# Is it possible to subclass a C struct in C++ and use pointers to the struct in C code?

Asked 16 years, 3 months ago  Modified 8 years, 6 months ago  Viewed 14k times

25

Is there a side effect in doing this:

C code:

```
struct foo {
      int k;
};

int ret_foo(const struct foo* f){
    return f.k;
}
```

C++ code:

```
class bar : public foo {

   int my_bar() {
       return ret_foo( (foo)this );
   }

};
```

There's an `extern "C"` around the C++ code and each code is inside its own compilation unit.

Is this portable across compilers?

c++ c gcc extern-c

Share

Improve this question

Follow

edited May 30, 2016 at 23:38

Jonathan Leffler

752k ● 145 ● 946 ● 1.3k

asked Sep 24, 2008 at 13:56

Edu Felipe

10.4k ● 13 ● 46 ● 41

I'm guessing you mean (foo*)this, or even better: static_cast<foo*>(this) – Richard Corden Sep 24, 2008 at 14:18

I'm curious about the effect of an extern "C" around a class with a method... Still, the reason you want to do this would help us offering info or alternative solutions. – paercebal Sep 24, 2008 at 17:25

When you say "portable across compilers", do you mean compiling the C code with one compiler, and the C++ with another, and linking the two together? – Roddy Sep 24, 2008 at 22:19

## 10 Answers

Sorted by: Highest score (default)

30

This is entirely legal. In C++, classes and structs are identical concepts, with the exception that all struct members are public by default. That's the only difference. So asking whether you can extend a struct is no different than asking if you can extend a class.

There is one caveat here. There is *no guarantee* of layout consistency from compiler to compiler. So if you compile your C code with a different compiler than your C++ code, you may run into problems related to member layout (padding especially). This can even occur when using C and C++ compilers from the same vendor.

I *have* had this happen with gcc and g++. I worked on a project which used several large structs. Unfortunately, g++ packed the structs significantly looser than gcc, which caused significant problems sharing objects between C and C++ code. We eventually had to manually set packing and insert padding to make the C and C++ code treat the structs the same. Note however, that this problem can occur regardless of subclassing. In fact we weren't subclassing the C struct in this case.

Share Improve this answer Follow

answered Sep 24, 2008 at 16:30

Derek Park
46.8k ● 16 ● 59 ● 76

2 Actually, there is another difference between structs and classes: default visibility of base classes is public in structs, but private in classes. – Aconcagua Aug 17, 2016 at 9:58

11

I certainly not recommend using such weird subclassing. It would be better to change your design to use composition instead of inheritance. Just make one member

```
foo* m_pfoo;
```

in the bar class and it will do the same job.

Other thing you can do is to make one more class FooWrapper, containing the structure in itself with the corresponding getter method. Then you can subclass the wrapper. This way the problem with the virtual destructor is gone.

Share

Improve this answer

Follow

edited Sep 24, 2008 at 14:09

answered Sep 24, 2008 at 14:04

m_pGladiator

8,620 ● 8 ● 45 ● 61

this is the best answer. Unlike the accepted answer there a few options pointed out help you avoid problems with vtables and the structure layout. – Alexander Oh Jan 22, 2014 at 10:30

3

> “Never derive from concrete classes.” — Sutter
>
> “Make non-leaf classes abstract.” — Meyers

It’s simply wrong to subclass non-interface classes. You should refactor your libraries.

Technically, you can do what you want, so long as you don’t invoke undefined behavior by, e. g., deleting a pointer to the derived class by a pointer to its base class subobject. You don’t even need `extern "C"` for the C++ code. Yes, it’s portable. But it’s poor design.

Share

Improve this answer

Follow

edited Jun 20, 2020 at 9:12

Community Bot

1 ● 1

answered Sep 24, 2008 at 14:21

Roman Odaisky

2,931 ● 22 ● 26

5 I don't agree at all with such definitive statements by Meyer and Sutter. There is a lot you can do by inheriting from concrete classes, especially if you only add methods in the derived classes, not data members. This way you don't have the problem of slicing. – QBziZ Sep 24, 2008 at 18:36

4 I think you're missing the point. My guess is the OP has existing C modules and wishes to 'wrap' some of the structs into classes, while still offering a C interface for legacy systems. If he's stuck with C for the structs, Sutter and Meyer are no help at all. – Roddy Sep 24, 2008 at 22:16

3

This is perfectly legal, though it might be confusing for other programmers.

You can use inheritance to extend C-structs with methods and constructors.

Sample :

```
struct POINT { int x, y; }
class CPoint : POINT
{
public:
    CPoint( int x_, int y_ ) { x = x_; y = y_; }
```

```
    const CPoint& operator+=( const POINT& op2 )
    { x += op2.x; y += op2.y; return *this; }

    // etc.
};
```

Extending structs might be "more" evil, but is not something you are forbidden to do.

Share Improve this answer Follow

answered Sep 24, 2008 at 14:31

Christopher
**8,982** ● 3 ● 34 ● 38

I'm not so sure about that. I believe (at least in the current version of the standard) that the layout of a class as a base class subobject may be different than if you have an instance of that class. (See note under ISO C++ 10/5) – Richard Corden Sep 24, 2008 at 14:39

1 If the derived class adds no data, and adds no virtual methods, then nothing bad should happen. Now, this is perhaps evil in design, but if the need is to add a default constructor and some helper functions, then... Well... This is not design. It is just correcting a missing feature, nothing more. – paercebal Sep 24, 2008 at 17:27

---

2

Wow, that's evil.

> Is this portable across compilers?

Most definitely not. Consider the following:

```
foo* x = new bar();
delete x;
```

In order for this to work, foo's destructor must be virtual which it clearly isn't. As long as you don't use `new` and as long as the derived objectd don't have custom destructors, though, you could be lucky.

/EDIT: On the other hand, if the code is only used as in the question, inheritance has no advantage over composition. Just follow the advice given by m_pGladiator.

Share

Improve this answer

Follow

edited Sep 24, 2008 at 14:12

answered Sep 24, 2008 at 14:00

Konrad Rudolph
**545k** ● 139 ● 956 ● 1.2k

But that's not the code I intend to write. bar can only be casted to foo for function calls, and all C functions that operate on foo are const. (just as in the example.) – Edu Felipe Sep 24, 2008 at 14:05

Yes, in that case the code is kinda OK. It just begs for misuse. ;-) – Konrad Rudolph Sep 24, 2008 at 14:11

It seems the foo* interface is used by C only. The C code would not be able to delete it anyway: [1] it's not the owner and [2] it's not C++. – MSalters Sep 24, 2008 at 14:52

Should be noted that any C++ class with non-virtual destructor imposes the same problem. That's not an issue of portability, but occurs within one and the same language (C++) and one and the same compiler, too, if the base class is not designed to be inherited from. – Aconcagua Aug 17, 2016 at 10:05

---

2

This is perfectly legal, and you can see it in practice with the MFC CRect and CPoint classes. CPoint derives from POINT (defined in windef.h), and CRect derives from RECT. You are simply decorating an object with member functions. As long as you don't extend the object with more data, you're fine. In fact, if you have a complex C struct that is a pain to default-initialize, extending it with a class that contains a default constructor is an easy way to deal with that issue.

Even if you do this:

```
foo *pFoo = new bar;
delete pFoo;
```

then you're fine, since your constructor and destructor are trivial, and you haven't allocated any extra memory.

You also don't have to wrap your C++ object with 'extern "C"', since you're not actually passing a C++ *type* to the C functions.

Share Improve this answer Follow

answered Sep 24, 2008 at 15:43

moswald

11.7k ● 7 ● 56 ● 78

You are wrong. Deleting a pointer to a derived class by a pointer to a base class is undefined behavior. Period. – Roman Odaisky Sep 24, 2008 at 16:42

---

1

I don't think it is necessarily a problem. The behaviour is well defined, and as long as you are careful with life-time issues (don't mix and match allocations between the C++ and C code) will do what you want. It should be perfectly portable across compilers.

The problem with destructors is real, but applies any time the base class destructor isn't virtual not just for C structs. It is something you need to be aware of but doesn't preclude using this pattern.

Share Improve this answer Follow

answered Sep 24, 2008 at 14:07

Rob Walker

47.4k ● 15 ● 100 ● 137

The behaviour is not well defined - at least in my reading of the std. Do you have a reference to the text that says it's legal? – Richard Corden Sep 24, 2008 at 14:12

No reference ... but I don't fully grok the quote you used below. Wouldn't that prevent any useful down-casting, I don't see how it is specific to the case of the base class being a C struct – Rob Walker Sep 24, 2008 at 14:54

Unfortunately, I am unable to find a normative reference. But it was pointed out to me by an ISO C++ committee member that an object is a POD only if it is a concrete instance of a POD type. ie. there is no such thing as a POD base sub object. I'll add more detail to my answer below. – Richard Corden Sep 24, 2008 at 16:08

1

It will work, and portably BUT you cannot use any virtual functions (which includes destructors).

I would recommend that instead of doing this you have Bar contain a Foo.

```
class Bar
{
private:
   Foo  mFoo;
};
```

Share Improve this answer Follow

answered Sep 24, 2008 at 14:08

Airsource Ltd

32.6k ● 13 ● 74 ● 76

0

I don't get why you don't simply make ret_foo a member method. Your current way makes your code awfully hard to understand. What is so difficult about using a real class in the first place with a member variable and get/set methods?

I know it's possible to subclass structs in C++, but the danger is that others won't be able to understand what you coded because it's so seldom that somebody actually does it. I'd go for a robust and common solution instead.

Share Improve this answer Follow

answered Sep 24, 2008 at 14:10

Thorsten79

10.1k ● 6 ● 40 ● 54

Simply because ret_foo is written in pure C, where I don't have member methods. Also, I don't have the code for ret_foo, it's an API provided for me. – Edu Felipe Sep 24, 2008 at 14:15

Don't subclass stuff you don't have control over. Make it a member variable and handle it there. Wrap it in an object if useful. Composition is much better here than derivation.
– Thorsten79 Sep 24, 2008 at 14:24

0

It probably will work but I do not believe it is guaranteed to. The following is a quote from ISO C++ 10/5:

> A base class subobject might have a layout (3.7) different from the layout of a most derived object of the same type.

It's hard to see how in the "real world" this could actually be the case.

**EDIT:**

The bottom line is that the standard has not limited the number of places where a base class subobject layout can be different from a concrete object with that same Base type. The result is that any assumptions you may have, such as POD-ness etc. are not necessarily true for the base class subobject.

**EDIT:**

An alternative approach, and one whose behaviour is well defined is to make 'foo' a member of 'bar' and to provide a conversion operator where it's necessary.

```
class bar {
public:
   int my_bar() {
       return ret_foo( foo_ );
   }

   //
   // This allows a 'bar' to be used where a 'foo' is expected
   inline operator foo& () {
     return foo_;
   }

private:
  foo foo_;
};
```

Share

Improve this answer

Follow

edited Sep 25, 2008 at 12:06

answered Sep 24, 2008 at 14:20

Richard Corden
21.7k ● 9 ● 61 ● 87

I suspect that this would be due to virtual inheritance. – bk1e Sep 24, 2008 at 14:49

The problem is, I don't believe that the standard actually limits the place that it can happen. – Richard Corden Sep 24, 2008 at 16:13

Richard, that quote applies regardless of whether you are using structs or classes, or some funky mix. All that says is that the compiler can change the underlying layout of the object how it sees fit. This doesn't break anything. – Derek Park Sep 24, 2008 at 16:17

But a POD struct *must* have a very specific layout. If the compiler changes that layout then the C code may not work? Am I missing something? – Richard Corden Sep 25, 2008 at 7:40