Semantic Stability Across Contract-Checking Build Modes

Document #:	P2834R1
Date:	2023-06-08
Project:	Programming Language C++
Audience:	SG21 (Contracts)

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

Abstract

The Contracts MVP — along with any fully featured Contracts facility into which the MVP eventually evolves — will allow for multiple modes of program translation (a.k.a. build modes) in which a contract-checking annotation (CCA) may have distinct semantics. If differences across build modes are permitted to affect the semantics of other features of the language, then the source code of a program that is built using one contract-checking mode might compile to a substantially different program when built using another. In particular, if evaluating a precondition check on a function that has a *nonthrowing* exception specification were permitted to propagate an exception emanating from that CCA to the caller in only a checked build mode, that distinct behavior would be readily apparent across checked and unchecked build modes via routine use of the `noexcept` operator. After carefully examining and demonstrating the implications of contract-checking semantics affecting other language features in ways that are programmatically observable, we conclude generally that allowing such changes in program behavior across contact-checking build modes is actively harmful to the efficacy of Contracts as a tool for achieving correctness and improving safety.

Contents

Revision History

Revision 1 (Prior to 2023-06 Varna Meeting)

- Each appendix now ends in a proposal clarifying exactly our recommendation regarding that issue.

Revision 0 (Prior to 2023-05-18 SG21 telecon)

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

1 Introduction

SG21 is striving to complete an MVP for Contracts[1] within a timeframe that enables the MVP to ship with C++26.[2] As part of that process, a number of choices are being considered that might allow the evaluation of a contract-checking annotation (CCA) to emit an exception, which is not possible with the current MVP. Any design choice that allows an exception to be emitted from a *checked* CCA must also determine whether an exception arising from the evaluation of a precondition or postcondition check on the declaration of a `noexcept` function may be allowed to propagate back to the caller.

Treating a precondition check on a given function as occurring outside the purview of that function's exception specification is tempting in that an exception thrown as a result of evaluating such a contract check would not subject the program to immediate forced termination. As alluring as this design choice might at first seem, it is inherently fatally flawed.

In what follows, we consider the implications of allowing a thrown exception resulting from the evaluation of *any* CCA associated with a function to bypass the guarantees afforded by the exception specification of that function, arriving at one inevitable controlling principle.

Principle: Build-Mode Independence

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

Note that, by *proximate*, we are referring code that is either textually adjacent or applied (directly or indirectly) to the CCA, which we will further discuss shortly.

Before digging into those details, let's take a moment to unpack what this principle is trying to convey. Recall that we have a well-reasoned aversion to side effects in contract-checking predicates. That is, if a side effect occurs only when contract-checking predicates are evaluated (*enforce* or *observe* semantics), that side effect might result in program execution that is substantially different from when they aren't evaluated (*ignore* or *assume* semantics):

```
// main.cpp  (bad idea)

bool b = false:
```

[1] See [P2521R2].
[2] See [P2695R1].

```
void f() [[ pre: b = true ]] { }  // Precondition has a side effect.

int main() { f(); return b; }     // Return value is affected by build mode.
```

These same concerns — namely, where changing a CCA's semantics across build modes might result in substantially different program behaviors due to side effects, which occur at *run time* — appertain even more forcefully to unexpected changes in the intended *compile-time* semantics of the `noexcept` operator when applied to (the invocation of) a function declared to have both a *nonthrowing* exception specification and a precondition or postcondition check. This incontrovertible essential conclusion will be demonstrated using compelling examples.

Next let's talk a bit more about what we mean by *proximate* as compared to more general ones. We can imagine build modes that might affect one-definition-rule (ODR) use or somehow alter valid overload sets, thereby resulting in manifestly different code paths in distinct builds.[3] Preventing such abuses is neither within the remit of SG21 nor reasonably addressable by any contract-checking facility. What a well-designed Contracts facility *can* ensure, however, is that function specifiers, such as `noexcept`, imply the same semantics in all valid forms of use, irrespective of which particular CCA semantic[4] (e.g., *enforce*, *ignore*) is in force in the current build mode.

For the remainder of this paper, let's assume that we have some C++ contract-checking facility having at least one build mode in which the evaluation of a CCA may emit a thrown exception as either (1) a consequence of the predicate evaluating to something other than `true` or (2) the result of evaluating the predicate itself. Note that these assumptions might be met, respectively, through (1) the adoption of a configurable contract-violation handler, such as that proposed by [P2811R0], or (2) by allowing exceptions that escape the evaluation of a CCA's predicate to propagate to the caller, as might be considered based on some of the poll results during SG21's discussion of [P2751R0]. Either way, evaluating a CCA might generate an uncaught exception.[5]

Recall that, according to the definition of a *nonthrowing* exception specification for a function (or constructor), any thrown exception that escapes from its body, function `try` block, or (for constructors only) initializer list will result in a call to `std::terminate`. Our next step will be to consider the interaction of precondition checks (which might throw) with the `noexcept` operator when applied on a function that — per its `noexcept` exception specification — is prohibited by the language from emitting an exception.

[3] Here we are considering, for example, that a possible distinction might exist between which templates get instantiated by a predicate that is potentially evaluated and one that, due to it being in an unchecked build, is effectively discarded. We lack concrete examples of such obscurity but think we cannot or should not act to prevent such possibilities that might already exist in any other similar situation where a function template with no realized evaluations is not instantiated.

[4] The current MVP mentions two build modes, each of which when chosen assigns a single semantic to all build modes: `No_eval` uses the *ignore* semantic) and `Eval_and_abort` uses the *enforce* semantic. We are aware that this approach will not easily scale to more contract-checking build modes nor multiple modes within a single TU, so we have chosen to instead refer to the four known CCA semantics: *ignore*/*assume* (unchecked) and *enforce*/*observe* (checked). This paper applies in its entirety irrespective of whether the *observe* and/or *assume* semantics are ever adopted by the Standard.

[5] For simplicity and correctness, we will refer to only this *checked* build mode, which allows an exception to be emitted, and the *unchecked* build mode, such as `No_eval`, which emits no exceptions. We will ignore, without loss of generality, that additional *checked* build modes do not enable exceptions to be thrown or to otherwise escape the CCA could exist.

For example, consider an arbitrary function, `my_func`, that takes a non-negative integer, `i`, and returns void:

```
void my_func(int i) [[pre: i >= 0]];
```

This function has a CCA that, in some build mode(s), checks to make sure that the argument `i` passed to `my_func` at run time is in fact non-negative, and if it is negative, the CCA might throw.

Whether the result of evaluating `my_func`'s precondition check is governed by the guarantees of the exception specification makes no difference here because `my_func` isn't declared to be `noexcept` and, hence, *doesn't* profess *not* to throw. (Recall that a destructor's exception-specification is the opposite in that it defaults to nonthrowing — even when user-defined.) Consequently, applying the `noexcept` operator to any invocation of `my_func` in any build mode will naturally evaluate to `false`:

```
int x;                                                  // Value is not used.
static_assert( false == noexcept(my_func(x)) );  // always well-formed
```

Note that the value of `x` is irrelevant because the `noexcept` operator is concerned with only whether any of the subexpressions of its argument can throw.

Now let's consider another function, `your_func`, that is the same as `my_func` but is declared `noexcept`:

```
void your_func(int i) noexcept [[pre: i >= 0]];
```

Whenever we apply the `noexcept` operator to an invocation of `your_func`, we would naturally expect it to evaluate to `true`:

```
int x;                                                   // Value is not used.
static_assert( true == noexcept(your_func(x)) );  // unchecked build
```

Neither assertion above cares what runtime arguments are passed to `my_func` or `your_func`. Because the predicate in the CCA above involves only a single scalar and thus invokes no functions that might throw, the result of applying the `noexcept` operator is determined entirely at compile time by the presence (or absence) of a `noexcept` specifier on `your_func` (or `my_func`).

Note that an exception emitted as the result of evaluating an argument to the function proper is not similarly problematic because, in all build modes, that exception must necessarily be thrown prior to invoking the function. On the other hand, *all* exceptions that are allowed to escape from a CCA — whether from evaluating the predicate itself or in response to a successfully detected contract violation — are equally problematic because *all* are similarly dependent on the contract-checking build mode (see Appendix A).

Let's now consider the conundrum that ensues in a checked build mode where (1) evaluation of a precondition check can result in a thrown exception being emitted from that CCA and (2) its evaluation is presumed to occur *prior* to invoking the function and thus is *not* subject to the guarantees associated with a function having a nonthrowing exception specification.

There is no mechanism in the language to observe whether the exception originates before or after the function begins to execute. Hence, in a *checked* build mode, invoking `your_func` will clearly be capable of emitting an exception. Therefore, if the precondition check is not within the purview of

the function's exception specification, no choice is available but for the `noexcept` operator to return `false`[6] in every checked build mode where that CCA might emit an exception:

```
int x;                                                  // Value is not used.
static_assert(  true == noexcept(your_func(x)) );  // unchecked build
static_assert( false == noexcept(your_func(x)) );  // checked build
```

That is, the `noexcept` operator behaves differently depending on whether we are in a *checked* or *unchecked* build mode!

To introduce this polemic against the notion of build-mode-dependent semantics for the `noexcept` operator, suppose that we have some generic function, `theirfunc<F>(F& function, int y)`, that, based on whether the supplied `function` is `noexcept`, executes one of two distinct algorithms, one being fast and the other slow (the slow algorithm is slow only because it does some extra cleanup):

```
class SystemState { /* ... */ };
const SystemState& getCurrentState();  // Retrieve current global state.
void setCurrentState(SystemState&&);   // Update global state.

template <typename F>
void theirfunc(F& function, int y)
    // Invoke the specified function with the specified y, guaranteeing
    // that, if function throws, the global system state will be preserved.
{
    if constexpr (noexcept(function(y)))  // fast algorithm
    {
        function(y);  // does not throw
    }
    else                                  // slow algorithm
    {
        SystemState origState = getCurrentState();  // Record original state.

        if (INT_MIN == y) ++y;  // pathological case
        y = std::abs(y);        // Guarantee y is non-negative.

        try
        {
            function(y);  // might throw
        }
        catch (...)
        {
            setCurrentState(std::move(origState));   // Restore original state.
            throw;  // Rethrow exception from function.
        }
    }
}
```

[6]One might think to ask if we could, hypothetically, define some other operator, say, operator `noexcept_except_contracts`, that answers the question as to whether a function can throw in cases *other* than contract checking violations. If we could, that would mean that a function could throw in four, not just two, ways: nothing, general exception, contract-violation exception, or both. The question then becomes what operator should one use in production code? It turns out that, regardless of the reason an exception might be thrown, *whether* it may be thrown is the question one needs to ask, or the code that's asking won't work.

In the example above, whenever we pass, as an argument to `theirfunc`, a function that has a *throwing* exception specification, the *slow* version of the implementation for `theirfunc` will be instantiated — irrespective of anything else. If the function argument type instead has a *nonthrowing* exception specification and *no* precondition (or postcondition) checks on its declaration, the *fast* version of the implementation will be instantiated. But if an argument function having a *nonthrowing* exception specification has, say, a precondition, then for `theirfunc` to be instantiated with the *fast* implementation, it *must* have been built in an *unchecked* build mode; otherwise, our decision to evaluate the precondition check *before* `theirfunc` is invoked will force `their_func` to be instantiated with the *slow* implementation.

At the risk of being overly repetitive, having the semantics of commonly used C++ language constructs be dependent on the contract-checking build mode is anathema to any sensible notion of sound software engineering practice. To drive this point home, we need go no further than to create a couple of simple applications that illustrate the potential pitfall we're striving to avoid.

Let's first consider `My Application`, which uses the functionality presented above to invoke `their_func` on `my_func` with the value `-5`:

```
// My Application (my_app)
int main()
{
    their_func(my_func, -5); // my_func has a throwing exception spec.
}
```

Because `my_func` has a *throwing* exception specification, it will never have the *fast* path within `theirfunc` applied to its invocation in *any* build mode. Hence, the extra adjustments made to the inputs on the slow path will ensure that this application will never experience an out-of-contract call to `my_func`.

In contrast, let's now consider `Your Application`, which instead invokes `theirfunc` on `your_func`, again with the value `-5`:

```
// Your Application (your_app)
int main()
{
    their_func(your_func, -5); // your_func has a nonthrowing exception spec.
}
```

In this case, because it has a *nonthrowing* exception specification, `your_func` will cause the *fast* algorithm to be instantiated if and only if compiled in an *unchecked* build mode. That is, in a *checked* build mode, the invocation of `your_func` will be determined by the `noexcept` *operator* to be throwing, the *slow* algorithm will be instantiated for the implementation of `their_func`, and, hence, `your_func` will be safely called *in contract*.

Importantly, the *slow* algorithm *adjusts* the value passed to it to ensure that the value is non-negative and will thus always satisfy the precondition checks of both `my_func` and `your_func`. The *fast* algorithm, on the other hand, skips the adjustment step, and thus, in an *unchecked* build, `your_func` *will* be called *out of contract*, resulting in (at least) *library* undefined behavior and, in any event, a latent defect.

Catastrophically, this bug remains undetected even after switching to a *checked* build since the distinct code path followed to handle exception safety inadvertently provides substantially different — i.e., sufficiently constrained — values to the specified function, masking the defect (and resulting undefined behavior) in such builds.

Bottom line: Our decision to evaluate an associated precondition before (or postcondition after) the invocation of a function having a nonthrowing exception specification just to skirt the nonthrowing exception specification in checked builds isn't just dubious, it's untenable; the only viable alternative is that the exception specification observable from the declaration of a function governs the behavior of an applied `noexcept` operator — or any other compile-time query (see Appendix A) — precisely the same way in *every* contract-checking build mode.

2 Proposals

First, we restate with more clarity the founding principle which will guide the decisions we make in further proposals, and for which we hope we have, by now, sufficiently described the motivation.

Principle 1: Build-Mode Independence

A function having a CCA, especially one associated with its declaration, will not result in it having *proximate* compile-time semantics — e.g., with respect to the `noexcept` operator (or any other compile-time query) — that vary across contract-checking build modes. In particular, if a function having a CCA is well-formed when that annotation is compiled, then the `noexcept` operator will have the *same* behavior in all build modes, i.e., regardless of whether the CCA has been compile to have *ignore* or *enforce* semantics.

Recall that, by *proximate*, we mean the effect (e.g., of a CCA) on adjacent language constructs is a direct and inevitable consequence (e.g., of the current build mode) and not an accidental interaction resulting from global side effects — such as varying ODR-use leading to different overload sets — that might arise in more rare, if not pathological, cases. This same principle applies to other similar situations, e.g., involving traits; see Appendix A.2.

Also consider that we must allow for the possibility that some build modes might render a program ill formed, while others might leave it well-formed. For example, consider that the compile-time evaluation of a `constexpr` function that results in a contract violation might be well-formed (but have a latent logical defect) in an *unchecked* build mode yet be ill formed (exposing the defect) in a *checked* build. We'll need to flesh out the subtleties of such compile-time evaluations of functions having CCAs (see Appendix A). Nonetheless, for the subset of build modes for which a program is well-formed, the *proximate* compile-time semantics of all language constructs *must* remain the same.

Being compelled to adopt this first proposal leads us to another well-known principle.[7]

[7] See [N3279].

Principle 2: The Lakos Rule

Nonthrowing exception specifications and narrow contracts are inherently incompatible and contradictory.

Note that utility of the Lakos Rule transcends the inability for a function having a nonthrowing exception specification to fully support a robust contract checking facility (including one that might throw) and applies to the practical maintainability of software in general.[8] When presented with functions that violate this principle (perhaps even for sound engineering reasons), we must define how the language will respond. Given that the behavior of the `noexcept` operator *must* be the same across all build modes, exactly two fundamentally distinct alternatives are available from which we may choose.

1. Codify the Lakos Rule directly into the language. Adding a precondition or postcondition check to the declaration of a function that would otherwise have a nonthrowing exception specification makes that function automatically *not* `noexcept` — i.e., its exception specification silently becomes `noexcept(false)` — in *all* build modes; alternatively, we might even choose to make the function simply ill formed.

2. Allow a `noexcept` function to have arbitrary CCAs. If a CCA throws, however, the result is a call to `std::terminate`, just as if the exception had been thrown from an `[[assert ...]]` inside the body of the function or constructor.

The first alternative is draconian, brittle, and inconsistent with other well-reasoned language decisions at the heart of the design of C++. Having the exception specification silently change to nonthrowing when any CCA is added to its declaration would be surprising and error prone, to say the least, and doesn't address what's supposed to happen if a CCA is placed in the (perhaps nonvisible) body of said function.

Another entirely separate reason, however, arises for not wanting the addition of CCA to affect the *proximate* semantics of the *noexcept* specifier even if that change would apply to all build modes.

Principle 3: Zero Overhead for Ignored Predicates

When the predicate of a CCA is *ignored*, the generated code *proximate* to the CCA is *as if* the CCA had been commented out, i.e., removed from the source entirely.

Violating the zero-overhead principle for CCAs above would imply that just having a contract check in the source code — even when not built to be checked at run time — could potentially degrade performance and thereby generally discourage widespread adoption of our burgeoning contract-checking facility. In particular, we require that neither run time nor the layout of any object be affected by the addition of any CCA. Note that, to ensure equivalent semantics, anything referenced in a contract check is ODR-used in *all* build modes. Hence, we do acknowledge the possibility of additional template instantiations that result from adding a contract check, even when the entire program is built to have the *ignore* semantics.

The alternative — i.e., making the addition of a precondition check to a nonthrowing function ill

[8] See [P2861R0].

formed — would mean that those who want to use `noexcept` on a narrow function (for whatever reason) would be denied the benefits of contract checking — not even forced termination. In particular, Standard-Library vendors who choose to strengthen their exception specification on narrow contracts and still want to provide attenuated support for contract checking (i.e., all but throwing handlers) would be denied that opportunity. *Why?*

Consider that C++ already provides multiple instances of explicit syntax for declaring a property that would enable prospective clients to exploit that property, regardless of whether the property is statically provable or even strictly true but which will be true — or close enough to true — in practice.

Let's suppose we have a function whose contract states that it has no side effects (i.e., that it is *pure*), and this property can somehow be made accessible programmatically at compile time to generic clients.[9] These clients would, of course, be free to skip calling said function if they have no need for the result or can similarly exploit this *pure* property — e.g., cached result for given arguments.

Now suppose that a developer, experimenting in a local sandbox, adds a print statement within that function. Should that function now be considered ill formed[10] because the compiler can observe that it might have a side effect?

Such brittleness resulting from draconian compiler enforcement of user-declared properties[11] has been demonstrated repeatedly to be counterproductive as we'll expound upon below. Designed properly, the function will still compile and, in practice, behave correctly, the worst case being that the function was elided since it deliberately but innocuously misrepresented the truth about having no side effect, such as printing.

Similarly, when we put `noexcept` on a function, we are *not* necessarily claiming that there is no possibility of anything in the body of the function ever throwing. We *are* allowing the client to use this function in a context that requires knowing programmatically that no exception will ever escape from that function. Hence, if something inside the body of the function does throw, the natural consequence is program termination.

Developers have legitimate engineering reasons for wanting to represent a stronger exception specification than what might hold in general. For example, imagine that we have an object for which there is no nonthrowing move constructor, yet we know that the only way a copy operation can throw would be due to system memory exhaustion. We want to be able to place objects of this type in standard containers, such as `std::vector`, and achieve the full efficiency that we would realize with a type sporting a move constructor having a nonthrowing exception specification.

As the owners of the application, we know that we have ample memory and therefore are unconcerned with exceeding system-memory limits. Moreover, if our application is not safety critical and instead does batch work where the priority is maximizing throughput, we might actively prefer that, should

[9] See, e.g., [N3744] and [P0078R0].

[10] We acknowledge that this analogy is not strictly applicable in that purity is not generally provable (decidable), whereas whether a function having a nonthrowing exception specification has CCAs (at least on its declaration) is trivial for the compiler to check at the call site.

[11] The original design of Concepts for C++0X required that all concepts declared by the user be verified by the compiler.

our memory limits somehow be exceeded, the application crash loudly rather than degrade quietly. Hence, providing a generic wrapper that exposes a `noexcept` move operation offers real-world practical economic value.[12]

In the interest of avoiding historical mistakes, note that the initial design of C++11 specified as ill formed an explicit exception specification that did not match the one produced when defaulting a special member function. That brittle behavior was replaced in 2014, so that such a special member function would be well-formed but *deleted*. Implementing this new behavior proved problematic, and besides, the Standards Committee recognized that valid use cases did exist for telling a white lie to the compiler (see above), so in 2019, the behavior changed to what it is today: An explicit exception specification takes precedence over the defaulted one.[13]

Choosing to inseparably couple the existence of a nonthrowing exception specification with the absence of precondition and postcondition checks (and vice versa) would do a grave disservice to C++ users since a developer might reasonably want to have some of the benefits of both. That is, although not every action of a general-purpose contract-violation handler can be supported for a function having a nonthrowing exception specification, some can, which might be better than nothing in some cases. Unilaterally forcing everyone always to choose between (perhaps *algorithmically* faster) nonthrowing exception specifications and fully functional contract checks is not something that even John Lakos himself would ever want to see.[14]

Codifying the Lakos Rule into the language such that nonthrowing exception specifications are syntactically incompatible with preconditions or postconditions is unnecessary and would serve only to occasionally impede effective software design. This observation leads us to the second alternative, our second proposal, and the ultimate conclusion of this paper.

Proposal 1: `noexcept` Functions[15]

After the initialization of function arguments, any further steps in the evaluation of a `noexcept` function that emits an unhandled exception will follow [except.spec]p5, resulting in the invocation of `std::terminate`.

This second proposal is merely a special case of the first one; anything else would be new territory and necessarily contradict our underlying principal requiring the same *as-if* behavior across all build modes.[16]

Finally, this result does not preclude implementations from providing the caller's `source_location` to the violation handler when a precondition is violated, and thus some of the goals of [P2780R0] can still be achieved.

[12] See [lakos21], Section 3.1."`noexcept` Specifier," "Use Cases," "A wrapper that provides `noexcept` move operations," pp. 1099–1101.

[13] See [lakos21], Section 3.1."`noexcept` Specifier," "Description," "Unconditional Exception Specifications," pp. 1085–1089, specifically footnote 2, p. 1086.

[14] See [P2861R0].

[15] SG21 consensus at the 2023-05-19 telecon was to adopt this proposal.

[16] A third alternative would be to treat throwing from a handler as forced termination – even for functions having a *throwing* exception specification. Although such an approach would not be satisfactory for people who want to make use of throwing from the handler, it would nonetheless be self consistent.

3 Conclusion

The long-standing debate about whether a precondition check behaves as if it is evaluated *before* or *after* a function is invoked has finally been resolved — here and now — in favor of the latter, yet this result still allows for collecting the caller's source location when a violation occurs. The dispositive proof we have exhibited comes down to the absolute need to avoid dangerous pitfalls that arise from allowing the compile-time semantics of language constructs *proximate* to a contract-checking annotation (CCA) to vary, in programmatically detectable ways, from one contract-checking build mode to another.

The efficacy of a Contracts facility as a tool for detecting software defects and improving safety demands that any defects resulting in unintended (e.g., undefined) behavior that would manifest in an *unchecked* build are not somehow inadvertently hidden or otherwise rendered harmless in a *checked* build. At the same time, we must avoid design choices that — while consistent — might affect runtime performances even when the program is built in a mode where all CCAs are ignored, thereby presenting a strong disincentive to widespread adoption. Making the wrong choice with respect to how CCAs interact with exception specifications can easily lead to such unfortunate scenarios.

To demonstrate why a function having a nonthrowing exception specification must *never* allow an exception to escape from *any* CCA in *any* build mode, we assumed the contrary — namely, that an exception emitted from a CCA *can* bypass a nonthrowing exception specification on a function. We then created two similar functions (`my_func` and `your_func`), each taking a non-negative integer argument guarded by a precondition check but with only the latter declared `noexcept`.

Next we exhibited a generic function (`their_func`) that chooses a more efficient, *fast* path when the `noexcept` operator, applied to its function parameter, indicated that invoking that function argument would not throw an exception. We observe that, for the first function, the slow path was always instantiated, whereas for the second function, which is `noexcept`, the fast path was instantiated *if and only if* compiled in an *unchecked* build, thus exhibiting dramatically different code paths across distinct contract-checking build modes.

We then created two applications (`my_app` and `your_app`), each invoking its respective function on an invalid input. An abundance of caution along the *slow* path of the generic function resulted in suppressing a defect (undefined behavior), while the *fast* path skipped this corrective step, thereby allowing that defect to manifest. To our horror, enabling runtime checking failed to detect this defect, while disabling runtime checking removed the correcting code and restored the catastrophic bug.

Our hypothetical decision to allow an exception to escape from the CCA of a function having a `noxcept` specifier caused the variation in the *proximate* semantics of the `noexcept` *operator*, which in turn enabled the pitfall of creating and masking a latent defect (undefined behavior). We therefore concluded that we must never allow the `noexcept` operator to behave differently across contract-checking build modes and thus that we cannot allow an exception to escape from *any* CCA for a function having a nonthrowing exception specification.

After completing this *proof by contradiction*, we were faced with exactly two alternatives: (1) codify the Lakos Rule in the language, making a nonthrowing exception specification syntactically

incompatible with precondition and postcondition checks or (2) treat every CCA as being subject to the same exception specification as the function itself.

We then proceeded to show why the former alternative is needlessly draconian, brittle, and not in keeping with the design practices typical of modern C++. We also noted that any design choice that achieved consistency at the expense of runtime overhead for even *ignored* CCAs would serve only to stifle widespread adoption of this burgeoning Contracts facility.

Using detailed examples, we demonstrated that compiler enforcement of similar user-specified properties, such as *purity* (having side effects), would — even where practicable — be woefully counterproductive. We also observed that the original choice to prevent users from explicitly changing default exception specifications on special member functions was notoriously ill-advised.

We therefore embraced the latter approach with the clear-eyed understanding that an exception emitted from any CCA associated with a function having a nonthrowing exception specification will necessarily result in forced program termination. We also concluded that the general advice of not allowing proximate compile-time semantics to differ among well-formed programs across various build modes was helpful in guiding us to a successful resolution of other, related issues (see Appendix A).

In short, the evaluation of *all* CCAs shall be treated *as if* occurring *after* the invocation of the function and *before* it returns and thus subject to the same guarantees afforded by the exception specification of that function, irrespective of the contract-checking build mode. To do otherwise is to forgo the significant benefits in consistency, correctness, safety, and performance that our C++ Contracts facility seeks to provide.

A Related Considerations

The primary purpose of this paper was to reiterate the absolute requirement that exceptions are never to be emitted when invoking a function having a nonthrowing exception specification in any contract-checking build mode. While writing this paper, we encountered several related issues, some of which we've already touched on briefly. This appendix will now introduce several of these related topics, which we'll need to address as they become relevant.

A.1 Exceptions emitted directly from a CCA's predicate

In the discussions so far, we've made no distinction as to whether an exception emitted from a CCA results (1) *directly* from evaluating the CCA's predicate or (2) *indirectly* as a consequence of that predicate evaluating to `false` and then, as part of the process of handling a contract violation, an exception being explicitly thrown.

Recall that, unlike the evaluation of arguments to a function proper, which are always necessarily evaluated prior to invoking the function in every build mode, allowing an exception to emanate directly from the evaluation of a precondition check's predicate produces precisely the same inconsistency in *proximate* compile-time behavior across checked and unchecked build modes that would have occurred had the exception been thrown indirectly, e.g., via a configurable violation handler.

In fact, allowing an exception to propagate back to the client directly from the evaluation of a CCA's predicate itself would pose precisely the same problem as with a throwing handler, and this would be true even absent a build mode in which contract violations are handled via throwing an exception. Hence, we must conclude that no exception thrown as the result of merely evaluating the predicate of a CCA itself may ever be allowed to escape back to the caller of a function having a nonthrowing exception specification.

One simple approach — the one employed by the current MVP[17] — that avoids the creation of new control paths when evaluating CCA's predicate throws is to always catch it and call `std::terminate`, even for functions having *throwing* exception specifications. Alternatively, we can restore some of the flexibility to handle exceptions thrown directly from a CCA's predicate — in a disciplined manner — by treating these exceptions as contract violations as recommended by Proposal 2.3 in [P2751R0], enabling a custom contract-violation handler to be aware of this dichotomy and thus, when desired, to choose to rethrow the exception, as described in Section 4.3 of [P2811R1].

SG21 consensus as of the 2023-05-18 telecon was to adopt the proposal in [P2751R0] and treat exceptiosn thrown from a CCA predicate as contract violations.

A.2 Trivial functions that have preconditions or postconditions

A trivial operation — constructor, destructor, or assignment operator — is one that is (1) 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.

Providing a body — even an empty one — for a special member function never results in the special member function being trivial. Defaulting a user-declared special member function (via `= default` on the first declaration) will result in its 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* (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.[18]

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

[17] See [P2695R1].

[18] Feel free to ask the authors for war stories about the pain that comes from developers who decided 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 non-trivial 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.

```
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 member function `isValid` to *ensure* (or perhaps just *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.[19]

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() = default [[ pre: isValidYMD(d_year, d_month, d_day)]];
  // Destroy this object.
```

Notice that the precondition itself applies the public class member function, `isValid`, to the private data members, e.g., `d_year`, of the class. The Committee has long since agreed[20] that all CCAs are part of a function's implementation; hence, when a CCA is associated with a member function, it fairly deserves private access to the data members of the class.

In an *unchecked* build, the destructor does nothing, and because there is no user-supplied definition, 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*[21] trivially destructible and thus, at least *notionally*, trivially copyable.

Absent the seminal information presented earlier in this paper, we might be tempted to treat this `Date` class *as if* it were *trivially destructible* and *trivially copyable* in an *unchecked* build but *not* in a *checked* build.

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

[20] During a Standards Committee meeting (c. 2015) that involved CCAs, Bjarne Stroustrup expressed the need for a CCA on a member function to have private access to the implementation of its class so that it 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 discussed and achieved consensus in November 2018. This result has remained the SG21 consensus, as indicated in Section 4.3 of [P2521R3].

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

Let's now consider the problems that would manifest should the `memcpy` invoked by a library for a trivially copyable type have an off-by-one error in the range it chooses to copy:

```
<template type T>
fastCopy(T *dst, T* src, std::size_t n)
{
    if constexpr (std::is_trivially_copyable<T>::value) // fast algorithm
        std::memcpy(dst, src, (n + 1) * sizeof * dst);  // Oops, off by one!
    else                                                // slow algorithm
        for (int i = 0; i < n; ++i) *dst == *src*;      // OK
}
```

If we were to assume that `Date` were trivially copyable in *only* an *unchecked* build, then in a *checked* build, we'd again get the slow but correct algorithm, whereas, in an *unchecked* build, we'd get the fast and incorrect one, thus violating our first principle — i.e., that *proximate* compile-time semantics (e.g., of the `std::is_trivially_copyable` trait) must *not* vary across (e.g., contract-checking) build modes.

Moreover, if enabling contract checking skipped this problematic, fast code path entirely, then the tools we use to diagnose such problems — e.g., debuggers, core file analysis, and even contract checks themselves — would be analyzing a program state where the problem will not occur. The only sound conclusion is again that *proximate* compile-time semantics must be invariant across *all* build modes.

Once again, we are left two alternatives.

1. Specify that defaulted special member functions that have associated CCAs are never trivial in any build mode. The implication is that just having an inactive precondition could substantially impact performance[22] even in an unchecked (e.g., *ignored*) build.

2. 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 executing the check might have produced.) If, however, the compiler would be the one 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` could be substantial, perhaps even dramatic, and would introduce a strong disincentive to adding such defensive checks to otherwise trivial functions, thus violating Principle 3. Hence, we do not consider the first alternative to be viable.

[22] 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, being trivially copyable gives the compiler special permission to bypass calling its copy constructor. 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 it trivial, nor does it give license to the *library* to use `memmove` for a type that would otherwise be *trivially copyable* but for its almost trivial destructor.

The second 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. This latter approach does come with the risk of perhaps omitting checks that an uninitiated author might have intended to be run by the client in every case — i.e., including even those cases where 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.
```

Just by providing an empty function body (see the example above), a function that was otherwise *trivial* can easily be made non-trivial 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 needed 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.

Again, compiler optimizations would be treated quite differently. A compiler is permitted to initiate an optimization *only* when it can prove that the resulting optimized code has the same *as-if* (defined) behavior as the original code. Hence, a compiler optimization that replaced a manual `for` loop with, say, a `memmove` in an *unchecked* build might — due to `CCA` on a `trivial` member function of the type of object being copied — be unable to make that same optimization in a *checked* build and still reliably provide the expected checks. In that case, the compiler would be forced to forgo that optimization, much the same as if we had removed `-O2` from the command line.

For example, suppose we wanted to add a defensive precondition check to the copy-assignment operator of the `Date` class to verify redundantly that the class invariants initially hold:

```
Date& operator=(const Date& date) = default
                                    [[ pre: isValid(d_year, d_month, d_day)]];
  // Assign to this object the value of specified Date object.
```

If the compiler can prove at compile time that it can provide the requisite checks and still perform the `memmove` optimization, it is free — but under no *obligation* — to do so.

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 be able to 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.

Putting these suggestions together, we arrive at one proposal regarding the triviality of special member functions with contract-checking annotations:

Proposal 2: CCAs can be Trivial

A special member function with preconditions or postconditions may be *trivial*. Note: this may result in situations where the preconditions or postconditions are not checked due to the invocation of the special member function being skipped or replaced by a bitwise copy.

A.3 Implicit lambda captures from CCAs

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 an expression in a lambda ODR-uses a local entity. In such cases, that entity will be implicitly captured as part of the generated closure object, thus affecting its size.

Now consider that the predicate of a CCA within a lambda will involve an expression. If that predicate is allowed to do such implicit by-value capture, it has the potential to affect not only the size of the closure but also the lifetime of the captured object.

First, suppose we have simple `struct`, `S`:

```
struct S {
    int d_x;
};
```

Now imagine a function `f` that takes a shared pointer to an `S`, `sp`, by value and returns, also by value, a `lambda` converted to an `std::function` that takes no arguments and returns an `int`:

```
std::function<int()> f(std::shared_ptr<S> sp)
{
  int *x = &sp->d_x;  // Raw pointer x is invalidated when *sp is destroyed.

  return [=]()        // lambda having by-value capture default
  [[ pre : sp ]]      // precondition that odr-uses sp
  {
    return *x;        // lambda body that does not  odr-use sp
  };
}
```

Notice that the raw pointer `x` is extracted first and is captured in the body of the `lambda`. Also notice that the precondition check captures the `std::shared_ptr<S>`, `sp`, by value, thus extending the lifetime of the referenced `S` object.

When checking is enabled, allowing the capture of `sp` will keep the referenced `S` object alive as long as the lambda is alive, thus making the raw pointer `x` remain valid even after `f` returns (and `sp` is destroyed). If disabling checking were to remove that capture, any invocation of the lambda returned from `f` will occur *after* the `sp` has been destroyed. Depending on the rest of the program, that might mean that no remaining shared pointers are keeping the `S` object alive, and thus any access via that pointer would result in (language) undefined behavior. The closure size would be different too.

By now we should all clearly understand that we simply cannot allow observably different program behaviors across contract-checking build modes as the result of adding any CCA. To guard against

such defects, we yet again have just two alternatives.

1. An ODR-use of a local entity from a CCA's predicate will capture the named local entity in every build mode, i.e., even when contract-checking is disabled.
2. An ODR-use of a local entity from a CCA's predicate will be ill-formed if that same entity is not already ODR-used by the lambda from outside of a CCA predicate.

Choosing the first alternative would violate Principle 3, ensuring that there is absolutely no runtime or object-size overhead associated with any contract checks when those contract checks are *ignored*. Even the relatively small spacial overhead associated with adopting the first alternative might tend to discourage adoption of contracts in general. Hence, we consider this first option to be nonviable.

The second alternative — i.e., making it ill-formed to ODR-use a variable 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 makes abundantly clear to users when they are referencing a value that is not directly relevant to the body of the lambda. The workaround, for those who want to capture that dubious value anyway, is simply to add the appropriate ODR-use directly within the body of the lambda:

```
std::function<int()> f(std::shared_ptr<S> sp)
{
  int *x = &sp->d_x;  // Raw pointer x is invalidated when *sp is destroyed.

  return [=]()        // lambda having by-value capture default
  [[ pre : sp ]]      // precondition that ODR-uses sp
  {
    (void) sp;        // sp is now ODR-used in all build modes.
    return *x;        // lambda body that now does odr-use sp
  };
}
```

Such explicit ODR-use of `(void) sp` (the `void` suppresses unused warnings) in the body of the lambda will force the corresponding capture in *all* build modes. This workaround is analogous to adding a user-provided empty definition to replace a trivial destructor to suppress optimizing away a check for a notionally trivial function (see Section A.2).

Another example to consider is where an additional implicit capture that a CCA might introduce would have significant performance impact that should only be taken on with careful consideration, even in a checked build mode. Consider a situation where a potentially large container would be inadvertently copied by such a capture:

```
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
  }
}
```

Should `v` be captured by the closure object returned by `foo` the fundamental behavior of `foo` would be changed — it would change from something that likely has a constant-time algorithmic performance to one with linear time requirements, it would begin to allocate memory where previously it did not, and all of this for a property that could easily be checked once after the initialization of `ndx` and not on each invocation of the function object. Making this function ill-formed removes the risk of inferior implementations resulting from an innocuous-seeming contract check.

This all comes together in one proposal regarding lambdas:

Proposal 3: CCA ODR use does not Implicitly Capture

Within the expression of a CCA that is attached to or within a lambda, the odr-use of a local entity does *not* implicitly capture that entity.

The reference to the local entity from the CCA will still continue to instead attempt to name the member of the closure object instead of the local entity itself. Therefore, if there is no other factor that 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.

A.4 Contract violations that occur at compile time

This final section on compile-time contract violations is a somewhat more esoteric yet related topic to what we have seen so far, and thus we have incorporated it here. We will try to apply the same sorts of principles we've used previously, although the results are somewhat less elegant. Every program that is well-formed in a checked build is also well-formed in an unchecked one but not vice versa. In particular, some programs that are unchecked might compile and have latent defects yet fail to compile due to those very defects. At a minimum, we must ensure that all well-formed programs have the same behavior in all contract-checking build modes and that checked builds that fail to compile help us weed out additional bugs.

The C++ language provides us two primary decorators to enable constant evaluation for functions (and constructors): `constexpr` and `consteval`. Certain operations performed within the implementation of a `constexpr` or `consteval` function — such as invoking an ordinary function, e.g., `h()` in the code snippet below — make the function no longer eligible to be evaluated at compile time (see [expr.const] in the Standard):

```
void h() { };                                // not compile-time evaluable

constexpr int ff(int x) { h(); return x; }  // usable only at run time
consteval int gg(int x) { h(); return x; }  // well-formed but not usable

constexpr int f(int x) { if (x) h(); return x; }  // compile and run time both
consteval int g(int x) { if (x) h(); return x; }  // compile time only
```

In the definitions above, function `ff`, which is declared to be `constexpr`, calls `h()` for every value of `x`; hence, there is no input for which it can be evaluated at compile time. Function `gg` is declared to be `consteval` so it cannot be evaluated at run time; hence, it's useless. Both `f` and `g` have a branch, so they don't call `h()` unconditionally. Either one can be called with `0` at compile time. Since `f` is

declared `constexpr`, it can be called at either compile time (with 0) or run time (with anything else), but because `g` is declared `consteval`, it can be called only at compile time (with `0`).

Note that, as of C++23,[23] only the failed *use* of either of these functions in a *manifestly constant evaluated* expression is ill-formed; we can think of a `consteval` function as if it were a `constexpr` function that can be invoked only in manifestly constant evaluated expressions.

Another useful decorator (for variables only) is `constinit`, which says that the evaluation of the initialization of the variable *must* occur at compile time (i.e., the initializer is a *manifestly constant evaluated* expression), but the variable itself is not `const` or `constexpr`:

```
constinit int v  = 5;     // OK,           manifestly constant evaluated
const     int v0 = f(v);  // OK,       not manifestly constant evaluated
const     int v1 = f(5);  // OK,       not manifestly constant evaluated
constexpr int v2 = f(5);  // Error,        manifestly constant evaluated
constinit int v3 = f(5);  // Error,        manifestly constant evaluated
          int v4 = g(5);  // Error, always manifestly constant evaluated
```

In the definitions above, `v` is initialized at compile time to be a modifiable global variable. Both `v0` and `v1` are `const` but are not required to be evaluated at compile time, so they are potentially runtime evaluated, yet the compiler is allowed to optimize if it has visibility into the bodies of `f` and `h`. The (`const`) `v3` and (non`coonst`) `v4` are each required to be initialized at compile time due to the respective `constexpr` and `constinit` but cannot be when passed `5` because `h()` is not compile-time evaluable; hence, both are compile-time errors. Finally, (non`const`) `v4` is initialized using `g`, which is declared `consteval` and hence always manifestly constant evaluated; since `g` can be compile-time evaluated only for an argument of `0`, it too is a compile-time error.

When invoked, the compiler might try to evaluate a `constexpr` function, such as `f` on its specific arguments at compile time; for a `consteval` function, such as `g`, it *must*:

```
int v5 = f(v);  // Evaluated at run time, v is not a compile-time constant.
int v6 = f(5);  // Evaluated at run time, f(5) isn't evaluable at compile time.
int v7 = f(0);  // Perhaps evaluated at compile time, f is constexpr.
int v8 = g(0);  // To be evaluated at compile time, g is consteval.
```

Adding a CCA to a function that is decorated with `constexper` or `consteval` might, depending on its inputs, impact the potential compile-time evaluation of such a function input at compile time in either of two ways.

1. The predicate is compile-time evaluable and fails to evaluate to `true`, thereby indicating that a contract violation has occurred at compile time.
2. The predicate might itself be ineligible to be evaluated at compile time, e.g., by invoking a non-compile-time-evaluable function or accessing global non`constexpr` state.

If a CCA is enabled, evaluated in a manifestly constant evaluated expression, and observed (at compile time) to not evaluate to `true`, then the program is straight-up ill-formed for at least two reasons, irrespective of whether the function itself is compile-time evaluable:

[23] See [P2448R2].

1. Attempting to provide some sort of alternative defined behavior for a compile-time contract violation would risk running afoul of our first principle by allowing two distinct build modes to produce well-formed programs having distinct internal paths and, quite possibly, observably distinct behaviors. By making ill-formed a compile-time contract violation (i.e., the predicate of the CCA evaluates to something other than `true` at compile time), the behavior of a successfully compiled program is the same in every contract-checking build mode, and we detect the violation in *every checked* build mode, thereby staying consistent with our principles.
2. How we handle a contract violation will be, at least in part, a link-time decision; hence, we cannot in practice evaluate the contract violation at compile time. Fortunately, by the previous item, we have no desire (and thus no need) to support making a compile-time contract violation well formed.

As one would expect, when a CCA successfully evaluates to `true` at compile time, it has no semantic effect whatsoever.

Next let's consider a non-compile-time-evaluable predicate, `b()`:

```
bool b() { return true; }  // non-compile-time-evaluable function
```

Let's suppose we were to add a non-compile-time-evaluable CCA to the `constexpr` function `f` from the previous example:

```
constexpr int f(int x) [[ pre: b(); ]] { if (x) h(); return x; }
```

Let's now consider the implications of invoking this new `f` in three different contexts.

1. `f` is not compile-time evaluable even in an unchecked build mode:

   ```
   int a = f(0);            // f is evaluated at run time in all build modes.
   ```

2. `f` is required to be compile-time evaluable in an *unchecked* build mode:

   ```
   constexpr int b = f(5);  // f is evaluated at compile time in an
                            // unchecked build mode and ill-formed otherwise.
   ```

3. `f` is optionally compile-time evaluable in an *unchecked* build mode:

   ```
   int c = f(5);            // f is evaluated at compile time in an
                            // unchecked build mode and at run time otherwise.
   ```

Case 1 presents no issue since the function is always evaluated at run time, regardless of whether the CCA happens to be evaluable at run time.

In case 2, the program is well-formed only in an *unchecked* build mode simply because no other viable options are available. We can't put off the evaluation of the function until run time because it is explicitly needed (required to be evaluated) at compile time. Silently skipping the check would open up its own safety pitfall, one that would be at odds with the remit of this entire feature. The only other option would be to make using the new function in a *manifestly constant evaluated* expression ill formed in *every* build mode but that would (1) require checking the predicate to determine whether it was compile-time evaluable with the given arguments even in an unchecked build and (2) block perfectly valid use in an *unchecked* build.

Choosing to make the check ill-formed in only *checked* builds has several (largely positive) implications.

- Doing so (at least) informs the caller that the check is not fully compatible with the self-declared compile-time-evaluable nature of the function being provided.
- However the ill-formedness of the evaluation of a function is determined, it should be a hard error — i.e., not one which can be detected with SFINAE or as part of a concept check —- to attempt to evaluate the predicate of a CCA in a manifestly constant-evaluated context and then have the predicate evaluation prevent the evaluation from being considered a constant expression.

  In other words, the following code should be ill-formed (given the `f` defined above):[24]

  ```
  template<int x> concept C = requires { std::bool_constant<f(x)>(); };
  constexpr bool P = C<5>;  // Could be false in checked builds, but
                            // evaluation of f fails to be a  constant
                            // expression because of a CCA predicate.
  ```

- We do, however, narrow the set of programs that are valid in checked builds, indicating that the nature of this check may preclude its use in valid situations.

The general guidance would be that, for functions declared to be `constexpr` or `consteval`, the compile-time-evaluable domain of the predicate should contain that of the function itself. In those vanishingly rare cases where some or all of the compile-time domain simply cannot be checked in the predicate, we offer another workaround, which enables the implementer of the predicate to opt in to allowing the CCA to be skipped when the alternative would be to leave the program ill-formed:

```
template <typename F>
constexpr bool runtime_only(F&& predicate)
{
    if consteval {
        return true;  // Skip test if during manifestly constant evaluation.
    }
    else {
        return predicate();  // checked at run time
    }
}
```

Now the author can write a new predicate for the CCA for `f` accordingly:

```
constexpr int f(int x) [[ pre: runtime_only(b()); ]] { if (x) h(); return x; }
```

In case 3, the program is always well-formed but, in a checked build, evaluation will necessarily degrade from compile time to run time. Again, when compile-time evaluation is an optional optimization performed by the compiler, the choice to perform that optimization cannot affect any observable semantics. Because compile-time evaluation is not mandated for an expression that is not *manifestly constant evaluated*, under the *as if* rule the compiler is completely within its rights to skip that optimization, and in most cases, no semantic difference and little extra impact on runtime

[24] Thanks to Ed Catmur on the SG21 reflector for bringing this up (See http://lists.isocpp.org/sg21/2023/05/3774.php).

performance will occur (i.e., the function body will still evaluate to a constant that is assigned at run time after the contract check is evaluated).

A scenario does, however, arise in which runtime evaluation might lead to undesirable race conditions due to namespace-scope runtime static initialization across translation units (this is a bad idea). In the unlikely event that we would need to require that a function used to initialize a modifiable global variable be evaluated entirely at compile time, use of `constinit` gets that job done:

```
constinit int d = f(5);  // f is evaluated at compile time in an
                         // unchecked build mode and ill-formed otherwise.
```

Note that this third scenario applies only to `constexpr` functions since `consteval` functions do not have the option to degrade to runtime evaluation.

Finally we note that both `constexpr` and `consteval` imply `inline` and, to be usable at compile time, *require* visibility into the body. Hence, a CCA in the visible body of a compile-time evaluable function would have the same implications as one associated with the declaration, whereas such would not be the case for the same function whose body was not visible at the call site; that function, however, would simply not be compile-time evaluable.

Note that we have been discussing the behavior of *checked* and *ignored* CCAs. The fourth semantic, *assume*, brings with it nuance that has been more thoroughly explored during the addition of `[[assume]]` in C++23.[25] For assumed contract checks, we do not require that the compiler ever evaluate the CCA's predicate; ignoring a CCA is always a valid option, though possibly not the best available option, for a compiler to take when asked to assume it. Therefore, similar to how `[[assume]]` is treated in the Standard, whether assumed CCAs are checked at compile time should be unspecified. Like an *ignored* CCA, when used in a manifestly constant evaluated context, a violation will have no effect, although it might make a program ill-formed if the compiler does choose to (and can) evaluate the predicate at compile time. Further exploration of this particular subject should occur alongside future proposals to reintroduce the *assume* semantic to the C++ Contracts facility.

Putting this all together leads to one proposal that covers the handling of checking CCas during constant evaluation:

Proposal 4: Constant Evaluation of CCAs

If an expression or conversion is manfiestly constant-evaluated, it is ill-formed if the evaluation of the predicate of a CCA with a runtime-checked semantic disqualifies that expression from being a core constant expression.

Acknowledgements

Thanks to Andrzej Krzemieński, Marshall Clow, Timur Doumler, Mungo Gill, Ville Voutilainen, and Ed Catmur for feedback on earlier drafts of this paper.

[25] See [P1774R8], Section 4.6, "Behaviour of assumptions during constant evaluation."

Bibliography

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

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

[N3744] Walter E. Brown, "Proposing [[pure]]", 2013
http://wg21.link/N3744

[P0078R0] Karl-Étienne Perron, "The [[pure]] attribute", 2015
http://wg21.link/P0078R0

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

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

[P2448R2] Barry Revzin, "Relaxing some constexpr restrictions", 2022
http://wg21.link/P2448R2

[P2521R2] Andrzej Krzemieński, Gašper Ažman, Joshua Berne, Bronek Kozicki, Ryan McDougall, and Caleb Sunstrum, "Contract Support — Working Paper", 2022
http://wg21.link/P2521R2

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

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

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

[P2780R0] Ville Voutilainen, "Caller-side precondition checking, and Eval_and_throw", 2023
http://wg21.link/P2780R0

[P2811R0] Joshua Berne, "Contract Violation Handlers", 2023
http://wg21.link/P2811R0

[P2811R1] Joshua Berne, "Contract Violation Handlers", 2023
http://wg21.link/P2811R1

[P2861R0] John Lakos, "The Lakos Rule: Narrow Contracts And 'noexcept' Are Inherently Incompatible", 2023
http://wg21.link/P2861R0