The deleted copy-constructor of `propagate_const`

ISO/IEC JTC1 SC22 WG21 DXXXX 2015-04-28

Jonathan Coe <jbcoe@me.com>

Robert Mill <rob.mill.uk@gmail.com>

Guy Davidson <guy.davidson@creative-assembly.com>

Titus Winters <titus@google.com>

Andy Sawyer <andy.sawyer@gmail.com>

Abstract

This document gives motivation for the decision to delete the copy constructor and copy assignment from `propagate_const` and lays out the case for the inclusion of this class in a Library Fundamentals TS.

A summary of `propagate_const`

The template class `propagate_const` [N4388] allows a pointer type to be wrapped so that, when accessed through a `const` access path, the `const`ness is propagated to the pointee.

A class with pointer members can access non-`const` member functions of the pointees even when accessed through a `const` access path.

```
struct A {
  B* b_;

  void foo() const;
  void foo();
};
```

Both the `const` and non-`const` `foo()` are able to invoke non-`const` methods of the class `B` through the member `b_`.

Using `propagate_const`:

```
struct C {
  propagate_const<B*> b_;

  void foo() const;
  void foo();
};
```

only the non-`const` `foo()` is able to invoke non-`const` methods of the class `B` through the member `b_`.

The intention of `propagate_const` is not to unduly restrict the user but to prevent accidental `const`-incorrect use. A free-standing function, `get_underlying`, allows the wrapped pointer type to be explicitly extracted.

Copy construction and assignment

The copy and assignment operators of `propagate_const<T>` are deleted because copies from `const` `propagate_const<T>` references would otherwise allow creation of non-`const` `propagate_const<T>` objects and consequently non-`const` access to potentially shared state.

```
void bar(const C& cc) {
  C c(cc);
  c.foo(); // calls non-const foo
}
```

Move construction and assignment are allowed as they cannot be performed from `const` references.

For `propagate_const<std::unique_ptr<T>>` the deletion of copy and assigment is not a restriction as `std::unique_ptr` is itself a move-only type. For `propagate_const<std::shared_ptr<T>>` and `propagate_const<T*>` the restriction is limiting. This is intended behaviour: if copy construction from a `const` reference were possible then accidental loss of `const`-ness would be possible. It is also excessively restrictive.

Non-`const` copy construction and assignment

Copy-construction and assignment from a non-`const` reference does not present any danger of accidental loss of `const`-ness:

```
propagate_const(propagate_const& p);
```

could be made legal.

This design interacts poorly with other Standard Library components, like `std::vector`, which make use of copy and assignment from `const` references. Rather than propose a Standard Library addition that is known to work poorly with existing components, the authors have opted for a stronger than necessary restriction.

User-experience, TS 2 and extensions

As proposed, `propagate_const` identifies problems with accidental `const`-incorrectness where pointer members represent ownership and the pointee is part of an object's logical state.

Standard Library Fundamental Technical Specification 2 is the intended target for `propagate_const`. While the authors have real-world experience with `propagate_const` in its proposed, restrictive, form; they have not made broader amendments required for copy and assignment from non-`const` references to work with the Standard Library.

The authors are seeking further user experience and community feedback with `propagate_const` in its proposed form, using `get_underlying` to allow manual implementation of copy constructors where required. Once best-practice for copy construction from non-`const` references becomes clear then it will be incorporated into a further proposal.

References

[N4388] A Proposal to Add a Const-Propagating Wrapper to the Standard Library