Non-transient constexpr allocation using `propconst`

Document #:	P1974R0
Date:	2020-05-14
Project:	Programming Language C++
Audience:	EWG, LWG, LEWG

Reply-to:	Jeff Snyder <jeff-isocpp@caffeinated.me.uk> Louis Dionne <ldionne@apple.com> Daveed Vandevoorde <daveed@edg.com>

Contents

1 Abstract

This paper explores the problems of non-transient constexpr allocations, and proposes a set of constness-based requirements that govern when it is safe to allow such allocations, and when they can be placed in immutable storage. Unfortunately, the current (C++20) requirements are too strict to allow current implementations of many important allocating library types (e.g. `vector`) to be used as top-level `constexpr` variables, and it is necessary for them to be so. This is because even though such types meet the requirements in spirit, C++'s type system lacks the expressive power required for the compiler to be able to prove that such types meet them in

actuality. To bridge this gap, this paper proposes `propconst`, a new cv-qualifier that expresses "deep constness" and is closely related to `propagate_const`[N4388][N4600]. Lastly, we demonstrate how usage of `propconst` solves the `constexpr vector` problem, lets users protect themselves from unintentional const-correctness violations, removes the need for `propagate_const` and gives us an owning `propagate_const` almost for free.

2 Status of this paper

This paper is a new core language feature proposal targeting C++23.

3 Motivation

3.1 Non-transient constexpr allocation

Promotion of non-transient constexpr allocations to static storage was proposed by [P0784R5], but the feature was not accepted into C++20 due to concerns surrounding composability. For example, should the following snippet compile?

```
constexpr unique_ptr<unique_ptr<int>> uui
    = make_unique<unique_ptr<int>>(make_unique<int>());
int main() {
    unique_ptr<int>& ui = *uui;
    ui.reset();
}
```

As part of the test for whether a constexpr allocation can be promoted to static storage, [P0784R5] proposed that "evaluating that destructor would be a valid core constant expression and would deallocate all the non-transient allocations produced by the evaluation of expr.", and the above example satisfies that part of the test. However, as P0784 also points out, the test is meaningless because the destructor reads from a variable that is mutable at runtime (the internal `int*` in the inner `unique_ptr<int>`), and so without further guarantees about its immutability being provided, it must be rejected.

For the destructor evaluation test to be meaningful, we need to guarantee that a real destruction after program execution would behave the same as the the destructor did in the test, and this suggests that all we need is an extra condition for the destructor evaluation test:

> The simulated destruction is unsound and invalid unless all variables read during the destructor evaluation test are `const`, and are not part of an object that could be destructed at runtime

The example above violates this condition because the `unique_ptr<int>`'s internal `int*` member is not `const`, making it ill-formed as desired. Similarly, `unique_ptr<string>` and `unique_ptr<vector<int>>` fail to meet this condition and are ill-formed, as they must be. On the other hand, this condition would allow `constexpr unique_ptr<int>` and `constexpr vector<int>`, since their internal pointers are `const` by virtue of being non-static data members of a `const` object. So far so good.

Next, let's consider the same example, but using single-element `vector`s instead of `unique_ptr`s:

```
constexpr vector<vector<int>> vvi = {{1}};
int main() {
    vector<int>& vi = vi[0]; // ill-formed: discards const qualifier
    vi = vector<int>{}
}
```

Unlike `unique_ptr`, `vector` is a "deep const" type, meaning that a `const vector` does not let the user mutate its elements. Without the ability to mutate the `vector<vector<int>>`'s elements, we can't write code that would lead to a double-free like we could in the `unique_ptr` example, and so the reasoning for making `constexpr unique_ptr<unique_ptr<int>>` ill-formed does not carry over to this example.

Unfortunately, by the same reasoning we used for the `constexpr unique_ptr<unique_ptr<int>>` example, this `constexpr vector<vector<int>>` example would fail to satisfy our proposed condition. The problem here is that the "deep const" property of `vector` is informal, it is expressed by the author of `vector` taking great care to ensure that no `const`-qualified member function leaks a non-`const` reference to one of its members (either directly or indirectly, e.g. via an iterator). Since the compiler has no understanding of deep constness, we cannot modify our non-transient `constexpr` allocation condition in a way that will differentiate between `unique_ptr` and `vector`.

If we could make the compiler aware that `vector` was a deep-const type, we could allow `constexpr vector<vector<int>>` without modifying the non-transient constexpr allocation condition. One approach to this was proposed by [P0784R5], in the form of the `std::mark_immutable_if_constexpr` library function, which would be called by the authors of deep-const types during their constructors. This would require the authors of deep-const types to call `std::mark_immutable_if_constexpr` when and *only* when it is appropriate to do so, adding to the burden of writing a deep-const type. This was discussed at the Kona 2019 meeting, but did not get approval from EWG. As a result, non-transient constexpr allocations were removed from C++20.

This paper takes another approach to giving the compiler visibility into deep constness: adding it into the type system. In addition to solving the problem of allowing `constexpr vector<vector<int>>` whilst disallowing `constexpr unique_ptr<unique_ptr<int>>`, it would also reduce the burden on the authors of deep-const types by making it possible for the compiler to diagnose leaks of mutable references.

3.2 Writing deep-`const` types safely

C++'s `const` rules provide a great deal of help to users in getting their code to be `const`-correct: attempts to mutate a `const` variable will not compile, neither will attempts to mutate a variable via pointer/reference-to-`const` or form a pointer/reference to non-`const` from a `const` one. Furthermore, `const`-qualification of member functions separates those that mutate state from those that don't, and within a `const`-qualified member function, attempts to mutate member variables are ill-formed (excluding `mutable` members).

However, these protections only go as far as the first pointer or reference indirection. If a class has a non-static data member of pointer type, then whilst the pointer itself cannot be changed within a `const` member function, the data pointed to may be changed.

A common pattern in C++ types, including almost all variable-size container types, is to have heap-allocated data owned by the class which is considered to be part of the value of the class. Such types are often called "deep `const`", because their `const`-qualified member functions promise not to modify the data owned by the class. Since the data is heap-allocated and accessed via a pointer or reference, the C++ type system does nothing to help here, laying the burden of ensuring that the class's value is not changed by a `const` member function entirely upon the author of the class. The creators of the D programming language considered this to be a substantial enough issue that they chose a transitive model of constness for D, where dereferencing a `const` pointer can not yield non-`const` access to the underlying data (a choice that has its own downsides).

As an example, consider this usage of the PImpl idiom:

```
struct S
{
    S(int i) : m_impl{std::make_unique<Impl>(i)} {}
    void apply(auto fn)       { fn(m_impl->i); }
    void apply(auto fn) const { fn(m_impl->i); }
private:
    struct Impl { int i; };
    std::unique_ptr<Impl> m_impl;
};

int main()
{
    const S s{42};
```

```
    s.apply([](int& i){ ++i; }); // Compile error? no.
};
```

Whilst it looks like the call to apply should fail, since the object s is const and it will therefore call the const-qualified overload of S::apply, it will nevertheless compile and mutate s.m_impl->i.

This has been a limitation of C++'s type system since its early days, and our proposed propconst offers a way to resolve it. In the above example, replacing std::unique_ptr<Impl> by std::unique_ptr<propconst Impl> concisely provides the guarantees the author of the class expected—that an attempt to modify m_impl->i from a const-qualified member function of S will not compile, at least not without using const_cast or its ilk.

4 The propconst cv-qualifier

In order to give the compiler visibility of the "deep const" property of types such as vector, we propose adding a new cv-qualifier, tentatively named propconst. The propconst qualifier is only meaningful within a pointer or reference type, i.e. T propconst* or T propconst&. In the pointer case, propconst resolves to const if and only if the pointer is immutable, and is a no-op otherwise. In the reference case propconst behaves as-if the reference was a pointer. This results in T propconst* behaving very similarly to propagate_const<T*>. Top-level propconst is ignored, and we could choose to make it ill-formed if it is written verbatim in a variable declaration.

Below are some examples of how propconst within pointer types resolves when a variable whose type involves propconst is used in an expression:

Variable type	Resolves to expression type
`int propconst*`	`int      *`
`int propconst* const`	`int const* const`
`int           * propconst`	`int      *`
`int const     * propconst`	`int const*`
`int propconst* propconst`	`int      *`
`int propconst*           *`	`int      *      *`
`int propconst* const     *`	`int const* const*`
`int           * propconst*`	`int      *      *`
`int const     * propconst*`	`int const*      *`
`int propconst* propconst*`	`int      *      *`
`int propconst*           * const`	`int      *      * const`
`int propconst* const     * const`	`int const* const* const`
`int           * propconst* const`	`int      * const* const`
`int const     * propconst* const`	`int const* const* const`
`int propconst* propconst* const`	`int const* const* const`
`int           *           * propconst`	`int      *      *`
`int const     *           * propconst`	`int const*      *`
`int propconst*           * propconst`	`int      *      *`
`int           * const     * propconst`	`int      * const*`
`int const     * const     * propconst`	`int const* const*`
`int propconst* const     * propconst`	`int const* const*`
`int           * propconst* propconst`	`int      *      *`
`int const     * propconst* propconst`	`int const*      *`
`int propconst* propconst* propconst`	`int      *      *`

Reference types resolve similarly:

Variable type	Resolves to expression type
`int propconst&`	`int`
`int propconst*         &`	`int      *`
`int propconst* const   &`	`int const* const`
`int          * propconst&`	`int      *`
`int const    * propconst&`	`int const*`
`int propconst* propconst&`	`int      *`

When an object of pointer type is declared, and that object is not a non-static data member of a class, the mutability of the pointer itself is known—it's either `const` or it's not. Therefore, if the pointee type is `propconst`-qualified, whether that `propconst` will resolve to `const` or a no-op is also known. For example, the variable declaration `int propconst* ip` will resolve to `int* ip`, and `int propconst* const ip` will resolve to `int const* const ip`, as shown in the table above. However, if `int propconst* m_ip` is a non-static data member declaration, the constness of the pointer changes depending on whether the member is accessed via a `const` or non-`const` pointer (or reference) to the class, and this in turn affects how the `propconst` resolves. For example, within a member function that is not `const`-qualified, `m_ip` would resolve to `int* m_ip`, whereas in a `const`-qualified member function it would resolve to `int const* const mp`.

```
struct S {
  int propconst *ppi;
  void f() const {
    // The type of the expresion (ppi) here is "int const *const",
    // as-if ppi was declared with "int const *"
  }
  void f() {
    // this->ppi's type here is "int*"
    // The type of the expresion (ppi) here is "int *",
    // as-if ppi was declared with "int*"
  }
};
```

Below are some examples of how `propconst` resolution for non-static data members changes depending on the constness of the object that they are a member of:

Non-static data member type (of a class `T`)	Expression type when accessed via `T&`	Expression type when accessed via `T const&`
`int propconst*`	`int      *`	`int const* const`
`int propconst*         *`	`int      *      *`	`int      *      * const`
`int propconst* const   *`	`int const* const*`	`int const* const* const`
`int          * propconst*`	`int      *      *`	`int      * const* const`
`int const    * propconst*`	`int const*      *`	`int const* const* const`
`int propconst* propconst*`	`int      *      *`	`int const* const* const`
`int propconst&`	`int`	`int const`
`int propconst*         &`	`int      *`	`int      *`
`int propconst* const   &`	`int const* const`	`int const* const`
`int          * propconst&`	`int      *`	`int      * const`
`int const    * propconst&`	`int const*`	`int const* const`
`int propconst* propconst&`	`int      *`	`int const* const`

It is important to note that there is no new rule involved here—the above is just the interaction of the `propconst` rule with status-quo non-static data member constness.

4.1 Declared types and expression types

The `propconst` qualifier on a declaration resolves to either `const` or "nothing" (i.e. it is a no-op) at the point that the declared entity is used in an expression. This prevents the `propconst` qualifier from appearing directly in the type of an expression, which in turn ensures `propconst` qualification is not involved in overload resolution, and thus `propconst` is never needed as a part of a parameter type declarator or as a member function qualifier. `propconst` can, however, appear in type aliases, type template parameters, and types resulting from the use of `decltype`.

The resolution of `propconst` can be observed comparing `decltype(x)` to `decltype((x))`. The former yields the type of the variable as-written, and the latter yields the type of that variable used as an expression.

```
struct A
{
    int propconst* i_pc_ptr;

    void f()
    {
        static_assert(is_same_v<decltype(i_pc_ptr), int propconst*>);
        static_assert(is_same_v<decltype((i_pc_ptr)), int*&>);
    };

    void f() const
    {
        static_assert(is_same_v<decltype(i_pc_ptr), int propconst*>);
        static_assert(is_same_v<decltype((i_pc_ptr)), int const* const&>);
    };
};
```

4.2 Contexts in which `propconst` may be used

There are two questions to consider here: (1) "Where can the `propconst` keyword appear?" and (2) "Where can a type with an unresolved `propconst`—possibly an alias or a template parameter—be used?" The first is a question of where the user can write `propconst`, the second is about where that `propconst` may reach in the type system, either by direct use of the keyword or via type aliases and type template parameters.

Informally, we propose to permit the `propconst` keyword only in contexts where its effect is context-dependent, but a type involving an unresolved `propconst` would have no such limitation. For example, in the block-scope variable declaration `int propconst* i`, the `propconst` will always resolve to a no-op, and so that usage would be invalid. However, if the same declaration were of a non-static data member, the `propconst` could resolve in different ways (e.g. in `const` vs non-`const` member functions), which means it would be a meaningful and valid usage of the `propconst` keyword.

4.2.1 Examples

The interaction of these rules for is demonstrated in the following examples:

—
```
propconst int i1; // Error: invalid use of the `propconst` keyword
```

In type `propconst int`, the `propconst` is immediately resolvable, so the keyword may not be used here. Making this ill-formed helps the programmer catch a potential misunderstanding.

—
```
using T1 = propconst int;  // Valid: effect of propconst depends on context
                           //        in which T1 is used
```

In `T1`, the `propconst` keyword is not immediately resolvable. E.g. `decltype(declval<T1*>())` is `int*`, but `decltype(declval<T1* const>)` is `int const*`. Thus, use of the `propconst` keyword is allowed.

—
```
void f1(int propconst* const) {} // Error: invalid use of the `propconst` keyword
```

In the function type `void(int propconst* const)` the `propconst` is immediately resolvable (it would be equivalent to `const`), so the keyword may not be used here.

—
```
using T2 = int(propconst int*); // Error: invalid use of the `propconst` keyword
```

Even though `T2` is a type alias, there is no way to use `T2` that makes the `propconst` resolve to `const`, so the keyword may not be used here.

—
```
using T3 = int(T1*); // Valid: the `propconst` keyword does not appear directly
```

`T3` is also `int(propconst int*)`. The `propconst` here is also immediately resolvable, but the keyword is not used so there is no error.

4.3 Relationship to `std::experimental::propagate_const`

In intent and semantics, `propconst T*` is a drop-in replacement for `propagate_const<T*>`, except for cases where member functions of `propagate_const<T*>` are being called directly (e.g. `get()`, `get_underlying()`). However, they differ in the way they compose with smart pointers. `propagate_const` works as a wrapper, e.g. `propagate_const<unique_ptr<T>>`, whereas `propconst` composes with smart pointer types by being used as a qualifier on their pointee type, e.g. `unique_ptr<propconst T>`. There are advantages and disadvantages to both of these composition models: with `propagate_const`, users must call `get_underlying` to call methods specific to the smart pointer type (e.g. `unique_ptr::release`), whereas with `propconst` some of `unique_ptr`'s member functions need tweaks in order to work when the pointee type is `propconst`-qualified (see Smart pointer types below).

`propagate_const`	`propconst`

`propagate_const<unique_ptr<int>> p;`
 	
`p.get_underlying().reset()`
	`unique_ptr<propconst int> p;`
	
	`p.reset()`

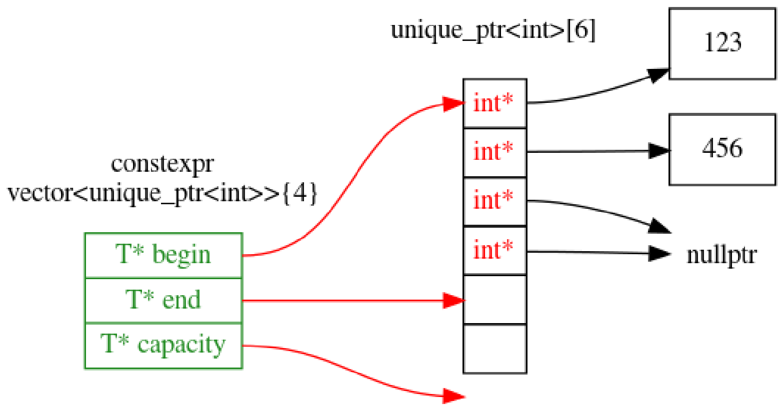
5 Non-transient constexpr allocations using `propconst`

We propose permitting non-transient constexpr allocations, provided that no variable read during constant destruction is reachable as mutable. If `const_cast` were used, undefined behaviour would ensue, and thus it wouldn't be constant destruction since core constant expressions cannot elicit undefined behaviour.

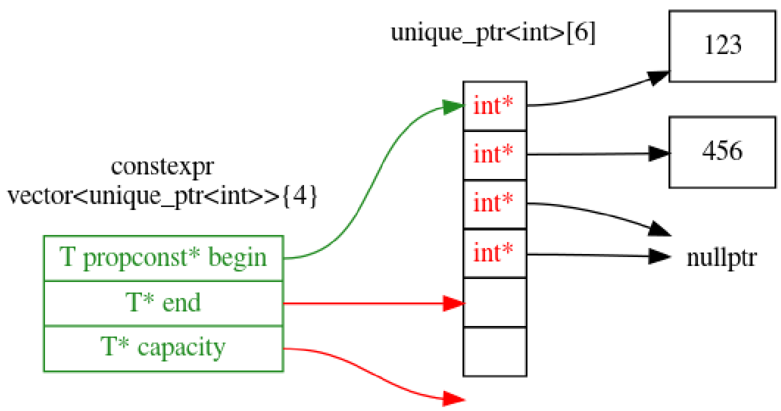
When `propconst` is correctly applied, this permits nested containers—like `vector<vector<int>>`—that do not violate this rule (see Existing deep-const types below).

Consider a constexpr global variable of type `vector<unique_ptr<int>>`. Note that the `int`s are mutable at runtime, and this is OK because their values do not affect constant destruction.

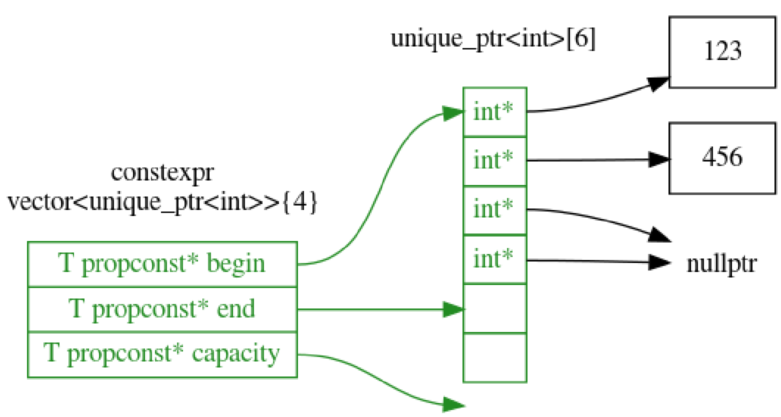
With a C++20 implementation of `vector`, this fails our proposed non-transient constexpr allocation test, because the raw pointers within `unique_ptr`s are reachable as mutable via the `begin`/`end`/`capacity` pointers in the vector:



If we make just the `begin` pointer `propconst`, then nothing changes, as the array of `unique_ptr<int>`s is still reachable as mutable via the `end`/`capacity` pointers:



However, if all three are `propconst` then there is no route to the array of `unique_ptr<int>`s that does not `const`-qualify it, and so we can be sure that its contents will not change at runtime, and thus it passes our non-transient constexpr allocation test:



6 Library Impact

6.1 Existing deep-const types

Existing deep-const library types (e.g. `tuple`/`variant`/`vector`/`set`/`map`) would be required to be usable as `constexpr` variables as long as any allocator involved is the standard default allocator. Implementation of this should be a no-op where there is no allocation involved (e.g. `tuple`/`variant`), and should be a matter of applying `propconst` appropriately in types that rely on dynamic allocation (e.g. container types).

Deep-const types with no implementation impact:

— `pair`
— `tuple`
— `optional`
— `variant`
— `array`

Deep-const types with implementation impact:

— `any`
— `basic_string`
— `vector`
— `deque`
— `forward_list`
— `list`
— [`unordered_`][`multi`]`set`
— [`unordered_`][`multi`]`map`

6.2 Smart pointer types

Some library types, notably smart pointer types, can be made to exhibit `propconst` behaviour if instantiated for `propconst T`, opening up the benefits of `propconst` to the users of smart pointers. In order to make this work, some accessors on such classes will need to change.

Changes are needed to member functions that give access to the underlying object, but are `const`-qualified. The simplest example of this is `observer_ptr::get`:

```
template <typename T>
struct observer_ptr
{
    // ...
    T* get() const { return m_ptr; }
    // ...
private:
    T* m_ptr;
};
```

Here, if `T` is `propconst U`, then `get()` will not compile, since the return type will resolve to `U*`, but the returned expression is of type `U const*`. Whilst this prevents the class from leaking const access to the pointee from a `const` member function, it also makes it useless.

To resolve this, we need to have both `const` and non-`const` overloads of `get`, and a type trait to determine type return type of the `const`-qualified `get` call:

```
template <typename T>
struct propconst_to_const { using type = T; };

template <typename T>
struct propconst_to_const<T propconst> { using type = T const; };
```

```cpp
template <typename T>
using propconst_to_const_t = typename propconst_to_const<T>::type;

template <typename T>
struct observer_ptr
{
    // ...
    T*                       get()       { return _ptr; }
    propconst_to_const_t<T>* get() const { return _ptr; }
    // ...
private:
    T* m_ptr;
};
```

If we take advantage of the function return types extension to this proposal (see below), the signature is more succinct and the type trait is unnecessary:

```cpp
template <typename T>
struct observer_ptr
{
    // ...
    T*       get()       { return _ptr; }
    T* const get() const { return _ptr; }
    // ...
private:
    T* m_ptr;
};
```

Types affected:

- `unique_ptr`
- `observer_ptr`
- `span`?
- `shared_ptr`
- `weak_ptr`

6.3 Type traits

With the introduction of a new qualifier like `const` and `volatile`, we might want to add new type traits. For example, we could consider adding `std::remove_propconst` to mirror `std::remove_const` and `std::remove_volatile`. We should also consider what to do with existing type traits such as `std::remove_cv` – should they change behaviour to also handle `propconst`?

Furthermore, Standard Library implementations may need to be adjusted to handle the new qualifier. For example, if a trait has a specialization on `T const` and `T volatile`, one should now consider whether a specialization on `T propconst` is required, and if so what the desired behaviour should be. This determination must be done on a case-per-case basis. Also note that this observation holds not only for the Standard Library, but for all generic libraries.

6.4 Forward compatibility

The introduction of `propconst` may have an impact on user code that does not handle `propconst` yet. For example, imagine the Standard Library returns a `propconst`-qualified type to the user through one of its type traits. If the user's code is not ready to handle the `propconst` qualifier (for example if the user has template specializations which do not handle `propconst`), it may fail to interoperate with the `propconst`-enabled library.

However, we would like to note that these issues are similar to the issues that were faced when introducing rvalue-references. We believe that these difficulties are worth overcoming given the type system improvement that `propconst` represents.

7 Extension: function return types

If we specify that `propconst` in function return types is resolved prior to discarding top-level cv-qualifiers on fundamental types, we can avoid the need for type trait usage in the specification of smart pointer library types.

```
template <typename T>
struct unique_ptr
{
    T* get();              // Resolves to U* if T is propconst U
    T* const get() const; // Resolves to const U* if T is propconst U
};
```

The return type of the `const`-qualified overload of `get()` resolves as follows:

1. We start with `T* const`
2. Suppose `T` is `propconst U`; the return type is now `propconst U* const`
3. `propconst` resolution yields `const U* const`
4. Top-level cv-qualifiers on fundamental types are discarded, yielding `const U*`

If `propconst` is not involved, `T* const` just has its top-level const discarded (per the current IS), and becomes `T*` as it was before.

Supporting this pattern for references would require a novel usage `const` in the grammar, but otherwise works the same as for pointers:

```
template <typename T>
struct unique_ptr
{
    T& operator*();              // Resolves to U* if T is `propconst U`
    T& const operator*() const; // Resolves to const U* if T is `propconst U`
};
```

8 Extension: `const`-qualified constructors

When a copyable smart pointer type is parameterized on `propconst`, copying the smart pointer (via a typical copy constructor taking `const T&`) must be disallowed since copying to a mutable variable would grant mutable access to the pointee:

```
void f(const observer_ptr<propconst int>& pi)
{
    observer_ptr<propconst int> pi2 = pi; // must be invalid:
    int& ri = *pi2;                        // would allow this mutability leak
    const observer_ptr<propconst int> pi2 = cpi; // also invalid, but need not be
}
```

The above will not compile, since it falls out of the proposed rules for `propconst` that `observer_ptr<propconst int>`'s copy constructor is ill-formed:

```
template <typename T>
struct observer_ptr
{
    observer_ptr(const observer_ptr& rhs)
        : m_ptr(rhs.m_ptr) // ill-formed
```

```
    {}

    T* m_ptr;
};
```

In the above snippet, if `T` is `propconst int`, the type of the expression `rhs.m_ptr` is `const int* const`. The type of `m_ptr` is `propconst int*`, but since the propconst may resolve either way in later use, it must be initialized with an `int*`. Therefore, initializing it from `rhs.m_ptr` would discard qualifiers.

However, if we knew that we were initializing a const object, it would be valid to initialize `rhs.m_ptr` with a `const int*` since the `propconst` within the const smart pointer object would always resolve to `const`. We could enable this pattern by adding `const`-qualified constructors:

```
template <typename T>
struct observer_ptr
{
    observer_ptr(const observer_ptr& rhs) const
        : m_ptr(rhs.m_ptr) // OK
    {}

    T* m_ptr;
};
```

In this `const`-qualified constructor, we know that the complete object will be `const` for its entire lifetime, so `m_ptr` is considered const. Therefore, it can be initialized with a pointer-to-const even when `T` is qualified with `propconst`.

```
void f(const observer_ptr<propconst int>& pi)
{
    observer_ptr<propconst int> pi2 = pi;       // still invalid
    const observer_ptr<propconst int> pi2 = pi; // OK
}
```

When a `const`-qualified constructor is available, the destructor must be compiled as if it were `const`-qualified to avoid use of `m_ptr` as non-const in the destructor. We may also be able to allow separate `const`-qualified destructors, if we are always able to determine whether the object was constructed as const.

9 References

[N4388] Jonathan Coe, Robert Mill. 2015. A Proposal to Add a Const-Propagating Wrapper to the Standard Library.
https://wg21.link/n4388

[N4600] Geoffrey Romer. 2016. Working Draft, C++ Extensions for Library Fundamentals, Version 2.
https://wg21.link/n4600

[P0784R5] Peter Dimov, Louis Dionne, Nina Ranns, Richard Smith, Daveed Vandevoorde. 2019. More constexpr containers.
https://wg21.link/p0784r5