Constification should not be included in the MVP

John Spicer (jhs@edg.com)

Document number: P3478R0
Date: 2024-10-16
Audience: SG21, EWG

Introduction

The contracts MVP (minimum viable product) includes a feature that is colloquially called “constification”. This is discussed in P2900R8 and also in P3261R1.

Constification makes some set of variables and parameters const inside contract-checking annotations (CCA).

The purpose of this paper is to advocate for this feature not being part of the MVP.

Summary of concerns

An overview of my concerns given here. These are explained in more detail below.

1. The feature gives novel semantics to expressions in CCAs. This will be surprising to many users and will harm the adoption of contracts.
2. The benefits of constification are very small and can generally be provided in a superior way without changing the semantics of expressions in a CCA.
3. Constification changes overload resolution results.
4. Others have proposed features that would improve the safety of code in CCAs. We should address the full set of features for an alternate evaluation mode rather than settle for what I consider to be a trivial improvement but which could make it harder to add a better mode in the future (for example, see P3285R0).

Constification makes evaluation of a CCA fundamentally different that every other expression

It is my expectation, and I believe that it would be the expectation of most C++ users, that the following would all share the same semantics:

```
assert(x);
my_own_assert(x);
if (!x) abort();
contract_assert(x);
```

But the contract_assert will have significantly different semantics from the others, and there is no facility to get the non-constified behavior for an expression (there is a way to unconstify an individual variable use, but that must be done for every variable and for every user of that variable).

I view this as unacceptable and to be a huge burden on C++ users who want to move to contracts from either C assert or their own facility.

It will also be surprising to uses who are not using an existing facility and expect things to “just work”.

The benefits are small and can be achieved in superior ways

Given this example:

```
struct X {int val; int *pval;};
void f(int *const p, X *const px, int const i, X const x) {
        // errors
        contract_assert((p=0));
        contract_assert((i=0));
        contract_assert((x.val=0));
        // okay
        contract_assert((*p=0));
        contract_assert((px->val=0));
        contract_assert((*x.pval=0));
}
```

Note that contracts already have a syntax that makes `contract_assert(p=0)` ill-formed. The additional parentheses are needed to allow an assignment at the top-level.

Constification will result in errors on the first three cases because those variables have been made const.

It will not give errors on the last three cases.

Saying `p=0` or `i=0` is odd, but at least the impact is local to the function. The indirection cases change the global state of the program yet remain undiagnosed.

I had briefly proposed a semantic change that would prohibit all modifying operations in a CCA but later retracted that in favor of letting compilers choose to diagnose those cases. But we could still go in the direction of a required semantic check.

A feature of constification is that it prevents binding a non-const reference to a constified variable. This is not as easily addressed in some other way, but compilers can warn about such cases in a CCA, and static analyzers that do whole program analysis (which is most of the better ones) can definitively let you know whether your CCA is modifying something that is bound to a non-const reference.

Constification changes overload resolution

If you change the types of some things, it will affect overload resolution. This is (or should be) a fundamental law of the universe.

There are three main failure modes of overload resolution:

1. The function selected is not the one you intended.
2. There is no matching function found.
3. The call is ambiguous.

These are listed where #1 is the worse outcome and #3 is the least bad outcome.

For #1, it can be very hard to figure out why you did not get the expected function. This is particularly true if the function is called from generic code that works most of the time.

For #2, you might end up with a *lot* of compiler diagnostic output, but at least you have a chance of finding your answer in there somewhere.

For #3, you at least have a set of things to start with. It should be noted that you might have gotten to #3 because of an error in your code, and fixing it leads to #1 or #2.

A trivial example of code that changes is:

```
void f(int, int); // #1
void f(char&, char&); // #2

int main() {
        char c1 = 1;
        const char c2 = 1;
        f(c1, c1); // calls #2
        f(c2, c2); // calls #1
}
```

This is not intended to be a “real world” example, just to show that making something const can cause a completely different function to be called.

I have discussed examples like this with advocates of constification and the answer is essentially “real code doesn’t do that, and if it does then it is broken”.

I’m sure real code does sometimes bind something to a non-const lvalue, and might have good reason to. For example, it might be able to be called in ways where something is modified and in other cases where you know the thing is not modified.

We spend a lot of time in CWG looking at overload examples that involve user-defined conversions, converting constructors, partial ordering, etc. It is very hard for people to understand what one set of arguments does in overload resolution. It is much harder for people to understand what happens when some things are made const.

It is also the case that in generic code you don’t actually know what is changing. Consider this example:

```
template <typename T, typename U, typename V, typename W>
void f(T t, U u, V v, W w) {
      contract_assert(g(t, u, v, w));
}
```

Any or all of those parameters could be const to begin with, so you don’t know how things will change in your assert.

If you decide you don’t want that call to be constified, you simply need to write:

```
template <typename T, typename U, typename V, typename W>
void f(T t, U u, V v, W w) {
          contract_assert(g(const_cast<std::add_lvalue_reference_t<decltype(t)>>(t),
const_cast<std::add_lvalue_reference_t<decltype(u)>>(u),
const_cast<std::add_lvalue_reference_t<decltype(v)>>(v),
const_cast<std::add_lvalue_reference_t<decltype(w)>>(w)));
```

}

Of course, the boiler-plate part of that can be put in a macro, but is that really the code we want people to be required to write to get an expression that works the same way as any other expression in C++?

And, of course, in addition to causing overload resolution to change/fail for all of the usual reason, constification can cause calls to fail to meet concepts.

Evolution of the language

As mentioned in the introduction, there are other proposals to improve the safety of contracts.

Whatever we do should address those concerns and the concerns of constification together (to the extent that we can't address constification more simply by just encouraging compiler diagnostics).

I think it would be okay for C++ to have two evaluation modes for a CCA. If we have more than two, IMO, that will suggest to users that we don't really know how to safely handle contracts in C++.

P3285 strives to provide a "safe" model for evaluation of a CCA where you are assured that your CCA does not pass because it accidentally ended up using undefined behavior.

It is hard to summarize what constification does. It will help you find some bugs of some level of importance, but in doing so it will break some amount of code.

To me, this is a very poor tradeoff. As described above, it will find some bugs, but probably not the ones that you care about the most.

Many have expressed that safety is the most important thing for C++ to address.

The contracts feature improves safety by providing a convenient way for people to add preconditions, postconditions, and assertions to their code in a way that can be enabled or disabled, and which allows the introduction of a violation handler. It also provides the quick_enforce mode which allows contracts to be deployed in release builds with very low cost.

But to obtain those benefits, it must be widely adopted.

I believe that constification will be a hurdle to such adoption, and will be a hurdle to adopting a CCA evaluation model that provides meaningful improvements in safety.

In my opinion, constification provides no real safety improvements. It may allow diagnosis of some changes to variables that you did not intended, but as described above, it only addresses a small number of cases, and not the most important cases.

But it also makes code less safe by changing what a CCA does. In many cases you are no longer testing the code that you thought you were testing because you have changed the types on which you are operating and you have changed the function being called by overload resolution.

An example from Ville Voutilainen

You might find this interesting: https://godbolt.org/z/7Ev4eeM3v

When constification is on, f() has a constness 'mismatch' between the if-condition and the precondition. g() does not.

When constification is off, neither function has a constness mismatch.

When there's no constness mismatch, the compiler proves expression equivalence between the expression used in the if-condition and the precondition, and (correctly) nukes the precondition, and there's no need to call a violation handler, the precondition is proven satisfied.

When there's a constness mismatch, both the if-condition and the precondition call pred(), and a cold clone of f() that calls the violation handler is generated, and it's jumped to if pred() returns false.

Furthermore, this is incidentally exactly the kind of proof Gaby's P3285 strives for. Strict predicates and conveyor functions are stronger than [[gnu::pure]], but from static analysis and code optimization standpoint they are very similar. So, a practical example of how that analysis works is right there in
that godbolt example. For this particular analyzer, which is baked into the compiler, the analysis is used for optimizing away a redundant precondition evaluation and a branch for calling the violation handler. For a correctness analyzer, the analysis would be used for being able to tell the programmer that the precondition of X::f() is proven satisfied and the code is therefore contract-correct.

In Conclusion

I would like to have contracts in C++26, but not if that means living with constification.

Constification is an experiment being conducted on C++ users, and we lack sufficient data to know its impact.

If people insist on constification in the MVP, then we should make it a TS and not incorporate it into the IS.