be particularly nice as long as it can be reasonably automated.

[7] See [P1774R8].

5.4 Preconditions and Postconditions

Transforming a precondition or postcondition entails performing the appropriate transformation at the right point of evaluation and that the transformation exactly matches the one performed for an assertion (and we elaborated on this above).

The primary distinction is that, for preconditions, we might wish to move the evaluation point (in a manner that is unobservable to the abstract machine) from within the function invocation to immediately prior to it, to thus facilitate providing the call-site source location to the violation-handling process. This approach is often known as *call-side checking*. Should an implementation choose to do this, documenting well what will control the semantic that checks will have — i.e., will such control be based on compiler flags of the caller, the callee, or some mix of both? — would be important. These are implementation choices that must be made and would all be within the allowed behaviors of our specification.

6 Conclusion

SG21 members arrive with a wide variety of experiences using real-world contract-checking facilities for a broad range of purposes. Many of these use cases require significant flexibility from a contract-checking facility. Providing sufficient flexibility to allow platforms to explore satisfying these use cases enables the community to take advantage of contract checking immediately while gaining the experience needed to reach consensus on the additional features that will layer on top of the initial Contracts facility that we provide in C++.

In particular, moving away from the concept of singular monolithic build modes enables platforms to meet the realistic needs that will be immediately apparent when an attempt is made to use and deploy contracts in any modern C++ ecosystem. Allowing the known range of correctness-oriented contract-checking semantics — *ignore*, *enforce*, and *observe* — will enable developers as well as organizations of all sizes and from a variety of industries to deploy robust contract-checking systems in real-world environments as soon as a Standard ships with Contracts included and compilers make Contracts available to their users.

Acknowledgements

Thanks to Gašper Ažman, John Lakos, and Ville Voutilainen for actively participating in the initial discussion that led to the formalization of this paper, as well as all of those who participated in the reflector and telecon discussions that led up to it. Thanks to Andrzej Krzemieński, Timur Doumler, and others for futher feedback on 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 authors.

Bibliography

[P1769R0] Ville Voutilainen, “The "default" contract build-level and continuation-mode should be implementation-defined”, 2019 http://wg21.link/P1769R0

[P1774R8] Timur Doumler, “Portable assumptions”, 2022
http://wg21.link/P1774R8

[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

[P2751R1] Joshua Berne, “Evaluation of Checked Contracts”, 2023
http://wg21.link/P2751R1

[P2755R0] Joshua Berne and Jake Fevold and John Lakos, “A Bold Plan for a Complete Contracts Facility (forthcoming)”, 2023

[P2811R5] Joshua Berne, “Contract Violation Handlers”, 2023
http://wg21.link/P2811R5

[P2834R1] Joshua Berne and John Lakos, “Semantic Stability Across Contract-Checking Build Modes”, 2023
http://wg21.link/P2834R1