A Principled Approach to Open Design Questions for Contracts

Document #:	P2932R3
Date:	2024-01-15
Project:	Programming Language C++
Audience:	SG21 (Contracts)

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

Abstract

SG21 has made significant progress toward producing a complete design for a Contracts facility MVP. As work proceeds for completing this feature proposal, open questions must be considered and answered such that the feature will eventually have widespread adoption throughout the entire C++ ecosystem. Fundamental design principles that help guide such decisions for the Contracts facility are presented in this paper as are proposals that address most (if not all) of the open questions affecting the design of the Contracts facility MVP.

Contents

1 Introduction **2**

2 Guiding Principles **3**
- 2.1 CCAs Check Plain-Language Contracts 3
- 2.2 Concepts Do Not See Contracts 4
- 2.3 Choose Ill-Formed to Enable Flexible Evolution 8
- 2.4 Contract Reliability 9
- 2.5 Feature Orthogonality 10
- 2.6 Other Principles 10

3 Proposals **10**
- 3.1 Trivial Special Member Functions 11
- 3.2 Implicit Lamba Captures 15
- 3.3 Compile-Time Evaluation 19
- 3.4 Virtual Functions 25
- 3.5 Coroutines 27
- 3.6 Contracts on First Declarations 29
- 3.7 Are CCAs potentially throwing? 31

4 Conclusion **36**

A ODR Unuse Is Bad **37**

Revision History

Revision 3

- Clarified foundational Principle 2

Revision 2 (Presented in November 2023 meeting in Kona)

- Clarifies constant expression handling section, particularly regarding non-runtime evaluable predicates in observed CCAs
- Corrected Proposal 1.B to apply to *trivial* special member functions, added Proposal 1.C
- Updated to the adopt SG21 syntax from [P2961R0]
- NOTE: This revision includes proposals that were edited in realtime and has not been fully editted, it is being published to record what was presented in a face-to-face WG21 meeting

Revision 1 (October 2023 mailing)

- Clarified workarounds for lambda captures
- Added principles of reliability, orthogonality, discussion of others
- Clarifications of approach for handling compile-time evaluation
- Alternate proposal 1.B

Revision 0 (September 2023 mailing)

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

1 Introduction

The Contracts MVP being developed by SG21[1] is progressing smoothly toward a complete proposal according to the agreed-upon schedule.[2] SG21 has reviewed several papers and has reached a general consensus on what will happen when a contract-checking annotation (CCA) is evaluated.

- [P2751R1], "Evaluation of *Checked* Contract-Checking Annotations," clarified what content will be allowed in the predicate of a CCA and how the results of that evaluation will be interpreted.
- [P2811R7], "Contract-Violation Handlers," described a link-time replaceable contract-violation handler to allow for customized handling of contract violations at run time.
- [P2834R1], "Semantic Stability Across Contract-Checking Build Modes," established that having preconditions and postconditions evaluated outside the `noexcept` guarantee of a function that is marked `noexcept` should not be considered.
- [P2877R0], "Contract Build Modes, Semantics, and Implementation Strategies," made the choice of semantics for a CCA implementation defined and, in particular, showed that such a choice might vary between evaluations and is thus not a compile-time property of a CCA.

[1] See [P2521R4] and the forthcoming [P2900R2].
[2] See [P2695R1].

Taking the above resolutions together, on top of the existing core MVP proposal, produces an almost complete specification for Contracts as a C++ language feature.[3] Some edge cases and other design points, however, still need to be addressed.[4]

To motivate the choices we intend to make when suggesting answers to these design questions, we will begin by identifying fundamental principles that will codify our design intentions for the C++ Contracts facility. These principles start with basic goals and build up to concrete requirements that all proposals should meet (or maximize) when making engineering decisions for what should become part of C++.

2 Guiding Principles

Each principle proposed in this paper comes from the author's experience, follows from SG21's earlier principles, or aims to maximize some aspect of the quality of the Contracts facility that eventually becomes adopted by the C++ Standard.

2.1 CCAs Check Plain-Language Contracts

When programmers of any language set out to write a subroutine (or function), they have in mind some notion of what will happen when the subroutine is invoked (for a given subset of syntactically valid inputs). The part of that notion to which the programmer is willing to commit as unchanging with changes to the implementation is what we generally refer to as the function's *contract*. Hence, for any function that can be relied upon by clients, a *contract* must exist between the callers of that function and its implementers. This contract, which we call the *plain-language contract*, exists as a combination of the explicit (through documentation) and implicit (through choice of programming language, coding style, local conventions, etc) agreements to which both parties agree.

A *plain-language contract* can capture a boundless variety of different conditions, requirements, and promises that might exist between callers and callees. Some of these are prone to being validated with a simple C++ expression, such as `x > 0`, while others are completely beyond the scope of the C++ abstract machine, such as "clients of this function have paid their bills recently."

That a plain-language contract is absolute is important to understand; if both parties agree to something, they make no guarantees about what happens if that agreement is violated, i.e., the behavior is undefined. Often, such as when transmitting input across untrusted boundaries, this contract *must* include guarantees about the handling of bad input, and neither the existence of bad input nor the handling of it is in the purview of contract checking.[5]

A *contract-checking annotation* encodes in C++ an algorithm that can detect when a plain-language contract has been violated; i.e., the fundamental agreement between caller and callee cannot be met given the detected condition. When writing a CCA, that new annotation must be checking some aspect of a plain-language contract to detect violations of the agreement that exists between the caller and callee of the function.

[3] I.e., barring an actual syntax for the feature, which is the subject of a separate, ongoing series of discussions orthogonal to the issues in this paper. For this paper, we will be using the C++20 attribute-like syntax (see [P0542R0]).

[4] See [P2834R1] for the initial explorations of some of these solutions based on the principles described by that paper. See also [P2896R0], which enumerates the open design questions addressed by this paper.

[5] See [P2053R0] for more on this subject.

Any CCA that identifies a condition as a defect that is not related to its associated *plain-language contract* is clearly doing something above and beyond checking the requirements of that *plain-language contract*. If, for any kind or placement of CCA, there are ways in which that CCA might be evaluated where the predicate might indicate a violation when the CCA's associated plain-language contract is not violated or not relevant, then a false-positive bug will have been diagnosed, which would be a defect in the structure of the Contracts facility itself.

Therefore, a CCA, when introduced anywhere, must always evaluate in ways that validate its associated plain-language contract.

> **Principle 1: CCAs Check a Plain-Language Contract**
>
> Each contract-checking annotation placed on the declaration of a function must, when it is evaluated, identify violations of the *plain-language contract* of that function.

2.2 Concepts Do Not See Contracts

The fundamental conceit of a contract-checking facility is that running a program with CCAs checked at run time should tell us something meaningful about what happens when the same program is executed with no CCAs present at all. If a design choice made by the contract-checking facility leads to the inability to detect bugs that occur in a program without CCAs present, then the facility will have failed on a fundamental level.

The heart of this requirement is that the act of checking a contract should not be *destructive* — the presence of or evaluation of a contract check should not cause a contract to be violated, nor should it prevent a contract from being violated. The presence of or evaluation of a contract check might become destructive in a variety of ways:

1. **Runtime Side Effects** — A side effect in the evaluation of a predicate might alter state that directly impacts the evaluation of the function and results in its failing to meet its own postconditions:

   ```
   int x = 0;
   void f()
     pre( x++ < 10)
     post( x < 10)
   {}
   ```

2. **Compile-time Side Effects** — The predicate of a CCA, being normal C++ code, might ODR-use entities that are not otherwise ODR-used and thus result in changing control flow in the program simply by being present in the program. This effect might happen without even evaluating the CCA, largely as a result of introducing new static data members whose initialization alters control flow.[6]

[6] ODR-use can also trigger template instantiation which would introduce new friends into the overload resolution process and result in stateful alterations to program behavior based on constant evaluations. See [CWG2118] for more information on this, although this technique does not work on most platforms and it seems to be the general consensus that this is a bug that needs fixing and is not behavior which should be relied upon.

Consider, for example, a type that maintains a count of the number of objects of that type which have been default constructed:

```
class CountInstances {
private:
  static int s_count;
public:
  CountInstances() { ++s_count; }
  int count() { return s_count; }
};
int CountInstances::s_count = 0;
```

A template with a static data member of this `CountInstances` type can then be used to alter runtime behavior based on odr-use:

```
template <typename T>
struct X {
  static CountInstances counter;
};
```

The `main` function might then branch on there being no instances yet created before it begins:

```
void testFunction();

int main()
{
   if (CountInstances::count() == 0) {
     testFunction();
   }
}
```

On the other hand, should we add a precondition to `testFunction` that, through ODR-use, triggers the creation of a static `CountInstances` object, we would find that our `main` function never even reaches an invocation of `testFuntion` to validate the newly added precondition:

```
void testFunction()
  pre( X<int>::counter.count() >= 0 );
```

By introducing this CCA, even if we never build our program in a way that evaluates the predicate, our main function will promptly return in lieu of doing its primary task do to the static initialization of an additional instance of `Counter`.

3. **Additional Computation** — The evaluation of a predicate, even should it have no observable side effects, might result in measurable changes in performance characterstics of a function which results in it failing to meet guarantees provided by its contract. Consider, for example, a binary search algorithm that has, as a precondition, a check that the input is sorted:

```
bool binary_search(int *first, int *last, int value)
  pre( std::is_sorted(first, last);
```

When enabled, the above check will make a function that guarantees logarithmic performance have linear performance instead, even though there is no distinction in behavior (when passed a sorted range) observable within the C++ abstract machine itself.

Even non-algorithmic changes in behavior can introduce CPU-level performance-impacting changes, such as additional branches or computation, which at the very least dissipate additional heat when evaluated in a correct program.

4. **Observable Properties of a Function** — C++ has powerful introspective abilities, ranging from standard library traits, tools such as the `noexcept` operator, to the ability to control overload resolution using Concepts. Any one of these tools might, should they be able to detect the presence or expected behavior of a contract check, lead to different code paths being followed when that contract check is present.

   In [P2834R1] we demonstrated that allowing the configured semantic of a CCA to impact whether a function was `noexcept` could result in a checked build experiencing a completely disjoint set of bugs from those which the unchecked build exhbits, even before a single contract check is actually evaluated.

A predicate with fully defined behavior and no observable side effects as far as the abstract machine is concerned might *still* cause a function contract to be violaed due to complexity guarantees of that function. At the other extreme, a predicate that updates shared state in a program might be destructive for a function directly related to that state, and the same predicate might be perfectly innocuous on a different function. Given those facts, it quickly becomes clear that we cannot mandate that all CCAs encode non-destructive contract checks — there is no clear-cut definition that the language could provide which would identify destructive and only destructive checks.

What the language design *can* do, however, is not get in the way of allowing developers to write non-destructive checks, and that is imperative. In particular, the mere presence of a CCA in a piece of code — even one with a vacuous predicate — must not be capable of becoming destructive. With a vacuous predicate, the only impact that a CCA might have is to alter the semantics of the code in which the CCA is present. We already have within the language a tool to identify when an entity does or does have certain semantics, and that tool is concepts — therefore we arrive at the conclusion that concepts and contracts must be, as much as possible, orthogonal to one another:

Principle 2: The Presence of CCAs Does Not Alter Concepts

Introducing a CCA into an entity should not alter the concepts satisfied by that entity any more than the predicate itself would.[a]

[a]Thanks to Lisa Lippincott for help in narrowing down this formulation.

The program semantics that a concept could detect, which would allow an alternate control path to be followed that might never even reach a CCA, includes a variety of semantic properties:

- Changing the result of SFINAE
- Altering overload resolution results
- Altering the result of a compile-time operator, such as `noexcept` or `sizeof`

- Altering the value of a type trait that might be checked in an `if constexpr`

Now, clearly, the entire Contracts facility's purpose is to allow for writing more code that alters a program's behavior. This principle provides a boundary for how that behavior alters a program, facilitating reasoning about the program with CCAs present as well as the program that existed before those CCAs were added to it.

There are, in fact, a number of different principles that build upon this one, each making clear how certain aspects of allowing a CCA's presence or behavior to impact program semantics must be limitted.

The first correlary principle, originally elucidated in [P2834R1], is that we must not allow the semantic with which a CCA will be evaluated to impact the semantics of the program around the CCA. In our original paper, we made it clear, with detailed examples showing the risks if this was notdone, that a precondition check on a `noexcept` function should not make invocations of that function potentially-throwing when the CCA is *observed* or *enforced* while having it be nonthrowing when the CCA is *igored*. In general, the landscape has clarified to make this functionally impossible with the adoption of [P2877R0], but the basic guiding principle that leads to that still remains.

> **Principle 3: Program Semantics Are Independent of Chosen CCA Semantics**
>
> The semantic with which a CCA is evaluated must not affect the compile-time semantics surrounding that annotation, e.g., any observable traits of expressions or function invocations involving evaluation of the CCA.

An alternative way to think about this principle is that, with respect to the semantics of the program surrounding the CCA, the semantics *ignore*, *observe*, and *enforce* are indistinguishable. Obviously, the checked semantics will result in the predicate evaluating, and a contract violation with those semantics will clearly alter what happens when the CCA is evaluated, but in the proximate area leading up to the CCA evaluating the semantic will not matter. Fundamentally, it cannot be known what that semantic will be until the CCA evaluation is reached. This prevents the semantic choice from producing heisenbugs — turning on or off checking will not make the check itself no longer be reached.

A third side of this coin is about what happens when a CCA is introduced into code that previously did not have contract checking at all. For a program that is otherwise unchanged, contract checking should simply verify its correctness and not fundamentally alter it. More importantly, when it is decided *not* to check the contracts (i.e., evaluate them with the *ignore* semantic) it is important that the program that runs match the performance and behavior of the program that had no CCAs at all. Any deviation from this would result in a strong motivation to either not use the Contracts facility at all or, at best, use macros to remove them entirely from production builds.

To avoid this problem, it is thus imperative that any CCA, especially one evaluated with the *ignore* semantic, has the same semantic impact on a program (i.e., the same impact on control flow, such as what concepts are satisfied) as if the CCA had not been there at all. Put another way, we must see the same behavior from a program with all CCAs *ignored* as we would see if they had a *nothing* semantic where the CCA and its predicate were entirely removed from the program.

Principle 4: Zero Overhead for Ignored Predicates

The behavior of code surrounding a CCA evaluated with the *ignore* semantic is *as if* the CCA's predicate was an unevaluated operand, i.e., placed as the argument of a `sizeof` operator.

While fundamentally just another way to state Principle 2, this view of it is essential for partially non-technical but social reasons — the adoption of Contracts must have no friction or supposition of impact on the benefits C++ brings to software without Contracts, otherwise we risk bifurcating the community into those who improve correctness with Contracts and those who use C++ to maximize other factors, such as performance. Contracts can, and should, play well with all C++ software and improve its correctness, and that facet of its design must be maintained.

2.3 Choose Ill-Formed to Enable Flexible Evolution

Early in the process of identifying a path for bringing Contracts back into the C++ draft Standard, SG21 solicited and amassed[7] a wide array of use cases for guiding the scope of our next attempt at a viable MVP for a C++ Contracts facility. Despite valiant efforts to refine the semantics and functionality addressed by the MVP, a large number of these use cases will remain unsupported until more work is done and further consensus has been achieved.

When standardizing a solution, we would be wise to leave room for satisfying the use cases that SG21 will address after an initial feature is integrated into C++. To facilitate that evolution, decisions made today must not prevent migration to a more broadly applicable feature in future revisions.

When deciding how to provide this essential freedom, a few *specification tools* are available to us.

- One choice, which was made for quite a few aspects of the C++20 Contracts facility, is to leave the behavior that is not finalized as undefined behavior. This option has the advantage of allowing implementations the flexibility to experiment with various possible solutions along with the severe disadvantage of all those solutions being nonportable and possibly conflicting. SG21 had early consensus to refrain from using undefined behavior in this fashion because doing so is generally perceived to be a source of reduced safety, not increased flexibility or correctness.

- A middle ground rarely chosen is to simply make certain behaviors *implementation defined*. While generally appearing safer than *undefined behavior*, this approach still results in code that is nonportable when different platforms define distinct and incompatible behaviors for the same constructs.

- Another specification tool available to us is to make what we are unsure of ill-formed. That is, given two or more plausible solutions whose behaviors largely but incompletely overlap, rather than making all of that behavior undefined or even implementation defined, we might choose to define the overlapping behavior and (to strongly discourage inconsistency) to leave the remaining, nonoverlapping behavior ill-formed. Implementations are still, in this case, free to provide conforming extensions that implement potential evolutionary paths but, when asked to strictly conform to the Standard, will have consistent, well-defined behavior.

[7] See [P1995R1].

- Finally, when we have consensus on a solution but are unclear whether the solution in question will be implementable on all platforms, we might choose to specify the full solution and make it *conditionally supported*. Using this approach, however, effectively introduces multiple *dialects* of the language on different platforms, which is often an approach the WG21 community has avoided.

In general, given the history of SG21's consensus to focus on the safety and portability of the Contracts feature being developed, the preferable option is to leave room for evolution by making the available design space ill-formed in the current implementation.

> Principle 5: Make Undecided Behaviors Ill-Formed
>
> To accommodate use cases that are not yet supported by the current contracts proposal, prefer to keep extensibility flexible by declaring unresolved behavior ill-formed, rather than either undefined ("unsafe") or implementation defined (nonportable).

2.4 Contract Reliability

Depending the checking of contracts is often a sign of bad design; CCAs, by their very nature, are carefully designed to introduce *defensive* checks that are redundant in any correct program. On the other hand, when diagnosing a problem, being able to identify which checks have passed often becomes critically important so that we can reason about where a defect might actually be occurring. In other words, when checks are being enforced, knowing that such checks are going to be evaluated and enforced is quite helpful.

> Principle 6: Checked Contracts Must Be Checked
>
> The evaluation of a CCA — and thus the checking of a CCA with a checked semantic — must not be silently elided.

For normal function invocations, Principle 6 is easy to follow. When the function is invoked, all CCAs on that function will be evaluated. Special member functions, however, alter this analysis slightly due to the possibility of eliding the invocation of certain special member functions entirely.

- Copy elision allows what would appear to be places where a copy constructor or a destructor should be invoked to be skipped entirely.
- Trivial default constructors and destructors can be elided by the compiler since they do nothing. Libraries may also elide their invocation through identifying triviality via Standard Library traits such as `std::is_trivially_destructible`.
- Trivial copy constructors and assignment operators can be replaced by an equivalent bitwise copy. Just as with the other potentially trivially operations, this property may be detected through Standard Library traits, and higher-level libraries can replace explicitly copying or assigning with corresponding calls to `std::memcpy` or `std::memmov`.

Given the common understanding of copy elision in the language, requiring CCAs be evaluated if they appertained to an elided copy constructor, move constructor, or destructor seems ill advised.

The other operations that are elidible when trivial, however, pose a separate concern that we will address in Section 3.1.

2.5 Feature Orthogonality

C++ is a rich, powerful, multi-paradigm language. An impedance mismatch between features, where a developer is forced to choose to use one — *only* one — of a set of features, greatly harms adoption of all those features. For unrelated features, we cannot, as designers, assume that users will not prioritize using *all* features and then be incapable of meeting their own needs if the features are incompatible.

> **Principle 7: Unrelated Language Features Should Remain Orthogonal**
>
> A language feature must minimize how much its use impedes the use of other language features.

2.6 Other Principles

Many other principles guide all our individual design decisions and form an input into any developer's calculus regarding the quality of any given piece of language design.

- Do not pay for what you do not use. — New language features must impose a cost only when being used.
- Maximize teachability and simplicity. — Language features that cannot be easily understood lead to defects when they are used, so all designs should strive to remain intuitive, understandable, and clearly specified.
- Refer to Occam's Razor. — When multiple solutions are available, choose the simplest one.

Each of these principles is individually important, yet they do not supersede the more focused principles we have already articulated that apply directly to allow Contracts achieve the goals they are attempting to achieve, which we believe apply specifically to the proposals in this paper.

For example, we do not believe it is important to choose simplicity for simplicity's sake, or to follow naive assumptions whose application would be in conflict with our guiding principles.

3 Proposals

Each subsection within this section discusses a distinct open question in the specification of the Contracts MVP and provides a proposed resolution based on the principles we have introduced above.

Some proposals are made of multiple constituent parts. The individual parts throughout a section are not numbered, and a final complete, numbered proposal will be made near the end of each section.

Some proposals have multiple options that are consistent with our principles, and these options address cases in which conflicting needs arise from distinct principles that would result in distinct proposals when prioritized differently.

3.1 Trivial Special Member Functions

A trivial operation — constructor, destructor, or assignment operator — is one that (1) is generated entirely by the compiler and (2) invokes no user-provided code. The copy and move operations, when trivial, become bitwise copyable (and thus *may* be replaced by, e.g., `memmove`). The default constructor and destructor, when trivial, do nothing.[8]

Providing a body — even an empty one — for a special member function results in that function never being trivial. Defaulting a user-declared special member function (via `= default` on the first declaration) will result in it being trivial as long as it doesn't need to invoke any user-provided code for a base-class or member object.

The definition of the *trivially copyable* trait has evolved over the years, but the main point is that all the default-generated copy and move constructors and copy and move assignment operators are *trivial* (and hence, can treat the object as pure data) as long as at least one of them is not deleted. For a type to be *trivially copyable*, it must also be *trivially destructible*; i.e., the compiler-generated destructor is a no-op.

Now consider that, to minimize runtime overhead and still get substantial coverage, common practice allows for any type that happens to have one or more (programmatically verifiable) class (object) invariants to assert them in the one place where the flow of control must pass for every constructed object: the destructor. This defensive-checking strategy is particularly effective at catching memory overwrites.[9]

Now imagine we have a trivially copyable value type, such as this heavily elided `Date` class:

```
class Date {
  int d_year;   // [ 1 .. 9999 ]
  int d_month;  // [ 1 .. 12   ]
  int d_day;    // [ 1 .. 31   ]
 public:
  static bool isValidYMD(int year, int month, int day);
    // Return true if year/month/day represents a valid date.

  Date(int year, int month, int day) pre(isValidYMD(year, month, day));
    // Create a Date object having a valid date.

  int year()  const { return d_year; }
  // ...
};
```

In the `Date` class above, the user-declared value constructor creates a valid `Date` object set to the specified `year`, `month`, and `day`, and in a *checked* build, its precondition check invokes the class

[8]See also the mention of this issue in [P2521R4], Section 4.4, "Annotations on trivial ops.," `{con.trv}`.

[9]Feel free to ask the author for stories about the pain that comes from developers who decide to implement types that overwrite their *own* members by calling `memset(this,0,sizeof(this))` in their constructor bodies, under some misguided belief that doing so improved performance and correctness. In particular, if such a class has any member, such as `std::string`, that has a nontrivial default constructor that does more than just zero-initialize, this ill-fated choice leads to painful-to-diagnose problems that such invariant checking can often readily detect.

member function `isValid` to *ensure* (or perhaps just to *observe*) that the date is, in fact, valid. From then on, the only way to change the value of this object is through the use of its compiler-generated assignment operations.[10]

As previously stated, the `Date` class above is trivially copyable, but let's now add a defaulted destructor having a precondition check that validates its invariants:

```
~Date() pre(isValidYMD(d_year, d_month, d_day)) = default;
  // Destroy this object.
```

Notice that the precondition itself applies the public class member function, `isValidYMD`, to the private data members, e.g., `d_year`, of the class. The Committee has long since agreed[11] that all CCAs are part of a function's implementation; hence, a CCA that is associated with a member function fairly deserves private access to the data members of the class.

If CCAs are *ignored*, the destructor does nothing, and because no user-supplied definition is available, one might reasonably presume that this special member function remains *trivial*. In a *checked* build, however, code provided by the user is expected to run when an object of this type is destroyed. In some sense, this `Date` type is *almost*[12] trivially destructible and thus, at least *notionally*, trivially copyable.

Following Principle 3, i.e., we cannot change the semantics of code based on which semantic CCAs are evaluated with, means that we *cannot* take the approach of treating this `Date` class *as if* it were *trivially destructible* and *trivially copyable* in an *unchecked* build but *not* in a *checked* build.

Once again, we are left with three alternatives.

1. Specify that defaulted special member functions that have associated CCAs are never trivial in any build mode. The implication is that merely having an inactive precondition could substantially impact performance,[13] even in an unchecked (e.g., *ignored*) build.

2. Apply Principle 5 and make the application of CCAs to a defaulted, trivial special member function ill-formed. In a future Standard, the option to opt out of this restriction explicitly with a label (such as `pre skippable (true)`) would remain open for those who wish to use both features.

[10]Recall that, just by declaring any non-special-member constructor, we suppress even the declaration of the default constructor. All five remaining special member functions, however, are generated as usual.

[11]During a Standards Committee meeting in 2015 that involved CCAs, Bjarne Stroustrup expressed the need for a CCA on a member function to have private access to the implementation of the function's class so that the CCA could write efficient preconditions without having to expose extra functions in a public API that were not directly relevant to clients. This discussion led to a paper, [P1289R1], which was considered and achieved consensus in November 2018. This result has remained the SG21 consensus, as indicated in Section 4.3 of [P2521R4].

[12]See [lakos21], Section 2.1."Generalized Pods," "Use Cases," "Skippable destructors (notionally trivially destructible)," pp. 464–470, especially pp. 469–470.

[13]Although a special member function that has an empty body supplied by the user will provide no code and therefore would seem to be no different than the empty body supplied by the compiler, the copy constructor being trivial gives the compiler special permission to bypass calling that copy constructor entirely. This optimization is particularly effective for contiguous sequences of such objects — e.g., `std::vector<my_trivially_copyable_type>` — since repeated calls to the copy constructor can be replaced by a single call to `memmove`. For an object to be considered trivially copyable, however, it must have a trivial destructor. Note that the compiler's ability to see that the body of a user-supplied destructor is empty doesn't make that destructor trivial, nor does such compiler ability give license to the *library* to use `memmove` for a type that would otherwise be *trivially copyable* but for its almost trivial destructor.

3. Treat contract checks on trivial functions as *skippable* without notice. That is, if the function itself is considered trivial in an *unchecked* build mode, it will report so in *every* build mode. Libraries may circumvent the execution of these special member functions, even in checked build modes. (Note that we are skipping the *check* itself, not just any side effects that executing the check might have produced.) If, however, the compiler would be eliding invocation of the trivial function, it might nonetheless choose to evaluate the CCAs, followed by the trivial operation, depending on the user's chosen build mode and optimization level.

In the case of the first alternative above, the potential loss in runtime performance from not doing the `memmove` invocation could be substantial, perhaps even dramatic, and would introduce a strong disincentive to adding such defensive checks to otherwise trivial functions, thus violating Principle 4. Hence, we consider the first alternative to be nonviable.

The second alternative allows us to add an opt-in for the third alternative in the future and adheres most strongly to Principle 6 by preventing a situation where CCAs might become unreliable.

The third alternative is much more consistent with our conclusion with respect to the `noexcept` specifier in that it keeps the two language features — namely triviality and contract checking — maximally orthogonal, adhering to Principle 7. This latter approach does come with the risk of perhaps omitting checks that an uninitiated author might have intended the client to run in every case — i.e., including even those cases in which use of an equivalent `memmove` would be substantially more runtime performant. Fortunately, the duly informed author of a CCA for a special member function can always easily opt into this other slower but safer behavior:

```
~Date() pre(isValid(d_year, d_month, d_day)) { }  // empty body
  // Destroy this object.
```

Simply by providing an empty function body (see the example above), a function that was otherwise *trivial* can easily be made nontrivial in every build mode, thus removing the permission for libraries or the compiler to skip the invocation of this destructor and the evaluation of its associated checks. Consider that the checks are on the destructor, so there's nothing to skip on the copy constructor, and the compiler could easily run the checks on destruction in a checked build even though no code had to run to destroy the object. If the author of the class fails to realize the triviality of the function and, as a result, some check isn't run, no affirmative harm is done since the check was defensive (redundant) anyway and therefore entirely useless in every correctly written program.

Finally, we note that *all* contract checks are presumed to be purely defensive and thus entirely redundant in a defect-free program. Turning off runtime contract checking on otherwise *trivial* functions is, in effect, just one more way for developers to control if and to what extent their programs are checked at run time. Hence, as long as the presence of a CCA doesn't affect the compile-time result of evaluating a trait on a type, we might imagine giving the compiler explicit (e.g., via a compiler switch) freedom to sometimes refrain from invoking such runtime checks on trivial types, perhaps in collaboration with the totality of the (e.g., optimization) build modes.

If we prioritize Principle 7 (orthogonality) and accept the risk to Principle 6 (reliability) imposed by potentially skipping CCAs, then we arrive at our proposed resolution for this design decision.

Proposal 1.A: CCAs Do Not Affect Triviality

A (defaulted) special member function having preconditions or postconditions may still be trivial. Note that trivial special member functions might be replaced (by the language or a library) by bitwise copies or even elided completely, both of which would skip evaluation of any CCAs associated with that function.

Alternatively, we can favor Principle 6 with the Contracts MVP and make function triviality incompatible with having CCAs.

Proposal 1.B: No CCAs on Trivial Special Member Functions

A *trivial* (defaulted) special member function having preconditions or postconditions is ill-formed.

Alternatively, we also have the general question for defaulted functions in general over whether you are defaulting the interface of the function or defaulting the implementation of the function. Delaying this question requires delaying that decision for all defaulted functions.

Proposal 1.C: No CCAs on Defaulted Functions

A function defaulted on its first declaration having preconditions or postconditions on that declaration is ill-formed.

These proposals were polled in an in-person SG21 at the November 2023 WG21 meeting in Kona.

2023-11 Poll

For the Contracts MVP, allow defaulted special member functions to have preconditions and postconditions without affecting the function's triviality, as proposed in P2932R2 Proposal 1.A.[14]

SF	F	N	A	SA
0	5	3	3	6

Result: No Consensus

2023-11 Poll

For the Contracts MVP, make it ill-formed for a trivial defaulted special member function to have preconditions and postconditions, as proposed in P2932R2 Proposal 1.B.[15]

SF	F	N	A	SA
1	7	4	5	1

Result: No Consensus

[14] This is also Proposal 1.A in this revision of this paper.
[15] This is also Proposal 1.B in this revision of this paper.

2023-11 Poll

For the Contracts MVP, make it ill-formed for any function defaulted on its first declaration to have preconditions and postconditions on that declaration, as proposed in P2932R2 Proposal 1.C.[16]

SF	F	N	A	SA
12	4	2	1	0

Result: Consensus

The original poll also overlooked consideration of deleted functions, to which most of the same considerations applied, and this detail was polled in the 2023-12-07 SG21 telecon:

Poll 1

For the Contracts MVP, a deleted function having preconditions or postconditions is ill-formed.

SF	F	N	A	SA
10	4	3	0	0

Result: Consensus

Therefore, for the SG21 MVP there will be no CCAs on defaulted or deleted functions.

3.2 Implicit Lamba Captures

As with any other function that may be defined in C++, functions defined using a lambda expression must be annotatable with precondition and postcondition CCAs and must contain assertion CCAs within their bodies. Any issues related to name resolution or appurtenance, such as those raised for trailing return types by [P2036R3], must be resolved by the syntax adopted for Contracts, such as is done in [P2935R0].

Proposal 2.1: CCAs Allowed on Lambdas

Precondition and postcondition CCAs may be placed on a lambda expression such that they appertain to the closure object's call operator. Assertion CCAs may appear within the body of a lambda expression. *id-expressions* that reference captured local entities will be transformed as if they appeared within the body of the lambda expression.

Similar to affecting the triviality of a special member function, another area in which a CCA might be capable of having an impact on program behavior occurs when the predicate of a CCA within a lambda ODR-uses a local entity. In such cases, that entity might be implicitly captured as part of the generated closure object, thus affecting its size and possibly incurring a large cost to initialize it when that capture is by-value.

In some sense, *not* capturing at all when a CCA will be *ignored* and capturing only when a CCA would have other semantics might be possible. With the adoption of [P2877R0], no mechanism is available to have such properties as what is captured be based on the semantic that a CCA will have when evaluated. Additionally, having such a fundamental property of the closure object as the

[16] This is also Proposal 1.C in this revision of this paper.

list of members it contains (and thus its size) be dependent on the chosen semantic of a CCA would be a violation of Principle 3 and thus should not be considered.

While simply allowing a capture might seem like a minimal violation of the zero-overhead principle (Principle 4), consider that the object referenced might far exceed the cost and scope of the lambda itself:

```
std::function<int()> foo(const std::vector<S>& v)
{
  int ndx = pickIndexAtRandom(v);
  return [=]()
    pre( 0 <= ndx && ndx < v.size() )  // needs to capture v
  {
    return ndx;  // Obviously we intend this to capture ndx by value.
  };
}
```

Because the full v object must be captured by-value due to referencing it to get its size in the precondition of the lambda, this simple constant-time function becomes linear in the size of v due to the capture. That is a subtle and significant performance hit due to the presence of a CCA.

We are left with two options.

1. Allow the implicit capture from a CCA expression, which will happen regardless of the semantic with which the CCA is evaluated.

2. Make an implicit capture due to a CCA's expression ill-formed, preventing subtle costs due to the presence of a CCA.

Choosing the first alternative would violate Principle 4, ensuring that no runtime or object-size overhead is associated with any contract checks compared to simply not having such checks present in the program in the first place. The potentially significant and hidden cost of such captures might discourage the adoption of the Contracts facility in general.

The second alternative — i.e., making ill-formed the ODR-use of a local entity that is not already ODR-used in the body of the lambda apart from any CCA — creates zero overhead when *ignored*, has no effect on object size, and yet clearly informs users when they are referencing a value that is not directly relevant to the body of the lambda. This approach is both consistent with Principle 4 and, if there is any doubt about whether it is the correct long-term direction, is also an application of Principle 5.

One workaround, for those who want to capture that dubious value anyway, is simply to add an odr-use of the varaible in question within the body of the lambda:

```
std::function<int()> foo(const std::vector<S>& v)
{
  int ndx = pickIndexAtRandom(v);
  return [=]()
    pre( 0 <= ndx && ndx < v.size() )  // needs to capture v
  {
    static_cast<void>(v);  // force implicit capture of v
    return ndx;  // Obviously we intend this to capture ndx by value.
```

```
  };
}
```

Another option is to add an init-capture to make an explicit copy of the variable in question:

```
std::function<int()> foo(const std::vector<S>& v)
{
  int ndx = pickIndexAtRandom(v);
  return [=,v=v]()  // capture v in a new variable named v
    pre( 0 <= ndx && ndx < v.size() )  // uses captured variable v
  {
    return ndx;  // Obviously we intend this to capture ndx by value.
  };
}
```

Of course, when thinking about the compilation error due to the lack of capture of `v`, a developer will quickly realize that much better alternatives are available: `assert` that `ndx` is in the proper range prior to initializing the lambda or capture only `v.size()` for use in the lambda's precondition.

Even more significantly, should the captured variable be a *new* capture not only is extra work done by the closure object's initializer, the very nature of the closure object changes from a captureless lambda to one with a capture. Consider the following functions which would distinguish between closure objects with and without captures:

```
template <typename T>
std::true_type f(T t)
  { return {}; }

template <typename T>
std::false_type f(T t)
  requires std::is_convertible_v<T, bool(*)()>
  { return {}; }
```

Given the above, we could easily see how lambdas which alter whether they capture values would result in different code paths being taken based solely on the presence of contract checks:

```
void g()
{
   auto x = [](){ return true; }
   static_assert( ! decltype( f(x) )::value );  // convertible to bool(*)()

   auto y = []() pre(x()) { return true; }
   static_assert(   decltype( f(x) )::value );  // not convertible
}
```

This reasoning all comes together into one additional proposal regarding lambdas.

Proposal 2.2: CCA ODR-Use Does Not Implicitly Capture

It is ill-formed if all potential references to a local entity that cause that entity to be implicitly captured occur within CCAs attached to or within the body of that lambda.

The reference to the local entity from the CCA will still continue to attempt to name the member of the closure object instead of the local entity itself. Therefore, if no other factor creates that member of the closure (either an explicit capture or an ODR-use that is *not* in a CCA), a program will be ill-formed.

In comparison to the proposal in [P2890R0], we have complete agreement on the need to support CCAs on lambda functions, and the only difference is whether the exception for captures of local entities from within CCAs should be made.

These proposals were discussed and polled in the 2023-12-07 and 2023-12-14 SG21 telecons. The general ability to have CCAs on lambda expressions was polled once, and clarified with a followup poll regarding details of name lookup:

Poll 3

For the Contracts MVP, allow preconditions and postconditions on lambdas with the syntax proposed in [P2961R2] and the name lookup rules proposed in [P2890R1].

SF	F	N	A	SA
9	9	1	0	0

Result: Consensus

Poll 1

For the Contracts MVP, adopt the name lookup rules for pre and post on lambdas proposed in P2890R2. Name lookup for pre and post on a lambda should follow the existing rules for pre and post on functions in the Contracts MVP: it should be performed as-if the predicate were at the beginning of the lambda's body.

SF	F	N	A	SA
10	5	0	0	0

Result: consensus

Additionally, whether CCAs are able to implicitly capture entities was polled:

Poll 4

For the Contracts MVP, if an entity implicitly captured by a lambda expression L is only referenced within the preconditions and postconditions of L and within contract assertions inside the body of L, the program is ill-formed.[a]

SF	F	N	A	SA
7	7	1	3	0

Result: Consensus

[a]This is equivalent to Proposal 2.2

Therefore, for the SG21 MVP, precondition and postcondition CCAs may be put on lambda expressions and cannot implicitly capture entities on their own.

3.3 Compile-Time Evaluation

The specifics of evaluating a CCA at compile time have not been thoroughly pinned down in the MVP and, more importantly, might violate some of our fundamental principles if no additional changes are made.

Currently, after the adoption of [P2877R0], any evaluation of a CCA might or might not evaluate the predicate and detect a violation, and nothing about that proposal was specific to runtime evaluations. Therefore, even during compile-time evaluations, whether a CCA would be checked (i.e., have the *observe* or *enforce* semantic) or not (i.e., have the *ignore* semantic) is implementation defined. Note that this semantic is allowed to vary not only from one CCA to another, but also from one evaluation of a given CCA to the next evaluation of the same CCA.

For runtime evaluations, the contract-violation handling process involves invoking the global, replaceable contract-violation handler, which clearly can't be known when compiling an individual translation unit (TU).

Egregious differences between the behavior of compile-time evaluation and runtime evaluation are ill advised, so we propose retaining the meanings of the potential semantics of CCA evaluation at compile time and simply changing the effects of violations in a way that has similar spirit but does not allow the same customizability; we change an attempt to invoke the contract-violation handler into one that simply emits a diagnostic, the mechanism we use for the compiler to emit information about a problem to the user.

When a CCA being enforced is violated, we must also decide what happens when the *implementation-defined program termination* occurs, and again we have two potential choices.

1. Make the enclosing constant-evaluated expression ineligible to be a constant expression. This option is the typical specification tool used to say something *cannot* be done at compile time.
2. Make the program ill-formed, with no ability to choose a separate control flow path or to evaluate at run time instead.

With Option 1, one can then decide, based on the evaluation semantic, what to do in other contexts, altering program semantics based on the chosen semantic.[17] Allowing this form of detection of the chosen semantic would violate Principle 3, and we, therefore, consider this option nonviable. We are thus obliged to choose Option 2, making any constant evaluation that violates a checked CCA ill-formed.

Proposal: Compile-Time Contract Violations Emit Diagnostics
When the contract-violation process is evaluated at compile time (i.e., a CCA is violated when being evaluated with the *observe* or *enforce* semantic), a diagnostic will be emitted. When the evaluation is performed with the *enforce* semantic, the program is ill-formed.

The other major aspect of CCA evaluation that we must consider is the evaluation of the predicate itself and whether that must be eligible for compile-time evaluation. More importantly, can that

[17] See the example of detecting violations of the precondition of `sqrt` on page 22 to understand how being evaluable at compile time can be used to alter program semantics.

predicate being ineligible for constant evaluation render the enclosing expression ineligible for being a constant expression?

Recall from [P2877R0] that we can effectively consider the evaluation of a CCA with expression `X` to be of this form:

```
switch (__current_contract_semantic()) {
case semantic::ignore:
    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;
}
```

The above proposal is equivalent to saying that the intrinsic `__current_contract_semantic()` is a valid core constant expression, and that `__current_contract_semantic` is `constexpr` or `consteval`. As described in [P2877R0], implementations have a wide range of flexibility in what this function may do.

- An implementation that provides a single, global switch to choose the semantics for all CCAs might, for example, when that chosen semantic is enforced, install a version of semantic computation:

  ```
  consteval semantic __current_contract_semantic()
  {
    return semantic::enforce;
  }
  ```

- A different configuration might choose to provide a mode in which the semantic computation function that is installed makes different decisions at compile or run time:

  ```
  constexpr semantic __current_contract_semantic()
  {
    if consteval {
      // constant-evaluation semantic:
      return semantic::enforce;  // or something else, based on compiler flags
    }
    else {
      // runtime semantic
      return semantic::ignore;   // or something else, based on compiler flags
    }
  }
  ```

- Other implementations might inspect link-time properties to determine the runtime semantic, provide mechanisms to select the compile-time or runtime semantic based on the location of the CCA being evaluated, or do any number of other operations that produce the behavior such implementations have defined for the selection of CCA semantics.

When determining if an expression containing a CCA can be evaluated at compile time, we will need to identify two things:

1. Is each individual evaluation within the process of evaluating a CCA eligible to be part of a constant expression?
2. If the expression is not eligible, what happens?

The first question we have already answered above for the violation handler and termination: Neither can be performed at compile time, and it is ill-formed if the termination on enforced violation happens at compile time. The full expansion above, however, shows us we must answer questions about additional parts of this expansion as well.

First, is the computation of the semantic, which is the implementation-defined manner in which a semantic is chosen for the evaluation of a CCA, eligible to be done as part of a constant expression? If we were to allow the answer to this question to be *no*, then doing contract checking at compile time would be impossible; the very act of determining the semantic for a CCA would make the enclosing expression no longer a constant expression. This decision would be highly unfortunate, effectively making the use of CCAs completely incompatible with compile-time programming. Therefore, we propose that the selection can be done at compile time.

> Proposal: Semantic May Be Selected At Compile Time
>
> The implementation-defined choice of semantic when evaluating a CCA may be evaluated as part of a core constant expression; i.e., selecting the semantic as part of evaluating a CCA does not make the expression containing the CCA ineligible to be a *core constant expression*.

The other question is what happens when the expression, `X`, is ineligible to be evaluated at compile time. Here we note that one chosen semantic — the *ignore* semantic — can always make the answer to this question irrelevant. When evaluating a CCA and given our previous proposal, if the *ignore* semantic is chosen, the rest of the evaluation will always be eligible to be part of a constant expression; no other evaluations will be performed. Therefore, because we are striving to maintain Principle 3 (program semantics are independent of chosen CCA semantics), the evaluation of the expression must be just as eligible to be part of a constant expression as the empty branch in the *ignore* case is. Therefore, if the evaluation of the predicate, `X`, is *not* eligible to be part of a constant expression, we must treat that as defect. In the case where the CCA semantc is *observe* we will expect a diagnostic, and where it is *enforce* the program will additionally be ill-formed.

Just as we do with contract violations, we should allow *observing* a CCA that fails to be evaluable at compile time to emit a diagnostic while *enforcing* such a CCA should be ill-formed.

Proposal: Use of Nonconstant-Eligible Predicates When Constant Evaluation Is Required Is Ill-Formed

In an expression that is a valid core constant expression, evaluation of a predicate that is not eligible to be a core constant expression emits a diagnostic. If the semantic of the corresponding CCA is *enforce*, the program is ill-formed.

Note the importance of this situation being ill-formed when *enforced*, rather than the expression simply being ineligible to be a constant expression; some manifestly constant-evaluated contexts are also SFINAE contexts and thus would allow control flow to alter based on which semantic is selected for evaluating a CCA.

Consider the following prototypical example of a function with a narrow contract, also a function one might consider beneficial to use at compile time:

```
constexpr double sqrt(double x) pre(x >= 0);
```

Now, interestingly, we can make a concept check that could tell us whether a given expression is a valid Boolean constant expression:

```
#include <type_traits>  // for std::bool_constant

template <double x>
concept can_constexpr_sqrt = requires { std::bool_constant<sqrt(x)>(); };
```

Now, code could be written in terms of this concept that would have highly unintuitive behavior since it depends entirely on the chosen semantic of the precondition check on `sqrt` at compile time:

```
template <double x>
void f()
{
  if constexpr (can_constexpr_sqrt<x>) {
    // #1 --- processing if x >= 0 or preconditions are disabled
  }
  else {
    // #2
  }
}
```

The code block at `#2` is reachable only if we were to allow the precondition check failure to be subject to SFINAE and thus let the concept check fail instead of making the program ill-formed when the attempt to form the result `sqrt(-1.0)` was made. With our proposals, the above code would be ill-formed if the precondition's predicate is evaluated; otherwise, the code would always flow through to the block at `#1`. This behavior is the same as what we would observe in evaluating this code at run time with no attempt to use it at compile time.

The same reasoning must be applied to potentially constant variables that are not `constexpr` — i.e., reference or non-volatile `const`-qualified variables of integral or enumeration types. These variables are usable in a constant expression if their initializers are core constant expressions, but they are also fine to initialize with noncore constant expressions (and thus dynamically initialize at run time).

Here, following Principle 3 is again important as is keeping any well-formed behavior consistent with the behavior when CCAs are *ignored*. To determine this, an evaluation with all CCAs ignored must first be done to ascertain if the expression, without CCAs, is eligible to be a core constant expression. When this trial evaluation is *not* eligible for the initialization of a potentially constant variable, the CCAs should not matter; the initialization will happen as a dynamic initialization at run time, and any contract violations that might be detected will happen then. A trial evaluation that determines that a CCA is a core constant expression will cause the variable initialization to be a manifestly constant-evaluated context, and thus the rules above will apply.

For such variables, we want to make ill-formed (and not SFINAE-able) any attempt to use them in a manifestly constant-evaluated context.

Proposal: Nonconstant Eligible Initializers Evaluate CCAs at Run Time

If the initializer of a non`constexpr` potentially-constant variable is not itself a core constant expression when all its CCAs are evaluated with the *ignore* semantic, then that initializer is not a core constant expression; hence, no attempt will be made to perform constant evaluation on the expression again with CCAs potentially having different semantics.

We call out this last proposal because we should not aggressively enforce contracts at compile time that might never be evaluated at run time. Consider, for example, a function that demands to be evaluated at runtime for certain inputs as it cannot be used at compile time in large parts of its domain:

```
int compute_value(int n);   // not constexpr

consteval bool at_compile_time()
  // Return true if evalutaed at compile time, false otherwise.
{
  if consteval { return true; }
  else { return false; }
}

constexpr bool do_compute(int n)
  pre( n == 0 || !at_compile_time() )
{
  if (n == 0) {
    return 17;
  }
  else {
    return compute_value(n);
  }
}

void g()
{
  const int i = do_compute(0);  // can be compile time
  const int i = do_compute(1);  // must be computed at runtime
}
```

In the above, `do_compute` is evaluable with an input of `0` at compile time, but any other input must be computed at runtime. Without the precondition check on `do_compute` that is exactly what happens in `g()`. Introducing the precondition check at compile time would result in a failure, even though it could be evaluated at compile time — and that is why we must not be checking CCAs until we *first* determine if an expression is a core constant expression or not.

For each CCA with a checked semantic during the second evaluation we need o then ask the question of if it *would* make the larger expression a non-core constant expression or if it would identify a contract violation. In either case we emit a diagnostic, and if the semantic is *enforce* the program should be ill-formed.

All these proposals related to constant evaluation address individual aspects of the constant evaluation process and how it relates to CCA evaluation. Putting this all together produces one complete proposal that accomplishes all the above goals:

> Proposal 3: Evaluation of CCAs at Compile Time
>
> When determining whether an expression is a core constant expression, first determine if the expression is a core constant expression by evaluating it with all CCAs having the *ignore* semantic.
>
> - If it is a core constant expression or if it is not a core constant expression but is in a manifestly constant-evaluated context, re-evaluate the expression while evaluating the CCAs with semantics chosen in an implementation-defined manner.
>   - If the semantic is *observe* or *enforce* and the CCA *would* cause the expression to fail to be a core constant expression if evaluated, or it *would* result in a contract-violation if evaluated, emit a diagnostic. If the semantic is *enforce*, the program is ill-formed.
>   - If the expression is not a core constant expression, the program is ill-formed.
> - Otherwise, the expression is not a core constant expression.

This is the same general proposal put forth in [P2894R1], though that paper goes to significant efforts to present the solution in a way more easily digestible to those unfamiliar to the mechanics of constant expression evaluation.

By only hypothetically evaluating CCAs, we enable the ability to *observe* CCAs whose predicates are not checkable at compile time and treat them as violations, continuing on to the rest of the core constant expression after recognizing that the CCA produces a diagnostic. This process also removes any side effects that might occur from impacting the rest of the evaluation of the CCA, and thus the result of the trial evaluation and the final evaluation will invariably be the same.

Note that for a runtime-evaluated CCA, it is possible to distinguish between *observed* and *enforced* CCAs programmatically by inspecting the return value of the `semantic()` member function of the `contract_violation` object passed into the contract-violation handler. For a constant-evaluated CCA however, such a distinction cannot be made programmatically but only observed through the program being either well-formed or ill-formed.

With the multi-step algorithm proposed above to determine whether an expression containing a CCA is a core constant expression, constant evaluation with contracts might seem hard to teach or understand. This algorithm does not, however, significantly add to the already complex rules for how a `const`-qualified variable of integral or enumeration type behaves like a `constexpr` variable based on how it is initialized.

For general users of modern C++, who are taught to prefer `constexpr` and `constinit` over abusing `const` variables at compile time, the simple rule that your program will fail to compile (or produce a warning) if a contract violation is detected during constant evaluation is easy to understand and follow and will benefit programmers by calling out defects in their compile-time evaluations.

This proposal was discussed, along with [P2894R2], in the 2024-01-11 SG21 telecon.

Poll 1

For the contracts MVP, adopt the rules for contract checking during constant evaluation as proposed in [P2894R2].[a]

SF	F	N	A	SA
9	8	1	1	0

Result: Consensus

[a] This proposal is largely equivalent to Proposal 3.

3.4 Virtual Functions

With the adoption of [P2954R0] by SG21, CCAs on virtual functions have the following behavior.

- CCAs may be placed on virtual functions that do not override any other virtual functions.
- A virtual function that overrides exactly one with CCAs inherits the CCAs of that overridden function.
- Virtual functions that override multiple functions may do so only if none of those functions have CCAs.

The problem is that inheritance of CCAs from a base class violates Principle 1. By introducing a CCA on a base-class virtual function, all derived class virtual functions implicitly get that same virtual function. The general case for well-defined object-oriented implementations might be okay with this, but in some cases that will misidentify defects in software that are not actually wrong.

In particular, consider a derived class that has a wider contract on a function than its base class:

```
class Car {
  virtual void drive(int speed);
    // The behavior is undefined unless the specified speed
    // is less than 100.
};

class FastCar {
  void drive(int speed) override;
    // The behavior is undefined unless the specified speed
    // is less than 169.
};
```

This `FastCar` is a completely fine derived class for `Car`and can be used anywhere `Car` can.

Now consider adding a CCA to `Car` to validate the precondition of drive:

```
class Car {
  virtual void drive(int speed) pre( speed < 100 );
};
```

The default inheritance of this CCA in `FastCar` will cause all code that drives a `FastCar` *fast* to break — even code that knowingly has a `FastCar` and reasonably wants to take advantage of the fact that it has a `FastCar` in hand.

This results in the following (partial) proposal to fix the current SG21 MVP behavior to adhere to our stated principles.

> **Proposal: CCAs Are Not Inherited**
>
> A virtual member function that has no precondition or postcondition CCAs on its declaration will have no precondition or postcondition CCAs (and will not inherit those of any functions it may be overriding).

Properly allowing variance of CCAs across class hierarchies requires checking both the CCAs of the virtual function being invoked as well as those of the concrete implementation found through virtual dispatch.[18] This solution might, however, not be ready for SG21 to adopt at this time.

Consider a test function where we use a virtual function via dynamic dispatch through both base-class and derived-class references:

```
void testCars()
{
  FastCar fc;

  Car& cr = fc;
  cr.drive(120);   // CCAs of Car::drive and FastCar::drive
                   // should be checked.

  FastCar& fcr = fc;
  fcr.drive(120);  // CCAs of FastCar::drive should be checked.
}
```

With the current MVP, the CCAs of `Car::drive` and `FastCar::drive` are the same, so the behavior will be compatible with the above. The current MVP behavior without inherited CCAs does not give a clear answer as to what CCAs would be checked in the above calls.

Without a more complete solution, we must then apply Principle 5 and remove the ability to put CCAs on virtual function declarations until that more complete solution is adopted:

[18] After the initial MVP achieves consensus, the forthcoming [P2755R0] — or possibly other papers — will propose that the best approach here is to check both the CCAs that are visible based on the static type through which the function is being invoked as well as those of the specific concrete function selected by virtual dispatch, thus guaranteeing that all expectations of both caller and callee are satisfied.

Proposal 4: No CCAs on Virtual Functions

It is ill-formed to place a `pre()` or `post()` CCA on a virtual function, i.e., a member function marked `virtual` or that overrides a virtual function in a base class.

3.5 Coroutines

When users begin to consider what `pre()` and `post()` might mean when applied to a coroutine, a few frequent misunderstandings and open questions would need to be resolved.

- Do preconditions get applied if the body of the coroutine does not begin to execute for a coroutine that starts suspended?
- Are the function parameters named in a precondition referring to the parameters of the function invocation or the copies made within the coroutine frame? What about parameters named by a postcondition?
- Is a return value named by a postcondition a reference to the function's returned value, to a value returned by `co_return`, or to something returned by `co_yield`?

While perhaps clear answers could be provided, they all leave unsatisfied other obvious needs for providing contract checks on coroutines: `pre` and `post` alone do not provide anything like a complete solution for the contract checks on the interface of a coroutine.

Some considerations must also be addressed.

- For coroutine handle types that perhaps are not awaitable and cannot themselves be used with `co_await`, such as `std::generator`, whether a given function is a coroutine is unclear from a client perspective. In this context, `pre` and `post` have clear meanings in terms of the production of that initial returned object. Callers must be able to treat `pre` and `post` in the same fashion independently of whether the function being invoked is a coroutine or not.
- For the implementer of a coroutine, however, many more entry and exit points are generally available between a coroutine and other code; each call to `co_return`, `co_yield`, and even `co_await` is a distinct part of the Promise–type-specific interface the coroutine has with the outside world. Any one of these boundaries might have requirements for correctness that would be beneficial for a user of the coroutine to know and thus could have meaning when placed on the coroutine's declaration.
- When invoking a coroutine, normally a return object is produced and returned to the caller, and little ambiguity arises about when a postcondition would get evaluated for that object. Invoking a coroutine within a `co_await` expression, however, is a fairly involved process, which may involve evaluation before and after suspension and resuming of the call-side coroutine, and when exactly postconditions should be evaluate or which specific options are even implementable becomes unclear.

  Should a `co_await` expression result in suspension, arbitrary amounts of other code might execute prior to resumption, any of which may invalidate a postcondition of the coroutine. Evaluation such as this would break the fundamental intuition many have that a postcondition hold in the code immediately following invocation of a function that had that postcondition.

- Caller-side CCA checks would naturally be able to apply only to the actual function arguments, not the copies within the coroutine frame. This restriction can lead to subtle bugs if properties are checked that do not propagate through such copies.

  Possibly worse, forcing a coroutine parameter to be `const` by referring to it from a postcondition not only prevents modification of that parameter within the coroutine body (which is usually a manageable cost), but also causes the initialization of the copy (normally done with an xvalue referring to the original function parameter) to be done with a potentially much more expensive *copy* operation, not a *move* operation.

  In general, having preconditions and postconditions refer to objects that the body of the function can never actually see is a pitfall that must be considered very carefully.

Currently, no concrete proposal covers the full breadth of the interface a coroutine has with its callers. Without this complete picture, we cannot yet know if `pre` and `post` will have a meaning that is correct and useful to those calling into or implementing coroutines. Possibly `pre` and `post` with their semantics applied caller-side are the optimal solution for coroutines, even with the potential risks. Additionally, coroutines might benefit more from having bespoke *kinds* of CCAs that have different, coroutine-specific semantics (which, due to callers generally being unaware if a function is coroutine, would need to be fully implemented in the coroutine definition) that avoid these pitfalls.

Therefore, we apply Principle 5 to decide that we should disallow *any* CCAs on a coroutine until we have a more complete picture of what we intend to provide.

Proposal 5: No CCAs on Coroutines

Specifying a precondition or postcondition on a function that is a coroutine is ill-formed. That is, a function that has a `co_return`, `co_yield`, or `co_await` expression in its definition and also has a `pre()` or `post()` CCA is ill-formed.

Note that Proposal 5 intentionally says nothing about assertions. Contract checks to identify defects can still be added to a coroutine (or within the function so defined by the Promise type); they must simply be manually put within the body and cannot have additional control points by which they can be automatically injected.

In comparison to [P2957R0], this paper agrees completely on the treatment of assertion CCAs within a coroutine body. That paper, however, proposes semantics for precondition and postcondition CCAs on coroutines, which we propose disallowing until a more complete and intuitive picture can be provided for the interactions between coroutines and Contracts.

These proposals were polled in an in-person SG21 at the November 2023 WG21 meeting in Kona.

2023-11 Poll

For the Contracts MVP, allow preconditions and postconditions on coroutines, with the semantics proposed in [P2957R0].

SF	F	N	A	SA
3	1	2	7	6

Result: No consensus

2023-11 Poll

For the Contracts MVP, allow preconditions on coroutines, with the semantics proposed in [P2957R0]; make it ill-formed for a coroutine to have postconditions.

SF	F	N	A	SA
1	3	5	7	2

Result: No Consensus

2023-11 Poll

For the Contracts MVP, make it ill-formed for a coroutine to have preconditions and postconditions, as proposed in P2932R2 Proposal 5.[19]

SF	F	N	A	SA
6	8	1	2	3

Result: Consensus

Therefore, for the SG21 MVP, preconditions and postconditions will not be allowed on coroutines.

3.6 Contracts on First Declarations

SG21 has agreed that, for the MVP, CCAs shall not be repeated on multiple declarations and must always be on the first declaration encountered for a function. This placement reduces the need for answering questions about behavior if, for instance, a function is invoked in one place without having seen the CCAs on that function yet and in another place after CCAs have been seen.

Note the great benefit to insulating CCAs from either readers of a function's primary declaration or from client translation units; i.e., long-term, allowing CCAs to be repeated on later declarations, omitted from the first declaration and repeated at a later point, or even partially declared on one declaration and then fully defined on a subsequent declaration will be necessary. None of these possibilities would circumvent the one approach being allowed by the MVP: to support CCAs on first declarations only. Therefore, SG21 has followed Principle 5 to leave flexibility for future evolution and must simply define the behavior of the minimal feature we are supporting.

To fully define what we mean by restricting CCAs to be on first declarations only, we need to break down a few points.

- What is a *first* declaration?
- If multiple *first* declarations can appear in the same program, when are such CCAs the same?
- What happens if CCAs are *not* the same?

To allow for functions to be declared in multiple modules or in different header files, modern C++ would determine *first* based on reachability. In general, one declaration (`A`) is reachable from another (`B`) if `A` precedes `B` within the same TU or if `A` is declared in a module imported (directly or transitively) by the TU containing `B`. A declaration `D` for an entity is a *first* declaration of that entity if no other declarations for `D` are reachable. Interestingly, by declaring the same function (which

[19] This is also Proposal 5 in this revision of this paper.

must be attached to the global module fragment) in multiple different modules and importing those modules into a third one, multiple declarations can be *first* declarations of that function.

Definition: First Declarations

A declaration `D` for an entity is a *first declaration* of that entity if no other declarations for that entity are reachable from `D` — i.e., if no such declaration occurs earlier in the same TU and no such declaration is exported or used by a module that has been imported.

Given a clear definition of what is or is not a *first* declaration, we can restate the general rule in the MVP for CCAs being on first declarations only.

Proposal 6.1: CCAs on First Declarations Only

A function declaration that is not a first declaration shall not have `pre()` or `post()` CCAs attached to it.

To define if CCAs are the same, we will want to use the one-definition rule (ODR). This rule, however, applies to *definitions*, not declarations. Using part of the approach taken in C++20 Contracts,[20] we can define CCAs on different declarations for the same entity to be the same if they would be the same if placed in a function definition (at the same place where their declaration occurs), excluding potential renaming of function parameters, template parameters, or return value identifiers.

Definition: CCA Sameness

A CCA, `c1`, on a function declaration, `d1`, is the same as a CCA, `c2`, on a function declaration, `d2`, if their predicates, `p1` and `p2`, would satisfy the one-definition rule if placed in function definitions on the declarations `d1` and `d2`, respectively, except for the renaming of parameters, return value identifiers, and template parameters.

With the ability to identify if two CCAs are the same, we can easily extend this definition to two lists of CCAs being the same if all corresponding elements of the lists are the same. Since CCAs are always going to be evaluated in lexical order, an observable difference occurs between two otherwise identical lists of CCAs if they are reordered:

```
void f(int x)  pre(x > 0) pre(x > 1);
void g(int x)  pre(x > 1) pre(x > 0);

void test()
{
  f(0);  // violates x > 0
  g(0);  // violates x > 1
}
```

Therefore, the order must be considered when comparing lists of CCAs, and we compare the corresponding elements (in order) when determining if lists are the same.

[20] See [N4820] for the draft Standard that included the C++20 Contracts proposal wording.

Finally, the question of when to check that CCAs on a function are declared consistently across multiple first declarations must be addressed. Should the function declarations be in distinct TUs with no compile-time awareness of one another, we have little choice but to make a mismatch ill-formed, no diagnostic required (IFNDR).

Whether CCAs match when there are multiple *first* declarations reachable from the same point *could* be checked in two situations:

1. When importing a module that contains a *first* declaration for a function that is already reachable
2. When invoking a function with multiple *first* declarations that are reachable

Both options would require potentially large implementation difficulty, so we recommend leaving detection of mismatched CCAs as a quality of implementation decision; in practice, CCAs should mismatch only when declarations are being copied and pasted inappropriately instead of sourced from a header file with a single, canonical definition, so demanding extraordinary effort to detect such defects does not present as an incredibly high priority.

Proposal 6.2: Function CCA Lists Must Be Consistent

The list of CCAs on all first declarations (in all TUs) for a function shall be the same, no diagnostic required.

Because nonfirst declarations may have no CCAs on them, nothing else needs to be said for the Contracts MVP. If CCAs on nonfirst declarations were allowed, they would generally be from different source locations, and validating that the redeclared CCA lists were the same as the original would be straightforward and needed, and mismatches would then be ill-formed, *not* IFNDR.

3.7 Are CCAs potentially throwing?

Whether a CCA should be considered potentially throwing is a question that needs to be answered for a variety of reasons:

- With the ability to use assertion CCAs as expressions, it is now possible to write `noexcept(contract_assert( X` and so we must decide if this is always `true`, always `false`, ill-formed, or somewhere in between.

  ```
  static_assert(noexcept(contract_assert( true )));  // true or false?
  static_assert(noexcept(contract_assert( false ))); // true or false?
  static_assert(noexcept(contract_assert( X )));     // true or false?
  ```

- The language has a number of locations where a function's exception specification is implicitly deduced. Implicitly-declared special member functions might result in the use of a default argument that has an assertion CCA in it:

  ```
  struct B
  {
    B(int i = contract_assert( false ) , 0);
  }
  struct D : B
  ```

```
{
};
static_assert(noexcept(D()));  // true or false?
```

Explicitly defaulted special member functions might even have precondition or postcondition CCAs attached to them which might be considered potentially throwing:

```
struct S
{
  S() pre(true) = default;
};
static_assert(noexcept(S());  // true or false?
```

In the absence of the CCAs, all of these constructors would deduce a non-throwing exception specification.

- New language features often get considered, such as `noexcept(auto)`,[21] which would deduce exception specifications in additional contexts, and the question of what exception specification should be deduced in such cases must be understood and answered:

  ```
  void f() noexcept(auto) pre(true);
  static_assert(noexcept(f()));  // true or false
  ```

Currently, these decisions all come down to needing an answer to the same question: Is a CCA potentially-throwing? How we answer that question is not as simple as it might appear:

- Installing a `noexcept(true)` contract-violation handler at link time will, as a side effect, effectively make all CCA evaluations in a program never capable of emitting an exception. Thus, by the time a program is actually executing it can be definitively known whether any CCA can or cannot throw.

- The fact that a throwing contract-violation handler might be installed leads to the possibility that, without further knowledge not available at compile time, any CCA might conceivably allow an exception to escape.

- If a CCA is considered potentially throwing then there are two cases where a CCA's introduction into code will fundamentally alter the semantics of code surrounding that CCA:

  1. Any code with a deduced exception specification will now deduce `noexcept(false)` in the presence of a CCA. For example, adding a postcondition to a defaulted special member function will result in that function's deduced exception specification no longer being `noexcept(false)`:

     ```
     class S {
       class S_impl;
       std::unique_ptr<S_impl> d_pimpl;

     public:
       S(S&& orig)  post(orig.d_impl == nullptr) = default;
     };
     ```

[21] See [N3207], [N4473], and [P0133R0].

As can be seen here, the simple desire to capture that a type's moved-from state is empty results in producing a type whose move constructor is now no longer `noexcept`, a change that can have significant performance impact on uses of this type in containers.

Similarly a CCA can alter the exception specification when introduced into a default member initializer or default function argument:

```
class B {
  int d_i1 = contract_assert( true  ), 17;
  B(int i = contract_assert( true  ) , 34);
};
class D : B {
};
```

In this example, the implicit default constructor of `D` without the above assertions would have been `noexcept(true)`. On the other hand, with the addition of assertions in the two initializers, both invoked by the implicitly defined constructor of D, the default constructor would now deduce `noexcept(false)`.

2. Code which applies the `noexcept` operator directly to an expression containing an assertion CCA would branch in a different direction should that CCA be considered potentially throwing.

```
void f()
{
  if constexpr (noexcept(contract_assert( true  ))) {
    // CCA is not potentially-throwing
  }
  else {
    // CCA is potentially-throwing
  }
}
```

Of course, such code as the above is nonsensical without the CCA, and so the zero-overhead principle has no impact on it.

There might, however be cases where macros are used to amend a third-party function with assertions to check that function's preconditions:

```
#define CHECKED_X() ( contract_assert( x_preconditions()  ), x() )
void g()
{
   bool x_noexcept = noexcept(x());
   bool checked_x_noexcept = noexcept(CHECKED_X());
}
```

Should the assertion CCA above be considered potentially throwing the above code would result in different values of `x_noexcept` and `checked_x_noexcept`. Code that branched on this property would thus potentially follow very different paths resulting in the addition of checking of the contract of `x` changing the behavior of the code around it, clearly violation Principle 4.

To find a solution we can consider what the question of whether an expression is potentially throwing is truly asking.

- Does this expression allow an exception to escape under any circumstances?
- Does this expression allow an exception to escape when it has well-defined behavior?
- Does this expression allow an exception to escape when no contract is violated?

The question clearly cannot be the first one, as then any expression with undefined behavior would have to be considered potentially throwing. The second two questions, however, differ today only in what they exclude to restrict their domain to only those programs for which they are most meaningful.

Limiting the question of potentially-throwing only those cases where an expression's evaluation does not violate a contract allows CCAs to continue to satisfy the zero-overhead principle by not altering fundamental properties of a program — whether a function is `noexcept` — when a CCA is introduced.

Proposal 7.A: Potentially-Throwing Does Not Consider CCAs

When determining if a set of expressions is potentially-throwing CCAs are not considered. If there are no non-CCA expressions the query is ill-formed.

This approach brings with it two important considerations:

- It appears to be surprising to many that `noexcept(contract_assert(true))` could be `true` even though, with a throwing contract-violation handler, the expression could conceivably throw.

  Of course, for all those who install a non-throwing contract-violation handler, and in all programs where a contract is not violated, readers might equally hope that the noexcept operator would return `true` and avoid pessimizing (or even fundamentally altering) correct programs for the sake of incorrect ones.

  Since there are no non-CCA expressions to which this `noexcept` operator is being applied, we therefore avoid this confusion by making the operation ill-formed. In effect we treat `noexcept(contract_assert(true))` as equivalent to the invalid expression `noexcept()` when removing the CCAs from consideration.

  On the other hand, an expression such as `noexcept(contract_assert(true) , 17)` would be equivalent to `noexcept(17)` and evaluate to `true`. We know of few real use-cases for even asking this question in code, and so see no major issues with the answer being potentially surprising.

- Situations where an exception specification is deduced will deduce `noexcept(true)` when a CCA would be the only potential source of exceptions in an expression. This means that a throwing violation handler would then throw straight into the `noexcept` boundary on the function should there be a violation, resulting in program termination instead of an opportunity to recover.

  However, to support recovery using a throwing violation handler a program must already adhere to a number of stringent guidelines related to the implicit and explicit presence of `noexcept` in the language:

  – Do not put CCAs on or call functions that use CCAs in a function that is explicitly `noexcept(true)`.

  – Do not put CCAs on or call functions that use CCAs in a function (the destructor) that is implicitly `noexcept(true)`

  – Do not put CCAs on or call functions that use CCAs in a destructor for an object that is used as an automatic variable or a directly or indirectly owned subobject of an automatic variable (i.e., any object which might be destroyed during stack unwinding), as emitting an exception during unwinding will result in program termination.

Ignoring CCAs when considering whether a function's implementation is potentially throwing would simply extend these guidelines to include functions with deduced exception specifications:

  – Do not put CCAs on or call functions that use CCAs in a member initializer, default argument of a special member function, or on a defaulted special member function that is not otherwise `noexcept(false)`.

Of course, to check contracts in any of these situations while still supporting recovery using a throwing violation handler one must simply add explicit `noexcept(false)` annotations to the functions that envelope the use of a CCA.

As an alternative until consensus is reached, the only remaining proposal that would meet or principles would be to apply Principle 5, and make any cases where we ask the question ill-formed:

Proposal 7.B: CCAs are Neither Nonthrowing nor Potentially-Throwing
A CCA is neither nonthrowing or potentially throwing, and any use of a CCA in a situation where this must be determined is ill-formed.

This approach would make ill-formed certain operations:

- The `noexcept` operator may not be applied to an expression containing an assertion CCA.
- A precondition or postcondition CCA cannot be added to a defaulted special member function that does not have an explicit exception specification.
- An special member function may not be defaulted if its definition would require evaluating a default member initializer or default argument that contains an assertion CCA.

Each of these may be worked around by providing explicit `noexcept` specifications in the appropriate places.

To accept the semantic impact and potential cost `noexcept(false)` can be added to defaulted special member functions:

```
struct S{
  S() noexcept(false) pre(true) = default;
};
```

To retain existing behavior in the presence of added CCAs an appropriate expression to deduce the same exception specification that would be implicitly deduced must be formulated, which can be accomplished though it might be burdensome:

```
template <typename L, typename R>
struct P {
  L d_lhs;
  R d_rhs;

  P()
    noexcept(noexcept(L()) && noexcept(R()))
    pre(true)
    = default;
};
```

The largest risk we see with this proposal is that the easier approach of simply adding `noexcept(true)` is much more likely to be taken, and that brings with it the risk of turning what would otherwise have been a potentially-throwing special member function into one which is now `noexcept(true)` as a result of wanting to add CCAs to that function. This could result in a recoverable `bad_alloc` from a default constructor becoming program termination.

4 Conclusion

SG21 has come a long way while finalizing a design for a Contracts MVP. Even with the small, focused scope of the MVP, open questions remain and must be addressed prior to having Contracts as an adopted feature in ISO C++. This paper has attempted to wrap up the remaining known edge cases that are essential to address to have a complete feature proposal ready to be integrated into the rich, powerful, and sometimes challenging language that C++ is. Should any further questions arise during the adoption of this proposal, we hope these same principles can guide remaining decisions to keep the Contracts facility's design clear and consistent.

As a foundation for reasoning about undecided features for the C++ Contracts facility, and the SG21 MVP in particular, we have presented four principles:

- Principle 1 — CCAs Check a Plain-Language Contract
- Principle 2 — The Presence of CCAs Does Not Alter Concepts
- Principle 3 — Program Semantics Are Independent of Chosen CCA Semantics
- Principle 4 — Zero Overhead for Ignored Predicates
- Principle 5 — Make Undecided Behaviors Ill-Formed
- Principle 6 — Checked Contracts Must Be Checked
- Principle 7 — Unrelated Language Features Should Remain Orthogonal

Some of the proposals in this paper have already been discussed and adopted by SG21:

- Proposal 1.C — No CCAs on Defaulted Functions

- Proposal 5 — No CCAs on Coroutines
- Proposal 2.1 — CCAs Allowed on Lambdas
- Proposal 2.2 — CCA ODR-Use Does Not Implicitly Capture
- Proposal 3 — Evaluation of CCAs at Compile Time

Based on these principles, the open questions for the Contracts MVP have been addressed by the following proposals.

- Proposal 4 — No CCAs on Virtual Functions
- Proposal 6.1 — CCAs on First Declarations Only
- Proposal 6.2 — Function CCA Lists Must Be Consistent
- Either of
  - Proposal 7.A — Potentially-Throwing Does Not Consider CCAs
  - Proposal 7.B — CCAs are Neither Nonthrowing nor Potentially-Throwing

Finally, adoption of each of these reasoned proposals will serve two important objectives.

1. All outstanding design issues presented in [P2896R0] will be resolved.
2. The SG21 Contracts proposal will be a robust, initially useful, easy-to-evolve facility that integrates well with the whole of the C++ language and is ready to adopt and release in a timely fashion.

Acknowledgments

Thanks to Lori Hughes, Jens Maurer, John Lakos, Andrzej Krzemieński, and Daniel Krügler, for reviewing this paper and providing feedback.

Extra thanks to Timur Doumler for extensive discussion and contributions to make this paper significantly easier to understand.

Even more thanks to Lisa Lippincott who, as always, provided profound insight into how to formulate and consider the principles that we have represented here.

A ODR Unuse Is Bad

Putting a contract check that succeeds in a program should not make it have a defect or remove a defect — both of these options are instances of checks being *destructive*.[22] Of course, with the full power of C++ available in contract predicates it is certainly not possible for the core language to prevent all possible destructive checks from being introduced into a program.

[22] See [P2751R1].

On the other hand, at the heart of many of our principles is the recognition that a well-designed Contracts feature will not allow a CCA to alter the semantics of the program simply by existing or being evaluated, as such an alteration would lead to contract checks silently becoming destructive.

The challenging question, however, is when to consider a change in program behavior based on a CCA being present or evaluated to be a change so significant as to require having the core language prevent it. Conceptually, it is the semantics of the program in the immediate neighborhood around (and most importantly immediately before) the CCA which should not be altered by the presence of the CCA. The further away the change is, the more likely it is that it alters something unrelated to the CCA and thus not important to the program's correctness.

As a first consideration, should a CCA's introduction alter overload resolution to cause the program to resolve different entities than originally resolved we would have indication that the language has allowed a fundamental change in semantics due to the presence of a CCA. Identifying when overload resolution has changed is challenging considering that we are at the same time adding a whole new odr-using expression to the program.

But one thing the addition of the odr use from the CCA's predicate cannot do is remove an odr use the program previously had. A change in overload resolution, however, *can* mean that the previously resolved function is no longer odr-used.

Therefore, we posit that a design decision that allows the introduction of a CCA to remove an odr-use from a function is a sign of a bad decision.

- Something that can be identified with a trait, such as trivial copyability, can definitely lead to changes in overload resolution:

  ```
  template <typename T>
  void do_copy(T* to, T* from, std::true_type)
  {
     std::memcpy(to, from, sizeof(T));
  }

  template <typename T>
  void do_copy(T* to, T* from, std::false_type)
  {
     *to = *from;
  }

  template <typename T>
  void copy(T* to, T* from)
  {
     do_copy(to, from, std::is_trivially_copyable<T>{});
  }
  ```

  Now consider a type with a trivial copy constructor to which you wish to add a precondition:

  ```
  struct S {
    S() = default;
    S(const S&) pre(true) = default;
  };
  ```

```
void f()
{
   S s1, s2;
   copy(&s1, &s2);
}
```

Without the precondition, `S` is trivially copyable and this program instantiates and odr-uses the function template `do_copy<S>(S*, S*, std::true_type)`. If adding the CCA to the copy constructor of `S` were to make that type no longer trivially copyable, this overload of `do_copy` would no longer be odr-used and instead the overload `do_copy<S>(S*, S*, std::false_type)` would be linked into the program.

- A similar example involving allowing lambdas to implicitly capture can be found on page 17.

There may be a wider set of destructive CCAs that we can identify in the language in the future, but as a starting point for how the language can help prevent introducing itself as the source of CCAs being destructive this litmus test is quite effective.

Bibliography

[CWG2118] Richard Smith, "Stateful metaprogramming via friend injection" https://wg21.link/cwg2118

[lakos21] John Lakos, Vittorio Romeo, Rostislav Khlebnikov, and Alisdair Meredith, *Embracing Modern C++ Safely* (Boston: Addison-Wesley, 2021)

[N3207] Jason Merrill, "noexcept(auto)", 2010 http://wg21.link/N3207

[N4473] Ville Voutilainen, "noexcept(auto), again", 2015 http://wg21.link/N4473

[N4820] Richard Smith, "Working Draft, Standard for Programming Language C++", 2019 http://wg21.link/N4820

[P0133R0] Ville Voutilainen, "Putting noexcept(auto) on hold, again", 2015 http://wg21.link/P0133R0

[P0542R0] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, and B. Stroustrup, "Support for contract based programming in C++", 2017 http://wg21.link/P0542R0

[P1289R1] J. Daniel Garcia and Ville Voutilainen, "Access control in contract conditions", 2018 http://wg21.link/P1289R1

[P1995R1] Joshua Berne, Andrzej Krzemieński, Ryan McDougall, Timur Doumler, and Herb Sutter, "Contracts — Use Cases", 2020 http://wg21.link/P1995R1

[P2036R3] Barry Revzin, "Changing scope for lambda trailing-return-type", 2021 http://wg21.link/P2036R3

[P2053R0] Rostislav Khlebnikov and John Lakos, “Defensive Checks Versus Input Validation”, 2020
http://wg21.link/P2053R0

[P2521R4] Andrzej Krzemieński, “Contract support – Record of SG21 consensus”, 2023
http://wg21.link/P2521R4

[P2695R1] Timur Doumler and John Spicer, “A proposed plan for contracts in C++”, 2023
http://wg21.link/P2695R1

[P2751R1] Joshua Berne, “Evaluation of *Checked* Contract-Checking Annotations”, 2023
http://wg21.link/P2751R1

[P2755R0] Joshua Berne, Jake Fevold, and John Lakos, “A Bold Plan for a Complete Contracts Facility”, 2023
http://wg21.link/P2755R0

[P2811R7] Joshua Berne, “Contract-Violation Handlers”, 2023
http://wg21.link/P2811R7

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

[P2877R0] Joshua Berne and Tom Honermann, “Contract Build Modes, Semantics, and Implementation Strategies”, 2023
http://wg21.link/P2877R0

[P2890R0] Timur Doumler, “Contracts on lambdas”, 2023
http://wg21.link/P2890R0

[P2890R1] Timur Doumler, “Contracts on lambdas”, 2023
http://wg21.link/P2890R1

[P2894R1] Timur Doumler, “Constant evaluation of Contracts”, 2023
http://wg21.link/P2894R1

[P2894R2] Timur Doumler, “Constant evaluation of Contracts”, 2024
http://wg21.link/P2894R2

[P2896R0] Timur Doumler, “Outstanding design questions for the Contracts MVP”, 2023
http://wg21.link/P2896R0

[P2900R2] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, “Contracts for C++”, 2023
http://wg21.link/P2900R2

[P2935R0] Joshua Berne, “An Attribute-Like Syntax for Contracts”, 2023
http://wg21.link/P2935R0

[P2954R0] Ville Voutilainen, “Contracts and virtual functions for the Contracts MVP”, 2023
http://wg21.link/P2954R0

[P2957R0] Andrzej Krzemieński and Iain Sandoe, "Contracts and coroutines", 2023
http://wg21.link/P2957R0

[P2961R0] Jens Maurer and Timur Doumler, "A natural syntax for Contracts", 2023
http://wg21.link/P2961R0

[P2961R2] Timur Doumler and Jens Maurer, "A natural syntax for Contracts", 2023
http://wg21.link/P2961R2