Resolving Concerns with `const`-ification

Document #:	P3592R0
Date:	2025-02-10
Project:	Programming Language C++
Audience:	SG21 (Contracts)

Reply-to:	Joshua Berne <jberne4@bloomberg.net>
	Timur Doumler <papers@timur.audio>
	Lisa Lippincott <lisa.e.lippincott@gmail.com>

Abstract

The Contracts MVP, [P2900R13], makes *id-expression*s that name a variable declared outside a contract-assertion predicate implicitly `const`, a process that has come to be known as `const`-ification. As a result of earlier discussions of [P3261R2] and recent discussions with John Spicer and Daveed Vandevoorde regarding their concerns about `const`-ification, this paper contains proposals for features that are (mostly) additive to the Contracts MVP and that we hope will address those concerns and increase the consensus for [P2900R13].

Contents

Revision History

Revision 0

- Original version of the paper

1 Introduction

In [P2900R13], contract-assertion predicates treat variables declared outside of them as `const`:

```
int g = 17;
void f()
  pre( g += 17 > 0 );  // ill-formed; g is const in this context.
```

This aspect of the feature has given rise to concerns, many of which are elaborated on more deeply in [P3261R2]. In particular, we believe that the following concerns can be addressed with additive features on top of [P2900R13].

- The expression within a contract assertion cannot be reused in the function body with the guarantee that it will have the same meaning. Overload resolution might quietly invoke different functions in the two contexts:

  ```
  struct X {
    bool is_const()       { return false; }
    bool is_const() const { return true; }
  }
  void g(X& x)
    pre( x.is_const() )  // always succeeds
  {
    if (x.is_const()) {
      // This branch is never taken.
    }
  }
  ```

  We also have no way to express a predicate within the function body that does have the same semantics as inside a contract assertion.

- Within contract-assertion predicates, using a variable in a non-`const` fashion requires a cumbersome use of the often-maligned `const_cast`:

  ```
  // legacy API
  struct X;
  bool check_valid(X& x);  // nonmodifying but missing const

  void f(X& x) pre(check_valid(x));                  // ill-formed
  void g(X& x) pre(check_valid(const_cast<X&>(x));  // OK
  ```

  We can achieve nearly the same result as `const_cast` with the following macro:

  ```
  #define NOCONST(x) const_cast<std::add_lvalue_reference_t<decltype(x)>>(x)
  ```

  This macro, however, can be used in other contexts inappropriately, such as in a `const` member function when applied to a nonstatic member variable.

To address these concerns, we offer three proposals.

1. To allow replicating the meaning of a contract-assertion predicate outside of a contract assertion, we propose the `contract_predicate` keyword that can be used as an operator, which will treat its expression operand the same way the predicate of a contract assertion would, including the application of `const`-ification.
2. To ease the removal of `const`-ification in a local fashion, we propose a new operator, `noconst`, that can be used on *id-expression*s within contract-assertion predicates to remove the implicitly added `const` on those *id-expression*s.
3. To remove `const`-ification completely from a contract-assertion predicate, we propose a new contextual keyword, `mutable`, that can be placed on contract assertions, between the introducer (`pre`, `post`, or `contract_assert`) of a contract assertion and its predicate expression.

2 Proposals

Here we include detailed descriptions of our three proposed additions on top of [P2900R13].

2.1 `const`-ification for Arbitrary Expressions

[P2900R13] currently provides no way to reproduce, outside of a contract-assertion predicate, an expression within that contract-assertion predicate nor to guarantee that the same meaning will be achieved. In particular, changes in overload resolution based on the `const`-ness of *id-expression*s can result in different meanings.

For that purpose, we propose a new function-like operator whose name is `contract_predicate`, a new keyword.

unary-expression :
    ...
    contract_predicate (conditional-expression)
    ...

Within the argument of a `contract_predicate`, the same rules apply as in a contract-assertion predicate — namely, that an *id-expression* that names an entity declared outside the expression will have `const` added to its type.

To allow extracting arbitrary subexpressions from a contract-assertion predicate and getting the same behavior, this operator can be applied to any expression, and its type and value category will be that of its operand. Other than applying `const`-ification, this operator is, effectively, invisible.

Revisiting our earlier example, we can see the use and the resulting effect:

```
struct X {
  bool is_const()       { return false; }
  bool is_const() const { return true; }
}
void g(X& x)
  pre( x.is_const() )  // always succeeds
{
  if (contract_predicate(x.is_const())) {
```

```
    // This branch is always taken.
  }
}
```

Proposal 1: Introduce the `contract_predicate` operator.

Add the `contract_predicate` operator, which takes a single expression as an operand. Within that expression, any *id-expression* that names a variable declared outside the expression will be implicitly `const`. The type and value category of the `contract_predicate` expression will be that of its operand.

A search on `grep.app`, which searches millions of GitHub repositories, finds no results for `contract_predicate` in C or C++ code, indicating that we can make it a keyword without any concerning amount of code breakage.

This proposal has not yet had implementation experience.

2.2 `const`-ification As A Distinct Feature

The concern that an expression have the same meaning in different places is not completely addressed by Proposal 1 because that proposal assumes that a reader will understand that the contents of the expression in a precondition, postcondition, or contract assertion are in fact contract predicates.

As an alternative, we can require that the same syntax be used both within contract assertions and outside it so that the expressions are completely portable with no change in meaning. To achieve that, we can adopt Proposal 1 with the additional requirement that the new keyword, `contract_predicate` must also precede the parenthesized predicate in the contract assertion.

precondition-specifier :
  `pre` *attribute-specifier-seq*$_{opt}$ <u>`contract_predicate`</u> `(` *conditional-expression* `)`

postcondition-specifier :
  `post` *attribute-specifier-seq*$_{opt}$ <u>`contract_predicate`</u> `(` *result-name-introducer*$_{opt}$ *conditional-expression* `)`

assertion-statement :
  `contract_assert` *attribute-specifier-seq*$_{opt}$ <u>`contract_predicate`</u> `(` *conditional-expression* `)` `;`

Note that this proposal *requires* that the new keyword be placed before any contract predicate, making it non-optional:

```
int f(const int x)
  pre contract_predicate (x != 1)                 // a precondition assertion
  post contract_predicate (r : r == x && r != 2)  // a postcondition assertion
{
  contract_assert contract_predicate (x != 3);   // an assertion statement
  return x;
}
```

If Proposal 4 (below) is adopted, the mutable could be allowed to be interchangeable with the `contract_predicate` keyword.

Proposal 2: Mandatory use of `contract_predicate`

Add the `contract_predicate` operator and require the `contract_predicate` keyword be included in every use of `pre`, `post`, and `contract_assert`.

An alternative keyword, especially one that is shorter due to the requirement that it always be used in a contract assertion, might be considered. `const` has been suggested, but it is not intuitive nor has the analysis been done to confirm that it can be unambiguously used as a function-like operator. Finding alternative keywords that are short and intuitive is challenging.

2.3 The `noconst` Operator

Within a contract-assertion predicate (or within the operand of a `contract_predicate` expression), a new operator may be used as a function-like operator named `noconst`:

unary-expression :

...
noconst (id-expression)
...

This expression is ill-formed if *id-expression* does not name an entity declared outside an enclosing contract-assertion predicate. The type of this expression is the type of the *id-expression* without `const` added to it:

The type of the `noconst` expression is the same as the entity referred to by the expression:

```
struct X {
  bool is_const()       { return false; }
  bool is_const() const { return true; }
}
void f(X& x, const X&y)
  pre(  x.is_const() )
  pre(  y.is_const() )
  pre( !noconst(x).is_const() )  // type of noconst(x) is X
  pre(  noconst(y).is_const() )  // type of noconst(y) is still const X
  pre( ![]() {
    X z;
    return z.is_const();
  }())
  pre( ![]() {
    X z;
    return noconst(z).is_const();  // error: z is not subject to const-ification.
  }());
```

Proposal 3: Introduce the `noconst` operator.

Add the `noconst` operator, which takes an *id-expression* as an operand. The `noconst` operator may be used only within contract-assertion predicates (or the operand of a `contract_predicate` operator), and the resulting expression will not implicitly have `const` added to its type; the type and value category of the `noconst` expression will be that which its operand would have outside a contract-assertion predicate.

A search on `grep.app` finds 155 results for `noconst` (most of which occur in C code rather than C++ code), indicating that it is a relatively rare identifier. If this frequency is deemed too great, we could consider other or longer alternatives, such as `unconst` (which has 429 results) or `contract_noconst` (which has 0).

This proposal has not yet had implementation experience.

2.4 The `mutable` Specifier

When migrating existing assertions from legacy facilities where `const`-ification was not available to `contract_assert`, the need to have `const`-correct code (and its uses within assertion predicates) can be a hindrance to the adoption of Contracts.

To that end, we propose a mechanism to simply turn off `const`-ification entirely within a contract-assertion predicate by adding the specifier `mutable` to a contract assertion:

precondition-specifier :
  `pre` `mutable`$_{opt}$ *attribute-specifier-seq*$_{opt}$ `(` *conditional-expression* `)`

postcondition-specifier :
  `post` `mutable`$_{opt}$ *attribute-specifier-seq*$_{opt}$ `(` *result-name-introducer*$_{opt}$ *conditional-expression* `)`

assertion-statement :
  `contract_assert` `mutable`$_{opt}$ *attribute-specifier-seq*$_{opt}$ `(` *conditional-expression* `)` `;`

When present, *id-expression*s that appear within the *conditional-expression* will not have `const` added to their type if they name an entity declared outside the enclosing contract-assertion predicate.

Proposal 4: Introduce the `mutable` specifier.

Allow the use of the keyword `mutable` between the introducer (`pre`, `post`, or `contract_assert`) of a contract assertion and and its predicate expression. When present, *id-expression*s within the contract assertion's predicate will not be made implicitly `const`.

This proposal reuses an existing keyword, so no search needs to be done for a free identifier. On the other hand, objections to the chosen specifier have been raised because, unlike the effect of `mutable` on a closure type when attached to a lambda expression, in this case it does not actually make the thing it is attached to — a contract assertion — into something that can be mutated where previously it was immutable. Therefore, consideration should be given to alternative proposals with different keywords, such as `allow_mutation` or `no_constification`.

This proposal has had implementation experience in the GCC implementation of [P2900R13].

3 Wording Changes

Wording is relative to the current state of [P2900R13] and is separated by proposal.

3.1 Proposal 1 (`const`-ification of Arbitrary Expressions)

Modify [basic.contract.general] before paragraph 2:

2-a An expression may be a *contract predicate context*. [*Note:* Implicit modification of values outside a contract predicate context is discouraged ([expr.prim.id.unqual]). — *end note*]

Modify [basic.contract.general], paragraph 2:

2 Each contract assertion has a *predicate*, which is an expression of type `bool` that is a contract predicate context. [*Note:* The value of the predicate is used to identify program states that are expected. — *end note*]

Modify [expr.unary.general], paragraph 1:

unary-expression :
  ...
  contract-predicate-expression
  ...

Add a new section [expr.unary.contract.predicate] after [expr.unary.noexcept]:

contract-predicate-expression :
  `contract_predicate` (*conditional-expression*)

1 The operand of the `contract_predicate` operator is a contract predicate context.

2 The result and value category of the `contract_predicate` operator are the result and value category of its operand.

Modify [expr.prim.id], paragraph 3+d:

3+d Otherwise, if the *unqualified-id* appears in a contract predicate context ~~the predicate of a contract assertion~~ C ([basic.contract]) and the entity is

- a variable declared outside of C of object type `T`, or
- a variable or template parameter declared outside of C of type "reference to `T`", or
- a structured binding of type `T` whose corresponding variable is declared outside of C,

then the type of the expression is `const T`.

Modify [expr.prim.id.qual], paragraph 5+a:

5+a If Q appears in a contract predicate context~~the predicate of a contract assertion~~ C ([basic.contract]) and the entity is

- a variable declared outside of C of object type `T`, or
- a variable declared outside of C of type "reference to `T`", or
- a structured binding of type `T` whose corresponding variable is declared outside of C,

then the type of the expression is `const T`.

3.2 Proposal 2 (`const`-ification As A Distinct Feature)

Make the changes required for Proposal 1.

Modify the grammar at the start of [dcl.contract.func]:

function-contract-specifier-seq :
function-contract-specifier function-contract-specifier-seq$_{opt}$

function-contract-specifier :
precondition-specifier
postcondition-specifier

precondition-specifier :
`pre` *attribute-specifier-seq*$_{opt}$ `contract_`*predicate* (*conditional-expression*)

postcondition-specifier :
`post` *attribute-specifier-seq*$_{opt}$ `contract_`*predicate* (*result-name-introducer*$_{opt}$ *conditional-expression*)

Modify the grammar at the start of [stmt.contract.assert]:

assertion-statement :
`contract_assert` *attribute-specifier-seq*$_{opt}$ `contract_`*predicate* (*conditional-expression*) ;

3.3 Proposal 3 (The `noconst` Operator)

Modify [expr.unary.general], paragraph 1:

unary-expression :
...
noconst-expression
...

Add a new section [expr.unary.noconst] after [expr.unary.noexcept] (replacing "the predicate of a contract assertion" with "contract predicate context" if Proposal 1 is also accepted):

contract-predicate-expression :
`noconst` (*id-expression*)

1 The `noconst` operator must appear within the predicate of a contract assertion. The operand of the `noconst` operator must be a variable, structured binding, or template parameter to which `const` would be added ([expr.prim.id.unqual], [expr.prim.id.qual]).

2 The type and value category of the *contract-predicate-expression* are those of its operand. [*Note:* `const` is not added to the type of the operand even though it is within a contract assertion predicate. — *end note*]

Modify [expr.prim.id], paragraph 3+d:

3+d Otherwise, if the *unqualified-id* appears in the predicate of a contract assertion and is not the operand of the `noconst` operator C ([basic.contract]) and the entity is

- a variable declared outside of C of object type `T`, or
- a variable or template parameter declared outside of C of type "reference to `T`", or
- a structured binding of type `T` whose corresponding variable is declared outside of C,

then the type of the expression is `const T`.

Modify [expr.prim.id.qual], paragraph 5+a:

5+a If Q appears in the predicate of a contract assertion and is not the operand of the `noconst` operator C ([basic.contract]) and the entity is

- a variable declared outside of C of object type `T`, or
- a variable declared outside of C of type "reference to `T`", or
- a structured binding of type `T` whose corresponding variable is declared outside of C,

then the type of the expression is `const T`.

3.4 Proposal 4 (The `mutable` Specifier)

If Proposal 1 is also adopted, explicitly describe how the `mutable` specifier can apply to a contract predicate context in [basic.contract.general]

Modify the grammar at the start of [dcl.contract.func]:

function-contract-specifier-seq :
  function-contract-specifier function-contract-specifier-seq$_{opt}$

function-contract-specifier :
  precondition-specifier
  postcondition-specifier

precondition-specifier :
  `pre` `mutable`$_{opt}$ *attribute-specifier-seq*$_{opt}$ (*conditional-expression*)

postcondition-specifier :
  `post` `mutable`$_{opt}$ *attribute-specifier-seq*$_{opt}$ (*result-name-introducer*$_{opt}$ *conditional-expression*)

Modify the grammar at the start of [stmt.contract.assert]:

assertion-statement :
  `contract_assert` `mutable`$_{opt}$ *attribute-specifier-seq*$_{opt}$ (*conditional-expression*) ;

Modify [expr.prim.id], paragraph 3+d:

3+d Otherwise, if the *unqualified-id* appears in the predicate of a contract assertion without the `mutable` specifier C ([basic.contract]) and the entity is

- a variable declared outside of C of object type `T`, or
- a variable or template parameter declared outside of C of type "reference to `T`", or

- a structured binding of type `T` whose corresponding variable is declared outside of C,

then the type of the expression is `const T`.

Modify [expr.prim.id.qual], paragraph 5+a:

5+a If Q appears in the predicate of a contract assertion without the `mutable` specifier C ([basic.contract]) and the entity is

- a variable declared outside of C of object type `T`, or
- a variable declared outside of C of type "reference to `T`", or
- a structured binding of type `T` whose corresponding variable is declared outside of C,

then the type of the expression is `const T`.

4 Conclusion

Each of the proposals presented above provides mechanisms to make use of contract assertions, as proposed in [P2900R13], in situations in which they currently would not be useable and thus increases both the usability of the feature and, hopefully, consensus for its adoption into C++26.

These proposals address at least some of the concrete concerns that have been raised with `const`-ification in [P2900R13].

Acknowledgements

Thanks to Daveed Vandevoorde and John Spicer for feedback on this paper. Thanks to Lori Hughes for improving the words to a level of usage such that English speakers should be able to readily digest it.

Bibliography

[P2900R13] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, "Contracts for C++", 2025 http://wg21.link/P2900R13

[P3261R2] Joshua Berne, "Revisiting `const`-ification in Contract Assertions", 2024 http://wg21.link/P3261R2