Document: P4015r0
Date: 2026-02-16
Reply-to: Lisa Lippincott <lisa.e.lippincott@gmail.com>
Audience: Evolution Working Group

Enforcing Contract Conditions with Statements

Lisa Lippincott

Abstract

C++26 contracts come with an enforcement mechanism that provides to certain parties (function authors, function callers, and individual compilers, but not entire implementations or program builders) only a nebulous threat of contract enforcement, rather than certainty. This nebulous behavior is not only by design, but is in fact essential to the design. The uncertainty surrounding enforcement makes it practical to specify contract conditions as part of a function's declaration.

Nevertheless, there is clearly an unmet need for function authors to specify that contract conditions will be enforced. Various last-minute proposals have attempted to meet that need, but have foundered when they have attempted to specify concrete behaviors within a function declaration.

C++ is at its best when program behavior comes from statements, and declarations merely specify points of agreement between components. C++26 contracts in function declarations work because they specify agreed-upon conditions for transfer of control from one function to another, but do not specify any behavior of either function.

This suggests a design direction: couple C++26 function contracts, which specify agreed-upon conditions, to novel statements that provide concrete enforcement behavior. I explore that design direction here.

This is purely an informational paper. While this paper presents a possible change to C++, I oppose making such a change at the last minute for C++26, and I do not currently intend to pursue such a change for C++29.

1 The difference between a rule and an offer

When we specify the consequences of breaking a rule to a party bound by the rule, the rule becomes an offer.

We understand how this plays out in a legal context. "Do not park in the red zone" is a rule: the red zone is a no-parking zone. But "People who park in the red zone must pay $100" is an offer, not a rule: the red zone becomes an *expensive* parking zone.

This principle applies particularly strongly in software because we empower optimizers to accept such offers. Let's look at an example.

To begin, a programmer, "F," writes a function `foo` that could behave dangerously when given the wrong input. For concreteness, I'll use a classic example.

```
// foo.cpp
#include "foo.h"

char foo_buffer[64];

void foo( char[] text, std::size_t text_length )
```

```
   {
    memcpy( foo_buffer, text, text_length );
   }
```

Recognizing the danger, F adds a precondition to the declaration of `foo`. Using a hypothetical language feature intended to increase safety, F specifies in the declaration that the precondition must always be enforced.

```
// foo.h
#include <cstdlib>
extern char foo_buffer[64];

void foo( char[] text, std::size_t text_length )
   always_enforced_pre( text_length <= sizeof(foo_buffer) );
```

By using `always_enforced_pre` rather that simple `pre`, F has turned a rule, "Don't pass large values for `text_length`," into an offer, "`foo` will terminate execution when `text_length` is large." But F likely does not intend `foo` to be used as program-termination method, and may continue to understand the condition as a rule.

F then compiles the function into a dynamic library, distributes it, and people begin to use it. One such user is B, who writes a function `bar`.

```
// bar.cpp
#include "foo.h"

char bar_buffer[64];

void bar( char[] text, std::size_t text_length )
  {
   foo( text, text_length );

   if ( text_length <= sizeof(bar_buffer) )
      memcpy( bar_buffer, text, text_length );
  }
```

Perhaps B has forgotten about the precondition of `foo`, or perhaps B, seeing that the precondition is always enforced, has accepted the offer of termination. It really doesn't matter. But B's optimizing compiler is aware of the offer, and deems the test `text_length <= sizeof(bar_buffer)` redundant. It is elided from B's compiled code.

Meanwhile, other users of F's dynamic library complain about the short length allowed for messages. F thinks "They're right. I will increase the size of the buffer, satisfying my users with a backward-compatible change." Relaxing a rule is a backward-compatible change. But reneging on an already-accepted offer is not.

The denouement comes when users link B's program to F's updated library, and are exposed to a buffer overflow vulnerability — exactly the sort of problem that both F and B took pains to avoid. We can assign blame: by linking the two parts, users have violated the one-definition rule, and also F was mistaken in thinking the change was backward-compatible. But this is little solace to people faced with a vulnerability that seems to contradict a simple reading of either party's source code.

2 Using Statements to Produce Behavior

To avoid the problems above, we need to be clear about establishing rules, and avoid turning those rules into offers. Specifically:

— It's OK for two parties to agree upon rules governing their interaction.

— It's OK for either party to insist upon enforcing those rules.

— But it's not OK for either party to offer to enforce those rules.

The enforcement behavior isn't a problem. The problem comes from making enforcement part of the agreement.

We can navigate these waters by separating the agreement from the enforcement. Declarations will continue to use `pre` and `post` to specify agreed-upon rules. But where we wish to insist upon enforcement, we will use statements, hidden safely within function bodies. Enforcement will remain an implementation choice, not an offer.

2.1 Starting simple: `enforce_condition`

To start, let's invent a statement that simply enforces a condition. We will use its semantics to describe more complicated statements.

enforce_condition_statement:
`enforce_condition (` *conditional-expression* `) ;`

We can give this statement the simplest enforcing behavior: when executed, it evaluates the contained expression; contextually converts the result to `bool`; and, if the result is `false`, terminates program execution in an implementation-defined manner.

2.2 Incorporating preconditions by reference: `enforce_preconditions`

Using `enforce_condition` to enforce a function's preconditions would involve repeating those conditions, creating redundancy in the program source and introducing an avenue for error. To simplify this situation, we can introduce a statement that incorporates the preconditions by reference.

enforce_preconditions_statement:
`enforce_preconditions ;`

This statement has behavior equivalent to a sequence of `enforce_condition` statements, one for each precondition:

```
enforce_condition( first_precondition_expression );
enforce_condition( second_precondition_expression );
enforce_condition( third_precondition_expression );
// ...
```

The expressions are taken from the corresponding preconditions of the innermost[1] enclosing function, and given the same meanings. Specifically, identifiers in the expressions refer to the same entities they did in the precondition context, and have the same cv-qualified types.

The core use of `enforce_preconditions` is to place it at the beginning[2] of a function body. It will then guard the remainder of the body from violations of sufficiently stable preconditions. It may also be used in other contexts for later or conditional enforcement, as in

```
if constexpr( precondition_enforcement_required )
   enforce_preconditions;
```

2.3 Deferred enforcement: `enforce_condition_on_return`

Postconditions present a more difficult problem, as there is no place to put a statement that is executed between the function result's initialization and exit from the function. But we are familiar with a way to defer execution to that point; that's the point when automatic local objects are destroyed.

enforce_return_condition_statement:
`enforce_condition_on_return (` *result-name-introducer*$_{opt}$ *conditional-expression* `) ;`

1) Within a lambda expression, the relevant preconditions are those of the lambda object's `operator()`.
2) We can give special grammatical dispensation for this statement to appear at the beginning of a *function-try-block*.

Like `enforce_condition`, this statement enforces a condition by conditionally halting execution. But there are several differences:

— Enforcement is deferred to the point where a similarly-situated automatic object would be destroyed.

— As with an initializing automatic variable declaration, a jump that bypasses this statement renders the program ill-formed.

— Enforcement is conditional upon the function returning. If, at the point to which enforcement is deferred, the function is exiting by exception or has not executed an (explicit or implicit) return or co_return statement, no enforcement behavior takes place.

— Within the expression, the name introduced by the *result-name-introducer* refers to the function's result object.

If we introduce postcondition captures in a future C++ standard, these should also be made a part of `enforce_condition_on_return` statements. Capture expressions are evaluated at the point the statement is executed, not the point to which enforcement is deferred.

2.4 Incorporating postconditions by reference: `enforce_postconditions_on_return`

As with preconditions, we can eliminate redundancy by incorporating postconditions by reference.

enforce_postconditions_statement:
`enforce_postconditions_on_return` ;

This statement acts as a sequence of `enforce_condition_on_return` statements, one for each postcondition. The core usage is to place this statement at the beginning of a function, possibly following `enforce_preconditions`. Doing so will ensure that the function cannot return in a way that violates sufficiently stable postconditions.

2.5 Enforcing caller-facing conditions

The `enforce_preconditions` and `enforce_postconditions_on_return` statements enforce callee-facing conditions within a called function. But there is a mirror image to the need for such enforcement: function callers should equally be able to enforce caller-facing conditions. The need for caller-facing enforcement will increase as we improve contract support for indirect calls, creating situations where the caller-facing conditions are both nontrivial and differ from the callee-facing conditions. The introduction of implicit preconditions for built-in operations will further increase this need.

As with postconditions, there is no easy way to insert a statement at the points where caller facing conditions can be enforced. But just as an `enforce_postconditions_on_return` statement applies to every `return` statement in a scope, we can introduce statements that apply to every function call and built-in operation within a scope. Simply doing that, however, may cast an inconveniently broad net. A complex expression may contain many operations with pre- and postconditions when only a few are of interest.

There are many ways to limit enforcement to particular function calls. One simple way allows the targeted functions to be named.

enforce_call_conditions_statement:
`enforce_preconditions_on_call` *enforcement-nomination-list* ;
`enforce_postconditions_on_call` *enforcement-nomination-list* ;

Like `enforce_preconditions` and `enforce_postconditions`, these statements perform contract enforcement, but:

— Enforcement is applied to every nominated function call in the remainder of the current scope. Should we adopt implicit conditions on built-in operations, enforcement will also be applied to each nominated built-in operation.

— Each enforcement action stemming from `enforce_preconditions_on_call` is deferred to a point after the nominated call's parameters are initialized, but before the (C++26) evaluation of the function precondition sequence. Postcondition captures are also performed at this point.

— Each enforcement action stemming from `enforce_postconditions_on_call` is deferred to a point after the (C++26) evaluation of the function postcondition sequence, but before further execution of the calling function.

Several forms of nomination can be combined to cover all the ways we call functions in C++. Depending on the particular methods we allow, we may need to decorate nominations to avoid ambiguity.

— An unqualified name ("`pop_back`"), including an operator name ("`operator/`"), can be used to nominate direct calls to any function or operation with that name, as well as virtually-dispatched calls where the overridden function is named. Qualification can be used to target a narrower set of functions.

— Constructions and destructions can be nominated with "`Type::Type`" and "`Type::~Type`".

— Should new contract-bearing function types be introduced, the name of such a type can be used to nominate indirect calls through expressions of that type, references to that type, or pointers to that type.

— Should new contract-bearing member function pointer types be introduced, the name of such a type can be used to nominate indirect calls using expressions of that type.