Home

PUBLIC

Stack Overflow

Tags

Users

Jobs

TEAMS What's this?

Q&A for Work

Calling virtual functions inside constructors

Ask Question

Asked 10 years, 1 month ago Active 9 months ago Viewed 88k times

217

78

Suppose I have two C++ classes:

```
class A
{
public:
  A() { fn(); }

  virtual void fn() { _n = 1; }
  int getn() { return _n; }

protected:
  int _n;
};

class B : public A
{
public:
  B() : A() {}

  virtual void fn() { _n = 2; }
};
```

If I write the following code:

```
int main()
{
  B b;
  int n = b.getn();
}
```

One might expect that `n` is set to 2.

It turns out that `n` is set to 1. Why?

c++ constructor overriding virtual-functions

share improve this question

edited Jul 9 '18 at 8:07
Cheers and hth. - Alf
126k ● 14 ● 169 ● 278

asked Jun 7 '09 at 15:46
David Coufal
2,738 ● 4 ● 24 ● 29

8 I'm asking and answering my own question because I want to get the explanation for this bit of C++ esoterica into Stack Overflow. A version of this issue has struck our development team twice, so I'm guessing this info might be of use to someone out there. Please write out an answer if you can explain it in a different/better

Blog

Find Help Online Mindfully — and Effectively!

Featured on Meta

Custom Filters release announcement

Unicorn Meta Zoo #6: Interview with Catija

The world is big and I am SO small. What are the implications for our meta…

Creating a system for featuring posts. Tell the mods what you want

We're removing "Hot Meta Posts" from Stack Overflow's sidebar for now;…

at 15:59

1 What surprises me is the lack of a compiler warning. The compiler substitutes a call to the “function defined in the class of the current constructor” for what would in any other case be the “most overridden” function in a derived class. If the compiler said “substituting Base::foo() for call to virtual function foo() in constructor” then the programmer would be warned that the code will not do what they expected. That would be a lot more helpful than making a silent substitution, leading to mysterious behavior, lots of debugging, and eventually a trip to stackoverflow for enlightenment. – Craig Reynolds Jan 31 at 20:02

add a comment

13 Answers

active oldest votes

199

Calling virtual functions from a constructor or destructor is dangerous and should be avoided whenever possible. All C++ implementations should call the version of the function defined at the level of the hierarchy in the current constructor and no further.

The C++ FAQ Lite covers this in section 23.7 in pretty good detail. I suggest reading that (and the rest of the FAQ) for a followup.

Excerpt:

> [...] In a constructor, the virtual call mechanism is disabled because overriding from derived classes hasn’t yet happened. Objects are constructed from the base up, “base before derived”.
>
> [...]
>
> Destruction is done “derived class before base class”, so virtual functions behave as in constructors: Only the local definitions are used – and no calls are made to overriding functions to avoid touching the (now destroyed) derived class part of the object.

EDIT Corrected Most to All (thanks litb)

share improve this answer

edited Oct 16 '18 at 7:04

melpomene
71.7k ● 5 ● 56 ● 100

answered Jun 7 '09 at 15:52

JaredPar
597k ● 126 ● 1100 ● 1363

49 Not most C++ implementations, but all C++ implementations have to call the current class's version. If some don't, then those have a bug :). I still agree with you that it's bad to call a virtual function from a base class - but semantics are precisely defined. – Johannes Schaub - litb Jun 7 '09 at 16:19

12 It's not dangerous, it's just non-virtual. In fact, if methods called from the constructor were called virtually, it would be dangerous because the method could access uninitialized members. – Steven Sudit Jun 18 '14 at 19:33

5 Why is calling virtual functions from destructor dangerous? Isn't the object still complete when destructor runs, and only destroyed after the destructor finishes? – Siyuan Ren Sep 11 '14 at 12:39

9 **−1** "is dangerous", no, it's dangerous in Java, where downcalls can happen; the C++ rules remove the danger through a pretty expensive mechanism. – Cheers and hth. - Alf Aug 14 '16 at 10:47

8 In what way is calling a virtual function from a constructor "dangerous"? This is total nonsense. – Lightness Races in Orbit Sep 5 '16 at 20:49

show **9** more comments

sql

More jobs near Atlanta…

Linked

57 C++ virtual function from constructor

4 Calling pure virtual function

2 C++ interview inheritance puzzle

1 Why is my override method not being called?

7 C++ virtual method overriding

0 calling a protected virtual method in C++

0 virtual function calls in constructor and destructor

3 Is it possible to use the template method pattern in the constructor?

-1 c++: Call derrived function from base constructor?

1 Binding within Constructors within Constructors

see more linked questions…

Related

1341 Calling the base constructor in C#

1237 Virtual member call in a constructor

632 What are the rules for calling the superclass constructor?

2075 How do I call one constructor from another in Java?

847 Can I call a constructor from another constructor (do constructor chaining) in C++?

1371 When to use virtual destructors?

209 Why does an overridden function in the derived class hide other overloads of the base class?

1161 Why do we need virtual functions in C++?

888 Call one constructor from another

80

Calling a polymorphic function from a constructor is a recipe for disaster in most OO languages. Different languages will perform differently when this situation is encountered.

The basic problem is that in all languages the Base type(s) must be constructed previous to the Derived type. Now, the problem is what does it mean to call a polymorphic method from the constructor. What do you expect it to behave like? There are two approaches: call the method at the Base level (C++ style) or call the polymorphic method on an unconstructed object at the bottom of the hierarchy (Java way).

In C++ the Base class will build its version of the virtual method table prior to entering its own construction. At this point a call to the virtual method will end up calling the Base version of the method or producing a *pure virtual method called* in case it has no implementation at that level of the hierarchy. After the Base has been fully constructed, the compiler will start building the Derived class, and it will override the method pointers to point to the implementations in the next level of the hierarchy.

```
class Base {
public:
   Base() { f(); }
   virtual void f() { std::cout << "Base" << std::endl; }
};
class Derived : public Base
{
public:
   Derived() : Base() {}
   virtual void f() { std::cout << "Derived" << std::endl; }
};
int main() {
   Derived d;
}
// outputs: "Base" as the vtable still points to Base::f() when Base::Base() is run
```

In Java, the compiler will build the virtual table equivalent at the very first step of construction, prior to entering the Base constructor or Derived constructor. The implications are different (and to my likings more dangerous). If the base class constructor calls a method that is overriden in the derived class the call will actually be handled at the derived level calling a method on an unconstructed object, yielding unexpected results. All attributes of the derived class that are initialized inside the constructor block are yet uninitialized, including 'final' attributes. Elements that have a default value defined at the class level will have that value.

```
public class Base {
   public Base() { polymorphic(); }
   public void polymorphic() {
      System.out.println( "Base" );
   }
}
public class Derived extends Base
{
   final int x;
   public Derived( int value ) {
      x = value;
      polymorphic();
   }
   public void polymorphic() {
      System.out.println( "Derived: " + x );
   }
```

2133 Why are elementwise additions much faster in separate loops than in a combined loop?

Hot Network Questions

- Assuring luggage isn't lost with short layover
- To find islands of 1 and 0 in matrix
- How should I quote American English speakers in a British English essay?
- Self-deportation of American Citizens from US
- Complaints from (junior) developers against solution architects: how can we show the benefits of our work and improve relationships?
- How well would the Moon protect the Earth from an Asteroid?
- Can a US President, after impeachment and removal, be re-elected or re-appointed?
- Why put copper in between battery contacts and clamps?
- Did Vladimir Lenin have a cat?
- What is the German equivalent of the proverb 水清ければ魚棲まず (if the water is clear, fish won't live there)?
- Why did I lose on time with 3 pawns vs Knight. Shouldn't it be a draw?
- Why force the nose of 737 Max down in the first place?
- Why are we moving in circles with a tandem kayak?
- How can chaotic entities be prevented from spreading beyond their domain?
- Classic vs Modern Experience
- Should I let a company know I've reverse engineered and rebuilt their app?
- Scam? Checks via Email
- Is SecureRandom.ints() secure?
- What is more environmentally friendly? An A320 or a car?
- Do 3/8 (37.5%) of Quadratics Have No x-Intercepts?
- Was Donald Trump at ground zero helping out on 9-11?
- Blank spaces in a font
- 2010 (?) science fiction TV show with new world
- How to season a character?

Question feed

```
   public static void main( String args[] ) {
      Derived d = new Derived( 5 );
   }
}
// outputs: Derived 0
//          Derived 5
// ... so much for final attributes never changing :P
```

As you see, calling a polymorphic (*virtual* in C++ terminology) methods is a common source of errors. In C++, at least you have the guarantee that it will never call a method on a yet unconstructed object...

share improve this answer

edited Jun 7 '09 at 17:02

answered Jun 7 '09 at 16:56

David Rodríguez - dribeas

178k ● 16 ● 241 ● 441

1 Good job explaining why the alternative is (also) are error-prone. – DS. Sep 21 '12 at 15:26

"If the base class constructor calls a method that is overriden in the derived class the call will actually be handled at the derived level calling a method on an unconstructed object..." How so if base is already initialized. There is no possiblity unless you explicilty call "init" before initializing other members. – Arek Bal Oct 1 '13 at 19:27

2 An explanation! +1, superior answer imho – underscore_d Jul 3 '16 at 0:30

For me the problem is that there are so many restrictions in C++ classes that its incredible hard to achieve any good design. C++ dictates that "If it could be dangerous forbid it" even if its intuitive causing problems such as: "Why this intuitive behavior doesn't work" to happen all the time. – VinGarcia Sep 22 '16 at 17:38

@VinGarcia What? C++ does not "forbid" anything in this case. The call is simply treated as a non-virtual call, to the method for the class whose constructor is currently executing. That is a logical consequence of the object construction timeline - not some draconian decision to stop you doing silly things. The fact that it coincidentally fulfills the latter purpose too is just a bonus to me. – underscore_d May 21 '17 at 13:15

show **3** more comments

54

The reason is that C++ objects are constructed like onions, from the inside out. Super-classes are constructed before derived classes. So, before a B can be made, an A must be made. When A's constructor is called, it's not a B yet, so the virtual function table still has the entry for A's copy of fn().

share improve this answer

answered Jun 7 '09 at 15:46

David Coufal

2,738 ● 4 ● 24 ● 29

16 C++ does not normally use the term "super class" - it prefers "base class". – anon Jun 7 '09 at 15:48

That is the same in most OO languages: you cannot possibly build a derived object without the base part being already constructed. – David Rodríguez - dribeas Jun 7 '09 at 16:12

2 @DavidRodríguez-dribeas other languages do actually do that. For example in Pascal, memory is allocated for the whole object first, but then only the most-derived constructor is invoked. A constructor must either contain an explicit call to its parent's constructor (which does not have to be the first action - it just has to be somewhere), or if it doesn't , it's as if the first line of the constructor made that call. – M.M Dec 15 '15 at 12:52

Thanks for the clarity and avoidance of details which doesnt go straight to the outcome – user5193682 Nov 1 '16 at 19:14

add a comment

The C++ FAQ Lite Covers this pretty well:

23

> Essentially, during the call to the base classes constructor, the object is not yet of the derived type and thus the base type's implementation of the virtual function is called and not the derived type's.

share improve this answer

edited Jul 9 '18 at 8:13

Cheers and hth. - Alf
126k ● 14 ● 169 ● 278

answered Jun 7 '09 at 16:03

Aaron Maenpaa
83k ● 9 ● 85 ● 105

2 Clear, straightforward, simplest answer. It's still a feature I would love to see get some love. I hate having to write all these silly initializeObject() functions that the user is forced to call right after the construction, just bad form for a very common use case. I understand the difficulty though. C'est la vie. – moodboom Dec 10 '16 at 17:23

1 @moodboom What "love" do you propose? Bear in mind that you can't just change how things currently work in-place, because that would horribly break reams of existing code. So, how would you do it instead? Not only what new syntax you would introduce to allow (actual, non-devirtualised) virtual calls in constructors - but also how you would somehow amend the models of object construction/lifetime so that those calls would have a complete object of the derived type on which to run. This'll be interesting. – underscore_d May 21 '17 at 13:17

@underscore_d I don't think any syntax changes would be needed. Maybe when creating an object, the compiler would add code to walk the vtable and look for this case and patch things then? I've never written a C++ compiler and I'm quite sure my initial comment to give this some "love" was naive and this will never happen. :-) A virtual initialize() function isn't a very painful workaround anyway, you just have to remember to call it after creating your object. – moodboom May 21 '17 at 16:07

@underscore_d I just noticed your other comment below, explaining that the vtable isn't available in constructor, emphasizing again the difficulty here. – moodboom May 21 '17 at 16:12

1 @moodboom I goofed when writing about the vtable not being available in the constructor. It **is** available, but the constructor only sees the vtable for its own class, because each derived constructor updates the instance's vptr to point at the vtable for the current derived type and no further. So, the current ctor sees a vtable that only has its own overrides, hence why it can't call any more-derived implementations of any virtual functions. – underscore_d May 21 '17 at 16:19

show **3** more comments

13

One solution to your problem is using factory methods to create your object.

- Define a common base class for your class hierarchy containing a virtual method afterConstruction():

```
class Object
{
public:
  virtual void afterConstruction() {}
  // ...
};
```

- Define a factory method:

```
template< class C >
C* factoryNew()
```

```
{
  C* pObject = new C();
  pObject->afterConstruction();

  return pObject;
}
```

- Use it like this:

```
class MyClass : public Object
{
public:
  virtual void afterConstruction()
  {
    // do something.
  }
  // ...
};

MyClass* pMyObject = factoryNew();
```

share improve this answer

edited Aug 15 '12 at 5:21

davidA
5,191 ●6 ●44 ●61

answered Jun 7 '09 at 17:09

Tobias
4,472 ●2 ●29 ●58

@meowsqueak Thanks for the formatting improvement! – Tobias Aug 16 '12 at 9:44

add a comment

1

Do you know the crash error from Windows explorer?! **"Pure virtual function call ..."**
Same problem ...

```
class AbstractClass
{
public:
    AbstractClass( ){
        //if you call pureVitualFunction I will crash...
    }
    virtual void pureVitualFunction() = 0;
};
```

Because there is no implemetation for the function pureVitualFunction() and the function is called in the constructor the program will crash.

share improve this answer

answered Jun 7 '09 at 16:51

TimW
7,476 ●1 ●24 ●29

It's hard to see how this is the same problem, as you didn't explain why. Calls to non-pure virtual functions during ctors are perfectly legal, but they just don't go through the (not yet constructed) virtual table, so the version of the method that gets executed is the one defined for the class type whose ctor we are in. So those don't crash. This one does because it's pure virtual and unimplemented (side note: one *can* implement pure

virtual functions in the base), so there is no version of the method to be called for this class type, & the compiler assumes you don't write bad code, so boom – underscore_d May 21 '17 at 13:23

D'oh. The calls do go through the vtable, but it hasn't yet been updated to point at the overrides for the most-derived class: only the one being constructed right now. Still, the result and reason for the crash remains the same. – underscore_d May 21 '17 at 16:15

add a comment

1

The vtables are created by the compiler. A class object has a pointer to its vtable. When it starts life, that vtable pointer points to the vtable of the base class. At the end of the constructor code, the compiler generates code to re-point the vtable pointer to the actual vtable for the class. This ensures that constructor code that calls virtual functions calls the base class implementations of those functions, not the override in the class.

share improve this answer

answered May 7 '13 at 10:14

Yogesh

403 ● 2 ● 16

The vptr isn't changed at the end of the ctor. In the body of ctor `C::C`, virtual function calls go the `C` overrider, not to any base class version. – curiousguy Jul 11 '16 at 13:32

@curiousguy can you explain more clearly. – Yogesh Aug 9 '16 at 7:08

The dynamic type of the object is defined after the ctor has called base class ctors and before it constructs its members. So the vptr isn't changed at the end of the ctor. – curiousguy Aug 9 '16 at 15:51

@curiousguy I am saying the same thing, that vptr is not changed at the end of constructor of base class, it will be changed at the end of constructor of derived class. I hope you are telling the same. Its an compiler/implementation dependent thing. When are you proposing that vptr should change. Any good reason for downvoting? – Yogesh Dec 4 '17 at 6:16

The timing of the change of vptr is not implementation dependent. It is prescribed by language semantics: the vptr changes when the dynamic behavior of the class instance changes. There is no freedom here. Inside the body of a ctor `T::T(params)`, the dynamic type is `T`. The vptr will reflect that: it will point to vtable for T. Do you disagree? – curiousguy Dec 4 '17 at 17:28

show **1** more comment

1

The C++ Standard (ISO/IEC 14882-2014) say's:

> Member functions, including virtual functions (10.3), can be called during construction or destruction (12.6.2). When a virtual function is called directly or indirectly from a constructor or from a destructor, including during the construction or destruction of the class's non-static data members, and the object to which the call applies is the object (call it x) under construction or destruction, the function called is the final overrider in the constructor's or destructor's class and not one overriding it in a more-derived class. If the virtual function call uses an explicit class member access (5.2.5) and the object expression refers to the complete object of x or one of that object's base class subobjects but not x or one of its base class subobjects, the behavior is **undefined**.

So, Don't invoke `virtual` functions from constructors or destructors that attempts to call into the object under construction or destruction, Because the order of construction starts from **base to derived** and the order of destructors starts from **derived to base class**.

So, attempting to call a derived class function from a base class under construction is dangerous.Similarly, an object is destroyed in reverse order from construction, so attempting to call a function in a more derived class from a destructor may access resources that have already been released.

share improve this answer

answered Aug 23 '17 at 7:03

msc

21.4k ● 7 ● 74 ● 148

add a comment

1

Other answers have already explained why `virtual` function calls don't work as expected when called from a constructor. I'd like to instead propose another possible work around for getting polymorphic-like behavior from a base type's constructor.

By adding a template constructor to the base type such that the template argument is always deduced to be the derived type it's possible to be aware of the derived type's concrete type. From there, you can call `static` member functions for that derived type.

This solution does not allow non-`static` member functions to be called. While execution is in the base type's constructor, the derived type's constructor hasn't even had time to go through it's member initialization list. The derived type portion of the instance being created hasn't begun being initialized it. And since non-`static` member functions almost certainly interact with data members it would be unusual to *want* to call the derived type's non-`static` member functions from the base type's constructor.

Here is a sample implementation :

```
#include <iostream>
#include <string>

struct Base {
protected:
    template<class T>
    explicit Base(const T*) : class_name(T::Name())
    {
        std::cout << class_name << " created\n";
    }

public:
    Base() : class_name(Name())
    {
        std::cout << class_name << " created\n";
    }


    virtual ~Base() {
        std::cout << class_name << " destroyed\n";
    }

    static std::string Name() {
        return "Base";
    }

private:
    std::string class_name;
};
```

```
struct Derived : public Base
{
    Derived() : Base(this) {} // `this` is used to allow Base::Base<T> to deduce T

    static std::string Name() {
        return "Derived";
```

This example should print

```
Derived created
Derived destroyed
Base created
Base destroyed
```

When a `Derived` is constructed, the `Base` constructor's behavior depends on the actual dynamic type of the object being constructed.

share improve this answer

answered Oct 19 '18 at 17:00

François Andrieux

17.7k ● 3 ● 30 ● 51

add a comment

0

Firstly,Object is created and then we assign it 's address to pointers.Constructors are called at the time of object creation and used to initializ the value of data members. Pointer to object comes into scenario after object creation. Thats why, C++ do not allows us to make constructors as virtual . .another reason is that, There is nothing like pointer to constructor , which can point to virtual constructor,because one of the property of virtual function is that it can be used by pointers only.

2. Virtual functions are used to assign value dynamically,as constructors are static,so we can not make them virtual.

share improve this answer

answered Nov 10 '17 at 0:48

Priya

1 ● 1

add a comment

0

As has been pointed out, the objects are created base-down upon construction. When the base object is being constructed, the derived object does not exist yet, so a virtual function override cannot work.

However, this can be solved with polymorphic getters that use **static polymorphism** instead of virtual functions if your getters return constants, or otherwise can be expressed in a static member function, This example uses CRTP (https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern).

```
template<typename DerivedClass>
class Base
{
public:
    inline Base() :
    foo(DerivedClass::getFoo())
    {}
```

```
    inline int fooSq() {
        return foo * foo;
    }

    const int foo;
};

class A : public Base<A>
{
public:
    inline static int getFoo() { return 1; }
};

class B : public Base<B>
{
public:
    inline static int getFoo() { return 2; }
};

class C : public Base<C>
{
public:
    inline static int getFoo() { return 3; }
};

int main()
{
    A a;
```

With the use of static polymorphism, the base class knows which class' getter to call as the information is provided at compile-time.

share improve this answer

answered Apr 11 '18 at 3:30

stands2reason

396 ● 3 ● 16

1 I think I will avoid to do this. This is not single Base class any more. You actually created lots of different Base class. – Wang Jul 31 '18 at 21:12

@Wang Exactly: `Base<T>` is just a helper class, not a common interface type that can be used for runtime polymorphism (f.ex. heterogeneous containers). These are useful too, just not for the same tasks. Some classes inherit both from a base class that's an interface type for runtime polymorphism and another that's a compile time template helper. – curiousguy Nov 7 '18 at 23:53

add a comment

-3

I am not seeing the importance of the virtual key word here. b is a static-typed variable, and its type is determined by compiler at compile time. The function calls would not reference the vtable. When b is constructed, its parent class's constructor is called, which is why the value of _n is set to 1.

share improve this answer

answered Oct 28 '13 at 6:58

user2305329

3 ● 2

1 The question is why `b` 's constructor calls the base `f()`, not the derived override of it. Type of the variable `b` is irrelevant to that. – underscore_d Jul 3 '16 at 0:33

"The function calls would not reference the vtable" That is not true. If you think virtual dispatch is only enabled when accessing through a `B*` or ` B&`, you are mistaken. – Lightness Races in Orbit Sep 5 '16 at 20:51

Aside from the fact that it follows its own logic to the wrong conclusion... The idea behind this answer, known static type, is misapplied. A compiler could devirtualise `b.getN()` because it knows the real type, & just directly dispatch to the version from `B`. But that's just an allowance made by the *as-if* rule. Everything still must act *as-if* the virtual table is used & followed to the letter. In the `A` constructor, the same is true: even if (probably not possible) it gets inlined w/ the `B` ctor, the virtual call must still act *as-if* it only has the base `A` vtable available to use. – underscore_d May 21 '17 at 16:31

@LightnessRacesinOrbit Can you give me an example for your assertion that virtual dispatch happens without calling through a reference or pointer (including the implicit `this`)? – Peter A. Schneider Aug 23 '18 at 11:59

@user2305329 You are right that the call `b.getn()` is non-virtual. `b` is a statically typed object, and whatever `getn()` is defined for its type will be called. But *inside member functions*, including the constructor, all member function calls are made through the implicit `this` pointer and are hence virtual function calls, if it is a polymorphic class. The reason and rationale for resolving the virtual `fn()` call to the base class's implementation -- even though it happens during the overall construction of a derived object -- is explained in the other answers. – Peter A. Schneider Aug 23 '18 at 12:07

show **1** more comment

-4

During the object's constructor call the virtual function pointer table is not completely built. Doing this will usually not give you the behavior you expect. Calling a virtual function in this situation may work but is not guaranteed and should be avoided to be portable and follow the C++ standard.

share improve this answer

answered Jun 7 '09 at 17:00

terry

76 ● 2

5 "*Calling a virtual function in this situation may work but is not guaranteed*" That is *not* correct. The behaviour **is** guaranteed. – curiousguy Dec 24 '11 at 2:16

1 @curiousguy ...guaranteed to call the base version if available, or to invoke UB if the vfunc is pure virtual. – underscore_d Jul 3 '16 at 0:38

add a comment

Your Answer

Sign up or log in

Sign up using Google

Sign up using Facebook

Sign up using Email and Password

Post as a guest

Name

Email

Required, but never shown

Post Your Answer

By clicking "Post Your Answer", you agree to our terms of service, privacy policy and cookie policy

Not the answer you're looking for? Browse other questions tagged c++ constructor overriding virtual-functions or ask your own question.