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

I. Introduction

We propose the introduction of a propagate_const wrapper class that propagates const-ness to pointer-like member variables.

II. Motivation

The behaviour of const member functions on objects with pointer-like data members is seen to be surprising by many experienced C++ developers. A const member function can call non-const functions on pointer-like data members and will do so by default without use of consVcast.

Example:

```cpp
struct A
{
    void bar() const
    {
        std::cout << "bar (const)" << std::endl;
    }

    void bar()
    {
        std::cout << "bar (non-const)" << std::endl;
    }
};

struct B
{
    B() : m_ptrA(std::make_unique<A>()) {}

    void foo() const
    {
        std::cout << "foo (const)" << std::endl;
        m_ptrA->bar();
    }

    void foo()
    {
        std::cout << "foo (non-const)" << std::endl;
        m_ptrA->bar();
    }

    std::unique_ptr<A> m_ptrA;
};
```
int main()
{
    B b;
    b.foo();
    const B consVb;
    consVb.foo();
}

Running this program gives the following output:

foo (non-const)
bar (non-const)
foo (const)
bar (non-const)

The behaviour above can be amended by re-writing `void B::foo() const` using `consVcast` to explicitly call the `const` member function of `A`. Such a change is unnatural and not common practice. We propose the introduction of a wrapper class which can be used on pointer-like member data to ensure propagation of `const`-ness.

**Introducing propagate_const**

The class `propagate_const` is designed to function as closely as possible to a traditional pointer or smart-pointer. Pointer-like member objects can be wrapped in a `propagate_const` object to ensure propagation of `const`-ness.

A `const`-propagating `B` would be written as

```cpp
struct B
{
    B();              // unchanged

    void foo() const; // unchanged
    void foo();       // unchanged

    std::propagate_const<std::unique_ptr<A>> m_ptrA;
};
```

With an amended `B`, running the program from the earlier example will give the following output:

foo (non-const)
bar (non-const)
foo (const)
bar (const)

**The pimpl idiom with propagate_const**

The pimpl (pointer-to-implementation) idiom pushes implementation details of a class into a separate object, a pointer to which is stored in the original class [2].

```cpp
class C
{
    void foo() const;
```
void foo();

std::unique_ptr<CImpl> m_pimpl;
};

void C::foo() const
{
    m_pimpl->foo();
}

void C::foo()
{
    m_pimpl->foo();
}

When using the pimpl idiom the compiler will not catch changes to member variables within const member functions. Member variables are kept in a separate object and the compiler only checks that the address of this object is unchanged. By introducing the pimpl idiom into a class to decouple interface and implementation, the author may have inadvertently lost compiler checks on const-correctness.

When the pimpl object is wrapped in propagate_const, const member functions will only be able to call const functions on the pimpl object and will be unable to modify (non-mutable) member variables of the pimpl object without explicit consVcasts; const-correctness is restored. The class above would be modified as follows:

class C
{
    void foo() const;  // unchanged
    void foo();        // unchanged

    std::propagate_const<std::unique_ptr<CImpl>> m_pimpl;
};

Thread-safety and propagate_const

Herb Sutter introduced the appealing notion that const implies thread-safe [3]. Without propagate_const, changes outside a class with pointer-like members can render the const methods of that class non-thread-safe. This means that maintaining the rule const=>thread-safe requires a global review of the code base.

With only the const version of foo() the code below is thread-safe. Introduction of a non-const (and non-thread-safe) foo() into D renders E non-thread-safe.

struct D
{
    int foo() const { /* thread-safe */}
    int foo() { /* non-thread-safe */}
};

struct E
{
    E(D& pD) : m_pD{&pD} {}

    void operator() () const
    {
        m_pD->foo();
    }
D* m_pD;
};

int main()
{
    D d;
    const E e1(d);
    const E e2(d);

    std::thread tl(e1);
    std::thread t2(e2);
    tl.join();
    t2.join();
}

One solution to the above is to forbid pointer-like member variables in classes if \texttt{const=>thread-safe}. This is undesirably restrictive. If instead all pointer-like member variables are decorated with \texttt{propagate\_const} then the compiler will catch violations of \texttt{const-ness} that could render code non-thread-safe.

```cpp
struct E
{
    E(D& pD);                 // unchanged
    void operator() () const; // unchanged

    std::propagate\_const<D*> m_pD;
};
```

Introduction of \texttt{propagate\_const} cannot automatically guarantee thread-safety but can allow \texttt{const=>thread-safe} to be locally verified during code review.

### III. Impact On the Standard

This proposal is a pure library extension. It does not require changes to any standard classes, functions or headers.

### IV. Design Decisions

Given absolute freedom we would propose changing the \texttt{const} keyword to propagate \texttt{const-ness}. That would be impractical, however, as it would break existing code and change behaviour in potentially undesirable ways. A second approach would be the introduction of a new keyword to modify \texttt{const}, for instance, \texttt{deep const}, which enforces \texttt{const-propagation}. Although this change would maintain backward-compatibility, it would require enhancements to the C++ compiler.

We suggest that the standard library supply a class that wraps member data where \texttt{const-propagating} behaviour is required. The \texttt{propagate\_const} wrapper can be used much like the \texttt{const} keyword and will cause compilation failure wherever \texttt{const-ness} is violated. \texttt{const-propagation} can be introduced into existing code by decorating pointer-like members of a class with \texttt{propagate\_const}.

The change required to introduce \texttt{const-propagation} to a class is simple and local enough to be enforced during code review and taught to C++ developers in the same way as smart-pointers are taught to ensure exception safety.
It is intended that propagate_const contain no member data besides the wrapped pointer. Inlining of function calls by the compiler will ensure that using propagate_const incurs no run-time cost.

**Encapsulation vs inheritance**

Inheritance from the wrapped pointer-like object (where it is a class type) was considered but ruled out. The purpose of this wrapper is to help the author ensure const-propagation; if propagate_const<T> were to inherit from T, then it would allow potentially non-const member functions of T to be called in a const context.

**Construction and assignment**

A propagate_const<T> should be constructable and assignable from a U or a propagate_const<U> where U is any type that T can be constructed or assigned from. There should be no additional cost of construction for a propagate_const<T> beyond that for construction of a T. The wrapped T should not be value-initialized as this would incur a cost for raw pointers. If value-initialization is desirable then it can be accomplished with another wrapper class like boost::value_initialized [4].

**Pointer-like functions**

operator* and operator-> are defined to preserve const-propagation. When a const propagate_const<T> is used only const member functions of T can be used without explicit casts.

**get**

The get function returns the address of the object pointed to by the wrapped pointer. get is intended to be used to ensure const-propagation is preserved when using interfaces which require raw C-style pointers

**operator value***

When T is a raw pointer operator value* exists and allows implicit conversion to a raw pointer. This avoids using get to access the raw pointer in contexts where it was unnecessary before addition of the propagate_const wrapper.

**Equality, inequality and comparison**

Free-standing equality, inequality and comparison operators are provided so that a propagate_const<T> can be used in any equality, inequality or comparison where a T could be used. const-propagation should not alter the result of any equality, inequality or comparison operation.

**swap**

The swap function should not add or remove const-ness but should not unduly restrict the types with which propagate_const<T> can be swapped. If T and U can be swapped then const-propagating T and U can be swapped.
get_underlying

get_underlying is a free-standing function which allows the underlying pointer to be accessed. The use of this function allows const-propagation to be dropped and is therefore discouraged. The function is named such that it will be easy to find in code review.

hash

The hash struct is specialized so that inclusion of propagate_const does not alter the result of hash evaluation.

noexcept

We have omitted noexcept specifications until we have stronger guidelines from the Library Evolution Working group. The noexcept status of all functions will depend on the noexcept status of the corresponding functions on the underlying pointer.

V. Expository Implementation

A sample form of std::propagate_const is given below and makes use of an overloaded templated helper function: underlying_pointer.

```
template <typename T>
class propagate_const
{
    typedef decltype(*std::declval<T>()) reference_type;

public:

    using value_type = std::enable_if_t<std::is_lvalue_reference<reference_type>::value, typename std::remove_reference<reference_type>::type>::type;

    -propagate_const() = default;
    propagate_const() = default;

    template <typename U, typename V = std::enable_if_t<std::is_convertible<U,T>::value>>
    propagate_const(U&& u) : t{std::forward<U>(u)}
    {
    }

    template <typename U, typename V = std::enable_if_t<std::is_convertible<U,T>::value>>
    propagate_const<T>& operator = (U&& u)
    {
        t = std::forward<U>(u);
        return *this;
    }

    template <typename U, typename V = std::enable_if_t<std::is_convertible<U,T>::value>>
    propagate_const(const propagate_const<U>& pu) : t{pu.t}
    {
    }
```

template <typename U,
    typename V = std::enable_if_t<std::isConvertible<U,T>::value>>
propagate_const(propagate_const<U>&& pu) : t{(std::move(pu.t))} {
}

template <typename U,
    typename V = std::enable_if_t<std::isConvertible<U,T>::value>>
propagate_const<T>& operator = (const propagate_const<U>& pt)
{
    t = pt.t;
    return *this;
}

value_type* operator->() {
    return underlying_pointer(t);
}

const value_type* operator->() const {
    return underlying_pointer(t);
}

value_type* get() {
    return underlying_pointer(t);
}

const value_type* get() const {
    return underlying_pointer(t);
}

template <typename T_ = T,
    typename V = std::enable_if_t<std::isConvertible<T_>::value>>
operator value_type*() {
    return underlying_pointer(t);
}

template <typename T_ = T,
    typename V = std::enable_if_t<std::isConvertible<T_>::value>>
operator const value_type*() const {
    return underlying_pointer(t);
}

value_type& operator*() {
    return *t;
}
const value_type& operator*() const
{
    return *t;
}

explicit operator bool () const
{
    return static_cast<bool>(t);
}

private:
    T t;

template<typename U>
static value_type* underlying_pointer(U* p)
{
    return p;
}

template<typename U>
static value_type* underlying_pointer(U& p)
{
    return p.get();
}

template<typename U>
static const value_type* underlying_pointer(const U* p)
{
    return p;
}

template<typename U>
static const value_type* underlying_pointer(const U& p)
{
    return p.get();
}
};

template <typename T, typename U>
bool operator == (const propagate_const<T>& pt, const propagate_const<U>& pu)
{
    return pt.t == pu.t;
}

template <typename T, typename U>
bool operator != (const propagate_const<T>& pt, const propagate_const<U>& pu)
{
    return pt.t != pu.t;
}

template <typename T, typename U>
bool operator < (const propagate_const<T>& pt, const propagate_const<U>& pu)
{
    return pt.t < pu.t;
}

template <typename T, typename U>
bool operator > (const propagate_const<T>& pt, const propagate_const<U>& pu)
{
    return pt.t > pu.t;
}
VI Acknowledgements

Thanks to Walter Brown, Kevin Channon, Nick Maclaren, Roger Orr, Ville Voutilainen, Jonathan Wakely, David Ward and others for helpful discussion.

VII References

- [3] Herb Sutter, C++ and Beyond 2012: Herb Sutter - You don't know [blank] and [blank]
- [4] boost value_initialized