Deducing `this` - P0847

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0847r4.html

presented by Amir Kirsh
in Core C++ Meetup :: 28-Oct-2020

The problem

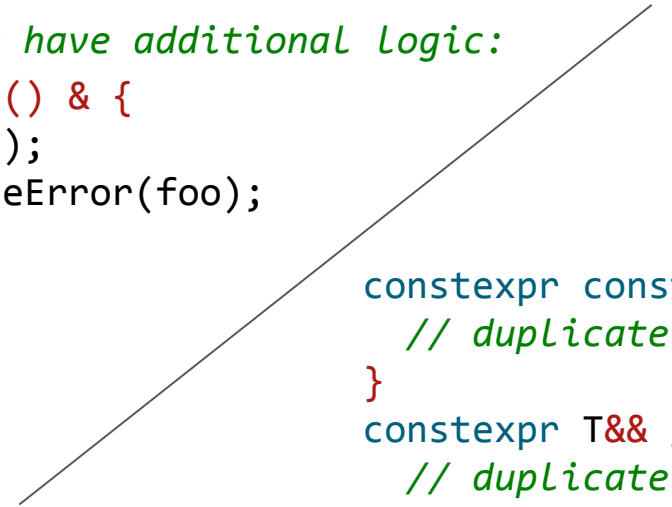
```
template <typename T>
class Holder {
  T value;
public:
  constexpr Holder(T v): value(std::move(v)) {}
  // 2 versions getValue():
  constexpr T& getValue() {
    return value;
  }
  constexpr const T& getValue() const {
    return value;
  }
};
```

The problem - it can get even worse

```
// there might be in fact 4 versions:
constexpr T& getValue() & {
  return value;
}
constexpr const T& getValue() const & {
  return value;
}
constexpr T&& getValue() && {
  return value;
}
constexpr const T&& getValue() const && {
  return value;
}
```

The problem - can become worser

```
// the 4 versions may have additional logic:
constexpr T& getValue() & {
  auto foo = callFoo();
  if(!foo) throw ValueError(foo);
  return value;
}
```



```
constexpr const T& getValue() const & {
  // duplicate code?
}
constexpr T&& getValue() && {
  // duplicate code?
}
constexpr const T&& getValue() const && {
  // duplicate code?
}
```

This is not a theoretical problem...

The problem arises in many implementations…

`std::optional::value()` - has all 4
`string operator[]` and at - has 2 (const and non-const)
etc.
and it appears also in user's code

I personally got to know P0847 when, annoyingly, implementing something like the above and looking for a better approach...

Is this copy-paste so bad?

Well we can think of worse things…
But if we can avoid it, why not?

A programmer has usually nothing against copy-pasting code – unless it is the language that forces the copy paste…

Solutions so far (1) - delegation with const cast

```
// careful delegation with const_cast
constexpr const T& getValue() & {
  auto foo = callFoo();
  if(!foo) throw ValueError(foo);
  return value;
}
```

```
constexpr T& getValue() const & {
  return const_cast<T&>
  (static_cast<const Holder&>(*this).getValue());
}
constexpr T&& getValue() && {
  return const_cast<T&&>
  (static_cast<const Holder&>(*this).getValue());
}
constexpr const T&& getValue() const && {
  return std::move(
    static_cast<const Holder&>().getValue());
}
```

Solutions so far (2) - helper with forwarding ref

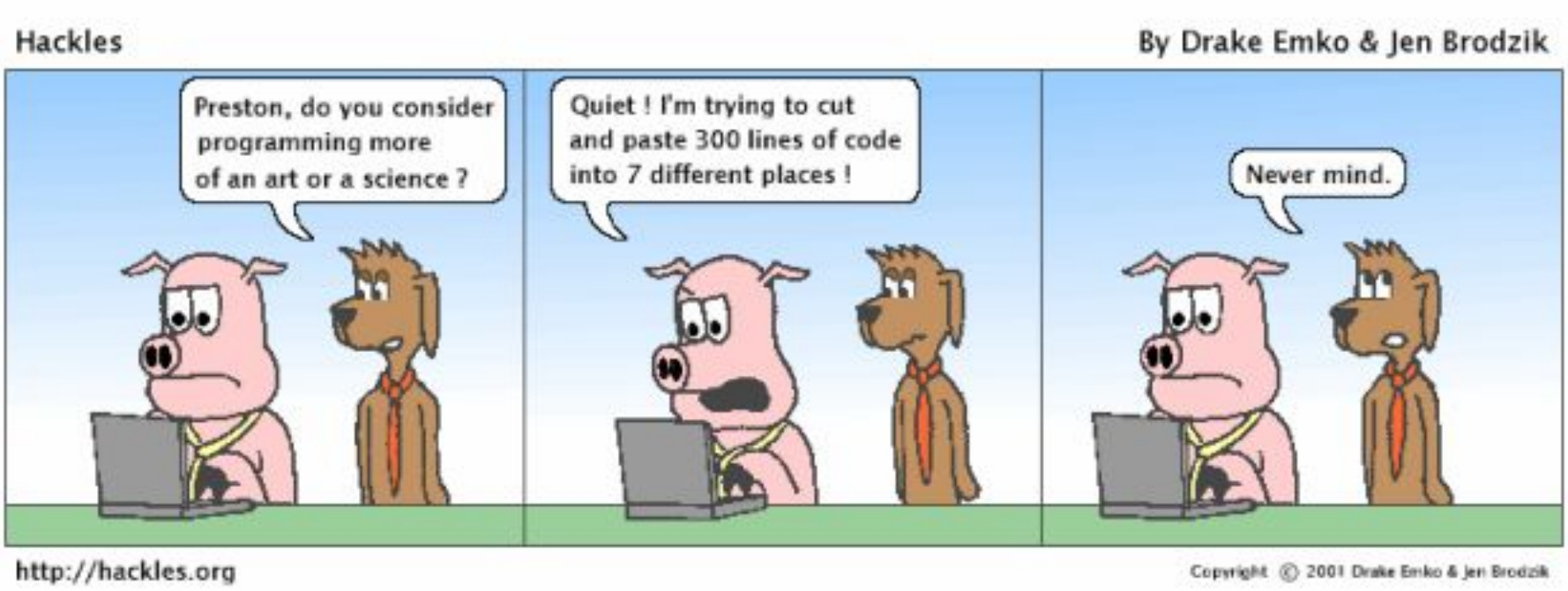
```cpp
// forward all 4 to helper method
template<typename Me>
static decltype(auto)
getValueImpl(Me&& me) {
  auto foo = callFoo();
  if(!foo) throw ValueError(foo);
  return std::forward<Me>(me).value;
}
```

```cpp
constexpr const T& getValue() & {
  return getValueImpl(*this);
}
constexpr T& getValue() const & {
  return getValueImpl(*this);
}
constexpr T&& getValue() && {
  return getValueImpl(std::move(*this));
}
constexpr const T&& getValue() const && {
  return getValueImpl(std::move(*this));
}
```

Solutions so far (3) - helper + a macro

```cpp
// forward all 4 to helper method
template<typename Me>
static decltype(auto)
getValueImpl(Me&& me) {
  auto foo = callFoo();
  if(!foo) throw ValueError(foo);
  return std::forward<Me>(me).value;
}
```

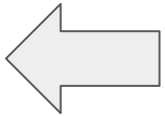
```cpp
// use a macro to create the 4...
CREATE_4_FORWARDERS(getValue, getValueImpl)
```


Hackles
By Drake Emko & Jen Brodzik
Preston, do you consider programming more of an art or a science ?
Quiet ! I'm trying to cut and paste 300 lines of code into 7 different places !
Never mind.
http://hackles.org
Copyright © 2001 Drake Emko & Jen Brodzik

Can we do better?

P0847r4

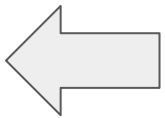
```
// forward to helper method
template<typename Me>
static decltype(auto)
getValueImpl(Me&& me) {
  auto foo = callFoo();
  if(!foo) throw ValueError(foo);
  return std::forward<Me>(me).value;
}
```



why not making this ***a member function*** that would service all 4 cases?

P0847r4

```
// not a helper anymore!!
template<typename Me>
static
auto&& getValue(this Me&& me) {
  auto foo = callFoo();
  if(!foo) throw ValueError(foo);
  return std::forward<Me>(me).value;
}
```



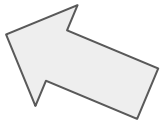
"python style" syntax:
getting *this* as the first parameter!

With a single member implementation!
Amen Hallelujah!!

This is a member function!

P0847r4

```
// not a helper anymore!!
template<typename Me>
static
auto&& getValue(this Me&& me) {
  auto foo = callFoo();
  if(!foo) throw ValueError(foo);
  return std::forward<Me>(me).value;
}
```



Note:

The method is *const*, if *“Me”* is const
It is && if *“Me”* is &&

This is exactly the same as it is with any other forwarding reference!

P0847r4

```
// not a helper anymore!!
template<typename Me>
static
auto&& getValue(this Me&& me) {
  auto foo = callFoo();
  if(!foo) throw ValueError(foo);
  return std::forward<Me>(me).value;
}
```

I called the *this* parameter **Me&& me**, but these are just names you can pick any names you like (e.g. P0847 calls it **Self&& self** which *may become a convention*)

Problem solved! but wait, that's not all...

P0847r4 - passing this by value (1)

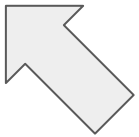
```
struct my_vector : vector<int> {
  auto sorted(this my_vector self) -> my_vector {
    sort(self.begin(), self.end());
    return self;
  }
};
```

the usage here is ***not*** for getting forwarding reference

it is in order to **get *this* by value**

P0847r4 - passing this by value (2)

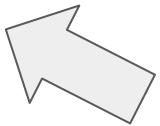
```
// proposed new version for std::less_than::operator()
struct less_than {
  template <typename T, typename U>
  bool operator()(this less_than,
          T const& lhs, U const& rhs) {
    return lhs < rhs;
  }
};
```



empty (stateless) class example
no need to pass a *reference* to this

P0847r4 - passing this by value (3)

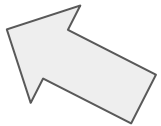
```
// proposed possible new version for std::string_view
template <class charT, class traits = char_traits<charT>>
class basic_string_view {
    const_pointer data;
    size_type size;
public:
    constexpr const_iterator begin(this basic_string_view self) {
        return self.data;
    }

    // ...
```



classes for which pass by value
is cheaper than reference indirection

P0847r4 - recursive lambda

```
// self here is the lambda closure object
auto fib = [](this auto const& self, int n) {
    if (n < 2) return n;
    return self(n-1) + self(n-2);
};
```



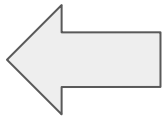
the paper gives another cool example of a recursive lambda

P0847r4 - CRTP, without the C, R, or even T

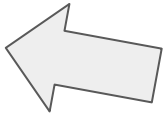
```
// current code - without using P0847 - with “old style CRTP”
template <typename Derived>
struct add_postfix_increment {
  Derived operator++(int) {
    auto& self = static_cast<Derived&>(*this);
    Derived tmp(self);
    ++self; // call the derived prefix increment
    return tmp;
  }
};

struct some_type : add_postfix_increment<some_type> {
    some_type& operator++() { ... }
};
```

P0847r4 - CRTP, without the C, R, or even T

```
// new code - with P0847
struct add_postfix_increment {
  template <typename Self>
  auto operator++(this Self&& self, int) {
    auto tmp = self;
    ++self; // may call derived prefix increment
    return tmp;
  }
};

struct some_type : add_postfix_increment {
    some_type& operator++() { ... }
};
```



according to the proposal, `this Self&& self` can deduce derived type which is a super-strong feature of this syntax



the paper gives additional CRTP without C, R and T examples

Thank you!

```
void conclude(auto greetings) {
    while(still_time() && have_questions()) {
        ask();
    }
    greetings();
}

conclude([]{ std::cout << "Thank you!"; });
```