CWG 3158 — `const`-ification of Splice Expressions

Document #:	P3598R0
Date:	2026-03-26
Project:	Programming Language C++
Audience:	EWG, CWG

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

Abstract

This paper collects various information that has been discussed recently on the reflectors regarding CWG 3158 so that a more informed decision can be made regarding whether a *splice-expression* applied to the reflection of a variable within a contract assertion should have `const` added to its type in the same manner we currently do for *id-expressions*.

Contents

1 Introduction

CWG 3158 points out that the specification for a *splice-expression* does not apply `const`-ification when a similar *id-expression* would.

The status quo is that when a *splice-expression* is applied to the reflection of a variable the variable's type is the type of the expression with only some of the alterations (such as the removal of reference type) applied to it that would be applied to an *id-expression*:

```
void f(int p)
  pre(( ++p, true))          // error: unqualified-id 'p' has const type
  pre(( ++[:^^p:], true))    // OK
{}
```

This behavior then begets a statement and a question:

1. When an *id-expression* denotes a variable that is declared outside a contract assertion we add `const` to its type.
2. If a *splice-expression* denotes a variable that is declared outside a contract assertion should we add `const` to its type?

There are a number of reasons to consider this a defect and to instead fix CWG3158 by applying the same logic to *splice-expressions* that is applied to *id-expressions*.

- A *splice-expression* that is applied to the reflection of a variable (or structured binding) can be thought of as a computed *id-expression*, and must therefore behave in the same manner as an *id-expression* to reduce cognitive load when deciding to compute any given *id-expression*'s target variable.
- No heroics are needed to determine whether `const`-ification should be applied or not, as the compiler knows all information about the target variable when it has a reflection of that variable.
- The GCC trunk implementation of reflection and contracts already has this behavior, so there is both implementation experience and an indication that this behavior is the most natural one to apply when implementing the language.
- [P3261R1] discussed many different reasons to adopt `const`-ification (or alternatives). Almost all of them apply equally well to preconditions that would use a *splice-expression* instead of an *id-expression*, with the added factor that the code where the problem occurs might be even harder to read.

  Consider a somewhat contrived but not entirely unrealistic scenario.

  – A developer decides to write a function whose purpose is to index into a container. Because they love their clients, they decide to allow clients to pass the container and the index in either order, and they will determine which parameter is the index by using reflection to see if it is an integral type:

  ```
  template <typename T1, typename T2>
  auto f(T1& t1, T2& t2)
  {
  ```

```
  constexpr std::meta::info key = meta::is_integral(^^T1) ? ^^t2 : ^^t1;
  constexpr std::meta::info container = meta::is_integral(^^T1) ? ^^t1 : ^^t2;
  return [:container:][[:key]];
}
```

– Next that developer also requires that the value is in the container and is a nonegative number:

```
template <typename T1, typename T2>
auto f(T1& t1, T2& t2)
  pre(
    [: meta::is_integral(^^T1) ? ^^t2 : ^^t1 :]
        // first operand is the first parameter that is non-integral
        // or the second parameter is both are integral
    [   // bracket operator, this better not be std::map :)
      [: meta::is_integral(^^T1) ? ^^t1 : ^^t2 ]
        // second operand is the first integral parameter
    ] >= 0 )
{ /*...*/ }
```

This code works great for many use cases:

```
void g(std::vector<int> &v)
{
  f(v,5);  // good
  f(5,v);  // good
}
template <std::size_t N>
void g(std::array<int,N> &v)
{
  f(v,5);  // good
  f(5,v);  // good
}
```

But then the user stumbles on one of the classic blunders:

```
void g(std::map<int,int> &m)
{
  f(m,5);  // Precondition check adds the key 5 to m!
  f(5,m);  // Precondition check adds the key 5 to m!
}
```

If `const`-ification had been applied to the *splice-expressions* in the preconditions of `f`, this last version of `g` would not compile — for very good reasons — and the user would immediately realize the problem and find alternatives to fix it.

- [P3261R1] also discussed a number of methods to work around `const`-ification when it is needed (to work with APIs that do not mutate state but are also not `const`-correct).

  Not fixing this core issue means that we have inadvertently added a new feature to the language where `[:^^:]` becomes an idiomatic `unconst` operator.

  - There is no intuitive reason that explains why this is an `unconst` operator other than "WG21 forgot to reconcile the specifications of Reflection and Contracts".
  - For those who do depend on the protections of `const`-ification but also use reflection, they get this escape hatch applied when they have no desire for it.
  - The `unconst` operator is syntactic sugar for functionality that can already be done without reflection, and which can be done more verbosely with reflection by accessing the object instead of the variable:

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

void f(int p)
  pre(( ++UNCONST(p) ));  // code smell!
```

    With reflection this can also be done, but in a way that makes it clear that something is being circumvented:

```
void f(int p)
  pre(( ++ [: std::meta::object_of(^^p) :] ));  // code smell!
```

    Both of these are, of course, much easier to see happening and question if they are problematic.

- It is important to recognize that this is a change to *splice-expression* so that it works more consistently as a computed *id-expression* and it is *not* a change to reflection. Reflection within contract assertions still, as always, sees only entities and `const`-ification does not play a part in that process.

Given all of the above, we come to a simple proposal for the resolution of CWG3158:

Proposal 1: Apply `const`-ification to splice expressions

When a splice expression (`[::]`) is applied to the reflection of a variable or structured binding within a contract assertion, add `const` to its type if the underlying variable is declared outside of the contract assertion (in a manner consistent with how it is done for id expressions).

2 Wording Changes

Modify [expr.prim.splice], paragraph 2:

2 For a *splice-expression* *E* of the form *splice-specifier*, let S be the construct designated by *splice-specifier*.

— The expression is ill-formed if S is

— a constructor,

— a destructor,

— an unnamed bit-field, or

— a local entity([basic.pre]) such that

— there is a lambda scope that intervenes between the expression and the point at which S was introduced and

— the expression would be potentially evaluated if the effect of any enclosing `typeid` expressions([expr.typeid]) were ignored.

— Otherwise, if S is a function F, the expression denotes an overload set containing all declarations of F that precede either the expression or the point immediately following the *class-specifier* of the outermost class for which the expression is in a complete-class context; overload resolution is performed([over.match,over.over]).

— Otherwise, if S is an object or a non-static data member, the expression is an lvalue designating S. The expression has the same type as that of S, and is a bit-field if and only if S is a bit-field.

[*Note*: The implicit transformation whereby an *id-expression* denoting a non-static member becomes a class member access([expr.prim.id]) does not apply to a *splice-expression*. — *end note*]

— Otherwise, if S is a direct base class relationship (`D`, `B`), the expression is an lvalue designating S. The expression has the type `B`.

— Otherwise, if S is a variable or a structured binding, S shall either have static or thread storage duration or shall inhabit a scope enclosing the expression. ~~The expression~~ $\underline{E}$ is an lvalue referring to the object or function X associated with or referenced by S, ~~has the same type as that of S,~~ and is a bit-field if and only if X is a bit-field. If E appears in the predicate of a contract assertion C([basic.contract]) and S is

— a variable declared outside of C of object type `T`,

— 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 E is `const T`, otherwise E has the same type as that of S.

[*Note*: The type of a *splice-expression* designating a variable or structured binding of reference type will be adjusted to a non-reference type([expr.type]). — *end note*]

— Otherwise, if S is a value or an enumerator, the expression is a prvalue that computes S and whose type is the same as that of S.

— Otherwise, the expression is ill-formed.

3 Conclusion

The right resolution for CWG3158 is to make splice expressions consistent with their intended behavior as computed id expressions. This solution is straightforward and already implemented.

As we ship C++26 we are getting large and important features finalized, it is on us to make sure they remain as coherent as possible. That will either be done now, or later at a much higher cost.

Bibliography

[P3261R1] Joshua Berne, "Revisiting `const`-ification in Contract Assertions", 2024
http://wg21.link/P3261R1

Acknowledgments

Thanks to all of those who contributed to the discussion of this issue on the reflector.