Call a non-const member function from a const member function

I would like to know if its possible to call a non-const member function from a const member function. In the example below First gives a compiler error. I understand why it gives an error, I would like to know if there is a way to work around it.

```
class Foo
{
   const int& First() const
   {
         return Second();
   }

   int& Second()
   {
        return m_bar;
   }

   int m_bar;
}
```

I don't really want to discuss the wisdom of doing this, I'm curious if its even possible.

c++ const

asked Oct 28 '10 at 21:06
Steve
5,476 6 50 86

3 You are not the first one: stackoverflow.com/questions/856542/… – Till Theis Oct 28 '10 at 21:13

thanks Till, that didn't come up in my search – Steve Oct 28 '10 at 21:26

7 Answers

```
return (const_cast<Foo*>(this))->Second();
```

Then cry, quietly.

answered Oct 28 '10 at 21:08
Adam Wright
36.7k 7 98 134

What if the functions have the same name? const int& GetBar() const; int& GetBar(); – phandinhlan Apr 24 at 22:15

It is *possible*:

```
const int& First() const
{
    return const_cast<Foo*>(this)->Second();
}

int& Second() { return m_bar; }
```

I wouldn't recommend this; it's ugly and dangerous (any use of `const_cast` is dangerous).

It's better to move as much common functionality as you can into helper functions, then have your const and non-const member functions each do as little work as they need to.

In the case of a simple accessor like this, it's just as easy to `return m_bar;` from both of the functions as it is to call one function from the other.

answered Oct 28 '10 at 21:07

James McNellis
223k 35 646 807

Or just turn `int& Second()` into `int& Second() const` . The non- `const` member functions will have no problem calling `Second()` , but now the `const` member functions will be able to call it without any jiggery-pokery, too. – Jeremy W. Sherman Oct 28 '10 at 21:30

@Jeremy: How do you suppose we return a non-const reference to a member, then? – GManNickG Oct 28 '10 at 21:32

@GMan: Touché. You can't without declaring `m_bar` as a `mutable int` . @Fred Larson's recommendation is quite good, but the direction it leads - writing every blessed accessor twice, once `const` -modified and once not - is not so cheery. – Jeremy W. Sherman Oct 28 '10 at 21:39

@Jeremy: In most cases (not all, but most), it isn't necessary to have a non-const accessor (at least that's been my experience). Even if you do need both, since you have to write both functions anyway, writing the `return m_bar;` in both of them isn't usually too much of an additional burden. – James McNellis Oct 28 '10 at 21:43

the issue was inheritance of a zip iterator interface in a const version of the zip iterator, so it isn't as simple as the example suggests – Steve Oct 29 '10 at 1:54

By the definition of `const` , a function should not modify the state of an object. But if it calls another non-const member, the object's state might get changed, so it's disallowed.

I know you said you didn't want to hear about this, but I think it's important for others that happen upon the question.

answered Oct 28 '10 at 21:09

community wiki
Mark Ransom

Overload on `const` :

```
const int& Second() const
{
    return m_bar;
}
```

You can add this method and keep the original non-const version.

answered Oct 28 '10 at 21:27

Fred Larson
33.1k 7 71 106

iterators are similar in this and make an interesting study.

const iterators are often the base for 'non const' iterators, and you will often find `const_cast<>()` or C style casts used to discard const from the base class with accessors in the child.

Edit: Comment was

> I have a zip iterator where the const one inherits from the non-const

This would generally be the wrong inheritence structure (if your saying what I think you are), the reason being that children should not be less restrictive than parents.

say you had some algorithm taking your zip iterator, would it be appropriate to pass a const iterator to a non const ?

if you had a const container, could only ask it for a const iterator, but then the const iterator is derived from an iterator so you just use the features on the parent to have non const access.

Here is a quick outline of suggested inheritence following the traditional stl model

```
class ConstIterator:
    public std::_Bidit< myType, int, const myType *, const mType & >
{
  reference operator*() const { return m_p; }
}

class Iterator : public ConstIterator
```

```
{
  typedef ConstIterator _Mybase;
  // overide the types provided by ConstIterator
  typedef myType * pointer;
  typedef myType & reference;

  reference operator*() const
  {
    return ((reference)**(_Mybase *)this);
  }
}

typedef std::reverse_iterator<ConstIterator> ConstReverseIterator;
typedef std::reverse_iterator<Iterator> ReverseIterator;
```

edited Oct 29 '10 at 22:30

answered Oct 28 '10 at 21:28

Greg Domjan
6,837 4 24 42

I tend to write iterators as template classes and used type traits to manipulate the constness of the member functions; it avoids the need for `const_cast` and makes the code easier to read. – James McNellis Oct 28 '10 at 22:22

Sounds interesting, though does it allow for using iterators where you might use const_iterator? I'm making this original observation on the stl(s) I've seen - which is mainly to ones with MSVC. – Greg Domjan Oct 28 '10 at 22:38

this was actually my use case for this. I have a zip iterator where the const one inherits from the non-const. @James, could you answer this question with the type traits answer? I'm really interested in not doing this if possible – Steve Oct 29 '10 at 1:56

@Steve: I'll try and dig something up; I don't have my PC with all my projects on it at the moment. The general approach is to create a single class template `template <typename T> class iterator_impl` and then in the container that is using it provide a `typedef iterator_impl<const T> const_iterator` and `typedef iterator_impl<T> iterator`. You can then inside of the `iterator_impl` template extract the constness from `T` and set all the return types and pointers and such appropriately. Is that what you're looking for? (I just want to make sure I understand exactly what you need). – James McNellis Oct 29 '10 at 2:45

I found myself trying to call a non-const member function that was inherited, but was actually const because of the API I was using. Finally I settled on a different solution: re-negotiate the API so that the function I inherit is properly const.

It won't always be possible to negotiate changes to others' functions, but doing so when possible seems cleaner and nicer than needing to use const_cast and it benefits other users as well.

answered Dec 27 '12 at 18:02

sage
809 9 23

The restriction of const member methods are came from compile time. If you can fool the compiler, then yes.

```
class CFoo
{
public:
    CFoo() {m_Foo = this;}
    void tee();

    void bar() const
    {
        m_Foo->m_val++;  // fine
        m_Foo->tee();    // fine
    }
private:
   CFoo * m_Foo;
   int    m_Val;

};
```

This actually abolishes the purpose of const member function, so it is better not to do it when design a new class. It is no harm to know that there is a way to do it,especially it can be used as an work-around on these old class that was not well designed on the concept of const member function.

edited Feb 12 '14 at 21:20

answered Feb 12 '14 at 21:10

Jack Wu
51 1 3