Revisiting `const`-ification in Contract Assertions

Document #:	P3261R2
Date:	2024-11-25
Project:	Programming Language C++
Audience:	SG21 (Contracts), EWG (Evolution)

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

Abstract

The SG21 proposal for the Contracts facility seeks to reduce the chance of accidentally writing destructive[1] contract-assertion predicates by making `const` certain expressions that would not otherwise be `const` when used outside a contract assertion. SG21 has repeatedly shown strong consensus for pursuing this approach, dubbed `const`-ification, to making the Contracts facility vastly easier to use correctly. In this paper, we attempt to categorize the various concerns with different approaches to `const`-ification and clarify that the current design in [P2900R9] is optimal.

Contents

[1] A destructive contract-assertion predicate is one whose presence or evaluation would change the correctness (including essential behavior) of a program. The Contracts facility has been designed to support nondestructive contract-assertion predicates and, when reasonable, discourage the accidental or intentional use of destructive contract-assertion predicates.

Revision History

Revision 2 (pre-Wrocław November 2024 WG21 meeting)

- Presented results of SG21 discussions on [P3261R1]
- Added a presentation on analyzing relevance of concerns, Section 2.3
- Revised the Conclusion to address EWG
- Added Proposal G to allow for consistency if `const`-ification is removed
- Added another existing escape hatch to consider for `contract_assert`

Revision 1 (for discussion at the October 10, 2024 SG21 telecon)

- Enumeration of proposals about what to `const`-ify changed from numerals (1–6) to letters (A–E)
- New concerns added
  - Teachability of Contracts — How much each proposal impacts the ability to teach Contracts
  - Replaceable with Warnings — Whether proposals could feasibly be implemented as warnings
  - Don't Misnavigate Broken Overload Sets — Which proposals produce bad results for overload sets polluted by ADL
  - Increased Cost of Static Analysis — When contract assertions impact the ability to do static analysis
  - Silently Fixing Broken Predicates — Whether proposals hide errors by making them well-formed
  - Minimize Effort — Which escape hatches require the least effort to use
- Clarified assessment of warnings only
- Formal proposals (E1–E3) put forward for language-based escape hatches

Revision 0 (Presented at October 3, 2024 SG21 telecon)

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

1 Introduction

SG21's review of [P3071R1] generated some discussion about the exact set of expressions to which what has since been dubbed `const`-ification should be applied as well as whether we should build in any protections at all to avoid and discourage writing *destructive* contract assertions that change the correctness of a program.

As originally proposed and as integrated into [P2900R8], `const`-ification — which treats an expression as if it were wrapped in a cast that adds `const` to the expression's type — was applied to the following.

- Id-expressions that name variables having automatic storage duration, including structured bindings, are implicitly treated as `const`. This transformation applies to function parameters and non-`static` block-scope variables used within a contract-assertion's predicate.
- `*this` is `const`, which then applies the same rules as used within a `const` member function to any data members due to the implicit transformation of an id-expression denoting a data member, `d_x`, into a class member access expression `(*this).d_x`. For the same reason, member functions invoked either implicitly or explicitly through `this` will select `const` overloads within a contract-assertion's predicate.

After extensive discussions of the previous revision of this paper ([P3261R1]), one change was adopted into [P2900R9] to extend `const`-ification to apply to all variables declared outside of the contract-assertion predicate.

In this paper, we explore the motivations for the `const`-ification aspect of the Contracts facility, discuss possible concerns and how they apply to the design in [P2900R9] and to the various alternative designs that have been considered, and attempt to quantify how impactful those concerns would be on real software.

2 Motivation

Many reasons motivate the consideration to remove, keep, or adjust the `const`-ification aspect of [P2900R8]. Some reasons are inherent to the design and unique purpose of Contracts, and some are guided by what will produce a tool that maximizes utility for users of the C++ language.

2.1 Why `const`-ification?

The Contracts feature being designed by SG21 is built around a central purpose for contract assertions. Each contract assertion describes a single, discrete algorithm that identifies whether a contract violation has occurred. Importantly, these checks are encoding in a program parts of the *plain-language contract* that itself defines when the evaluation of that program is *correct*.

For contract assertions to benignly provide information about the program to which they are being applied, rather than simply producing a different program with functionally different behavior, they must never themselves alter the correctness of that program; i.e., they must follow the prime directive of the design for Contracts described in section 3.1 of [P2900R8].

> Principle 1: Contract Assertions Do Not Alter Correctness
>
> Neither the presence of a contract assertion nor the evaluation of a contract predicate should alter the correctness of a program's evaluation.

From this principle follow many of the design decisions that have been made in [P2900R8]. Importantly, this principle can be seen to underlie the principles and design decisions that were laid out in papers such as [P2834R1] and [P2932R3]. This essential property of contract assertions is also a key part of why adopting the flexible semantics model introduced by [P2877R0] is both viable and effective.

Contract assertions whose predicates would violate the above principle when evaluated are said to have *destructive predicates*, so-named in [P2712R0]. By design, the specification of contract assertions in [P2900R8] does its best to ensure that simply introducing a contract assertion into code does not make the assertion destructive. The evaluation of a predicate can, however, be destructive in some cases.

- The simple presence of certain predicates in a contract assertion within a function might violate a plain-language contract that prohibits their use, such as a promise a library might make to refrain from using certain other libraries or language features or to avoid using profanity when spelling function names.
- The evaluation of a contract predicate might require enough computation to violate complexity guarantees of the function, such as a linear check that input is sorted on a binary search function.
- The evaluation of a predicate might make modifications to program state that introduce, into a program, defects that violate later plain-language contracts.

In particular, the last category is what we often refer to as *destructive side effects*. One approach that could be taken is to simply declare that contract predicates may contain no side effects, but such a prohibition has a few major drawbacks.

- Only one of the above categories of destructive contract assertions contains predicates that have side effects at all.
- The core-language definition of side effects[2] is specific and hard to avoid in all software. This issue can largely be alleviated by allowing modifications of nonvolatile objects whose lifetimes begin and end during the evaluation of the contract predicate — a categorization that is often referred to as not allowing side effects *outside the cone of evaluation* of the contract assertion.
- Some core-language side effects — such as reading a volatile variable, allocating and deallocating memory, or caching the results of complex computations in a `mutable` variable — are quite infrequently a change in state that is easily observable and are thus highly unlikely, in practice, to alter the correctness (or expected behavior) of a program.
- Even many observable side effects — such as logging a trace message about function invocation — might be desired for evaluation and will often not alter the correctness of a program.
- The simple act of requiring any particular evaluation restrictions that apply to the entire evaluation of a contract predicate would preclude either the use of arbitrary functions inside a predicate (and thus practically all user-defined types or types from the Standard Library) or

[2]The C++ Standard defines a side effect as reading a `volatile` glvalue, modifying an object, calling a library I/O function, or calling a function that does any of those things.

the introduction of a new class of functions that guarantee this property.[3] The use of Contracts would be reduced to only toy applications and slideware if we could not, for example, use `std::vector::size()` or `std::string::operator==` within a contract-assertion predicate until such functions were updated with the appropriate new annotation (and possibly reimplemented using the new restrictions).

In general, however, the most frequent reason a contract predicate is destructive is a direct change to the state of an object that is relevant to the function's behavior, such as a function parameter or local variable. Invoking a function that is semantically nonmutating — even if not strictly free of side effects — is often an indication that the predicate itself is not destructive. C++, luckily, already has a concept for describing when operations are expected to be semantically nonmodifying operations: `const`.

We must understand the expectations users have for using a `const` reference to an object in comparison to using a non-`const` reference to the same object. In most cases today, including any expression composed of built-in operations or elements of the Standard Library, the behavior of any non-`const` operation that *could* be used with a `const` reference instead is to produce a Liskov-substitutable value as a result.

- The most common case, of course, is when no non-`const` alternative is available and both operations invoke the same overload.
- A frequent alternative is for containers to provide a form of deep-`const` by having a `const` overload return a `const` pointer, reference, or user-defined type that allows access but no modification of elements while the non-`const` overload returns a different type that would allow modification but is otherwise equivalent. The Standard Library contains functions, such as `begin()`, `end()`, `front()`, and `back()`, that all provide both `const` and non-`const` overloads. In all cases, the member functions themselves have no side effects, and the overloads are distinguished only by whether they return a `const` or non-`const` reference, pointer, or iterator that refers to an element of the container.

Therefore, a reasonable approach to reducing the chance that contract predicates are inadvertently destructive is to minimize their ability to execute non-`const` operations outside their cone of evaluation. The questions, then, are twofold.

1. What action should we take to prevent the use of modifying operations?
   - Do nothing, which would relegate all potential improvements to warnings provided by the platform.
   - Make selecting some set of non-`const` operations ill-formed.
   - Treat selected objects as `const` within the contract-assertion predicate.
2. How should we identify, within a contract predicate, which expressions will denote objects outside the cone of evaluation without risking a sufficiently large number of false positives, which would make the writing of contract assertions untenable?

[3]Introducing a new class of functions that provably avoid some forms of destructiveness is one of the stated goals of the features proposed in [P2680R1] and later [P3285R0], although those proposals aim to vastly further restrict what is allowed while providing no help to contract predicates that it considers `relaxed`.

Two in-depth analyses of the impact of `const`-ification as proposed in [P2900R8] have been published to help understand what it does to real software.

- [P3268R0] provided an analysis of a medium-sized codebase that made use of a homegrown assertion macro as well as `<cassert>` and then analyzed the used predicates to see if `const`-ification had any detrimental impact. In total, only a minuscule fraction of assertion predicates needed any change, and most changes resulted in making the used code `const`-correct where it was not completely so before.
- In [P3336R0], a sizeable set of libraries that made use of an internal assertion facility was rebuilt using the GCC implementation of Contracts (with `const`-ification) as the implementation of that macro. A similarly minuscule number of issues were encountered, one of which was a major bug and the rest of which were incompletely `const`-ified components. In addition, all unit tests passed with all assertions enabled, indicating an especially high likelihood that no changes in meaning related to `const`-ification impacted the correctness of the software.

An important point about both these analyses, however, is that they address concerns related to migrating existing, tested assertions to contract assertions with `const`-ification. In both case studies, libraries with existing mature assertion macros were analyzed, and an important part of such mature systems is that they are already tested and mostly correct; any critical bugs that might be caused by destructive contract assertions have likely been found and fixed long before the code in question was inspected. Therefore, the primary takeaway from these studies should be about ease of migration, not effectiveness at detecting bugs; the details of the bugs that would be detected are lost in the time spent already debugging and fixing those issues when they crept through development and testing processes and resulted in costly production issues.

2.2 Why Not `const`-ification?

The reasoning and case studies mentioned above suggest that `const`-ification can improve the ease and reliability of writing correct and nondestructive contract-assertion predicates. Of course, a number of major concerns arise with attempting to apply `const`-ification to contract-assertion predicates.

1. The reasons for selecting the entities chosen for `const`-ification by [P2900R8] are not obvious; why only local non-static variables?[4]
2. Modifying variables directly is not allowed, but modifications to objects pointed to by pointers can freely happen[5]:

```
void f(int * x)
  pre( x = nullptr )  // Error, x is const.
  pre( *x = 5 );      // Ok?
```

[4]The current limitation of `const`-ification is that it applies to only variables that are directly part of a function invocation since those are the most likely to be relevant to the correctness of that function's behavior. Variables with broader lifetimes not started or ended by the function invocation are much more likely to have state related to diagnostics or tracing and to not directly impact function correctness. Proposal B and Proposal C below explore the pros and cons of altering this decision.

[5]Proposal E below provides an alternative to address this concern.

3. Changing the overloads selected by expressions in a contract-assertion predicate — or the types deduced within those expressions — can have subtle and breaking implications compared to the natural assumption many would have that the expression will mean the same thing it means when used outside the contract-assertion predicate.[6]

   The simplest expression of this concern is that users will be surprised to see a contract assertion pass and then subsequently to see the same expression used in an `if` statement and not take the `true` branch of that conditional:

```
bool isConst(int& x) { return false; }
bool isConst(const int& x) { return true; }

void f(int x) pre( isConst(x) )
{
  if (isConst(x)) {
    std::cout << "Good!\n";
  }
  else {
    std::cout << "How Did I Get Here?\n";
  }
}
```

   Should a contract predicate be written that uses any form of type deduction to produce values based on whether function arguments are `const`, we would then, of course, see cases in which a contract assertion did not produce the result intended by the user. Consider, for example, an associative container that can be *locked* during evaluation so that it does not allow modifications:

```
template <typename K, typename V>
class MyMap {
  // ...
  void lock()   { d_locked = true; }
  void unlock() { d_locked = false; }
  //
};
```

   Unlike `std::map`, this container provides both `const` and non-`const` overloads of `operator[]`. Both the `const` overload and, when `d_locked` is `true`, the non-`const` overload will throw an exception when called with a key that is not in the map.

   The `const` overload is straightforward, throwing when a key is not found:

```
template <typename T, typename V>
V& MyMap<T,V>::operator[](const K& key) const
{
  auto* item = find(key);
  if (nullptr == item) {
    throw MissingItemError();
  }
```

[6] Proposals 1–4 suggest alternate ways to make potential modifications ill-formed without relying on changing expressions to be `const`.

```
  return item->value();
}
```

The non-`const` overload will add a new entry to the map in such cases as long as the map is not locked:

```
template <typename T, typename V>
V& MyMap<T,V>::operator[](const K& key)
{
  auto* item = find(key);
  if (nullptr == item) {
    if (d_locked) {
      throw MissingItemError();
    }
    else {
      item = insert(key, T{});
    }
  }
  return item->value();
}
```

Identifying when such a map is modifiable might be useful. For a non-`const` instance of `MyMap`, the map is modifiable when the `d_locked` flag is `false`:

```
template <typename T, typename V>
bool MyMap<K,V>::isModifiable() { return !d_locked; }
```

A `const` map, however, is never directly modifiable:

```
template <typename T, typename V>
bool MyMap<K,V>::isModifiable() const { return false; }
```

When the map is used in a precondition, we find ourselves getting a less than useful result. A function might use `!isModifiable()` to indicate that it expects its parameter to be passed in while in a locked state[7]:

```
template <typename K, typename V>
void f(MyMap<K,V> &map)
  pre(!map.isModifiable());
```

Now the above precondition is vacuous due to `const`-ification, and the bug of `f` being invoked with a nonlocked map is going to go undetected.[8]

4. For any type, essentially two distinct interfaces are potentially exposed to users — one that is `const` and one that is non-`const`. After all, we treat `T` and `const T` as different types within the language for a reason.

[7]If a function has a precondition that its argument not be modifiable, C++'s historical answer to this situation is to simply take the function parameter by `const&`, and then this runtime issue will not arise.

[8]There is, of course, a teaching moment here in terms of writing good contract assertions; they should always be checking the actual condition they want to check, not something that is assumed to be equivalent. In this case, the contract that the parameter not be locked is fundamentally *not* equivalent to the question of whether the contract-assertion predicate is able to modify the map, and the bug described here arises from that disconnect.

Any function as written will be using the objects accessible to it through one of these two interfaces, and its correctness will depend on which of those interfaces is being used. Having contract-assertion predicates silently be based on only the `const` interface means they can also silently be checking properties that differ from the interface that will be used by the function implementations themselves.

In almost all existing places in the language, code must be explicit when a switch is made from using the non-`const` interface of an object to using the `const` by binding to a `const` reference or pointer or by invoking a `const` member function (which binds an object to a `const` version of `this`).

This mismatch between which interface is being used by contract assertions and by the code they seek to guard is another way to view the underlying concern that leads to the issues described above with `MyMap::isModifiable`.

5. Undisciplined programmers and many legacy codebases do not consistently deploy `const`-correct programming styles, so many APIs that might be completely outside a user's control become significantly more difficult to use when `const_cast` must be deployed to invoke functions that are known to be nonproblematic:

```
namespace oldLib {
  struct OldMechanism { /* ... */ };
  bool isGood( OldMechanism &mech );  // does not modify mech
}
void f(OldMechanism& mech)
  pre( isGood(mech) )  // Error, no overload found.
  pre( isGood(const_cast<OldMechanism&>(mech)) );  // Ok, but ick.
```

Even the workaround above is problematic in many codebases due to a mandate to never use `const_cast` under any circumstances.[9] Should a codebase allow it, of course, `const_cast` is still verbose, macros to simplify it encounter common stigmas against any use of macros, and what remains is a motivation to refrain from using Contracts to increase program correctness from a population that could probably benefit most from its use.

6. A recent paper, [P3478R0], cites four concerns with `const`-ification in [P2900R8].

   (a) "The feature gives novel semantics to expressions in CCAs. This will be surprising to many users and will harm the adoption of contracts."

   Cases already exist in which the same identifier in different locations can be `const` or non-`const`. The most common example is in `const` member functions, though some claim that this behavior is unsurprising because of the (possibly far away yet still on the same function) visible use of `const`. Another common example is the implicit `const` applied to by-value captures in a lambda, another choice of the C++ language that rarely hinders users and where users are more than capable of learning to use the available escape hatch (applying `mutable` to the lambda, in this case). In Section 4.4, we discuss extensively the escape hatches to `const`-ification that are already available and how we could extend them with language features in the future.

[9] A codebase that combines both code that cannot be made `const`-correct and code that must never use `const_cast` probably also has other problems with reconciling the interaction between old and modern software.

The downsides of writing destructive contract assertions, however, are significant, and having the language steer users away from destructive contract assertions is worth the effort. In particular, nothing will harm the adoption of Contracts more than a user assuming that contract predicates can be used to *correct* bugs or as a shorthand for an `if` statement and then discovering that their production release is unquestionably broken when contract checking is turned off. Each step we take to prevent that situation while allowing users to write the checks that are actually meaningful to their programs increases the adoptability of Contracts as well as users' ability to independently learn how to use the Contracts facility.

(b) "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."

Simply applying `const`-ification to existing codebases has already found and fixed real bugs. More importantly, `const`-ification would have prevented many costly problems — that those who support in-house contract-checking facilities often have to help diagnose and fix — where users have employed assertions without thinking about how their semantics differ from an `if` statement.

No management decision will ever be made to disallow the Contracts facility simply because it is slightly harder to apply to codebases that are not `const`-correct after 30 years of having `const` in the language. A massive outage whose root cause is described as "we released a build where we used Contracts extensively and nothing worked," which can be the direct result of testing a system with destructive contract-assertion predicates enabled and then releasing an unchecked build, will swiftly lead to a management dictum to never use Contracts again.

(c) "Constification changes overload resolution results."

This statement is unquestionably true; making expressions `const` that would not otherwise be `const` will change overload resolution involving those expressions. On the other hand, we must consider when (if ever) this actually matters for expressions used in a contract-assertion predicate.

Some of the ramifications of this are explored more thoroughly in the concerns about different forms of `const`-ification explored in Section 4.1. In all cases, however, actual problems arise from broken overload sets or those that produce non–Liskov-substitutable results when invoked on non-`const` references instead of `const` ones. Such designs are already incredibly fragile to use in C++, and no meaningful *complete* example where contracts that use such functions would currently be employed has been offered.

(d) "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)."

None of the proposed features in [P3285R0] has been presented in an actionable and usable fashion. In particular, the rules proposed for contract assertions are so strict and highly limiting as to be of questionable use and provide absolutely no protections to any `relaxed` contract-assertion predicates. Given how limiting the rules would be, most moderate users of Contracts at anything beyond the most trivial level (such as those who use objects or the Standard Library) will most likely simply macro away any use of `strict` contracts and thus have no protection.

`const`-ification, on the other hand, rarely needs escape hatches applied, and when it does, they are applied in specific, local ways within an expression, thus disallowing or discouraging any attempt to turn off the feature entirely.

2.3 Success of `const`-ification

A central theme of the reasons to introduce `const`-ification of some sort to contract-assertion predicates is to help identify contract assertions with side effects that are likely to make them destructive. The most vocally expressed concerns with `const`-ification, however, relate to problems that might arise due to the change in semantics an expression has when used within a contract-assertion predicate and outside of it.

2.3.1 Decision Tree

The decision tree in Figure 1 illustrates all the various decisions that we must make when analyzing whether a given contract-assertion predicate is positively or negatively impacted by `const`-ification.

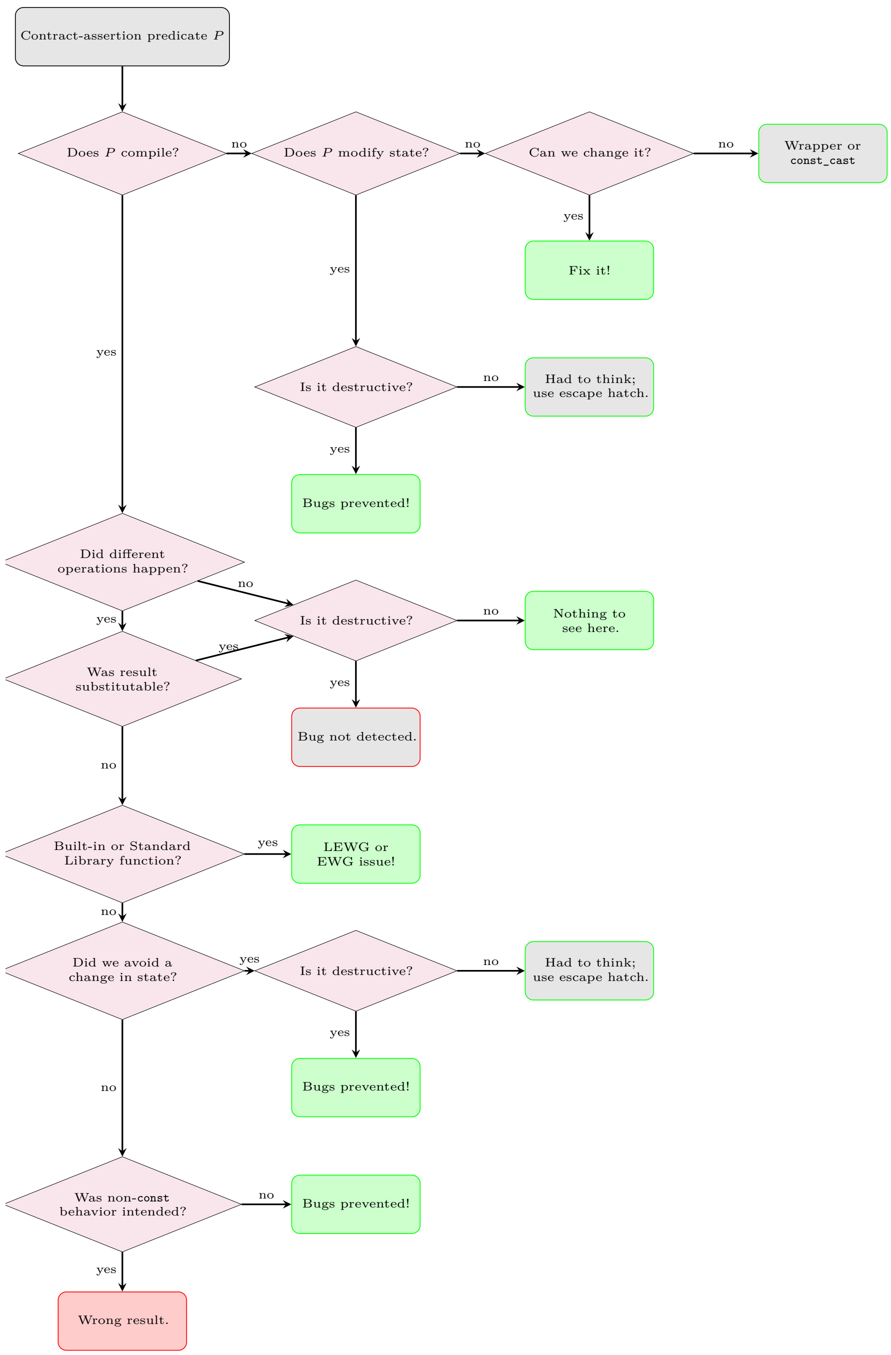
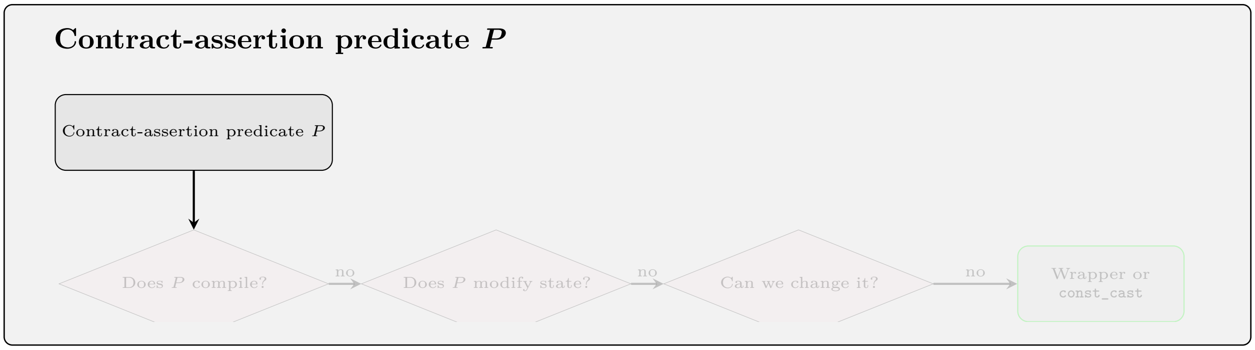
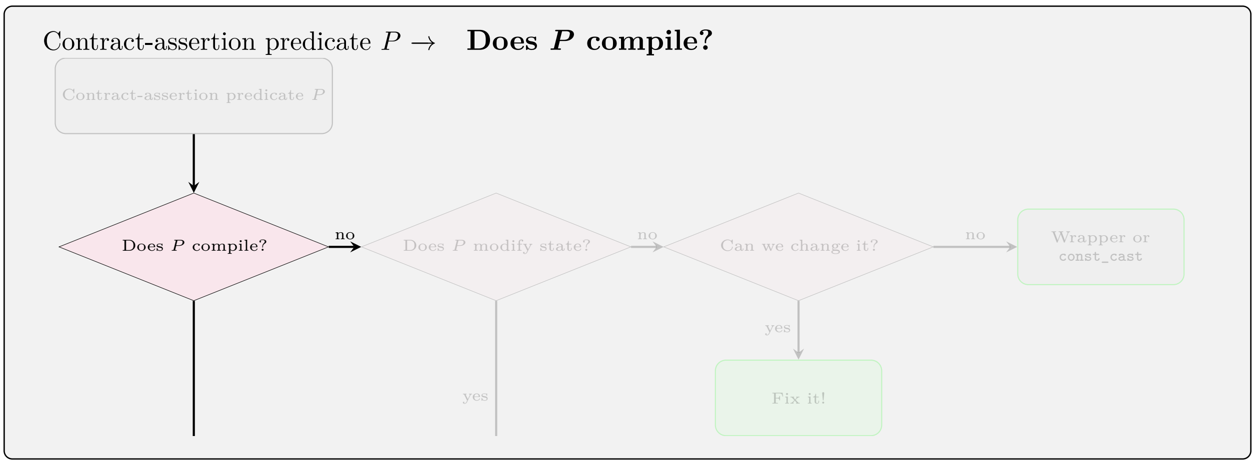


Figure 1: Does `const`-ification achieve its goal?



Given those opposing concerns, we can look at a tree of questions that tells us, for different circumstances, whether `const`-ification produces a good or a bad result. As our starting conditions, we will consider a contract assertion with a predicate, P, and analyze whether our result is good or bad when we apply the rules of `const`-ification, which are a combination of Proposal 5 and Proposal C, that are the current status quo in the Contracts MVP ([P2900R9]).



The first question to ask is whether `const`-ification made P fail to compile. P might not compile for many reasons when certain identifiers are `const` that were not previously `const`.

- Built-in operations that require a non-`const` lvalue might now be given a `const` lvalue, such as assignment operations:

  ```
  void f(int i) pre(( i = 5 ));
  ```

- Overload resolution might now not find a valid overload that takes a `const` reference to an object that has been `const`-ified:

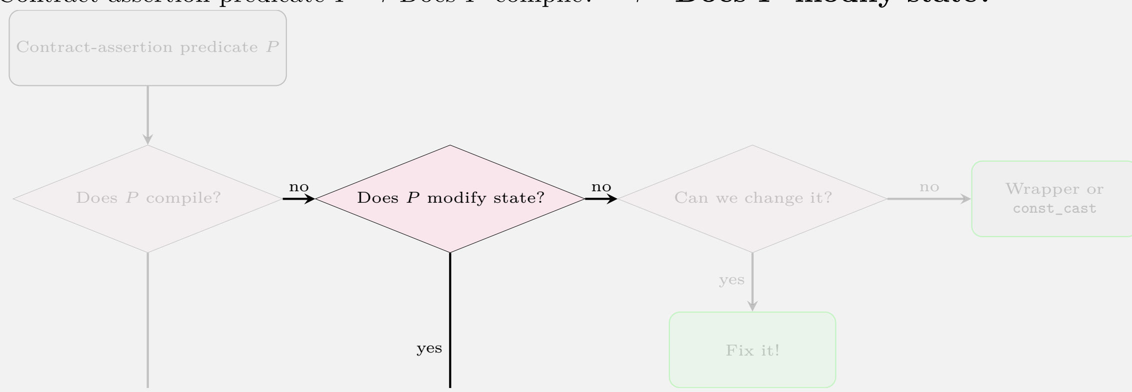
  ```
  void f(vector<intl> v) pre(( v.emplace_back(17) ));
  ```

- Overload resolution on a `const` variable might now be ambiguous:

  ```
  // best candidate for char lvalue
  bool f(char& c);

  // ambiguous candidates for const char lvalue
  template <typename T> requires (sizeof(T) < 4)
  ```

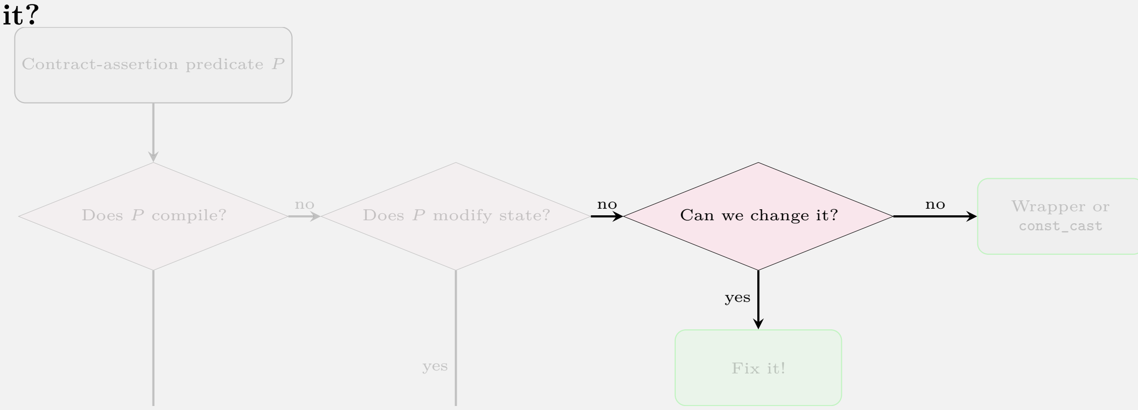
```
bool f(T t);
template <typename T> requires (std::is_integral<T>)
bool f(T t);

void f(char c) pre(f(c));
```

Contract-assertion predicate $P \rightarrow$ Does P compile? $\xrightarrow{no}$ **Does P modify state?**



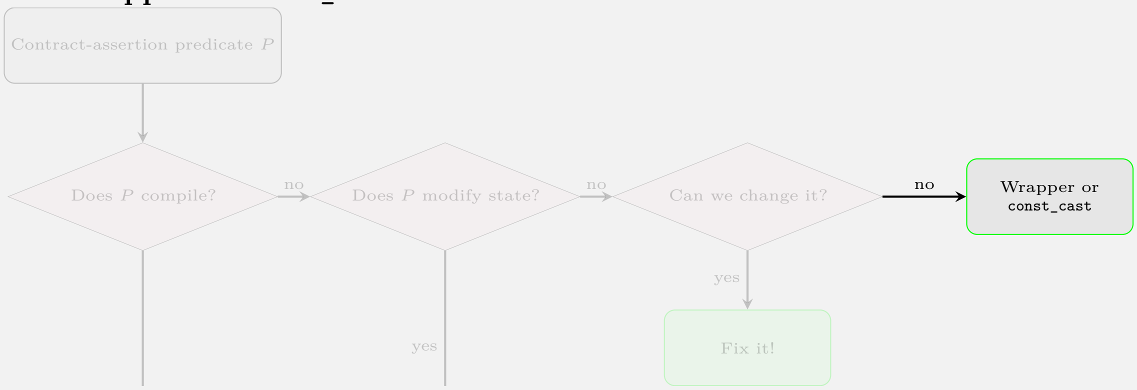
When an expression fails to compile, the first thing to then ask is whether the expression without `const`-ification would have modified state since avoiding modifications is the primary purpose of `const`-ification.

Contract-assertion predicate $P \rightarrow$ Does P compile? $\xrightarrow{no}$ Does P modify state? $\xrightarrow{no}$ **Can we change it?**



For functions that do *not* modify state, we have a contract-assertion predicate, P, that is likely not destructive and that we intend to evaluate but cannot. The solution we can apply depends first on whether we can change the functions being invoked.

Contract-assertion predicate $P \rightarrow$ Does P compile? $\xrightarrow{no}$ Does P modify state? $\xrightarrow{no}$ Can we change it? $\xrightarrow{no}$ **Wrapper or `const_cast`**



When given a function that modifies no state but that is in a library we do not own and thus cannot change, we must instead employ an escape hatch to invoke it. While not the easiest solution, these approaches are all valid and viable. In addition, programmers encountering this situation have been guided by the language to consider carefully the invocation of a potentially modifying function from within a contract-assertion predicate, but they are never outright prevented from doing so.

The first option is to write a simple, reusable wrapper around the external function, which provides an indirect mechanism to use the external function in a `const`-correct manner:

```
namespace other_lib {
  struct S { /* ... */ };
  bool is_valid(S& s);
}

namespace my_lib {
  bool is_valid(const other_lib::S& s)
  {
    return other_lib::is_valid( const_cast<other_lib::S&>(s) );
  }

  void f(S s)
     pre( my_lib::is_valid(s) );
}
```

The second option is to employ a `const_cast` within the contract predicate itself to directly invoke the third-party API that does not support `const` usage:

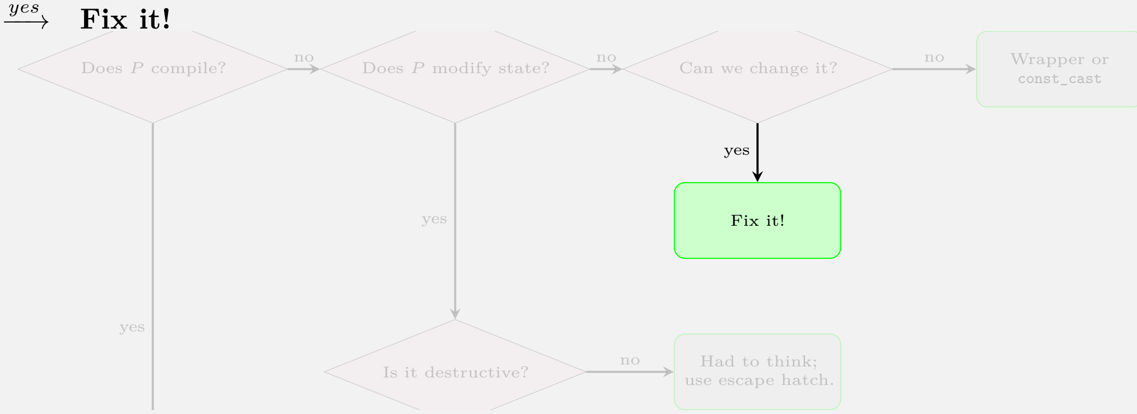
```
void f(S s)
  pre( other_lib::is_valid( const_cast<S&>(s) ) );
```

Finally, we can use a macro to avoid unneeded manual type deduction with the above `const_cast` as well as with many cases in which a `const_cast` might be blatantly removing `const` from something that is declared as `const`:

```
// some_utility_header.h
#define UNCONST(x) const_cast<std::add_lvalue_reference_t<decltype(x)>>(x)
```

```
// in my library
void f(S s)
  pre( other_lib::is_valid( UNCONST(s) );
```

Contract-assertion predicate $P \to$ Does P compile? $\xrightarrow{no}$ Does P modify state? $\xrightarrow{no}$ Can we change it? $\xrightarrow{yes}$ **Fix it!**



When a library can be changed, of course, the best result is to leave the contract-assertion predicate unchanged and make the API being used `const`-correct.

In some cases, making the API `const`-correct is a matter of simply adding `const` to a member function that could have and should have been `const` all along. So a simple container, such as

```
template <typename T>
class SimpleContainer
{
  // ...
  std::size_t size() { return d_size;      }  // forgot const
  bool empty()       { return d_size == 0; }  // forgot const
  // ...
};
```

would be modified to have `const` qualifiers on its members, such as

```
template <typename T>
class SimpleContainer
{
  // ...
  std::size_t size() const { return d_size;      }  // fixed
  bool empty() const       { return d_size == 0; }  // fixed
  // ...
};
```

No other code, barring the remote possibility that some user was assigning these functions to a member function pointer that was not itself `const`-qualified, would need to be changed.

Other cases, however, involve adding a second overload that is `const`-qualified to expose subobjects by `const` reference instead of non-`const` reference:

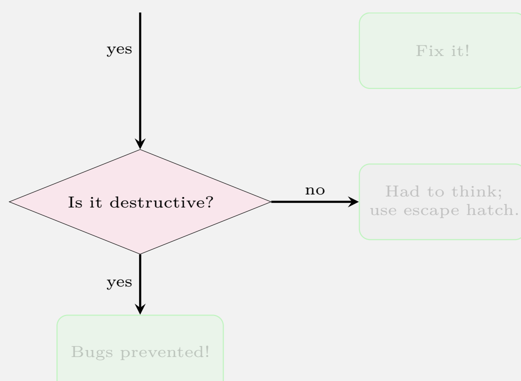
```
template <typename T>
class SimpleContainer
{
  // ...
  T& at(std::size_t index)   // provides non-const reference
  { return d_data[index]; }
  // ...
};
```

In this case, an additional overload with a syntactically identical implementation must be added:

```
template <typename T>
class SimpleContainer
{
  // ...
  T& at(std::size_t index)              // provides non-const reference
  { return d_data[index]; }
  const T& at(std::size_t index) const  // provides const reference
  { return d_data[index]; }
  // ...
};
```

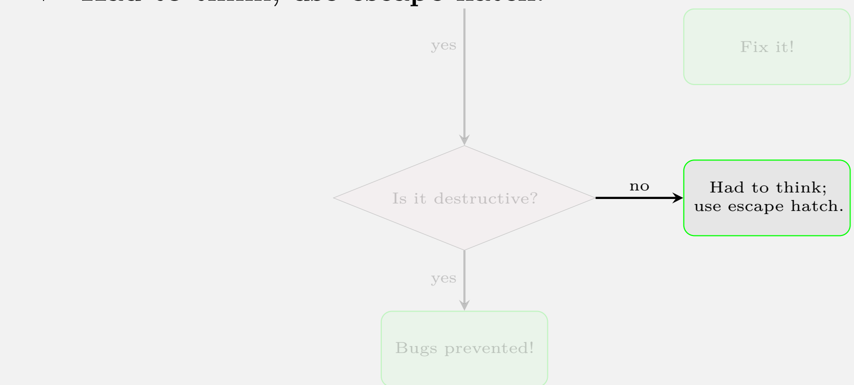
The only code that would need to be changed would be code that used a `const SimpleContainer<T>` and then attempted to modify its elements accessed through `at`, which is almost certainly a bug that is worth uncovering.

Contract-assertion predicate $P \to$ Does P compile? $\xrightarrow{no}$ Does P modify state? $\xrightarrow{yes}$ **Is it destructive?**



When modifications are happening, we must know if the modifications are destructive before we can determine the extent of the benefits of `const`-ification.

Contract-assertion predicate $P \rightarrow$ Does P compile? $\xrightarrow{no}$ Does P modify state? $\xrightarrow{yes}$ Is it destructive? $\xrightarrow{no}$ **Had to think; use escape hatch.**

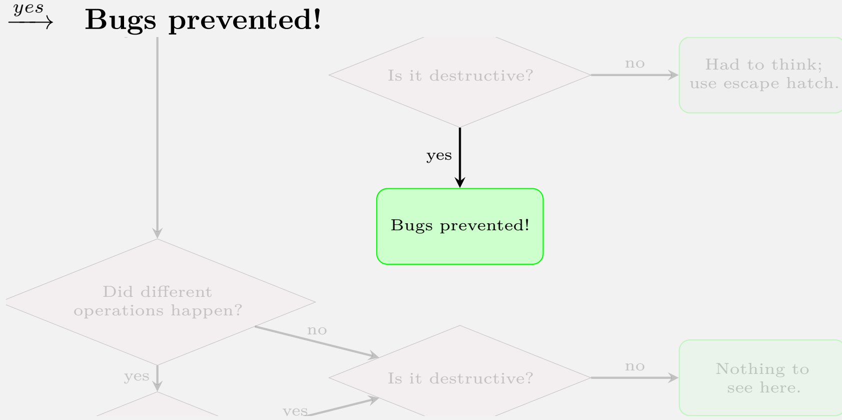


For a nondestructive predicate that does modify state, we have been forced to assess this situation due to `const`-ification. Making the API `const`-correct would be inappropriate in this scenario since the API does make modifications to state.

A common example of this sort of API is a function, such as `std::map::operator[]`, that inserts elements into the container when they are not found. Used carefully in a contract assertion that has checked that the index being used is already in the container, this function is viable,[10] but when used haphazardly, it can cause contract assertions to fill a map with unexpected entries that will simply not be there when assertions are disabled.

Therefore, in most such cases in which we have a modification that is both not destructive and one we want our contract-assertion predicate to make, the ideal solution is to indicate clearly that this reasoning has occurred by applying a `const_cast` or an `UNCONST` macro within the contract-assertion predicate itself.

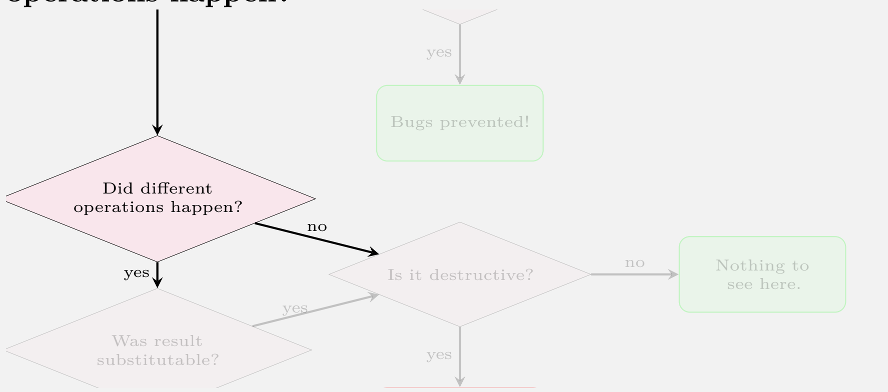
Contract-assertion predicate $P \rightarrow$ Does P compile? $\xrightarrow{no}$ Does P modify state? $\xrightarrow{yes}$ Is it destructive? $\xrightarrow{yes}$ **Bugs prevented!**



[10] Alternatively, an actual `const` API to access elements of a map, such as that proposed by [P3091R2], could be used.

When `const`-ification has identified a destructive predicate and triggered a failure to compile, it has perfectly achieved the goal for which it was designed. Success and profit will be had by all for avoiding such destructive contract-assertion predicates.

Contract-assertion predicate $P \to$ Does P compile? $\xrightarrow{yes}$ **Did different operations happen?**



For a predicate, P, that *does* compile, we must then identify if any of the operations evaluated as part of P were different from those that would have been evaluated if we did not have `const`-ification.

Some predicates, for example, are identical with constification:

```
void f(int i, std::vector<int> v)
  pre( i * 2 < 15 )
  pre( !v.empty() )
  pre( v.size() > 3 );
```

Some things that are subject to `const`-ification are also already `const`:

```
void g(const std::vector<int>& v)
  pre( std::is_sorted(v.begin(), v.end() ) );
```

In other cases, of course, a different overload will have been selected:

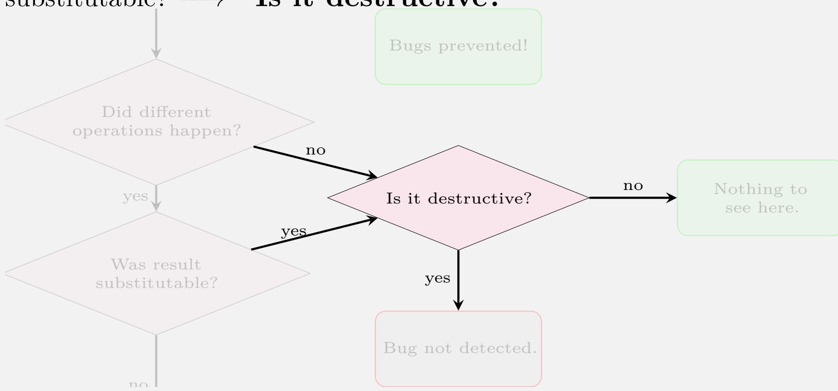
```
void g(std::vector<int>& v)
  pre( std::is_sorted(v.begin(), v.end() ) );
```

Another concern with invoking different operations is the impact that doing so might have on static analysis due to a lack of expression equivalence between the same expression used in a contract-assertion predicate and used outside as a conditional expression in an `if` statement. This kind of analysis, of course, can only be applied when additional annotations or assumptions are in play — e.g., the guarantees provided by `[[gnu::pure]]`, which was used in the example presented at the end of [P3478R0]. The problem with this concern, again, is where we would employ a *pure* function that overloads on whether its argument is `const` or not.

Contract-assertion predicate $P \to$ Does P compile? $\xrightarrow{yes}$ Did different operations happen? $\xrightarrow{no}$ **Is it destructive?**

or

Contract-assertion predicate $P \to$ Does P compile? $\xrightarrow{yes}$ Did different operations happen? $\xrightarrow{yes}$ Was result substitutable? $\xrightarrow{yes}$ **Is it destructive?**



When `const`-ification changed nothing or produced a correctly substitutable result, we must then identify whether we have missed an opportunity to detect a buggy contract-assertion predicate.

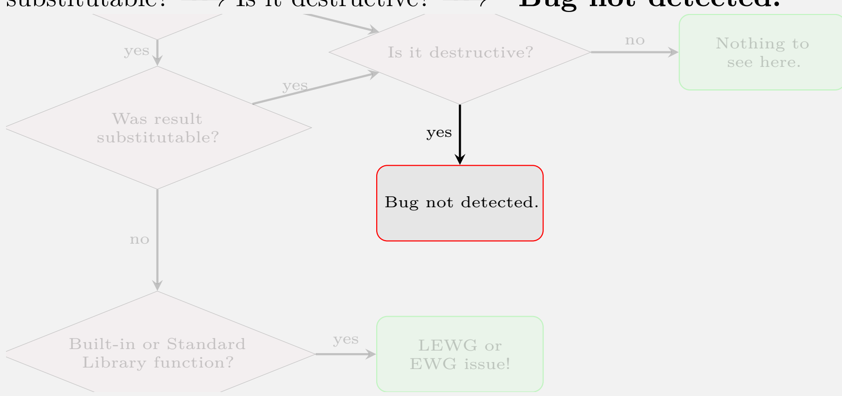
For example, because we are not applying *deep* `const` to things that are `const`-ified, modification can still be made through pointers:

```
void f(int *p)
  pre(( *p = 5 ));
```

Similarly, when handed a user-defined type with pointer semantics, the same thing will happen:

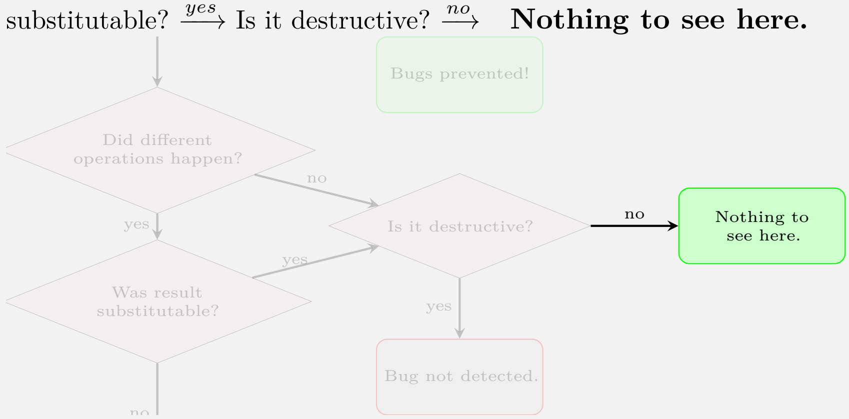
```
void f(std::shared_ptr<int> p)
  pre(( *p = 5 ));
```

Contract-assertion predicate P → Does P compile? $\xrightarrow{yes}$ Did different operations happen? $\xrightarrow{no}$ Is it destructive? $\xrightarrow{yes}$ **Bug not detected.**
or
Contract-assertion predicate P → Does P compile? $\xrightarrow{yes}$ Did different operations happen? $\xrightarrow{yes}$ Was result substitutable? $\xrightarrow{yes}$ Is it destructive? $\xrightarrow{yes}$ **Bug not detected.**



As with *all* solutions that have been considered for mitigating destructive predicates, `const`-ification is unable to prevent all possible cases. Given a destructive predicate that `const`-ification does not meaningfully change, a buggy predicate will remain a buggy predicate.

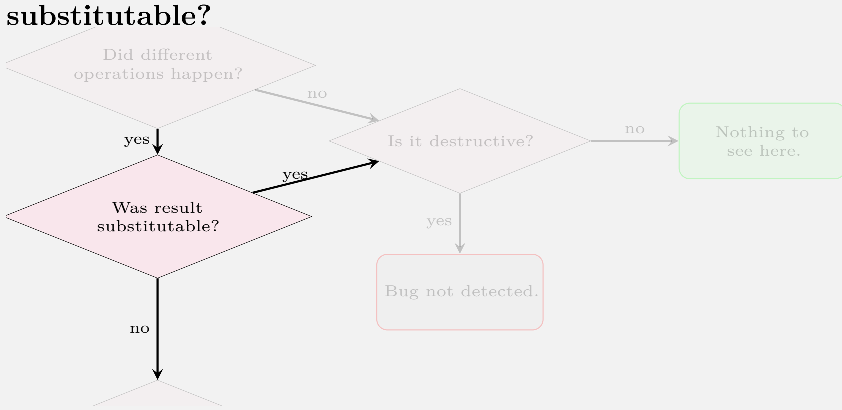
Contract-assertion predicate P → Does P compile? $\xrightarrow{yes}$ Did different operations happen? $\xrightarrow{no}$ Is it destructive? $\xrightarrow{no}$ **Nothing to see here.**
or
Contract-assertion predicate P → Does P compile? $\xrightarrow{yes}$ Did different operations happen? $\xrightarrow{yes}$ Was result substitutable? $\xrightarrow{yes}$ Is it destructive? $\xrightarrow{no}$ **Nothing to see here.**



Of course, when `const`-ification makes no changes to the predicate and the predicate is *not* destructive, users are left with exactly the same valid contract-assertion predicate they would have under any other scenario (that did not restrict them from writing any meaningful predicates in the first place).

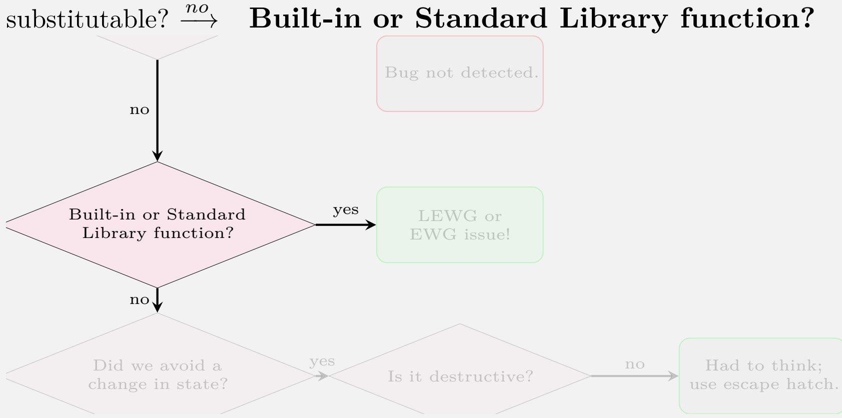
This scenario is likely to be the most common one when writing contract assertions. Simple expressions, especially those with scalar types that do not misuse assignment operators, will all fall into this category.

Contract-assertion predicate P → Does P compile? $\xrightarrow{yes}$ Did different operations happen? $\xrightarrow{yes}$ **Was result substitutable?**



Even when different operations happen, the common recommendation for design in C++ is to have `const` overloads of functions, when they are present, behave in a manner that is Liskov-substitutable for the non-`const` overloads.

Contract-assertion predicate P → Does P compile? $\xrightarrow{yes}$ Did different operations happen? $\xrightarrow{yes}$ Was result substitutable? $\xrightarrow{no}$ **Built-in or Standard Library function?**



Now we can decide if the function that produces a meaningfully different result when given a `const` parameter compared to when it is given a non-`const` parameter is one that is provided by the core language or Standard Library.

Of course, functionality that works on *types* and inspects whether they are `const` is an essential part of the Standard Library, and their use when applied to the types of `const`-ified *expressions* can still result in functions with nonsubstitutable results:

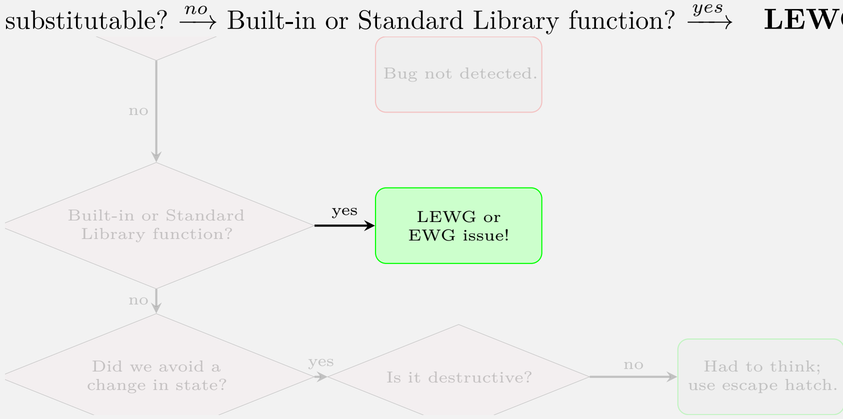
```
void f(int x)
  pre( std::is_const_v<decltype((x))> );  // always fails
```

What the Standard Library does not provide, however, is a *function* that does the same thing based on the types deduced for its argument:

```
template <typename T>
bool is_const(T&& t) { return std::is_const_v<T>; }  // not in std
```

Such a function would produce a nonsubstitutable result when given a `const` expression versus a non-`const` one — `true` in the first case, and `false` in the second. On the other hand, in generic code, a function like this one has never been deemed useful or usable enough to include in the Standard Library.

Contract-assertion predicate $P \rightarrow$ Does P compile? $\xrightarrow{yes}$ Did different operations happen? $\xrightarrow{yes}$ Was result substitutable? $\xrightarrow{no}$ Built-in or Standard Library function? $\xrightarrow{yes}$ **LEWG or EWG issue!**



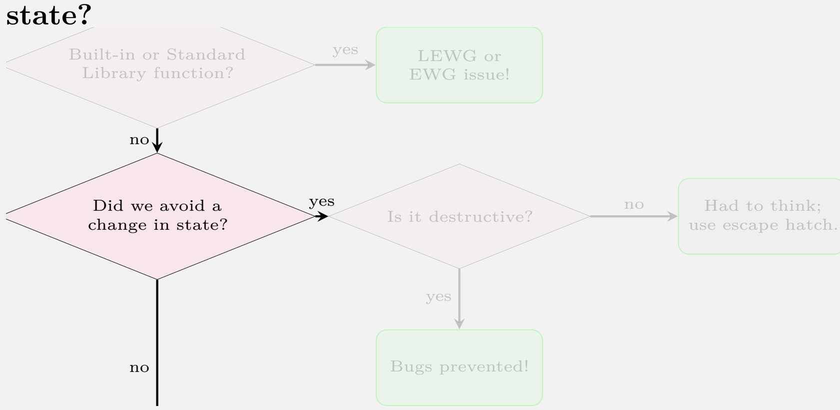
In all examples that we know of today, the core language and Standard Library provide a `const`-substitutable interface to anything that can be done with both `const` and non-`const` objects. The difficulty of dealing with APIs where this guideline is not upheld is so significant that any cases where this situation is encountered in the core language or Standard Library should be raised as major issues with the associated evolution groups.

Even operators that mix computation on objects with computation on types, such as `typeid`, ignore cv-qualifiers on objects passed to them.

Subtle cases where this design has not been upheld have occurred in the past. Consider, for example, the pre-C++11 support that was allowed for copy-on-write strings. In those cases, an invocation of the non-`const` `begin()` or `end()` members would instigate a copy if the string did not yet have unique ownership of its contents and would thus invalidate any iterators that were obtained through that string. Among many other reasons, this subtle variation between `const` and non-`const` behaviors eventually led to removing the allowance for copy-on-write strings and is a reason that copy-on-write containers in general are hard to deploy in C++.

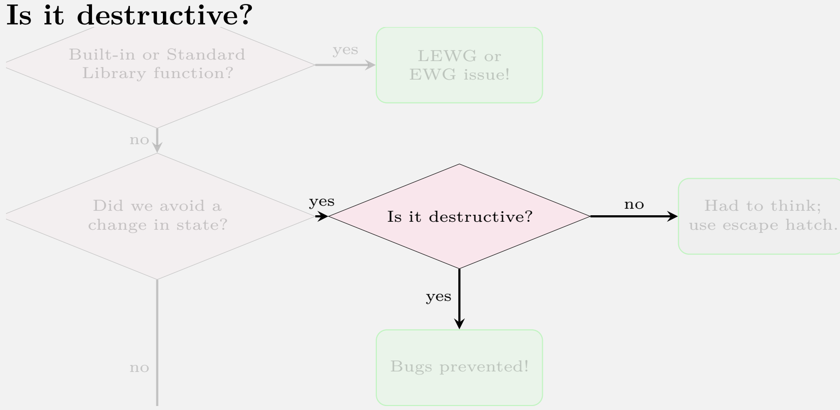
While the primary motivation for removing support for strings with copy-on-write semantics was the many concurrency issues that relate to such support, as described in [N2668], at the heart of these issues lie problems related to attempting to provide nonsubstitutable behavior for the non-`const` `begin()` and `end()` member functions of `string`.

Contract-assertion predicate $P \rightarrow$ Does P compile? $\xrightarrow{yes}$ Did different operations happen? $\xrightarrow{yes}$ Was result substitutable? $\xrightarrow{no}$ Built-in or Standard Library function? $\xrightarrow{no}$ **Did we avoid a change in state?**



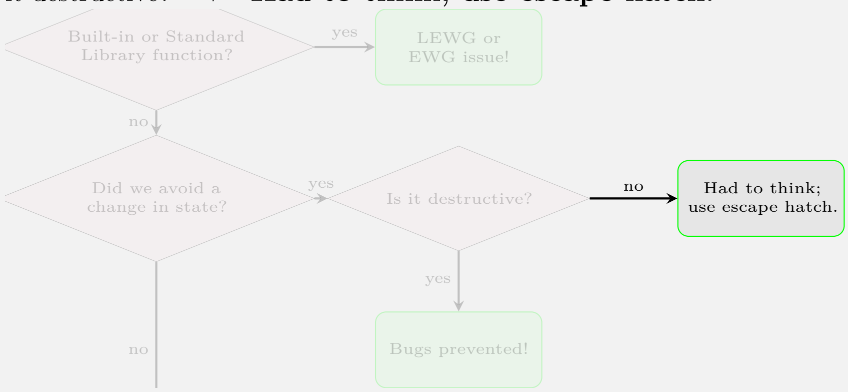
Now we are considering predicates, P, where the non-`const`-ified version of the predicate produced a different state than the `const` version did. Here we would treat a change in state that was avoided by `const`-ification differently than a function that simply produced a different result.

Contract-assertion predicate $P \rightarrow$ Does P compile? $\xrightarrow{yes}$ Did different operations happen? $\xrightarrow{yes}$ Was result substitutable? $\xrightarrow{no}$ Built-in or Standard Library function? $\xrightarrow{no}$ Did we avoid a change in state? $\xrightarrow{yes}$ **Is it destructive?**



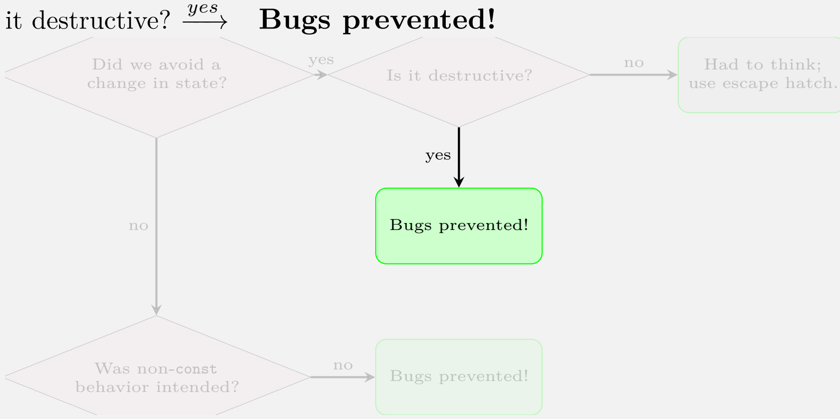
As usual, the first consideration when thinking about a contract-assertion predicate that modified state is to identify whether the predicate was destructive, and that answer will tell us how effectively `const`-ification helped us.

Contract-assertion predicate $P \rightarrow$ Does P compile? $\xrightarrow{yes}$ Did different operations happen? $\xrightarrow{yes}$ Was result substitutable? $\xrightarrow{no}$ Built-in or Standard Library function? $\xrightarrow{no}$ Did we avoid a change in state? $\xrightarrow{yes}$ Is it destructive? $\xrightarrow{no}$ **Had to think; use escape hatch.**



When switching from a predicate that produced a different value *and* changed program state but is somehow still not destructive to a predicate that is non-`const`, we have introduced a problem for users that they must think about, but we have also required that they explicitly opt-in to modifying state by applying an escape hatch. Overall, this mild imposition is a benefit to users.

Contract-assertion predicate $P \rightarrow$ Does P compile? $\xrightarrow{yes}$ Did different operations happen? $\xrightarrow{yes}$ Was result substitutable? $\xrightarrow{no}$ Built-in or Standard Library function? $\xrightarrow{no}$ Did we avoid a change in state? $\xrightarrow{yes}$ Is it destructive? $\xrightarrow{yes}$ **Bugs prevented!**

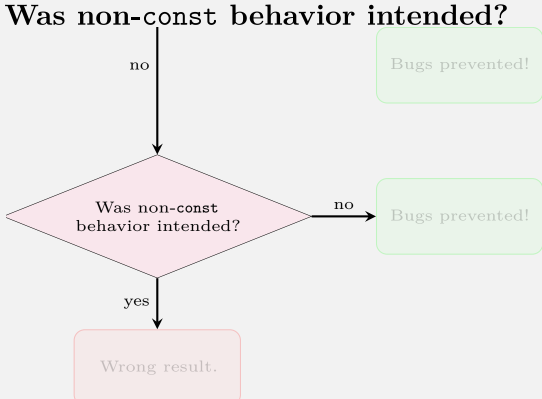


When switching to a different overload — even one that produced a different result — avoided a destructive alternative, we have unquestionably helped the program.

Consider the failure scenario here and how remarkably hard debugging the alternative is. When given a non-`const` function that modifies state and alters the correctness of a program (i.e., it is destructive), the breakage caused by such a change in state can often be very remote from the contract predicate that made the change. Far worse, however, is that this breakage is only identifiable by comparing the states of a program in which the contract-assertion predicate is evaluated to those of a program in which the predicate is not evaluated; i.e., two different builds of the program must be debugged in parallel.

Even if the `const` alternative that was selected produces the wrong result for the contract-assertion predicate, identifying that flaw happens relatively quickly, and debugging it requires analyzing only a single build of the program, a much more typical task for the average developer and one with which they are already highly acquainted.

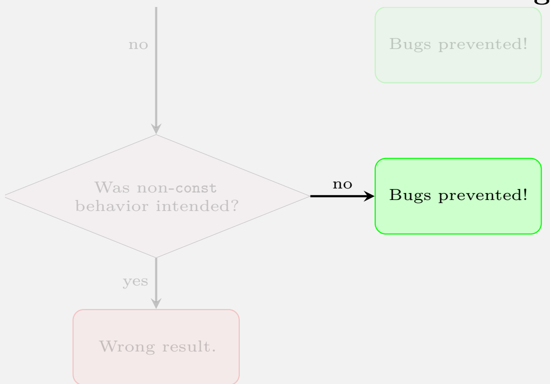
Contract-assertion predicate $P \rightarrow$ Does P compile? $\xrightarrow{yes}$ Did different operations happen? $\xrightarrow{yes}$ Was result substitutable? $\xrightarrow{no}$ Built-in or Standard Library function? $\xrightarrow{no}$ Did we avoid a change in state? $\xrightarrow{no}$ **Was non-`const` behavior intended?**



When the non-`const` alternative produces a different result from the `const` alternative, we must then determine if that different result was actually the result intended by the caller. Consider, for example, the overload sets described in Section 4.1, where in some cases a caller wants a `const` overload but might inadvertently get a non-`const` overload due to polluted overload sets.

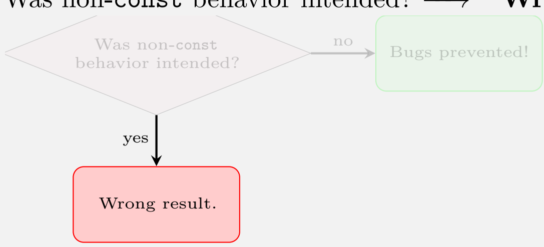
Considering that contract-assertion predicates are generally written with no intent to modify their parameters, a non-const overload being unintended should be quite unsurprising.

Contract-assertion predicate $P \rightarrow$ Does P compile? $\xrightarrow{yes}$ Did different operations happen? $\xrightarrow{yes}$ Was result substitutable? $\xrightarrow{no}$ Built-in or Standard Library function? $\xrightarrow{no}$ Did we avoid a change in state? $\xrightarrow{no}$ Was non-`const` behavior intended? $\xrightarrow{no}$ **Bugs prevented!**



When such a non-`const` overload is unintended but would have been selected without `const`-ification, `const`-ification is clearly a win; the user's intent was captured and applied correctly by the language, even in the presence of a polluted overload set that introduced functions that might have corrupted the results desired by the user.

Contract-assertion predicate $P \rightarrow$ Does P compile? $\xrightarrow{yes}$ Did different operations happen? $\xrightarrow{yes}$ Was result substitutable? $\xrightarrow{no}$ Built-in or Standard Library function? $\xrightarrow{no}$ Did we avoid a change in state? $\xrightarrow{no}$ Was non-`const` behavior intended? $\xrightarrow{yes}$ **Wrong result.**



Finally, we get to the case that is the source of great concern for some users. To get here, we must have a predicate, P, that meets all of the criteria above.

- P still compiles even with `const`-ification applied.
- P contains function calls that resolve differently due to the application of `const`-ification.
- The results of those different overloads are functionally different due to using the `const` overloads.
- The functions being called are all within libraries outside of the Standard Library and not following the examples it sets.

- The non-`const` operations that would be invoked by P modify no state.
- The non-`const` operation's result is the one intended to be used by the user.

The few examples that meet all these criteria invariably are those that attempt to explicitly distinguish between `const` and non-`const` parameters and to treat them differently but to do so based on parameter passing and type deduction, not through metaprogramming. Importantly, any program doing this to avoid using non-`const` functions will still have to resort to metaprogramming to avoid (through `if constexpr` blocks or other overloads) invoking functions that do not work on `const` parameters.

2.3.2 Experimental Results

Next, we can recontextualize some of the experiments that have been done applying implementations of [P2900R9] to real-world codebases.

- **Bloomberg Libraries Analyzed in [P3336R0]**

  In the experience report paper [P3336R0], a number of large libraries that use the `BSLS_ASSERT` family of contract-checking macros were compiled with those macros reimplemented in terms of `contract_assert` and thus with `const`-ification applied.

  - **Does *P* compile? — yes**: 17259/17287. Most contract assertions in these libraries compiled with `const`-ification.

    All the libraries that were examined are fairly thoroughly unit-tested, and all tests pass with assertions enabled or disabled. Therefore, we believe that *none* of the 17259 contract-assertion predicates that compiled are destructive.

    * ✔**Did different operations happen? — no**: ≈ 17120/17259. Manual inspection of a random selection of assertions that compiled without change identified certain common functions that were used that had both `const` and non-`const` overloads — e.g., `begin`, `end`, `front`, `back`, and so on. Expressions that made use of no such functions made up 99.2% of the (approximately 500) assertions that were manually inspected.

    * ✔**Did different operations happen? — yes**: ≈ 139/17259. The remaining ≈ 0.8% of the assertions selected different overloads with substitutable behavior for the non-`const` overloads they replaced.

  - **Does *P* compile? — no**: 28/17287.[11] Contract assertions in these libraries initially failed to compile with `const`-ification.

    * **Does *P* modify state? — no**: 15/28. More than half of the assertions that failed to compile did not modify state.

      · ✔**Can we change it? — yes**: 14/15. For almost all the assertions, a small change was made to make existing code `const`-correct. In one case, a `dynamic_cast` was used that was incorrect when applied to a `const`-ified variable, but changing the target type to itself be `const`-qualified achieved the correct result.

    * **Does *P* modify state? — yes**: 13/28. Less than half of the assertions that failed to compile *did* modify state.

      · ✓**Is it Destructive? — no — Had to check, use escape hatch.**: 4/13. Four of the predicates invoked functions with output parameters that were subsequently discarded, and a `const_cast` to remove `const`-ification in these cases achieved a correct result. In a codebase that did not still support older C++ Standards, the discarded output parameters could be moved into a nested lambda within the contract assertions and no use of `const_cast` would be needed.

[11] Note that the numbers presented in [P3336R0] were counts of issues discovered and bugs fixed, not the exact number of assertions that failed due to each issue, which is the data being presented here.

- ✔**Is it Destructive? — yes — Bugs Prevented!**: 9/13. Most of these cases were destructive checks that should not be in the codebase. Of these, eight were bugs that were quickly fixed and one was an intentional failure that existed as part of the testing framework for part of the `BSLS_ASSERT` contract-checking facility itself.

This data presents an important analysis of production libraries that are in active use with assertions both enabled and disabled. Any serious problems with destructive contract-assertion predicates have already happened in the past and been fixed at the cost of significant developer effort, but none could be expected to have survived in running software for long.

Analysis like this, therefore, is important for identifying how much negative impact applying `const`-ification at scale would cause, but such analysis highly downplays the benefits that would be achieved in reduced costs of writing those systems in the first place.

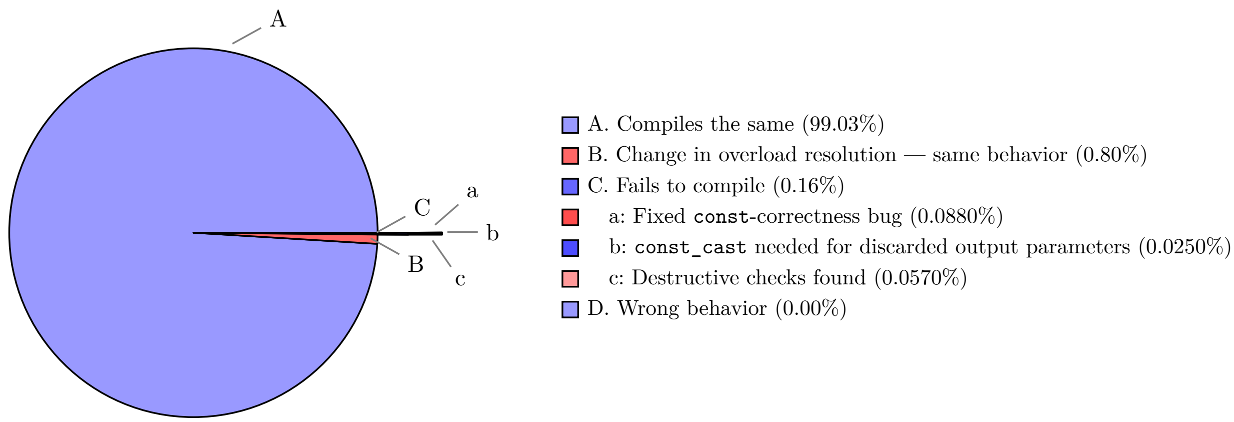


Figure 2: Results for Bloomberg libraries, with “Fails to compile” results subcategorized ([P3336R0])

- **Codebase Analyzed by [P3268R0]**

  Peter Bindels, in a report published as [P3268R0], manually inspected a large codebase to identify the impact of `const`-ification on that code.[12]

  That effort involved inspecting 7747 assertions in a codebase of approximately 2.5 million lines of code. The work done there did not distinguish between `const`-correct functions being invoked that did or did not have non-`const` overload but identified no cases in which there was a concern that `const`-ification would be the cause of an actual problem.

  - ✔**Does *P* compile? — yes**: 7722/7747. Most contract assertions in this codebase appeared ready to compile with `const`-ification. 160 involved functions that had `const` overloads already.
  - **Does *P* compile? — no**: 25/7747. A small fraction of contract assertions in this codebase had issues that would prevent them from compiling.
    - **Does *P* modify state? – no**: 21/25. Most of the predicates definitely do not modify state.
      - ✔**Can we change it? — yes**: 21/25. Eight invoked functions that were not `const`-qualified but which modified nothing and could be marked `const`. Thirteen invoked one of two functions that returned references or pointers to other objects yet lacked `const`-qualified overloads that return `const`-qualified references and pointers; adding the two `const` overloads was seen as an easy fix.
    - ✓**Does *P* modify state? – yes**: 4/25. A few predicates invoked functions that were not easily made `const`-correct, though none were seen as unfixable. These predicates would require minor refactoring or judicious use of `const_cast`.

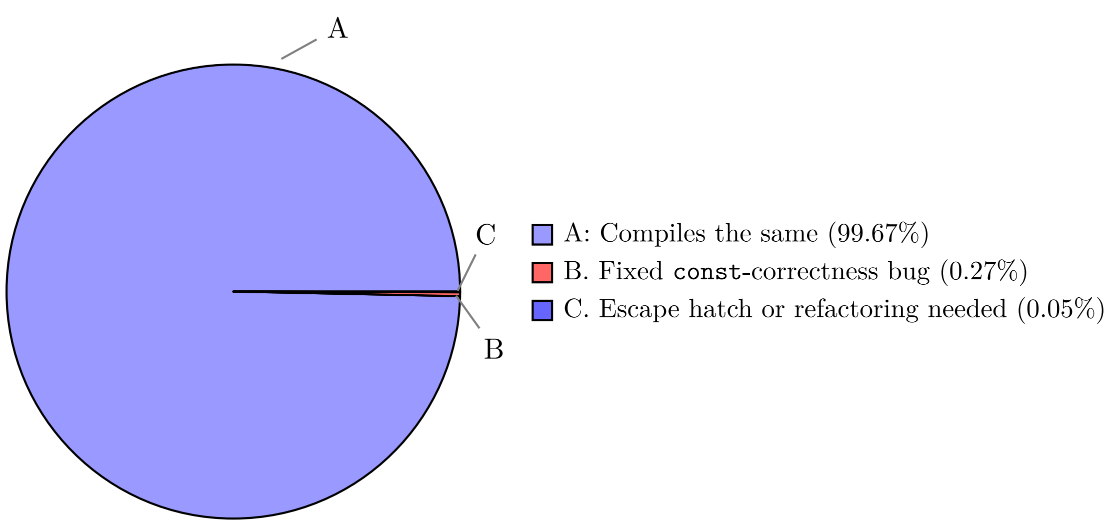


Figure 3: `const`-ification success for codebase in [P3268R0]

[12] Thanks to Peter Bindels for looking over this data to confirm it reflects his original analysis correctly.

LLVM and Libc++

Eric Fiselier has been implementing [P2900R9] in Clang. As part of that process, he has explored two approaches to making use of `contract_assert` with existing assertion facilities:

- In LibC++, 1266 assertions of various forms that currently use existing macro-based facilities were switched directly to `contract_assert`. Due to extensive use of these macros as expressions, this transformation required some creative work within the macro using an expression statement extension. Of the 1266 assertions, 1266 compiled and all tests passed, indicating that there were 0 issues introduced by `const`-ification.
- In LLVM, there were approximately $\sim 40,000$ uses of the `assert` macro. After fixing approximately 75 simple `const`-correctness bugs, $\sim 39,500$ (98.5%) of the assertions compiled. Visual inspection of the remaining assertions that failed to compile proceeded after that:
  - Approximately 75 easily-fixed `const`-correctness errors were uncovered, largely involving `const` not having been put on member functions.
  - Approximately 200 involved functions that had names that indicated access, and were likely functions that could be made `const` if more dedicated time were avialable for the task. Some of these involved data structures that lazily calculated state without storing it in mutable members, so a minor redesign would be needed to make these types `const` correct.
  - Approximately 75 involved using `map::operator[]` on maps that were also conditionally created only when assertions were enabled, a feature not yet supported by [P2900R10].
  - Approximately 200 involved other functions being used on conditionally compiled data structures.

3 Proposals

We can consider altering the design of `const`-ification along two primary axes:

1. How we avoid modification, ranging from compiler warnings, making certain constructs ill-formed, or changing the semantics of contract-assertion predicates by making some sub-expressions `const`
2. To which expressions we apply the above modifications, ranging from id-expressions that denote specific kinds of variables to chaining our reasoning to apply modifications to the results of member access and pointer dereference expressions

3.1 Mechanisms for the Avoidance of Modification

First, we will consider the variety of options regarding how we might address potentially modifying behavior within a contract assertion.

3.1.1 No Semantic Changes

The simplest approach to contract-assertion predicates is, of course, to treat them as any other expression and do nothing to attempt to dissuade destructive predicates. Importantly, this approach does *not* preclude any implementation warnings that might detect destructive contract assertions.

> Proposal 1: No `const`-ification (Only Warnings)
>
> Change nothing about what is allowed in contract-assertion predicates or what semantics any expressions have within contract-assertion predicates.

An important concern with the approach of abandoning `const`-ification and moving to producing warnings instead is whether warnings can identify approximately the same set of errors that our other proposals can. While many top-level errors can be reduced to a warning, a compiler is *not* freely permitted to perform arbitrary additional overload resolutions that are not already required for the existing potentially evaluated expressions in the program. Let's consider an example:

```
void f(std::vector<int> v)
  pre((std::sort(v.begin(), v.end()), true));
```

To determine if the above precondition is valid when the `vector` parameter `v` is treated as `const`, overload resolution must be performed on `std::sort` with parameters that are the return types of `vector<int>::begin() const` and `vector<int>::end() const` respectively, both of which are of the type `vector<int>::const_iterator`. Without `const`-ification, that overload resolution would never be performed. The problem here, however, is that overload resolution must instantiate template declarations, which can result in hard errors, new types being defined, and other observable changes in a program. To implement such a warning in a conforming way, a compiler would need to perform that extra overload resolution and then somehow *unwind* all those changes in state. No such unwinding is currently required anywhere else in the language and would be a huge implementation hurdle to produce. The introduction of more stateful compile-time evaluation to support reflection (see [P2996R5]) will only make such unwinding an even greater burden.

Even the determination of whether an overload set would accept a `const` argument instead of a non-`const` argument can result in hard errors that would require significant compiler efforts to attempt to unwind. An overload set might reject a `const` parameter for not having a member that matches. On the other hand, overload sets containing templates must do template argument deduction to determine if there is a match, and such deduction can fail in ways that are hard errors. Consider, for example, a function template that causes a precondition violation when instantiated with a non-`const` template parameter:

```
constexpr int constexpr_sqrt(int x) pre(x >= 0);

template <typename T>
void f(T&& t)
  noexcept( constexpr_sqrt( std::is_const_v<T> ? -1 : 1 ) );
```

Although odd, the semantics of the language would still restrict a compiler from letting a function template like this cause a warning to turn into a failure to compile, and changing the hard error to something recoverable contextually might require significant compiler re-engineering.

Implementing a warning such as this without the ability to totally unwind any effects of the extra overload-resolution attempts would either lead to a warning that escalates itself into an error or, much worse, a warning that introduces new template instantiations and overloads into a program and then *changes* the semantics of the program. Such changes to a program's semantics would be thoroughly nonconforming and potentially disastrous.

Therefore, we must consider that warnings alone will be unable to detect a similar range of real-world use cases with higher-level abstractions that we can detect with `const`-ification. Additional external tools or recompilation with a different nonstandard approach might be able to produce such warnings, but those solutions are outside the scope of what we aim to consider when deciding upon the best decision for the C++ language itself.

Below, in Section 4.1, we will discuss in more detail which of the other proposals presented here could be replaced in a compiler by warnings.

3.1.2 Prevent Modifying Operators

A second alternative that prevents some modifications is to make certain operators ill-formed when applied to an expression that we have determined should be `const`-ified.

> **Proposal 2: No Assignment Operations**
>
> Any assignment operator (`=`,`*=`,`/=`,`\%=`,`+=`,`-=`,`>>=`,`<<=`,`&=`,`^=`, and `|=`), increment operator (`++`), or decrement operator (`--`) is ill-formed if its modifying operand is subject to `const`-ification.

This first proposal to make operations ill-formed would therefore prevent contract assertions that increment or decrement local variables or that accidentally use `=` in lieu of `==`.[13] On the other hand,

[13] Note that a top-level assignment operation, such as `pre(x=0)`, is already ill-formed due to the choice of *conditional-expression* instead of *expression* in the grammar for contract assertions. Nested assignments, however, such as `pre((x = 0))` or `pre(x == 0 || y = 0 || z == 0)`, are still grammatically correct.

no issues related to modifying member functions or free functions are prevented, and hence this proposal fails to address a wide range of real-world use cases.

Note that this proposal is not limited to only built-in operations and scalar types; it is instead a restriction on the use of the operator syntax when applied to `const`-ified operands, and it will also apply for user-defined types when operators are overloaded.

3.1.3 Prevent Potentially Modifying Invocations

The other polar extreme is to prevent all operations that might be potentially modifying, which would include any function invocations that accept a `const`-ified object by pointer or non-`const` reference.

> Proposal 3: No Potentially Modifying Operations
>
> Any operation that could modify an operand subject to `const`-ification is ill-formed.

This aggressive approach to making operations ill-formed certainly prevents anything that might modify values but will quickly become problematic for all operations that allow for but do not directly perform modifications, such as `begin` and `end` on containers:

```
void f(vector<int> v)
  pre( std::is_sorted(v.begin(), v.end());  // Error, non-const begin and end
```

This, of course, could be worked around by manually casting to select only `const` overloads[14]:

```
void f(vector<int> v)
  pre( std::is_sorted( static_cast<const vector<int>&>(v).begin(),
                       static_cast<const vector<int>&>(v).end() ) );  // Ok
```

3.1.4 No Operations Without `const` Alternative

Next, we could consider, as a choice in the middle of the above two approaches, an alternative in which we perform overload resolution with the `const`-ified expressions treated as `const` but still continue to use the non-`const` selected overloads.

> Proposal 4: No Operations Without Equivalent `const` Operations
>
> When overload resolution is performed in a contract assertion with operands that are `const`-ified, perform the same overload resolution where those operands are `const`. If either component overload resolution fails, then the full overload resolution fails. If both succeed, the result of overload resolution without `const` applied will be used.

For example, when making use of a non-`const` Standard container in a contract assertion, we will be allowed to use functions such as `begin` or `end`, which have non-`const` overloads, but be prevented from doing so with a mutable-only member function such as `clear`:

[14] Note that we use `static_cast` here instead of the much-derided `const_cast` since a `static_cast` is able to add *cv*-qualifiers freely but is unable to perform the more risky operation of removing them.

```
void f(std::vector<int> &v)
  pre( v.begin() <= v.end() )  // Ok, const overloads exist.
  pre(( v.clear() , true ));   // Error, no vector::clear() const
```

On the other hand, because the non-`const` overload is selected by the expression, we would *not* detect problems, such as the call to `std::sort` shown earlier. While detecting this situation is potentially challenging, mandating that the second overload resolution be performed with the same overload set means that no significant implementation challenges are expected.

Note that this check is surface level; we don't attempt to instantiate template bodies or resolve the expression as if the return types of the `const` overload resolutions were used since that would lead to vastly more complexity. So the following example, where we mandate non-`const`-ness of a template parameter, would compile, even if the `const` overload would fail to compile were it actually used:

```
template <typename T>
bool foo(T&& t) {
  static_assert( !is_const_v<T> );
}
void f(int i) pre(foo(i));  // Ok, both overload resolutions succeed.
```

Alternately, when we use a requires clause to constrain a function to non-`const` parameters, our contract assertion would be ill-formed because the extra overload resolution would fail:

```
template <typename T>
bool void foo(T&& t) requires !is_const_v<T>;
void f(int i) pre(foo(i));  // Error, const overload resolution fails.
```

3.1.5 Make `const`-ified Expressions `const`

Finally, we can consider the approach taken by [P3071R1] and [P2900R8], which is to treat expressions subject to `const`-ification as if they were `const`.

> Proposal 5: Choose Nonmodifying Operations
>
> Any expression subject to `const`-ification is treated as `const`, selecting `const` overloads and being ill-formed if no `const` overloads are available. (This is the status quo in [P2900R8].)

This approach leverages the common understanding that the semantics of a `const` and non-`const` overload in an overload set should always be functionally equivalent when both are present yet allow types to express exactly those cases where `const` on an object should propagate to the return values produced by a function, such as when `begin` or `end` return `const` iterators.

With this approach, any function calls made with `const`-ified expressions as arguments will, therefore, both require that there be and choose the `const` overload of those functions, exactly as if the expression were wrapped in a cast that added `const` to its type.

3.2 Categories of Objects to Avoid Modifying

The approach taken in [P2900R8] to implement `const`-ification is to identify certain expressions and to alter the types of those expressions to be `const` but, importantly, to leave *unaltered* the types of the actual objects denoted by those expressions. This method is a very similar to the mechanism that makes a member access expression, through a pointer to `const`, give us `const` references to members, even when those members are not themselves `const`.

Let's consider the kinds of expressions and the types of objects that they might denote to which we could apply this process.

- Id-expressions can denote a number of different types of entities with different properties. The first factors to consider for such entities is their scope and storage duration.
  - Function parameters
  - Block-scope variables having automatic storage duration
  - Block-scope variables having other storage durations, i.e., thread local or static
  - Nonstatic data members of `this` within a member function (with an implicit object parameter) that are not tied to any specific function
  - Class members or namespace-scope variables having static or thread-local storage duration
  - Temporary objects, such as those returned by value from a function call within the contract predicate
  - Any variables declared within a lambda nested within a contract predicate — including its function parameters and block-scope variables of any storage duration — must be considered distinctly.

  Any such denoted entity might be one of a number of different types.
  - Nonreference, nonpointer objects, which have a value that could be modified
  - Pointers, which both have a value and denote another object at a different location
  - References, which denote only an object located somewhere else
  - Structured bindings, which name references or name parts of an object with its own storage duration
- `this` is a prvalue for a pointer to the implicit object parameter of a member function.
- Member access expressions select a member of a particular type from an object denoted by the left side of the expression. These members might have a number of distinct properties.
  - A member may be `mutable`, which would allow its mutation even if the member access is a member of a `const` expression.
  - A member may be a reference, which again would allow modification if the reference was non-`const` even if the access is a member of a `const` expression.

  - A member may be a pointer, which would not itself be modifiable if the access is of a `const` expression but would allow mutation of the object denoted by the pointer.

- Unary indirection expressions that use the `*` operator and that access the object pointed to by a pointer denote an object whose storage duration and scope are never explicitly known.

- Subscripting operators, when applied to pointers, transform into a combination of indirection and additive expressions — i.e., `p[n]` is equivalent to `*((p)+(n))`. Because, when applied to a built-in pointer, these expressions are accessing a subobject of the array pointed to by the pointer, `const`-ification could propagate through these operations in the same way that it does through indirect member access.

Each of the above expressions denotes objects whose lifetime can be inferred and which might be considered a candidate for `const`-ification.

A few considerations can be applied to the above categories to determine if they can potentially denote objects whose lifetime is outside the cone of evaluation of the contract predicate.

- Any object created outside the contract predicate will be outside the cone of evaluation.

- Any reference created outside the contract predicate or any pointer whose value is set before the contract predicate is evaluated will denote an object outside the cone of evaluation.

- Any temporary object or any variable declared within a nested lambda will denote an object *inside* the cone of evaluation.

- Any temporary reference or pointer will denote an object whose lifetime, relative to the evaluation of the contract predicate, is unlikely to be known at compile time.

Finally, we must consider some general concerns regarding whether modifying an object whose lifetime *is* outside the cone of evaluation of a contract predicate is likely to be a problem.

- Mutable members that are directly accessed might be considered mutable in all situations. In practice, however, the `mutable` keyword is often used to allow encapsulated methods to make changes to the mutable state while still presenting a nonmodified value to clients. Directly mutating a member without that encapsulation seems likely to be a source of errors that could be better expressed by enclosing the mutation in a `const` member function.

- Reference members are generally initialized when an object is initialized, and they cannot change. Therefore, reference members refer to an object whose lifetime necessarily encloses the lifetime of the reference member as well, and they are thus completely outside the cone of evaluation.

- Any of the above expressions, when they resolve to a user-defined overload, could be considered for `const`-ification, but that would be assigning a particular interpretation to such operators that C++ has not assigned to them in the past. A facility to incorporate user-defined deep `const` into the language itself and to define when overloaded operators or other functions should propagate `const` to their return values could be useful but would be a far larger feature than is needed for attaining a general benefit to the Contracts facility.

The current status quo in [P2900R8] applies `const`-ification based on the following criteria.

- Variables having automatic storage duration are local data whose values are likely to be pertinent to the local correctness of the program and are thus subject to `const`-ification.
- Variables having nonautomatic storage duration are assumed to be either locally created or intended for global non-`const` use and are thus not subject to `const`-ification.
- Variables that are references and subject to `const`-ification are assumed to have been initialized to something that is also pertinent to the local correctness of the program and are thus subject to `const`-ification.
- The implicit object parameter `*this` is again likely to be pertinent and is subject to `const`-ification.
- No further efforts are made to apply `const`-ification to members or when dereferencing any pointer other than `this`. Therefore, reference and mutable members of `*this` are both modifiable.

We can identify the following additional rules that could be added without falsely making `const` an object created within the cone of evaluation of the contract predicate. Note that, to avoid changing the semantics of an expression that is *inside* the cone of evaluation and that just happens to be `const`, these rules apply a form of deep `const` in only those situations where `const`-ification has been applied.

- Mutable and reference member accesses could be made `const` if the object being accessed (the left-side operand of the member access expression) is one to which `const`-ification has been applied.
- References that are initialized to either references or objects that have `const`-ification applied to them should carry forward that `const`-ification lest `x.d_x` and `static_cast<T&>(x).d_x` have `const` applied differently for no tenable reason, and more importantly, lambda captures by reference would not then have `const` applied to member access through those references.
- A pointer value to which `const`-ification is applied is ostensibly one that cannot have been modified during the evaluation of a contract predicate and thus will always point outside the cone of evaluation of the predicate. Therefore, the dereference operator applied to such a pointer value could be considered for `const`-ification as well.

  Because we would want the subscripting operator to apply `const`-ification in the same manner, `const`-ification should equally propagate through pointer arithmetic (`p+n`, `n+p`, and `p-n` where `p` is `const`-ified) and then the built-in subscripting operator will follow.

  It is possible, however, that a value which is a pointer might be modified through a well-defined `const_cast` to point to an object *within* the cone of evaluation of the predicate:

```
void assign(int* const & x) {
  const_cast<int*&>(x) = new int(0);
}
void f(int* p)
  pre( assign(p),
       *p *= 5,  // *p is within the cone of evaluation.
       *p > 3 );
```

The above example, however, already requires some breaking of the promises associated with `const` parameters — using a `const_cast` to forcibly modify a parameter that would otherwise not be modifiable — and does not seem overly concerning. Therefore we could take the approach of assuming the value of a `const`-ified pointer does not change during the evaluation of the contract-assertion predicate, and therefore it is sound to consider the denoted object to always be outside that cone of evaluation and be subject to `const`-ification as well.

This rule could be considered a generalization of how `this` is currently treated in [P2900R8]. Note, however, that this rule would be giving special treatment to built-in pointers; where any *smart* pointer type will not get the same treatment, consider that its overloaded `operator->` will be opaque to guaranteed analysis about the lifetime of its result. Such special treatment would potentially encourage users to continue to use raw pointers instead of migrating to the generally safer smart pointers.

- Objects of nonautomatic storage duration within block scope are generally going to be used in only that scope, and modifications to those objects are likely to turn contract predicates destructive.
- Objects at nonblock scopes (and static or thread-local storage duration) are certainly outside the cone of evaluation of a contract predicate. The primary reason to omit those scopes from `const`-ification is to allow them to maintain APIs that are not `const` correct. However, if we consider that an insufficient reason to drop `const`-ification from general use, then we should consider demanding better APIs for all objects.

  In general, APIs that should be a concern are often things such as logging APIs, and using such APIs directly within a contract predicate does not seem, in practice, to be essential. Within nested functions, we certainly must allow trace logging, but nothing in this proposal would alter the internal behavior of functions invoked from a contract predicate.

3.2.1 Scopes and Storage Duration

Now we can consider a variety of proposals for what expressions we should consider to be candidates for `const`-ification.

Proposal A: Minimal `const`-ification

Apply `const`-ification to

- id-expressions denoting variables having automatic storage duration
- the expressions `this` and `*this`, whether explicitly or implicitly used
- structured bindings whose corresponding variable would have `const`-ification applied to it
- parenthesized expressions that are `const`-ified, i.e., if `E` is `const`-ified, then so is `(E)`

(This is the status quo in [P2900R8].)

Then we offer two proposals for extending `const`-ification to nonautomatic variables.

Proposal B: Block Scope Nonautomatic

In addition to Proposal A, apply `const`-ification to id-expressions denoting variables at block scope having static or thread-local storage duration.

Proposal C: Global Scope Nonautomatic

In addition to Proposal B, apply `const`-ification to id-expressions denoting variables at class or namespace scope having static or thread-local storage duration. (Therefore, all id-expressions denoting variables will have `const`-ification applied to them.)

3.2.2 Deep `const`

Next, we contemplate two other extensions to more deeply expand `const`-ification, where we consider the results of certain operations to be `const`-ified if their operands are `const`-ified (not merely `const`).

This design would allow preventing modifications in some additional cases, but because we lack a concept of user-defined deep `const`-ness in the language, we would be unable to apply consistent benefits to user-defined pointer-like types, such as `std::shared_ptr` or `std::unique_ptr`.

The first extension allows us to propagate to members of an object, which can be taken on its own since direct subobject lifetimes are always going to match (barring obscure shenanigans) their complete object.

Proposal D: Reference and Mutable Members

Apply `const`-ification to member access expressions whose left-side operand is an expression to which `const`-ification has been applied.

Second, we can extend `const`-ification to follow indirection through pointers and pointer arithmetic.

Proposal E: Pointer Dereferencing

Apply `const`-ification to

- a unary expression whose *unary-operator* is `*` (i.e., an indirection expression) and whose operand is a pointer to which `const`-ification has been applied
- an additive expression whose operator is `+`, where one operand is a pointer to which `const`-ification has been applied (including a subscript expression using the built-in subscript operator to transform into an indirection applied to an additive expression)
- an additive expression whose operator is `-`, where the left-side operand is a pointer to which `const`-ification has been applied

3.2.3 Postcondition Result Name

In [P2900R9], an identifier, in one additional place, has `const` added to its type even when it denotes an entity that is not otherwise `const`, and that place is the result name in postcondition contract assertions. One of the original points raised during the discussion of `const`-ification was discomfort that `const` was being used to make modifications to the result of a function more difficult but not being consistently applied to function parameters.

To maintain consistency, we have two choices that should coincide with our choice for applying `const`-ification to everything else.

> Proposal F: Result Name
>
> Apply `const`-ification to the result name within the predicates of postcondition assertions, such that, for a function whose return type is `T`, the type of the result name id-expression associated with that function is `T`.

> Proposal G: No Result Name
>
> Do *not* apply `const`-ification to the result name within the predicates of postconditions assertions, such that, for a function whose return type is `T`, the type of the result name id-expression associated with that function is `const T`.

All concerns that apply to function parameters apply equally to the result name. Given that, considering solutions where we make the choice on `const`-ification differently for function parameters and the result name seems incorrect.

4 Overview of Solutions

We now have a large number of solutions, which we will summarize here.

First, we identify what we will choose to do for expressions that are subject to `const`-ification with five mutually exclusive alternatives.

- Proposal 1: Do Nothing (Warnings Only) — Produce only warnings; no semantic changes
- Proposal 2: No Assignment — No assignment, increment, or decrement operations allowed
- Proposal 3: No Modifications — No potentially modifying operations
- Proposal 4: No Modify-Only Operations — No operations without nonmodifying alternatives
- Proposal 5: Make `const` — Treat as `const`

Second, we can consider which entities should have `const`-ification applied to them initially, each of which builds on the set of entities identified by the previous proposal.

- Proposal A: Automatic Variables — Local non-`static` variables outside assertion
- Proposal B: Local Variables — Block-scope variables outside assertion
- Proposal C: All Variables — All variables outside assertion

Finally, we must determine whether we apply a deeper form of `const`-ification to certain expressions, which can each be considered orthogonally.

- Proposal D: Member Access — Deep `const` applies to member access expressions
- Proposal E: Pointer Dereference — Deep `const` applies to raw pointer dereference

While not all 60 combinations of the above choices are meaningful, we believe that the concerns that dictate decisions among each of the three categories above are fairly independent and that each category can be treated as a separate decision.

Throughout this section, we will use the following symbols to indicate different levels of satisfaction with the concerns we present, where check marks are good and xs are bad.

- ✔: A wide green check indicates a proposal has no concerns and will correctly identify any presented examples as modifying or nonmodifying.
- ✓: A narrow gray check indicates that this proposal has minor concerns that do not seem overwhelming.
- ✗: A narrow gray x indicates that this proposal has major concerns that are not totally disqualifying.
- ✘: A wide red x indicates that this proposal fails to satisfy the concern and fails to identify any presented example as modifying or nonmodifying.

4.1 Form of `const`-ification

We will now explore various concerns and code examples that will illuminate the differences between the various proposals for how to implement protections from modification of `const`-ified expressions. For each concern, we will identify how well each of the first five proposals addresses that concern.

- **Concern: Implementation Experience**

  Making no changes to how expressions are evaluated can be considered implemented in all existing compilers, and thus Proposal 1 can be considered implemented, although a thorough implementation of this approach that produces useful warnings has not yet been undertaken.

  ✔: Both the GCC and Clang implementations of Contracts have implemented `const`-ification as specified in [P2900R8], which means that Proposal 5 can be considered implemented.

  ✘: None of the other proposals in this section have implementation experience.

- **Concern: Implementation Feasibility**

  ✔: The proposals with implementation experience are obviously feasible to implement as well.

  ✓: Both Proposal 3 and Proposal 4 require performing an additional round of overload resolution with an already-built overload set, this time with `const` arguments. While this specification approach and implementation seem feasible, some situations could lead to surprising results and could require reconsideration.

- **Concern: Forward Compatibility**

  When presented with a variety of options to consider for standardization and if the choice is unclear or the room is divided, we can often delay a permanent decision if one option leaves open the choice to adopt one or more of the other options in the future. This concern led to a property that has guided many decisions in [P2900R8], i.e., undecided behaviors should be ill-formed, which was described in [P2932R3].

  ✔: Proposal 3 makes ill-formed many expressions to which the other proposals provide either normal semantics or `const` semantics, which means that Proposal 3 leaves open the maximal amount of opportunity to change to the other proposals in the future.

  ✓: Proposal 4 is similarly forward-compatible to any proposal other than Proposal 3.

  ✓: Proposal 5 could, in theory, be dropped if we were willing to risk changing some contract-assertion predicates that evaluate a `const` overload into expressions that evaluate a corresponding non-`const` overload. In general, these functions should be semantically similar, though someone might, for example, have an `operator[] const` on a container that threw exceptions when entries do not, while the corresponding `operator[]` inserted new entries in those cases.

  ✗: Proposal 2 could be removed completely (leaving Proposal 1 without code breakage) but is likely to prevent migration to any of the other proposals presented here.

  ✘: If we mandate no `const`-ification-related changes by choosing Proposal 1, then we are unlikely to ever be able to introduce any of the other ideas in a future Standard without significant code breakage.

- **Concern: Teachability of Contracts**

  With the introduction of a major new language feature, especially one we expect to be used regularly by developers of all skill levels, we must examine how effectively we can teach users both how the feature behaves and how to use it effectively. In particular, the question that we must answer is how effectively the tool can be used correctly without learning all its nuances and how easily more rarified and expert use cases can be understood.

  ✔: C++ developers are already aware of how `const` works and of other contexts (such as `const` member functions) where some expressions that might refer to non-`const` entities (such as member access expressions) become `const`. Importantly, by making non-`const` uses of variables declared outside a contract predicate harder to do, we naturally teach users unfamiliar with the best uses of Contracts to avoid making modifications of state within their contract-assertion predicates. Proposal 5 also proves more teachable when a user needs to work around limitations of `const`-ification within a contract assertion they are writing since the workarounds for it all involve clearly applied existing features of C++.

  ✓: Proposal 3 would be similarly easy to teach, but working around its limitations requires not only applying `const_cast`, but also encapsulating it in newly designed wrapper functions, a significantly larger hurdle for new developers.

  ✗: Proposal 2 and Proposal 4 introduce bespoke rules for what is and isn't allowed in a contract-assertion predicate, and those rules do not clearly resemble rules applied anywhere else in the language.

✘: Proposal 1 actively hinders the teaching of Contracts because it leaves users to navigate an unknown set of warnings of varying qualities while providing no actual guidance (outside of literature) as to how to write viable contract assertions.

- **Concern: Local Escape Hatch**

  If we adopt any form of semantic `const`-ification, cases are inevitable in which a nonmodifying function needs to be called as part of a contract assertion but is not marked `const`, either because that function is sometimes modifying or because it is from a library that has not provided a `const`-correct API.

  A common example is the use of `std::map::operator[]`, which inserts a new entry into a map when given a key that is *not* currently in that map but makes no modifications when used with a key that *is* in the map:

  ```
  void f(std::map<int,int> m, int k)
    pre( m.contains(k) && m[k] == 7 );
  ```

  Since all proposals presented in this paper do not involve restricting contract-assertion predicates to a special class of functions, all have available to them the same escape hatch of hiding a `const_cast` inside a wrapper function that takes a `const&` argument. The concern here, however, is that the availability of a direct escape hatch clearly conveys that the author of the contract-assertion predicate is intentionally working around `const`-ification.

  ✔: Proposal 1 allows for only warnings, which can always be disabled and thus worked around.

  ✓: Proposal 5 allows any `const`-ified expression to be turned into a non-`const` expression through the use of the appropriate `const_cast`. This use can even be fairly accurately encapsulated in a macro using `decltype`:

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

  This macro will do nothing when applied to most expressions, but when applied to an entity that has been `const`-ified, it will produce an expression with the same type as the declared entity.

  The use of `const_cast`, however, is frowned upon in many codebases, is verbose, and is often misunderstood. While allowing its encapsulation, as shown in the above macro, in any codebase seems reasonable, a future Standard might provide this facility within the Standard Library itself or make it a built-in operator guaranteed to work in only those cases in which a `const` qualifier can be safely removed from an expression.

  ✘: Proposal 2, Proposal 3, and Proposal 4 all make a range of expressions ill-formed and do not provide a clear mechanism to make those expressions well-formed since no semantics could be changed that would do so. We could, conceivably, `static_cast` a `const`-ified expression to its own type to remove the effects of these proposals, but doing so requires treating a `static_cast` of an expression to the type of that expression as meaningful when it otherwise never is.

- **Concern: Replaceable With Warnings**

  Barring forking another compiler to attempt recompilation with a different set of rules, any warning implementation is going to hit significant limitations in what it can do. Warnings that produce significant false positives are also a huge implementation burden for compilers since they produce endless bug reports and are eventually universally turned off.

✔: The analysis of Proposal 2 to simply identify the use of certain operator syntaxes on specific classes of operands is fairly straightforward and should be easily accomplished as a warning.

✓: The deeper inspection of the results of overload resolution needed for Proposal 3 should be similarly feasible, but that proposal also leads to significant false positives that would likely result in a hard to support warning.

✘: Proposal 4 would require additional overload resolutions in an already built overload set to be performed, leading to exactly the kinds of unwinding problems that we believe will prove intractable and unacceptable to compiler implementations. Proposal 5 would similarly require even more additional overload resolutions to be performed since the types of subexpressions will potentially differ from the non-`const`-ified version of the contract-assertion predicate.

Proposal 1 is obviously vacuous to talk about in this context. It gets a blank cell in the table.

- **Concern: Consistent Expression Behavior**

  Understandability of the language is always a concern when the same expression, in very similar locations, has different meanings. For example, some people might believe that the assertion in this example should never fail if the precondition passes:

```
#include <cassert>
bool g(int& x);
bool g(const int& x);
void f(int x)
   pre( g(x) )
{
   assert( g(x) );  // classic C assert macro, not contract_assert
}
```

  By deducing different types for template parameters during overload resolution, we can easily produce APIs in which behaviors change based on the `const`-ness of their arguments. Consider a metafunction that uses partial specialization to produce different results for `const` and non-`const` arguments:

```
template <typename T>
struct S {
  using type = long long;  // 8 bytes on most platforms
};
template <typename T>
struct S<const T> {
  using type = int;        // 4 bytes on most platforms
};
```

  Using that type, we could deduce the return type of a function based on what is passed to it:

```
template <typename T>
auto f(T&& x) -> typename S<T>::type;
```

Given the above, a precondition checking for properties of `f` would produce different results if an expression is treated as `const` by `const`-ification:

```
template <typename T>
void g(T t) pre( sizeof(f(t)) == 4 );
```

Of course, outside the precondition, `f(t)` might select a non-`const` overload and return `long long`. This inconsistency could result in subtle problems when attempting to reason about code within a function body and how assertion predicates relate to that code.

✘: Proposal 5 would, of course, invoke the `const int&` overload of `g`, which might produce a different result than the `int&` overload.

✓: Proposal 3 would make ill-formed attempting to bind the parameter `x` to the `int&` parameter of `g`, resulting in no change of behavior but instead making the program ill-formed. This proposal would similarly make ill-formed *any* cases where Proposal 5 would choose a different overload.

✔: The other proposals would allow the above example and invoke the `int&` overload of `g`.

- **Concern: Don't Misnavigate Broken Overload Sets**

  Argument-dependent lookup (ADL) is a powerful and yet dangerous tool in C++. In particular, it includes in an overload set functions with the right name from many associated namespaces, which can lead to highly surprising results when templates are constructed using arguments from different libraries that have conflicting uses of functions with the same name.

  Let's consider three different libraries all using the same ADL-customization point with different intentions.

  1. A library that has a `clean` free function that is intended to identify objects with contents in need of cleaning, defined as a function template taking a forwarding reference:

     ```
     namespace lib1 {
       template <typename T>
       bool clean(T&& t)
       {
         return t.size() > 0;  // Identify if there are contents to clean.
       }
     }
     ```

  2. A library that has a `clean` free function with the opposite meaning, this time implemented as a function taking a `const` lvalue reference:

     ```
     namespace lib2 {
       template <typename T>
       bool clean(const T& t)
       {
         return t.size() == 0;  // This object is clean.
       }
     }
     ```

3. A library that has a `clean` free function that cleans its parameter by calling `clear` on it:

   ```
   namespace lib3 {
     template <typename T>
     bool clean(T& t)
     {
       bool output = (t.size() > 0) ;  // number of items we are cleaning
       t.clear();                      // Clean out all items.
       return output;
     }
   }
   ```

Now, to experience the difficulty that arises from conflicting free functions intermingling with ADL, let's imagine that within each of these namespaces we have class templates that have `clear` and `size` members, obviously with many other unrelated differences in any real implementation:

```
namespace lib1 {
  template <typename T>
  class X {
  public:
    void clear();      // ...
    int size() const;  // ...
  };
}
// same definition for lib2::Y
// same definition for lib3::Z
```

Now we can consider users writing functions in each of these libraries, `lib1::f`, `lib2::g`, and `lib3::h`, all with similar forms:

```
namespace lib1 {
  template <typename T>
  void f(X<T>& x)
    pre( clean(x) );
}
// same declaration for lib2::g(Y<T>&)
// same declaration for lib3::h(Z<T>&)
```

In each case, the writer of these function has some expectation that their invocation of `clean` in a precondition is going to invoke `clean` from the namespace of the function template. ADL, however, has different sinister plans for the developer.

- The author of `lib1::f` clearly intended to write a precondition using their version of `clean` that works on objects of any type but, in spirit, does not modify the value even when the deduced template parameter is non-`const`.
- The author of `lib2::g` had a different interpretation of the word `clean` and similarly has an implementation that will work on (almost) any suitable object with a `const size` function.

– Finally, the author of lib3::h has made a terrible categorical error; their definition of clean actually does cleaning and requires a non-const parameter be passed to it. Clearly, this is a destructive predicate that will lead to critical differences between a checked and unchecked build of their program.

Note that, in all these cases, ADL is capable of subverting the intent of the function writer and picking up a different version of clean.

Template Parameter (T)	libf1::f	lib2::g	lib3::h
Without const-ification (Proposals 1–4)			
int	lib1::clean	lib2::clean	lib3::clean
lib1::X<int>	lib1::clean	lib1::clean	lib3::clean
lib2::Y<int>	lib1::clean	lib2::clean	lib3::clean
lib3::Z<int>	lib3::clean	lib3::clean	lib3::clean
lib1::X<lib2::Y<lib3::Z<int>>>	lib3::clean	lib3::clean	lib3::clean
With const-ification (Proposal 5)			
int	lib1::clean	lib2::clean	lib3::clean
lib1::X<int>	lib1::clean	lib2::clean	lib3::clean
lib2::Y<int>	lib2::clean	lib2::clean	lib2::clean
lib3::Z<int>	lib3::clean	lib2::clean	lib3::clean
lib1::X<lib2::Y<lib3::Z<int>>>	lib2::clean	lib2::clean	lib2::clean

In the table above, we show which overload of clean is invoked by the functions lib1::f, lib2::g, and lib3::h with different template parameters, depending on whether the function argument is const within the precondition assertion.[15]

– We have colored in red those cells in which a version of clean is selected by overload resolution that is not the one in the same namespace as the function template being instantiated.

– We have also colored in red those cells in which a version of clean will not compile, which in this context is when lib3::clean is instantiated for a const parameter that has only a non-const clear member.

– We have colored in gray those cells in which a function that will unintentionally modify state is invoked — i.e., lib3::clean — and the user has written a destructive predicate that we would want to discourage.

– When the template argument T is int, only the namespace of the function template is an associated namespace, and no surprising ADL resolution will occur. These cells always have the same result as the cells in which the template argument is a specialization of the class template (X, Y, or Z) from the same namespace as the function template.

– When the template argument T is lib1::X<int>, lib2::Y<int>, or lib3::Z<int>, two of the namespaces are associated (if different) — the namespace of the class template and the namespace of the function template.

[15] See https://godbolt.org/z/6cMTPn81j for the test program that verified these overload resolution results without the use of Contracts.

– When the template argument `T` is the burdensome mouthful that is `lib1::X<lib2::Y<lib3::Z<int>>>`, all three namespaces are associated namespaces.

The table above clearly shows that when we have function name collisions of ADL-enabled customization points, such as `clean`, C++ will not give us a result that is obvious to determine whether we have `const`-ification in play. To analyze each result, we will count how many of the 12 variations of associated namespaces are possible in the table above. Eight of these namespaces are ostensibly valid preconditions that would not modify any state and thus would not be those we wish to discourage, while 4, invoked from `lib3::h`, make modifications of state that we would want to prevent by default.

For the proposals below, we indicate whether the proposal would make the relevant pairings of template arguments and function templates well-formed. We will show ✘ if this allows an incorrect overload to be called, ✗ if `lib3::clean` is not detected as a problem, and ✔ otherwise.

✘ (3/12): Proposal 1 and Proposal 2 make no change to the type and make none of these examples ill-formed. Therefore, only 3 of the 8 valid preconditions pick the correct overload of `clean`, and 0 of the 4 invalid preconditions are ill-formed.

Template Parameter (`T`)	`libf1::f`	`lib2::g`	`lib3::h`
Without `const`-ification			
`lib1::X<int>`	✔: well-formed	✘: well-formed	✗: well-formed
`lib2::Y<int>`	✔: well-formed	✔: well-formed	✗: well-formed
`lib3::Z<int>`	✘: well-formed	✘: well-formed	✗: well-formed
`lib1::X<lib2::Y<lib3::Z<int>>>`	✘: well-formed	✘: well-formed	✗: well-formed

✓(10/12): Proposal 3 makes all uses of `lib1::clean` and on a non-`const` parameter ill-formed as well as all uses of `lib3::clean`.

Template Parameter (`T`)	`libf1::f`	`lib2::g`	`lib3::h`
Without `const`-ification			
`lib1::X<int>`	✘: ill-formed	✔: ill-formed	✔: ill-formed
`lib2::Y<int>`	✘: ill-formed	✔: well-formed	✔: ill-formed
`lib3::Z<int>`	✔: ill-formed	✔: well-formed	✔: ill-formed
`lib1::X<lib2::Y<lib3::Z<int>>>`	✔: ill-formed	✔: well-formed	✔: ill-formed

✘(4/12): Proposal 4 will find a `const` overload of `clean` in the overload set it examines in all cases where either `lib1` or `lib2` is an associated namespace that it searches. Therefore, it makes almost everything well-formed with very similar results to Proposal 1.

Template Parameter (`T`)	`libf1::f`	`lib2::g`	`lib3::h`
Without `const`-ification			
`lib1::X<int>`	✔: well-formed	✘: well-formed	✗: well-formed
`lib2::Y<int>`	✔: well-formed	✔: well-formed	✗: well-formed
`lib3::Z<int>`	✘: well-formed	✘: well-formed	✔: ill-formed
`lib1::X<lib2::Y<lib3::Z<int>>>`	✘: well-formed	✘: well-formed	✗: well-formed

✗(8/12): Proposal 5 is the only one that applies the bottom half of our table. Because the function parameter is `const`, all cases in which `lib3::clean` is selected will fail to compile. Note that, when `lib1::clean` is selected, being that it has a forwarding reference as its template parameter, the `const` version will be instantiated (and work as intended).

Template Parameter (`T`)	`libf1::f`	`lib2::g`	`lib3::h`
Without `const`-ification			
`lib1::X<int>`	✔: well-formed	✔: well-formed	✔: ill-formed
`lib2::Y<int>`	✘: well-formed	✔: well-formed	✘: well-formed
`lib3::Z<int>`	✔: ill-formed	✔: well-formed	✔: ill-formed
`lib1::X<lib2::Y<lib3::Z<int>>>`	✘: well-formed	✔: well-formed	✘: well-formed

Note that *none* of our proposals protect against all possible ADL-related mistakes here, nor do we particularly believe such a thing would be possible or appropriate to apply to just contract-assertion predicates, and therefore we gave no proposal a ✔.

- **Concern: Code Dependent on `const`-ness**

  In general, an overload set that differentiates its semantics based on whether the provided arguments are mutable is frowned upon in C++. Of course, one glaring exception occurs when an overload set is written to explicitly consider static properties of its arguments and return a value based on that evaluation. Consider, for example, a function that determines if its parameters are const — something usually done (as the implementation here does) with a compile-time type trait, not with a function call:

  ```
  template <typename T>
  bool is_const(T&& t) { return std::is_const_v<T>; }
  ```

  On its own, this function seems inferior to a `decltype` expression directly combined with `is_const_v`, but we might consider a more involved predicate that combines checking of runtime and compile-time properties to determine if two objects are swappable:

  ```
  template <typename T, typename U>
  bool is_swappable(T&& t, T&& u)
  {
    if constexpr (!std::is_same_v<std::remove_reference_t<T>,
                                  std::remove_reference_t<U>>) { return false; }
    if (!is_const(t) || !is_const(u)) { return false; }
    if constexpr (has_get_allocator<T>) {
      if (t.get_allocator() != u.get_allocator()) { return false; }
    }
    return true;
  }
  ```

  In general, when `is_swappable` is called immediately before `std::swap` and if that `swap` invocation is going to compile at all, the `is_const` checks will pass:

  ```
  template <typename T>
  void f()
  {
    T t1, t2;
  ```

```
    if (is_swappable(t1,t2)) {
      swap(t1,t2);
    }
  }
```

However, when this `is_swappable` function is used on `const`-ified parameters within a contract assertion, we will always be told that our variables are not swappable even when they otherwise are:

```
template <typename T>
void g()
{
  T t1, t2;
  contract_assert(is_swappable(t1,t2));  // always fails
  swap(t1,t2);
}
```

✔: Proposal 1 and Proposal 2 would both allow the above example to work as intended. Proposal 4 would make the above example compile due to the `is_swappable` function template being a valid match during both the `const` and non-`const` overload resolution on `t1` and `t2`.

✗: Proposal 3 would make the above example ill-formed due to `is_swappable` taking its parameters by non-`const` lvalue reference.

✘: Worst of all, Proposal 5 would change the meaning of the above code, making the contract assertion fail in all situations even when the function is called with otherwise swappable parameters.[16]

- **Concern: Interpret Semantics, Not Syntax**

  C++ provides numerous ways to perform operations, and for many operations, two mechanisms — through an overloaded operator and a named function — might be available to perform an operation:

```
class MyBigNum {
  MyBigNum& operator+=(const MyBigNum& rhs);
  MyBigNum& add(const MyBigNum& rhs);
};
```

  The language itself makes using the overloaded operator more natural in some cases, but in general does not otherwise distinguish between the two mechanisms for providing a user-defined operation on a type.

  ✘: Proposal 2 makes a clear distinction between member functions and overloaded operators, applying `const`-ification to only expressions involving assignment, increment, and decrement operators without considering any other user-defined functions.

  ✔: All other proposals take into consideration the exposed API of any function that is invoked through either operator overloading or the function-call syntax, considering only whether the

[16] On the other hand, this particular example will always fail when the program is first run, a situation that then provides a good learning experience and improved understanding of when to use static type checking and when to use contract assertions. While some might consider this semantic change a bug, others certainly consider it a feature.

parameters in question are `const` when deciding if the expression is considered likely to be problematic or not.

- **Concern: Non-`const`-Correct APIs**

  Many libraries do not make the effort to annotate nonmutating functions with `const` at all. When forced to use such third-party libraries, contract assertions that demand the use of `const` qualifiers make writing contract assertions significantly more difficult.

  ✔: Proposal 1 requires no extra use of `const` and makes all APIs as usable *within* contract assertions as they are *outside* them. Proposal 2 does not impact any use functions that a library might provide other than overloads of certain mutating operators that in all but vanishingly rare cases modify something anyway.

  ✗: Proposal 5 makes using non-`const`-correct APIs more difficult, but allows for a consistent escape hatch through the use of `const_cast`. Interest in supporting such use cases might increase the interest in providing a built-in operator to prevent constification.

  ✘: Proposal 3 and Proposal 4 both make using a non-`const`-correct API ill-formed.

- **Concern: Increased Cost of Static Analysis**

  A concern has been raised about the cost of static analysis increasing when the meaning of expressions within contract-assertion predicates is different from that outside those predicates. There are a few points to consider with this concern.

  – This concern does *not* apply to chaining of postconditions to preconditions of later function invocations or similar cases since that chaining can happen effectively as long as all contract assertions apply the same general rules for `const`-ification.

  – It has been suggested that static analysis should be able to prove, in general for any expression, that in the following example (or any example structurally similar to it, such as invoking another function with a match precondition), the contract assertion will never be violated:

  ```
  if (expression) {
    contract_assert(expression);
  }
  ```

  Of course, static analysis must always face the challenge that C++ is a complex language with a vast amount of power in the hands of any arbitrary function call. In particular, if the two above expressions actually do call *different* functions because of `const`-ification, then one of those functions is being passed a non-`const` reference or pointer to a local variable. If that function is an arbitrary opaque function in another TU, static analysis *must* assume that the variable's value might be modified, and thus the value of the expression cannot be considered to remain stable when entering the body of the `if` statement.

  – The opposite case, where an expression occurs *after* a contract assertion of a syntactically identical expression, might be a source of surprise to some users if a static analyzer cannot prove it to be true:

```
contract_assert(x);
if (x) {
  // We always take this branch.
}
else {
  contract_assert(false);  // This branch should be unreachable.
}
```

To prove that the unreachable contract assertion is actually unreachable, static analysis would need to be able to establish a correspondence between the truth of the contract assertion and the truth of the test in the `if` branch.

Such a correspondence, of course, won't even exist if `x` invokes functions that are opaque and that might modify the state of variables referenced by `x`. In truth, static analysis might not even be able to go that far with `const`-ification unless it makes the assumption that the values returned by the expression `x` are independent of (and do not change) global state as well.

- ✔: Proposal 3 limits contract assertions to those functions that take `const` parameters when passed `const`-ified arguments. Static analysis attempting to reason about identical expressions to those predicates will already be reasoning about expressions that treat `const`-ified expressions as `const` anyway, and static analysis will have the greatest chance of reasoning significantly about the values in a program.

- ✓: Proposal 5 minimizes the ability for contract assertions to break any assumptions that static analysis depends on. As long as static analysis trusts that passing a parameter by `const` reference or pointer will not lead to modifications of that parameter, significantly more static analysis can be performed than could be without such an assumption.

- ✗: Proposal 4 relies on the assumption that there is a relationship between the `const` and non-`const` overloads of a function when both exist and, most importantly, that the non-`const` overload is substitutable for the `const` one. Both humans and static analyzers should treat the two the same when either is applicable, although the non-`const` overload might provide additional capabilities, such as when `begin()` returns a non-`const` iterator that would enable further modification, and `begin() const` does not. If we make this assumption of substitutability the basis of when we apply `const`-ification, we can reasonably make the same assumption for static analysis. As with the other proposals that allow modifications, however, Proposal 4's static analysis without that level of trust will have to contend with significantly more opaque modifying functions to reason about, and any attempt at proof will travel significantly less distance.

- ✘: Proposal 1 provides the least benefit to static analysis since a static analyzer must contend with the fact that any contract assertion might be modifying any references to variables passed to functions by the contract-assertion predicate. Consider the following example:

```
bool foo(int& i);
bool foo(const int& i);

void bar()
{
```

```
  int i = 5;
  contract_assert(foo(i));  // might modify i without const-ification
  contract_assert(5 == i);  // provable with const-ification, not otherwise
}
```

Here we can see that, because `i` is not treated as `const` in the first contract-assertion statement, the second contract-assertion statement cannot be proven to be true. Given the power of the C++ language, of course, true proof of the second contract assertion would not be available even with `const`-ification, but static analysis that wants to detect real errors will often, by default, trust that variables passed by `const` reference or pointer will not be modified.

- ✘: Proposal 2 prevents some contract-assertion predicates that might break the ability for static analysis to perform its duty but otherwise does nothing to improve static analysis beyond what Proposal 1 does.

- **Concern: Silently Fixing Broken Predicates**

  Consider a contract-assertion predicate that attempts to move from a parameter into a function that *can* consume a value or, if given a `const` parameter, that will make a copy:

  ```
  bool foo(const S& s);
  bool foo(S&& s);

  void f(S s)
    pre( foo(std::move(s)) );
  ```

  The above contract-assertion predicate is clearly bad and is a sign that the developer in question does not understand the implications of having a predicate move from a parameter. Allowing the above example to compile, therefore, could be concerning.

  ✔: Proposal 3 identifies the above example as bad *and* makes it ill-formed.

  ✗: Proposal 5 clearly identifies the above example as concerning but transforms the predicate into one that calls the `const S&` overload of `foo`. Since the compiler did not produce an error, the developer is left both unaware that they are writing code whose intention is clearly wrong while also introducing a potentially expensive copy and associated allocations within the call to `foo`.

  ✘: Proposal 4 considers the above example to be a nonissue because it finds functions to evaluate both with and without `s` being `const`. Rather than fixing any issue, Proposal 4 lets the broken assertion evaluate and move from the parameter, consuming it before the body of `f` can use it.

  ✘: Proposal 1 and Proposal 2 both leave the above predicate unchanged, allowing the parameter to be moved from.

- **Concern: Handling of [P3336R0] Issues**

  Each of the proposals in this section would address different subsets of the issues that were identified in [P3336R0] when compiling a large number of libraries. Note that this analysis is still being applied to libraries that are in production and thus have already paid the (possibly large) cost of identifying and removing any critical errors that `const`-ification would have caught immediately.

✔: Proposal 5 is the implementation that was used in the analysis, so all issues and bugs identified by that analysis would be detected.

✓: Proposal 3 would make errors based on the issues identified in [P3336R0] and would also make errors based on all the fixed code without introducing many casts to manually *add* `const` to many expressions.

✗: Proposal 2 would detect the destructive predicate identified in BDE and the bugs detected in Library `#3` since those involved assignment and the increment operators. The (major) bug in Library `#4` would go undetected because it involved the invocation of a non-`const` member function with no `const` alternative. None of the other issues with `const`-correctness would be detected by this approach.

✗ Proposal 1 could, in theory, produce warnings matching any other proposal, but we do not believe it would be feasible, in practice, for compilers to produce warnings that require additional overload resolution. Producing warnings equivalent to Proposal 3 would be feasible, but warnings with significant quantities of false positives are often quickly turned off. Therefore, warnings that are applied only for situations that would be errors with Proposal 2 are the sole likely warnings that we will see, and those detect only a small subset of the issues identified in [P3336R0].

✗: Proposal 4 would identify the major identified issues but not all the potential issues that were detected by `const`-ification. In particular, in Library `#3`, using a base class function effectively returned `shared_from_this()`, which did not have a `const` overload. All the expressions that invoked non-`const` member functions through that accessor function would go undetected if the non-`const` overload of that base class function remained selected.

- **Concern: Direct Modification**

  Consider a contract assertion that captures the return code of an operation while also verifying that it is a success, where a user has taken what should be a normal expression and blindly wrapped it in a `contract_assert` to verify its value:

  ```
  int doImportantStuff();
    // Return zero on success and a nonzero value on failure.
  void f()
  {
     int rc;
     contract_assert( (rc = doImportantStuff()) == 0 );  // assert success?
     // ... code that depends on the value in rc
  }
  ```

  When the contract assertion is not evaluated, the above code, of course, fails catastrophically by not doing the important stuff it intended to do.

  ✔: All proposals *except* Proposal 1 would make this example ill-formed.

  ✓: Proposal 1 could reasonably be expected to produce a reliable warning for this case.

- **Concern: Encapsulated Modification**

  Now consider an example in which modification is performed through a non-`const` member function:

  ```
  struct Index {
    int d_index = 0;
    int increment() { return ++d_index; }
  }
  void f(Index index)
  {
    contract_assert(index.increment());
    // ...
  }
  ```

  ✔: Proposal 5 would make `index` a `const` expression within the `contract_assert`, and thus the above would be ill-formed. Both Proposal 3 and Proposal 4 make using this non-`const` member function ill-formed.

  ✗: Proposal 1 could produce a warning for this example but would require deeper analysis to identify this case and to avoid all the false positives that can be associated with Proposal 3.

  ✘: Proposal 2 makes the above example well-formed.

- **Concern: Nonmodifying Iteration**

  Now consider a case in which we might pass iterators to a container to a `const` algorithm to verify the contents of the container, such as whether an input vector is sorted:

  ```
  void f(std::vector<int> v)
    pre( std::is_sorted(v.begin(),v.end()) );
  ```

  ✔: Proposal 5 would allow the above code, invoking the `const` overloads of `begin` and `end` and passing the resulting `const` iterators to `is_sorted`. Proposal 4 allows the calls to `begin` and `end` due to the presence of their `const` overloads. Proposal 2 leaves the above example as is, and Proposal 1 would likely be silent on the above example, neither warning nor attempting to warn.

  ✘: Proposal 3 makes the above code ill-formed.

- **Concern: Modifying Iteration**

  Now consider a structurally similar example in which a user attempts to use a precondition to sort a function's input:

  ```
  void f(std::vector<int> v)
    pre(( std::sort(v.begin(), v.end()) , true ));
  ```

  ✔: Proposal 5 causes the attempt to find a usable overload of `sort` to fail because there is no viable candidate for `const` iterators. Proposal 3 does not allow the use of `begin` and `end` at all.

  ✘: Proposal 4 allows the calls to `begin` and `end` and then uses the results of the non-`const` overloads of those functions to find a valid `sort` to invoke, modifying the `vector`. Proposal 1 would be unable to viably identify the general case here to produce reliable warnings (though it could possibly have built-in knowledge of Standard Library templates to catch this particular case). Proposal 2 does nothing to prevent the above misuse.

- **Concern: SG21 Consensus**

  SG21 and EWG have discussed `const`-ification multiple times since the adoption of [P3071R1]. After reviewing a previous iteration ([P3261R1]) of this paper, Proposal 1 was polled while Proposal 5 remained the status quo. No sufficient interest to poll any of the other options here was expressed.

  https://wiki.edg.com/bin/view/Wg21telecons2024/Teleconference2024-10-10

  Remove constification from [P2900R8], as proposed in [P3261R1] Proposal 1.

SF	F	N	A	SA
3	1	2	10	8

  Result: Not Consensus

  ✔: Proposal 5, as the status quo, still maintains significant SG21 consensus.

Concern	1 Do Nothing (Warnings Only)	2 No Assignment	3 No Modifications	4 No Modify-Only Operations	5 Make `const`
Implementation Experience	✔	✘	✘	✘	✔
Implementation Feasibility	✔	✔	✓	✓	✔
Forward Compatibility	✘	✗	✔	✓	✓
Teachability of Contracts	✘	✗	✓	✗	✔
Local Escape Hatch	✔	✘	✘	✘	✓
Replaceable With Warnings		✔	✓	✘	✘
Consistent Expression Behavior	✔	✔	✓	✔	✘
Don't Misnavigate Broken Overload Sets	✘	✘	✓	✘	✗
Code Dependent on `const`-ness	✔	✔	✗	✔	✘
Interpret Semantics, Not Syntax	✔	✘	✔	✔	✔
Non-`const`-correct APIs	✔	✔	✘	✘	✗
Increased Cost of Static Analysis	✘	✘	✔	✗	✓
Silently Fixing Broken Predicates	✘	✘	✔	✘	✗
Handling of [P3336R0] Issues	✗	✗	✓	✗	✔
Direct Modification	✓	✔	✔	✔	✔
Encapsulated Modification	✗	✘	✔	✔	✔
Nonmodifying Iteration	✔	✔	✘	✔	✔
Modifying Iteration	✘	✘	✔	✘	✔
SG21 Consensus	✘	✘	✘	✘	✔

This analysis provides the following early conclusions.

- We believe being able to express contract assertions on basic data structures is essential to having a good contract-checking facility; Proposal 3 outright prevents the use of `begin` and `end` on a container, so it, therefore, is not one we will pursue.
- The implementation concerns related to doing a second set of overload resolution to implement Proposal 4 led us to discontinue pursuit of that option as well.

4.2 Entities `const`-ified

When considering to which entities we would apply `const`-ification, the first five proposals generally perform the same, so we need not belabor their consideration.

- With Proposal 1, we would not be standardizing any particular entities as `const`-ified, and implementations would have complete freedom to apply warnings to any range of entities they see as appropriate to warn on.

- All other proposals treat specific expressions either as invalid for certain operations or as `const` in certain contexts but otherwise have no essential differences to discuss in this section.

This leaves us with a few concerns to consider when deciding between the proposals presented for entities to `const`-ify, i.e., Proposal A, Proposal B, and Proposal C.

- **Concern: Implementation Experience**

  ✔: Proposal A has been implemented in both Clang and GCC as part of the implementation of [P2900R8].

- **Concern: Implementation Feasibility**

  ✔: All these proposals involve only a small change in the conditions under which an expression naming a variable will be `const`-ified, so all are equally feasible.

- **Concern: Forward Compatibility**

  Overall, once we pick a set of entities to which we will apply `const`-ification, changing that set of entities in a future Standard will be fairly difficult. Adding to the set will likely result in unacceptable code breakage, while narrowing the set of entities might be at least partially acceptable. How acceptable narrowing would be is largely dependent on the nature of a change in `const`-ification.

  ✘: Since significant broken code could result from applying any form of `const`-ification to a wider range of entities, Proposal A and Proposal B would struggle to expand the set of entities to which `const`-ification can be applied if we chose Proposal C as the solution we wanted to champion.

  ✗: Proposal C could, in theory, remove `const`-ification to reduce the set of entities to which it applies. Subtle changes in behavior, similar to those mentioned for Proposal 5 above, might be a concern but do not seem insurmountable.

- **Concern: Teachability of Contracts**

  As with concerns about how the form of `const`-ification can impact the teachability of Contracts as a feature, we must also consider the same impact our choice of `const`-ified entities will have.

  ✔: Proposal C has the easiest rule to teach and understand, while maximizing the number of cases where users are guided away from misuse.

  ✓: Proposal B both catches fewer mistakes and has a more complicated rule to understand, although the general formulation of that rule — don't reference anything declared as part of the function — is not *that* difficult to internalize.

  ✗: Proposal A combines having the most complicated rule, which must come with an understanding of different storage durations, with catching the fewest mistakes when learning to use Contracts in the first place.

- **Concern: Function Parameters**

  Modification of function parameters in a contract assertion is quite likely to result in a program whose correctness is independent of the correctness of the same program where the contract assertions are not evaluated. Here we can see that in action, where evaluating a precondition might change the result of a later contract-assertion statement:

```
void f(int x)
  pre( x-- > 0 )
{
  contract_assert( x > 0 );
}
```

✔: All the proposals for entities to `const`-ify will consider a function parameter as subject to `const`-ification.

- **Concern: Automatic Local Variables**

  Local variables at block scope are equally subject to causing problems when modified, and modification of such variable assertions is a frequent cause of bugs:

  ```
  void f(const std::vector<int> &v)
  {
    for (int i = 0; i < v.size(); ++i) {
       contract_assert(v[++i] >= 0);
       // Process every other element with contracts checked, and every element otherwise?
    }
  }
  ```

  ✔: All the proposals for entities to `const`-ify will consider a block-scope automatic variable as subject to `const`-ification.

- **Concern: `static` or `thread_local` Local Variables**

  Static local can be used to cache information about a function not specific to a particular invocation, such as attempting to track whether a function is being called recursively:

  ```
  void f()
  {
    thread_local int callDepth = 0;
    contract_assert( callDepth++ == 0 );

    someOtherFunction();

    contract_assert( --callDepth == 0 );
  }
  ```

  Of course, the above contract-assertion predicates are destructive; the correctness of later invocations of these assertion statements is dependent on having evaluated all earlier instances with a checked semantic, which is not a property guaranteed by Contracts in [P2900R8]. Discouraging attempts to tie contract-assertion predicates together like this makes the facility more robust to use confidently in a much broader set of situations.

  ✘: Proposal A would allow the above assertion statements because the variable `callDepth` has static storage duration.

  ✔: Both Proposal B and Proposal C would identify `callDepth` as eligible for `const`-ification.

- **Concern: Global Variables**

  The same situation that is implemented with a `thread_local` variable above might instead be implemented with a global store outside the function:

  ```
  class CallDepthTracker {
    int increment(const char *fname);
    int decrement(const char *fname);
  } globalCallTracker;
  void f()
  {
    contract_assert( globalCallTracker.increment("f") == 0 );
    someOtherFunction();
    contract_assert( globalCallTracker.decrement("f") == 0 );
  }
  ```

  ✘: Proposal A and Proposal B would allow the above assertion statements because the variable `callDepth` is a namespace-scope variable.

  ✔: Proposal C would identify `callDepth` as eligible for `const`-ification.

- **Concern: Direct Logging**

  Some global utilities are, however, not generally used to satisfy the contract of a program but rather are used for diagnostics. Logging facilities are an example, and in many programs that do not concern themselves with produced output and error streams in particular formats, `std::cout` and `std::cerr` are freely used for tracing and diagnostics. A contract assertion might, while being completely correct, be written to trace its evaluation with log messages written to standard error:

  ```
  void f(int x)
    pre( []{
      std::cout << "f called with x = " << x << std::endl;
      return x >= 0;
    }());  // Check inside immediately invoked lambda to allow for tracing.
  ```

  ✘: This contract assertion, which is likely to be nondestructive, would be made invalid if global variables are subject to `const`-ification with Proposal C.

  ✔: Both Proposal A and Proposal B assume that variables at global scope are more likely to be outside the set of states on which the correctness of the function might depend, so they do not subject global variables to `const`-ification.

- **Concern: Easy and Incorrect Workaround**

  In some cases, a user who does not particularly understand the nuances of the new Contracts facility in C++ might attempt to avoid warnings or errors that result from `const`-ification by simply tossing in keywords until their code compiles. Consider the simple case that produces a warning or error when any strategy for selecting entities is chosen:

  ```
  void f()
  {
    int i = 0;
  ```

```
  contract_assert(++i == 0);  // Error
}
```

✘: With Proposal A, the following variations might be thrown out to simply get the code to compile but also might be highly likely to be less correct than the code that the developer began with:

```
void g()
{
  static int i = 0;
  contract_assert(++i == 0);
}
```

By simply throwing in `static`, the code now compiles, and the user is left with a program that still behaves incorrectly when contracts are enabled and disabled.

✗: With Proposal B, the naive workaround will fail, but a user might still move a variable outside their function to achieve a similarly broken result:

```
int i = 0;  // global variable now
void h()
{
  contract_assert(++i == 0);  // just trying to make this compile
}
```

✔: Proposal C makes all variables declared outside the contract assertion subject to `const`-ification, so there is no simple way to move a variable around to forcibly achieve broken yet compiling code.

- **Concern: SG21 Consensus**

  SG21 and EWG have discussed `const`-ification multiple times since the adoption of [P3071R1]. After reviewing a previous iteration ([P3261R1]) of this paper, Proposal C was polled while Proposal A remained the status quo. No sufficient interest to poll Proposal B was expressed.

  https://wiki.edg.com/bin/view/Wg21telecons2024/Teleconference2024-10-10

  Apply constification to all variables, as proposed in [P3261R1] Proposal C (including referenced from within a lambda-expression within a predicate).

SF	F	N	A	SA
6	14	2	0	3

  Result: Consensus

  ✔: Proposal C had strong consensus to adopt, and it was incorporated into [P2900R9].

Concern	A Automatic Variables	B Local Variables	C All Variables
Implementation Experience	✔	✘	✘
Implementation Feasibility	✔	✔	✔
Forward Compatibility	✘	✘	✗
Teachability of Contracts	✗	✓	✔
Function Parameters	✔	✔	✔
Automatic Local Variables	✔	✔	✔
`static` Local Variables	✘	✔	✔
Global Variables	✘	✘	✔
Direct Logging	✔	✔	✘
Easy and Incorrect Workaround	✘	✗	✔
SG21 Consensus	✘	✘	✔

Early conclusions from this analysis tell us that extending `const`-ification to include all variables outside the contract-assertion predicate seems like a very strong proposal here. While some static APIs, like logging facilities, might have issues being used directly in a contract-assertion predicate, we believe that those APIs are both less likely to be used directly in such predicates and can be worked around in other ways when needed. The middle ground of extending just to nonautomatic local variables seems to offer less real benefit, so we will not pursue Proposal B further.

4.3 Deep `const`-ness

Finally, we consider the two possible cases, Proposal D and Proposal E, where we could apply a form of built-in deep `const` to entities.

- **Concern: Implementation Experience**

  ✘: There is no implementation experience with attempting to apply this form of deep `const`-ness within contract predicates.

- **Concern: Implementation Feasibility**

  ✗: While the analysis to implement these proposals is predominantly local, it is a novel approach that would need significant effort to both specify and implement, and that analysis has not yet been undertaken.

- **Concern: Forward Compatibility**

  ✘: As with earlier proposals, changing our decision on these proposals would involve applying `const`-ification to more or fewer expressions, both of which have potential concerns that would make such a change highly unlikely to be viable in a future revision of the Standard.

- **Concern: Teachability of Contracts**

  ✘: Any introduction of a locally applied deep `const` to contract assertions would require extensive effort to specify and teach, increasing the cost of understanding Contracts far more than any of the other proposals in this paper.

- **Concern: Pointer Dereference**

  When a contract-assertion predicate is provided a pointer value that is itself a value we would consider for `const`-ification, the object denoted by that pointer is almost certainly also one that is outside the cone of evaluation of that predicate:

  ```
  void f(int * p)
    pre( *p += 5 );
  ```

  ✔: Only Proposal E would make the above code with obviously potentially destructive side effects ill-formed.

- **Concern: Smart Pointer Dereference**

  The same example as written above but instead written using `std::unique_ptr` is quite different in that the pointer being dereferenced is one returned by a `const` member function and not necessarily one that points to an object outside the cone of evaluation of the contract-assertion predicate:

  ```
  void f(std::unique_ptr<int> p)
     pre( *p += 5 );
  ```

  ✘: None of the proposals would make the above example ill-formed.

- **Concern: Factory Function Dereference**

  To illustrate the issue with not having user-defined deep `const`, consider an object whose `operator->` returned a `std::unique_ptr` to a freshly created object:

  ```
  struct Validator {
    bool validate();  // not const
  };
  struct S {
    std::unique_ptr<G> operator->() const;
  };
  void f(S s)
    pre( s->validate() );   // Modify dynamically allocated object.
  ```

  Compared to the previous example, the return value of `unique_ptr::operator->()` is being dereferenced and modified in both cases, yet without user-provided guidance, we can't easily determine that one case should propagate `const`-ness and another one should not.

  ✔: None of the proposals would identify the above example as ill-formed.

- **Concern: Mutable Member Variables**

  Modifying in a contract predicate is probably still unintentional, and a `const` member function that encapsulates such modification *does* provide the promise of `const`-correct behavior even though mutation is happening. Without encapsulation, we have no such promise, and thus making a modification directly within a contract predicate is likely ill advised:

```
struct S {
  mutable bool d_computed = false;

  void compute()
    pre(( d_computed = true ));  // oops
};
```

✔: By having `const`-ification propagate through member access expressions, the implicitly constructed member access expression `this->d_computed` above would be made into a `const` expression, and the above example would be ill-formed.

- **Concern: Consistency of User-Defined and Built-In Types**

  Writing user-defined types in C++ that behave in almost all ways as a built-in type is possible. When doing so, operators are often overloaded with user-provided functions that do not by necessity have the same semantics as a built-in operator. When properties of a built-in operator are going to be used that cannot be replicated by an overloaded operator, a pressure arises to use built-in types more and lose the great benefits of user-defined types, such as smart pointers:

```
void f(int * p)                 pre( (*p) = 5 );
void g(std::shared_ptr<int> p) pre( (*p) = 5 );
```

  ✘: Proposal E introduces propagation of `const`-ification through pointer dereference that cannot be replicated for a user-defined type, giving inconsistent results for the above two preconditions with no ability to alter `std::shared_ptr` to match the behavior of a built-in pointer.

  ✔: Proposal D does not alter the behavior of an operation that users are able to override and treats both of the above preconditions equally.

- **Concern: Reliable Escape Hatch**

  As mentioned elsewhere, when we impose a rule to disallow an action in Contracts, we must either be certain that it can never be allowed or we must consider escape hatches that let those who are aware of the issues work around the rule locally. For `const`-ification, that escape hatch is to `const_cast` back to a modifiable type. In the case of an id-expression, we have an even better option because we can `const_cast` back to the type of the entity denoted by the id-expression using decltype:

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

  Importantly, if the entity is actually `const`, the above macro won't remove that `const` and no risk of the undefined behavior to which that `const_cast` often exposes us is there.[17]

[17] A built-in operation that removes `const`-ification (and only `const`-ification) could be made to work with any proposal, but such an operation is beyond the scope of this paper. That operation would also have the advantage of not necessarily working unexpectedly in other situations, such as when applied to a variable captured by value in a lambda.

✘: Solutions that apply `const`-ification to expressions other than id-expressions will be unable to rely on `decltype` being applied to the expression since that relies on the difference in the result of `decltype` when applied to a name of an entity instead of an arbitrary expression. Proposal D and Proposal E would introduce expressions in which the above escape hatch does not work.

Concern	No Deep `const`	D Member Access	E Pointer Dereference
Implementation Experience	✔	✘	✘
Implementation Feasibility	✔	✗	✗
Forward Compatibility	✘	✘	✘
Teachability of Contracts	✔	✘	✘
Pointer Dereference	✘	✘	✔
Smart Pointer Dereference	✘	✘	✘
Factory Function Dereference	✔	✔	✔
Mutable Member Variables	✘	✔	✘
Consistency of User-Defined and Built-In Types	✔	✔	✘
Reliable Escape Hatch	✔	✘	✘

Early conclusions from this analysis indicate that all options involving deep `const` seem to have more concerns than benefits, including serious issues regarding completing and implementing their specifications, so we will not propose pursuing these options.

4.4 Escape Hatches

Due to the nature of software design, we will inevitably encounter cases in which a contract assertion that is not destructive must still be written in terms of functions that are passed pointers and references that are not, themselves, `const`. The most common motivating cases for this scenario are unchangeable APIs that are not `const`-correct and APIs that mutate with some input values but are known to be nonmutating with other input values.

Situations with a contract assertion that cannot be expressed due to `const`-ification can, of course, be worked around. Some of these workarounds exist in the language already, and others would require the introduction of new syntax and semantics.

- **Make APIs `const`-Correct** — The ideal solution is for functions that make no modifications to their parameters to be properly marked with `const` qualifiers, removing any impedance to using those functions within contract assertions.

  When possible, this approach produces the ideal results: Not only is the quality of software improved by having contract assertions introduced into it, but the static properties of a program that are the results of `const`-correctness are better utilized. Of course, this approach is not always possible, in particular for very large codebases that do not follow modern standards or that are under third-party control.

- **Wrap APIs in `const`-Correct APIs** — For libraries that cannot be altered, users can write wrapper functions that accept `const` pointers and references and that perform the `const_cast` in a central, well-vetted location that then forwards arguments on to the underlying non-`const`-correct APIs. A function that is conditionally nonmutating, such as `std::map::operator[]`, can be given a `const` wrapper that throws if the key requested is not in the map and performs the `const_cast` otherwise:

```
template <typename K, typename V>
const V& nonmodifying_map_access(const std::map<K,V>& m, const K& key) {
  auto it = m.find(key);
  if (it == m.end()) {
    throw std::out_of_range("Key not found");
  }
  return it->second;
}
```

- **Apply `const_cast`** — The proposals here specifically do not apply to the results of a `const_cast`, and we can remove `const`-ification through the use of a `const_cast` to the type of the entity itself:

```
int i;
bool check(int&);
void f1() pre( check(i) );                    // Error, i is const.
void f2() pre( check( const_cast<int&>(i) ) );  // Ok
```

  Of note, `const_cast` is often considered inappropriate to use under any circumstances due to the risks of circumventing the assumptions of users that a `const` variable will not be modified and, even worse, is undefined behavior when the modification happens to a variable declared with a `const` qualifier on its complete object.

```
const int j;
bool modify(int&);
void f3() pre( modify( const_cast<int&>(j) ) );  // well-formed but UB
```

  Of course, the concerning undefined behavior happens only when an actual modification happens to the object with a top-level `const` qualifier. Users doing the above `const_cast` must take it upon themselves to not only construct the declared type of the variable properly, but to use such a cast only when the `const` qualifier is due to `const`-ification, not because it is in a `const` member function, it has a `const`-qualified object, or similar reasons.

  In practice, we will often see the need for wrappers like this when using an API written in a legacy C style where non-`const` pointers to structs are passed to functions that encapsulate business logic:

```
struct MyData {
  // ...
}
int isDataValid(MyData* data);  // Return 1 if data is valid; 0, otherwise.
```

  To use a function like this in a contract assertion we will, of course, need to apply the appropriate `const_cast`:

```
void f(MyData data)
  pre( isDataValid( &data ) )                          // Error
  pre( isDataValid( &const_cast<MyData&>(data) ) );  // Ok
```

- **Encapsulate `const_cast`**

  As was mentioned earlier in this paper, an encapsulated `const_cast` that uses `decltype` can provide a fair bit of protection against accidentally misusing `const_cast` to remove `const`-ification:

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

  This spelling has significant advantages over the direct use of `const_cast` within contract assertions.

  – The user does not have to figure out and reproduce the type of the variable.

  – When a variable is declared `const`, this macro will not remove `const` from the expression denoting that variable, protecting against at least some of the cases where `const_cast` would be deemed inappropriate by many coding standards.

  Of course, using this approach still has limitations.

  – Within a `const` member function, this macro will also remove the `const` applied to id-expressions that denote member variables.

  – Within a nonmutable nested lambda expression, this macro will also make a by-value capture within the lambda mutable.

  – This approach does not work if any proposal for deep `const` is adopted, such as Proposal D or Proposal E.

- **Evaluate Predicates Outside `contract_assert`**

  Another option that is applicable to `contract_assert` is to evaluate the contract-assertion predicate *outside* the assertion statement and then use only the boolean result as the predicate. Consider, for example, an existing macro, `my_assert`, that we would like to integrate with the Contracts facility but for which we cannot afford the costs imposed of supporting `const`-ification with the wide variety of uses we have for our macro.

  To integrate with Contracts, we can redefine our macro to use an immediately-invoked lambda:

  ```
  #define MY_ASSERT(X) [](bool b){ contract_assert(b); }(X)
  ```

  Of course, this new macro is subtly different from `contract_assert(X)` in a few ways.

  – `const`-ification is not applied to the expression `X`.

  – The evaluation of `X` is not explicitly elided if the `contract_assert` within the lambda body is *ignored*.

    * If `X` is known by the compiler to have no side effects, it can be fully optimized away due to its result being discarded, but such optimization depends on information that is not always available.

    * If `X` has otherwise benign side effects, such as allocating, deallocating, logging trace messages, or acquiring locks, then those side effects cannot be elided at all and must happen, regardless of the semantic chosen for the assertion.

  – The `comment` field of the `contract_violation` object passed to the contract-violation handler will, when a violation is detected by a macro like this one, invariably be populated with the not particularly informative value of `"b"`.

Even with these downsides, some environments will certainly find this escape hatch more than sufficient.

Another alternative to have the same number of evaluations that a `contract_assert` would otherwise have is to embed the expression in a local lambda that is then invoked by the `contract_assert`:

```
#define MY_ASSERT(X) do {           /* Embed in `do`/`while` for new scope. */\
  auto cond = [&] { return (X); };  /* lambda to evaluate predicate         */\
  contract_assert(cond());          /* Assert return value of lambda.       */\
} while (false)
```

A production implementation of the above macro would probably also use a different identifier than `cond` to avoid potentially colliding with identifiers used in the expression, so the identifier would likely be something along the lines of `mYaSsErTLaMbDaCoNd`, which would then appear in the `comment` passed to the contract-violation handler.

Finally, for those who also want an escape hatch for potential multiple evaluations of a contract assertion, a lambda can be used that evaluates its expression only once:

```
#define MY_ASSERT(X) do {               /* Embed in `do`/`while` for new scope.*/\
  auto cond = [&,result=false,evaluated=false](){                              \
    if (!evaluated) {                                                          \
      const_cast<bool&>(evaluated)=true;  /* Set `evaluated` first so we still */\
                                          /* fail if exception is thrown.      */\
      const_cast<bool&>(result)=(X);      /* Determine result.                 */\
    }                                                                          \
    return result;                        /* Return (possibly cached) result.  */\
  };                                                                           \
  contract_assert(cond());                /* Assert return value of lambda.    */\
} while (false)
```

Note that a mutable lambda cannot be used since the call operator on `cond` must be `const`-qualified, but the closure object itself is not `const`, so the `const_cast` usage within the lambda body is well defined. The two identifiers used above for the internal state of the closure object, `result` and `evaluated`, must also be given names that will not collide with the expression `X`.

- **Add an Operator to Prevent `const`-ification**

  We could add a built-in operator, which we might call `unconst`, that, when applied to an expression, removes any alternative interpretation of the `const`-ness that happens to that expression because of `const`-ification:

  ```
  int i;
  const int j;

  bool check(int&);
  void f4() pre( check( unconst(i) ) )   // Ok
            pre( check( unconst(j) ) );  // Error, j is still const.
  ```

  Such an operator could overcome the remaining flaws in the macro-based approach.

  – By not using a macro, we would avoid stigmas associated with the preprocessor and issues with tooling understanding this use of the language.

  – Errors in use, such as applying the operator outside a contract assertion or to a expression that is not subject to `const`-ification, would be prevented.

  – An operator like this would be able to understand deep `const` and remove its effects when asked to.

  – Such an operator could be defined to produce errors when used in places that have not been subjected to `const`-ification, improving the understanding of contract assertions for anyone attempting to blindly sprinkle uses of the operator everywhere.

  Formally, this operator would be proposed as follows.

  Proposal E1: Add an `unconst` Operator

  Add a new unary expression type, an `unconst` expression.

  – The expression is introduced by a new keyword, `unconst`.
  – The grammar for the expression will be a new production for *unary-expression*:
  unary-expression :
  `unconst (` *expression* `)`
  – The value category and result of an `unconst` expression will be the same as *expression*.
  – The type of the `unconst` expression will be that of *expression* without any `const` added to it because of `const`-ification.
  – An `unconst` expression appearing outside a contract-assertion predicate is ill-formed.
  – It is ill-formed if *expression* is not an expression to which `const`-ification has been applied.

- **Add a Label to Prevent `const`-ification**

  Also suggested is to provide a mechanism to turn `const`-ification off in an entire contract assertion. This option could be accomplished with a label (see [P2755R1] for an overview) that had this effect:

```
int i;
bool check(int&);

void f1() pre                   ( check(i) ); // Error, i is const.
void f2() pre no_constification ( check(i) ); // Ok
```

Such a label would have the downside of also removing `const`-ification from all parts of the expression that do not need it, allowing for accidental modifications to local scalars just because a non-`const`-correct API is in use:

```
int i;
void f3() pre no_constification( ++i && check(i) );  // Ok?
```

The contextual keyword suggested above is long to achieve two goals:

1. Make the keyword easy to find via search since any coding practice that recommends `const`-correctness will want to quickly identify any cases in which a contract assertion subverts that goal by working around it with a blanket tool.
2. Clarify exactly what is being undone — `const`-ification — without implying that extra mutations that would not normally be allowed are being enabled or that a contract assertion itself somehow has `mutable` or `const` state.

> **Proposal E2: Add an Optional Label that Suppresses `const`-ification to Contract Assertions**
>
> Add a new label supported by all contract-assertion specifiers.
> – Labels occur in the grammar of contract assertions where the *attribute-specifier-seq* can currently be placed.
> – This label is identified by the identifier with special meaning (which is not an attribute since it changes the semantics of the associated contract-assertion predicate), `no_constification`
> – Within a contract-assertion predicate and where the contract assertion has the `no_constification` label, `const` is not added to expressions that are subject to `const`-ification.

- **Add a Label to Enable `const`-ification**

  In addition to enabling `const`-ification, the same label could be introduced and *required*, leaving undecided what the default application of `const`-ification with no label would be:

```
int i;
bool check(int&);

void f3() pre constification ( check(i) );  // Error, i is const.
```

Proposal E3: Require Labels and Add a Label to Enable `const`-ification

In addition to Proposal E2, add another label to enable `const`-ification.
- The label is identified by the identifier with special meaning, `constification`.
- When present, the effects of `const`-ification are not applied.
- A `pre`, `post`, or `contract_assert` that has no `constification` or `no_constification` label is ill-formed.

This proposal would allow for user experience to guide the determination of an acceptable default for contract assertions.

For all above escape hatches, concerns must be considered, especially if we are going to explore a change to the language to facilitate working around `const`-ification.

- **Concern: Verbosity**

  The verbosity of any workaround can be seen as an advantage since it encourages users to update APIs to alternative `const`-correct ones. On the other hand, too much boilerplate, especially to manually reproduce the types of existing variable declarations, is a violation of software engineers' oft-repeated desire to not repeat themselves.

  ✔: Fixing APIs clearly has negative verbosity; software is improved, and no code remains specifically for the support of contract assertions.

  ✔: Evaluating predicates outside of `contract_assert` is a process that is mostly beneficial to those migrating legacy macros to Contracts, so the change will be made once in a single location (to a macro) without impacting most users.

  ✓: The macro or operator-based solutions are targeted tools to say exactly what they need to say and can be made as brief as desired, providing little syntactic overhead or need to repeat any already-known information.

  ✗: Wrapping APIs solely for the purpose of contract assertions is excessive overhead for many and can be considered overly verbose.

  ✘: Proposal E3 introduces the maximum amount of overhead to all uses of Contracts.

- **Concern: Don't Change Existing Code**

  Introducing contract checking into a codebase is generally done with the intent of verifying if that codebase is correct. Often the use of contract assertions to detect bugs will do so, and these bugs will also be fixed as part of the introduction of contract assertions. Changes to the actual code itself solely to facilitate the use of Contracts, however, are a concern to some.[18]

  ✘: Improving a non-`const`-correct API to be `const`-correct, while beneficial to the users of that API, is still a code change that some might not wish to undertake for the sake of introducing the use of contract assertions.

[18] Of course, just as software must be written to be testable, libraries must often be written with wider APIs for the purpose of writing contract checks that use types from those libraries. Very narrow APIs, such as those that do not allow full `const` usage, can always prove to be a hindrance to writing contract assertions that use information hiding within those APIs.

✗: Wrapping APIs in `const`-correct APIs widens the API surface available to clients significantly; an entire new layer becomes available and must be supported, and it can be considered a modification to code outside contract assertions for the sake of introducing contract assertions.

✔: All the other escape hatches are entirely used within a contract-assertion predicate and require no changes outside the newly introduced assertions.

- **Concern: Minimize Effort**

  Minimizing the effort to write contract assertions increases the level of adoption that Contracts might see.[19]

  ✘: Updating large non-`const`-correct APIs to be `const`-correct can be a huge design and engineering effort. Writing wrappers for such APIs is equally expensive.

  ✗: While limited to the contract assertions themselves, applying a `const_cast` manually and correctly is a challenging exercise.

  ✓: Using an encapsulated `const_cast` or an operator that removes `const`-ification is straightforward but demands that the programmer understand when to employ such tools within a contract-assertion predicate and thus might increase cognitive load.

  ✔: Simply removing `const`-ification from an entire contract assertion requires little thinking, understanding, or cognitive load and is the least-effort solution for escaping from whatever burdens might be perceived with `const`-ification.

  ✓: Making a change to an existing macro to use `contract_assert` within a lambda requires a small amount of effort for the maintainer of an existing macro but also might require deeper analysis for issues that might arise when an expression with intended side effects, such as maintaining the state of variables declared within `#ifndef NDEBUG` blocks, will not work correctly with the semantic model of [P2900R9] Contracts.

  ✘: Proposal E3 increases the effort required to write contract assertions for both predicates that use already `const`-correct types (including primitive types and standard library types) as well as those that are burdened with questionable legacy types that do not provide substitutable `const`-correct behavior.

- **Concern: Works With All Proposals**

  ✔: Switching to `const`-correct APIs will work with all the proposals in this paper, as would any new language feature we propose for this purpose.

  ✗: `const_cast` based alternatives will work poorly with deep `const` but should otherwise be effective.

  ✔: Evaluating predicates outside of the contract assertion in a lambda avoids any `const`-ification proposal in this paper.

[19] Of course, if introducing contract assertions into a codebase is highly error prone, adoption rates can quickly and unfortunately turn around into a rejection of using the feature.

- **Concern: Misusability**

  ✔: Writing `const`-correct APIs to wrap those that are not `const`-correct can, of course, be done incorrectly but is just as usable or misusable as the existing C++ language.

  ✘: Manually determining the proper target for a `const_cast` is highly error prone and is likely to result in mistakes or maintenance issues.

  ✓: Encapsulating `const_cast` along with the use of `decltype` works correctly in most practicable cases.

  ✔: A targeted operator can be designed that does nothing but prevent `const`-ification and reveal the type of the denoted entity.

  ✘: Proposal E2 and Proposal E3 that builds upon it both prevent `const`-ification from an entire contract assertion to work around problems of non-`const`-correct APIs but also leave the user open to all mistakes that might be related to accidental misuses of `const`-correct APIs, including built-in operators and anything in the Standard Library. Placing the contract expression in an external lambda has the same issues.

- **Concern: Specificity**

  Working around a non-`const`-correct API or carefully ensuring that specific parameters will not lead to modifications when using an API a certain way is an operation to which thought should be applied. A tool that removes all need for such thought is a step backward from the ideal feature if we want to maximize the probability of contract assertions being correct if they compile.

  ✘: A label to prevent all `const`-ification from a contract assertion is highly susceptible to a creative developer deploying a macro to apply it everywhere:

  ```
  #define mypre pre no_constification
  ```

  ✘: Evaluating an expression outside of a `contract_assert` is equally unspecific.

  ✔: All other solutions are specific to particular expressions or function calls that have been flagged as problematic by `const`-ification and must be worked around.

- **Concern: Bikeshedding**

  Any language feature that introduces new keywords or identifiers with special meaning must hit the inevitable delay of both finding and agreeing upon how to spell that identifier.

  ✔: Approaches that are not new language features need no bikeshedding.

  ✘: Approaches that do require a keyword obviously do require bikeshedding. In particular, the most common suggestion for a label is `mutable` simply because it is already a keyword, which has the fundamental problem that it is not the contract assertion itself that is in any way mutable. Similar issues, including a general decision on how labels should be chosen in a grammatically useful way, would need to be addressed for any new syntax proposal.

- **Concern: Complexity of Contracts Proposal**

  Any new operator or feature of the language brings with it cognitive load for users since they must be aware of what it does if they see it in use and why they would use it instead of other built-in features that support the same functionality. Any change we do suggest to the language that we expect to become part of the Contracts MVP also increases the complexity of that minimal product and must meet the high standard of being not only useful, but also necessary for the most basic uses of Contracts.

  ✘: All proposals for new language features increase the complexity of the language with features that are relevant to the use of Contracts in only very particular scenarios. In addition, these features introduce new special identifiers that users might need to be aware of even if they never use the new features. Given how they increase the complexity of Contracts as a feature, these approaches should first prove their utility in comparison to that cost.

  ✔: The other approaches involve no changes to the language and thus do not increase the complexity of the language.

- **Concern: Not Consuming Syntactic Real Estate**

  Contract assertions, being a new feature, have many paths of future syntactic evolution. Any new language feature that affects how we specify contract assertions must not only be cognizant of current uses of the feature, but also must be compatible with future evolutionary steps we might want to take.

  ✘: The syntactic space for labels — between the `pre`, `post`, or `contract_assert` and the parenthesized predicate — is currently unused and is open for any possible future evolution. Introducing a single label sets a possibly incorrect precedent for all future such evolutionary features.

  ✓: Although introducing an operator would consume syntactic real estate that other parts of the language could conceivably use due to using an identifier as a keyword, it would not require committing to a particular style of syntactic extensions to contract assertions themselves.

  ✔: Approaches not involving a language change do not prevent any form of future evolution nor does a new unary operator that removes `const`-ification.

- **Concern: Implementation Experience**

  We can identify whether there is implementation experience with any of these proposed options.

  ✔: All the proposals that do not involve a language feature have been tested with the existing implementations of [P2900R9].

  ✔: Proposal E2 has been implemented in GCC (with different identifiers).

  ✓: Proposal E3 has been partially implemented in GCC (with different identifiers) without the requirement that one of the two labels always be present.

  ✘: Proposal E1 has not been implemented.

- **Concern: Implementation Feasibility**

  A similar question is whether the proposals have large open questions about the complexity or challenges in implementing them.

  ✔: All the proposals that do not involve a language feature have been tested with the existing implementations of [P2900R9].

  ✔: Proposal E2 and Proposal E3 are equally straightforward tools that can be quickly added to existing implementations of Contracts, essentially requiring some checking of their use and a boolean value to track whether `const`-ification should be applied within a contract-assertion predicate.

  ✗: Proposal E1 requires the introduction of a new keyword — or potentially a contextual keyword — or possibly even a magic function-like tool with the appears (and name lookup rules) of a function in namespace `std`. All might prove challenging to implement and deploy, and research must be done to determine which is the best choice.

Concern	Make APIs `const`-Correct	Wrap APIs in `const`-Correct APIs	Apply `const_cast`	Encapsulate `const_cast`	Evaluate Predicates Outside `contract_assert`	E1 Add an Operator	E2 Add a Label	E3 Require Labels
Verbosity	✔	✗	✘	✓	✔	✓	✓	✘
Don't Change Existing Code	✘	✗	✔	✔	✔	✔	✔	✔
Minimize Effort	✘	✘	✗	✓	✓	✓	✔	✘
Works With All Proposals	✔	✔	✗	✗	✔	✔	✔	✔
Misusability	✔	✔	✘	✓	✘	✔	✘	✘
Specificity	✔	✔	✔	✔	✘	✔	✘	✘
Bikeshedding	✔	✔	✔	✔	✔	✘	✘	✘
Complexity of Contracts Proposal	✔	✔	✔	✔	✔	✘	✘	✘
Not Consuming Syntactic Real Estate	✔	✔	✔	✔	✔	✓	✘	✘
Implementation Experience	✔	✔	✔	✔	✔	✘	✔	✓
Implementation Feasibility	✔	✔	✔	✔	✔	✗	✔	✔

This analysis provides the following early conclusions.

- The approaches that are already supported by the language itself are more than sufficient for handling potential issues and making significant use of contract assertions in real software, so the language changes considered, although they improve quality of life somewhat for some small portion of potential users of contract assertions, do not actually enable any new uses.

- A new label (Proposal E2 or Proposal E3) shows significant concerns and provides no clearly apparent benefit over other options, so we do not believe that a label is the right solution to pursue.

- A new operator might be a viable solution of relatively high utility if we find that significant real-world use of contract assertions regularly encounters the need to work around `const`-ification. That has not been the case in either case study[20] that attempted to apply Contracts with `const`-ification to libraries with extensive existing use of assertion macros. An operator of this sort should be explored in the future but not as a requirement for the initial release of Contracts.

5 Conclusion

After deep analysis and long deliberation, the consensus in SG21 is to move forward with Contracts that have `const`-ification applied.

Overall, we hope this paper has clarified a few things points.

- `const`-ification is a sound approach to discouraging destructive side effects without taking away programmers' ability to write the contract-assertion predicates they need to write.

- For the simplest use cases, `const`-ification is a clear improvement that greatly helps educate new users of Contracts and improves their ability to write nondestructive predicates naturally.

- For experienced users of existing contract-assertion facilities, `const`-ification helps find real flaws and poses, in practice, no hindrance to the adoption of Contracts.

All combined, however, we are left with two options presented for EWG to consider:

1. The status quo of `const`-ification in [P2900R9], which is a combination of Proposal 5, Proposal C, and Proposal F in this paper

2. The removal of all `const`-ification expressions in contract-assertion predicates, as proposed by [P3478R0], which is a combination of Proposal 1 and Proposal G in this paper

Clearly, the preponderance of evidence in this paper would suggest that the first choice is the better solution, as repeated (and growing) SG21 consensus has concluded.

We additionally believe that pursuing one of the proposals to introduce language-based escape hatches to `const`-ification would be appropriate if consensus would be increased and the concerns raised above could be overcome:

- Proposal E1: Add an `unconst` Operator

- Proposal E2: Add an Optional Label that Suppresses `const`-ification to Contract Assertions

- Proposal E3: Require Labels and Add a Label to Enable `const`-ification

[20] See [P3268R0] and [P3336R0].

Acknowledgments

Thanks to Jens Maurer for originally introducing `const`-ification into the contracts MVP.

Thanks to Ville Voutilainen, John Spicer, Daveed Vandevoorde, Timur Doumler, Ran Regev, Oliver Rosten, John Lakos, Corentin Jabot, Greg Marr, Nina Ranns, Andrei Zissu, Attila Feher, Michael Wong, Andrzej Krzemieński, Frank Birbacher, Jason Merrill, and Richard Smith for useful feedback and discussions related to this paper.

Thanks to Lori Hughes for reviewing this paper, providing editorial feedback, and fussing with pie charts.

Bibliography

[N2668] A. Meredith, H. Boehm, L. Crowl, P. Dimov, and D. Krügler, "Concurrency Modifications to Basic String", 2008
http://wg21.link/N2668

[P2680R1] Gabriel Dos Reis, "Contracts for C++: Prioritizing Safety", 2023
http://wg21.link/P2680R1

[P2712R0] Joshua Berne, "Classification of Contract-Checking Predicates", 2022
http://wg21.link/P2712R0

[P2755R1] Joshua Berne, Jake Fevold, and John Lakos, "A Bold Plan for a Complete Contracts Facility", 2024
http://wg21.link/P2755R1

[P2834R1] Joshua Berne and John Lakos, "Semantic Stability Across Contract-Checking Build Modes", 2023
http://wg21.link/P2834R1

[P2877R0] Joshua Berne and Tom Honermann, "Contract Build Modes, Semantics, and Implementation Strategies", 2023
http://wg21.link/P2877R0

[P2900R10] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, "Contracts for C++", 2024
http://wg21.link/P2900R10

[P2900R8] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, "Contracts for C++", 2024
http://wg21.link/P2900R8

[P2900R9] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, "Contracts for C++", 2024
http://wg21.link/P2900R9

[P2932R3] Joshua Berne, "A Principled Approach to Open Design Questions for Contracts", 2024
http://wg21.link/P2932R3

[P2996R5] Barry Revzin, Wyatt Childers, Peter Dimov, Andrew Sutton, Faisal Vali, Daveed Vandevoorde, and Dan Katz, “Reflection for C++26”, 2024
http://wg21.link/P2996R5

[P3071R1] Jens Maurer, “Protection against modifications in contracts”, 2023
http://wg21.link/P3071R1

[P3091R2] Pablo Halpern, “Better lookups for `map` and `unordered_map`”, 2024
http://wg21.link/P3091R2

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

[P3268R0] Peter Bindels, “C++ Contracts Constification Challenges Concerning Current Code”, 2024
http://wg21.link/P3268R0

[P3285R0] Gabriel Dos Reis, “Contracts: Protecting The Protector”, 2024
http://wg21.link/P3285R0

[P3336R0] Joshua Berne, “Usage Experience for Contracts with BDE”, 2024
http://wg21.link/P3336R0

[P3478R0] John Spicer, “Constification should not be part of the MVP”, 2024
http://wg21.link/P3478R0