Doc number: N3973
Date: 2014-05-12
Project: Programming Language C++, Library Evolution Working Group
Reply-to: Jonathan Coe <jbcoe@me.com>
Robert Mill <rob.mill.uk@gmail.com>

A Proposal to Add a Logical Const Wrapper to the Standard Library Technical Report

I. Introduction

We propose the introduction of a `logical_const` wrapper class that propagates `const`-ness to pointer-like (or reference-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 `const_cast`; that is, `const` on member functions provides physical but not logical `const`-ness [1].

Example:

```
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 const_b;
  const_b.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 `const_cast` 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 logical `const`-ness.

Introducing `logical_const`

The class `logical_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 `logical_const` object to ensure propagation of logical `const`-ness.

A logically-`const B` would be written as

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

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

  std::logical_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 `logical_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].

```
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 inadventantly lost compiler checks on `const`-correctness.

When the pimpl object is wrapped in `logical_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 `const_casts`: `const`-correctness is restored. The class above would be modified as follows:

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

  std::logical_const<std::unique_ptr<CImpl>> m_pimpl;
};
```

Thread-safety and `logical_const`

Herb Sutter introduced the appealing notion that `const` implies thread-safe [3]. Without `logical_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 t1(e1);
  std::thread t2(e2);
  t1.join();
  t2.join();
}
```

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

```
struct E
{
  E(D& pD);                  // unchanged

  void operator() () const; //  unchanged

  std::logical_const<D*> m_pD;
};
```

Introduction of `logical_const` cannot automatically guarantee thread-safety but can allow `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 `const` keyword to propagate logical-`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 `const`, for instance, `logical const`, which enforces logical-`const`-ness. 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 logically-`const` behaviour is required. The `logical_const` wrapper can be used much like the `const` keyword and will cause compilation failure wherever logical-`const`-ness is violated. Logical-`const`-ness can be introduced into existing code by decorating pointer-like members of a class with `logical_const`.

The change required to introduce logical-`const`-ness 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 `logical_const` contains no member data besides the wrapped pointer. Inlining of function calls by the compiler will ensure that using `logical_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 logical `const`-ness; if `logical_const<T>` were to inherit from `T`, then it would allow potentially non-logical-`const` member functions of `T` to be called in a `const` context.

Construction and Assignment

A `logical_const<T>` should be constructable and assignable from a `U` or a `logical_const<U>` where `U` is any type that `T` can be constructed or assigned from.

Pointer-like functions

`operator*` and `operator->` are defined to preserve logical `const`-ness. When a `const logical_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 logical-`const`-ness is preserved when using interfaces which require raw C-style pointers

Equality, inequality and comparison

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

`swap`

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

`cast_away_logical_const`

`cast_away_logical_const` is a free-standing function which allows the underlying pointer to be accessed. The use of this function allows logical `const`-ness 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 logical-`const`-ness does not alter the result of hash evaluation.

V. Technical Specifications

The proposed form of `std::logical_const` is given below. Implementation is exposition-only.

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

public:

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

  ~logical_const() = default;

  logical_const(): t{}
  {
  }

  template <typename U>
  logical_const(U&& u) : t{std::forward<U>(u)}
  {
  }

  template <typename U>
  logical_const<T>& operator = (U&& u)
  {
    t = std::forward<U>(u);
    return *this;
  }

  template <typename U>
  logical_const(const logical_const<U>& pu) : t{pu.t} {}

  template <typename U>
  logical_const(logical_const<U>&& pu) : t{std::move(pu.t)} {}


  template <typename U>
  logical_const<T>& operator = (const logical_const<U>& pt)
  {
    t = pt.t;
    return *this;
  }

  template <typename U>
  logical_const<T>& operator = (logical_const<U>&& pt)
  {
    t = std::move(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);
  }

  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 logical_const<T>& pt, const logical_const<U>& pu)
{
  return pt.t == pu.t;
}
```

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

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

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

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

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

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

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

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

template <typename T, typename U>
void swap (logical_const<T>& pt1, logical_const<U>& pt2)
{
  swap(pt1.t, pt2.t);
}

template <typename T>
const T& cast_away_logical_const(const logical_const<T>& pt)
{
  return pt.t;
}

template <typename T>
T& cast_away_logical_const(logical_const<T>& pt)
```

```
{
  return pt.t;
}

template <typename T>
struct hash<logical_const<T>> : std::hash<T>
{
  size_t operator()(const logical_const<T>& p) const
  {
    return operator()(cast_away_logical_const(p));
  }
};
```

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

- [1] Bjarne Stroustrup, The C++ Programming Language, 4th edition, 2013, Addison Wesley ISBN-10: 0321563840 p464
- [2] Martin Reddy, API design for C++, 2011, Elsevier ISBN-10: 0123850037, Section 3.1
- [3] Herb Sutter, C++ and Beyond 2012: Herb Sutter - You don't know [blank] and [blank]