Are inline virtual functions really a non-sense?

Asked 12 years, 5 months ago Active 1 year, 3 months ago Viewed 86k times

183

64

I got this question when I received a code review comment saying virtual functions need not be inline.

I thought inline virtual functions could come in handy in scenarios where functions are called on objects directly. But the counter-argument came to my mind is -- why would one want to define virtual and then use objects to call methods?

Is it best not to use inline virtual functions, since they're almost never expanded anyway?

Code snippet I used for analysis:

```
class Temp
{
public:

    virtual ~Temp()
    {
    }
    virtual void myVirtualFunction() const
    {
        cout<<"Temp::myVirtualFunction"<<endl;
    }

};

class TempDerived : public Temp
{
public:

    void myVirtualFunction() const
    {
        cout<<"TempDerived::myVirtualFunction"<<endl;
    }

};

int main(void)
{
    TempDerived aDerivedObj;
    //Compiler thinks it's safe to expand the virtual functions
    aDerivedObj.myVirtualFunction();

    //type of object Temp points to is always known;
    //does compiler still expand virtual functions?
    //I doubt compiler would be this much intelligent!
    Temp* pTemp = &aDerivedObj;
    pTemp->myVirtualFunction();
```

```
    return 0;
}
```

c++ inline virtual-functions

Share Edit Follow

edited Jul 8 '13 at 17:43
Kara
5,764 ● 15 ● 49 ● 55

asked Apr 9 '09 at 11:03
aJ.
32.6k ● 21 ● 79 ● 126

1 Consider compiling an example with whatever switches you need to get an assembler listing, and then showing the code reviewer that, indeed, the compiler can inline virtual functions. – Thomas L Holaday Apr 9 '09 at 13:08

1 The above usually will not be inlined, because you are calling virtual function in aid of base class. Although it depends only on how smart the compiler is. If it would be able to point out that `pTemp->myVirtualFunction()` could be resolved as non-virtual call, it might have inline that call. This referenced call is inlined by g++ 3.4.2: `TempDerived & pTemp = aDerivedObj; pTemp.myVirtualFunction();` Your code is not. – doc Jul 1 '10 at 22:31

1 One thing gcc actually does is compare the vtable entry to a specific symbol and then use an inlined variant in a loop if it matches. This is especially useful if the inlined function is empty and the loop can be eliminated in this case. – Simon Richter Nov 21 '17 at 13:09

1 @doc Modern compiler try hard to determine at compile time the possible values of pointers. Just using a pointer isn't sufficient to prevent inlining at any significant optimization level; GCC even performs simplifications at optimization zero! – curiousguy Jun 13 '18 at 22:24

13 Answers

Active Oldest Votes

158

Virtual functions can be inlined sometimes. An excerpt from the excellent C++ faq:

> "The only time an inline virtual call can be inlined is when the compiler knows the "exact class" of the object which is the target of the virtual function call. This can happen only when the compiler has an actual object rather than a pointer or reference to an object. I.e., either with a local object, a global/static object, or a fully contained object inside a composite."

Share Edit Follow

edited Apr 2 '16 at 16:48
Wumpf
13 ● 7

answered Apr 9 '09 at 11:23
ya23
13.7k ● 8 ● 42 ● 42

8 True, but it's worth remembering that the compiler is free to ignore the inline specifier even if the call can be resolved at compile time and can be inlined. – sharptooth Apr 9 '09 at 11:39

6 Amother situation when I think inlining can happen is when you'd call the method for example

as this->Temp::myVirtualFunction() - such invokation skips the virtual table resolution and the function should be inlined without problems - why and if you'd want to do it is another topic :) – RnR Apr 9 '09 at 12:12

5 @RnR. It's not necessary to have 'this->', just using the qualified name is enough. And this behaviour takes place for destructors, constructors and in general for assignment operators (see my answer). – Richard Corden Apr 9 '09 at 13:10

2 sharptooth - true, but AFAIK this is true of all inline functions, not just virtual inline functions. – Colen Apr 9 '09 at 17:15

2 void f(const Base& lhs, const Base& rhs) { } ------In implementation of the function, you never know what lhs and rhs points to until runtime. – baye Aug 11 '10 at 23:44

77

C++11 has added `final`. This changes the accepted answer: it's no longer necessary to know the exact class of the object, it's sufficient to know the object has at least the class type in which the function was declared final:

```
class A {
  virtual void foo();
};
class B : public A {
  inline virtual void foo() final { }
};
class C : public B
{
};

void bar(B const& b) {
  A const& a = b; // Allowed, every B is an A.
  a.foo(); // Call to B::foo() can be inlined, even if b is actually a class C.
}
```

Share Edit Follow

answered Nov 13 '13 at 12:45

MSalters

162k ● 8 ● 144 ● 325

Wasn't able to inline it in VS 2017. – Yola Sep 13 '17 at 17:44

1 I don't think it works this way. Invocation of foo() through a pointer/reference of type A can never be inlined. Calling b.foo() should allow inlining. Unless you are suggesting that the compiler already knows this is a type B because it's aware of the previous line. But that's not the typical usage. – Jeffrey Faust Feb 10 '18 at 20:50

For example, compare the generated code for bar and bas here: godbolt.org/g/xy3rNh – Jeffrey Faust Feb 10 '18 at 21:01

1 @JeffreyFaust There's no reason that information shouldn't be propagated, is there? And `icc` seems to do it, according to that link. – Alexey Romanov Apr 11 '18 at 12:16

@AlexeyRomanov Compilers have freedom to optimize beyond the standard, and the certainly do! For simple cases like above, the compiler could know the type and do this optimization. Things are rarely this simple, and it's not typical to be able to determine the

actual type of a polymorphic variable at compile time. I think OP cares about 'in general' and not for these special cases. – Jeffrey Faust Apr 29 '18 at 17:09

38

There is one category of virtual functions where it still makes sense to have them inline. Consider the following case:

```
class Base {
public:
  inline virtual ~Base () { }
};

class Derived1 : public Base {
  inline virtual ~Derived1 () { } // Implicitly calls Base::~Base ();
};

class Derived2 : public Derived1 {
  inline virtual ~Derived2 () { } // Implicitly calls Derived1::~Derived1 ();
};

void foo (Base * base) {
  delete base;             // Virtual call
}
```

The call to delete 'base', will perform a virtual call to call correct derived class destructor, this call is not inlined. However because each destructor calls it's parent destructor (which in these cases are empty), the compiler can inline *those* calls, since they do not call the base class functions virtually.

The same principle exists for base class constructors or for any set of functions where the derived implementation also calls the base classes implementation.

Share Edit Follow

edited Apr 9 '09 at 12:58

answered Apr 9 '09 at 12:53

Richard Corden

20.7k ● 7 ● 57 ● 83

24 One should be aware though that empty braces don't always mean the destructor does nothing. Destructors default-destruct every member object in the class, so if you have a few vectors in the base class that could be quite a lot of work in those empty braces! – Philip Feb 17 '12 at 17:19

14

I've seen compilers that don't emit any v-table if no non-inline function at all exists (and defined in one implementation file instead of a header then). They would throw errors like `missing vtable-for-class-A` or something similar, and you would be confused as hell, as i was.

Indeed, that's not conformant with the Standard, but it happens so consider putting at least one virtual function not in the header (if only the virtual destructor), so that the compiler could emit a vtable for the class at that place. I know it happens with some versions of `gcc`.

As someone mentioned, inline virtual functions can be a benefit *sometimes*, but of course most often you will use it when you do *not* know the dynamic type of the object, because that was the whole reason for `virtual` in the first place.

The compiler however can't completely ignore `inline`. It has other semantics apart from speeding up a function-call. The *implicit inline* for in-class definitions is the mechanism which allows you to put the definition into the header: Only `inline` functions can be defined multiple times throughout the whole program without a violation any rules. In the end, it behaves as you would have defined it only once in the whole program, even though you included the header multiple times into different files linked together.

Share Edit Follow

answered Apr 9 '09 at 14:44

Johannes Schaub - litb
472k ● 118 ● 860 ● 1178

12

Well, actually **virtual functions can always be inlined**, as long they're statically linked together: suppose we have an abstract class `Base` with a virtual function `F` and derived classes `Derived1` and `Derived2`:

```
class Base {
  virtual void F() = 0;
};

class Derived1 : public Base {
  virtual void F();
};

class Derived2 : public Base {
  virtual void F();
};
```

An hypotetical call `b->F();` (with `b` of type `Base*`) is obviously virtual. But you (or the compiler...) could rewrite it like so (suppose `typeof` is a `typeid`-like function that returns a value that can be used in a `switch`)

```
switch (typeof(b)) {
  case Derived1: b->Derived1::F(); break; // static, inlineable call
  case Derived2: b->Derived2::F(); break; // static, inlineable call
  case Base:     assert(!"pure virtual function call!");
  default:       b->F(); break; // virtual call (dyn-loaded code)
}
```

while we still need RTTI for the `typeof`, the call can effectively be inlined by, basically, embedding the vtable inside the instruction stream and specializing the call for all the involved classes. This could be also generalized by specializing only a few classes (say, just `Derived1`):

```
switch (typeof(b)) {
  case Derived1: b->Derived1::F(); break; // hot path
  default:       b->F(); break; // default virtual call, cold path
}
```

Share Edit Follow

edited Jul 26 '17 at 7:07

answered Nov 18 '11 at 20:46

CAFxX

20.7k ● 4 ● 33 ● 62

1 Are they any compilers that do this? Or is this just speculation? Sorry if I'm overly skeptical, but your tone in the description above sounds sort of like -- "they totally could do this!", which is different from "some compilers do this". – Alex Meiburg Aug 19 '19 at 21:47

Yes, Graal does polymorphic inlining (also for LLVM bitcode via Sulong) – CAFxX Aug 20 '19 at 22:47

4

Marking a virtual method inline, helps in further optimizing virtual functions in following two cases:

- Curiously recurring template pattern (http://www.codeproject.com/Tips/537606/Cplusplus-Prefer-Curiously-Recurring-Template-Patt)
- Replacing virtual methods with templates (http://www.di.unipi.it/~nids/docs/templates_vs_inheritance.html)

Share Edit Follow

answered Sep 15 '13 at 4:41

Chenna Reddy

1,763 ● 15 ● 16

3

Inlined declared Virtual functions are inlined when called through objects and ignored when called via pointer or references.

Share Edit Follow

answered Aug 1 '11 at 2:31

tarachandverma

31 ● 1

inline really doesn't do anything - it's a hint. The compiler might ignore it or it might

3

inline a call event without *inline* if it sees the implementation and likes this idea. If code clarity is at stake the *inline* should be removed.

Share Edit Follow

answered Apr 9 '09 at 11:07

sharptooth

161k ● 88 ● 485 ● 923

3 For compilers that operate on single TUs only, they can only inline implicitly functions that they have the definition for. A function can only be defined in multiple TUs if you make it inline. 'inline' is more than a hint and it can have a dramatic performance improvement for a g++/makefile build. – Richard Corden Apr 9 '09 at 13:36

1

In the cases where the function call is unambiguous and the function a suitable candidate for inlining, the compiler is smart enough to inline the code anyway.

The rest of the time "inline virtual" is a nonsense, and indeed some compilers won't compile that code.

Share Edit Follow

edited Apr 9 '09 at 14:35

answered Apr 9 '09 at 11:09

moonshadow

78.2k ● 7 ● 80 ● 117

Which version of g++ won't compile inline virtuals? – Thomas L Holaday Apr 9 '09 at 13:04

Hm. The 4.1.1 I have here now appears to be happy. I first encountered problems with this codebase using a 4.0.x. Guess my info is out of date, edited. – moonshadow Apr 9 '09 at 14:35

1

A compiler can only inline a function when the call can be resolved unambiguously at compile time.

Virtual functions, however are resolved at runtime, and so the compiler cannot inline the call, since at compile type the dynamic type (and therefore the function implementation to be called) cannot be determined.

Share Edit Follow

edited Apr 9 '09 at 11:13

Jonathan Leffler

677k ● 127 ● 827 ● 1198

answered Apr 9 '09 at 11:06

PaulJWilliams

18.3k ● 3 ● 48 ● 78

1 When you call a base class method from the same or derived class the call is unambiguous and non-virtual – sharptooth Apr 9 '09 at 11:09

1 @sharptooth: but then it would be a non-virtual inline method. The compiler can inline functions you don't ask it to, and it probably knows better when to inline or not. Let it decide. – David Rodríguez - dribeas Apr 9 '09 at 11:13

1 @dribeas: Yes, that's exactly what I'm talking about. I only objected to the statement that virtual finctions are resolved at runtime - this is true only when the call is done virtually, not for the exact class. – sharptooth Apr 9 '09 at 11:34

I believe that's nonsense. Any function can *always* be inlined, no matter how big it is or whether it's virtual or not. It depends on how the compiler was written. If you do not agree, then I expect that your compiler cannot produce non-inlined code either. That is: The compiler can include code that at runtime tests for the conditions it could not resolve at compile-time. It's just like the modern compilers can resolve constant values/reduce nummeric expressions at compile-time. If a function/method is not inlined, it does not mean it cannot be inlined. – user1985657 Dec 15 '13 at 13:26

1

With modern compilers, it won't do any harm to inlibe them. Some ancient compiler/linker combos might have created multiple vtables, but I don't believe that is an issue anymore.

Share Edit Follow

answered Apr 9 '09 at 11:08

anon

0

Actually in some cases adding "inline" to a virtual final override can make your code not compile so there is sometimes a difference (at least under VS2017s compiler)!

Actually I was doing a virtual inline final override function in VS2017 adding c++17 standard to compile and link and for some reason it failed when I am using two projects.

I had a test project and an implementation DLL that I am unit testing. In the test project I am having a "linker_includes.cpp" file that #include the *.cpp files from the other project that are needed. I know... I know I can set up msbuild to use the object files from the DLL, but please bear in mind that it is a microsoft specific solution while including the cpp files is unrelated to build-system and much more easier to version a cpp file than xml files and project settings and such...

What was interesting is that I was constantly getting linker error from the test project. Even if I added the definition of the missing functions by copy paste and not through include! So weird. The other project have built and there are no connection between the two other than marking a project reference so there is a build order to ensure both is always built...

I think it is some kind of bug in the compiler. I have no idea if it exists in the compiler shipped with VS2020, because I am using an older version because some SDK only works with that properly :-(

I just wanted to add that not only marking them as inline can mean something, but might even make your code not build in some rare circumstances! This is weird, yet good to know.

PS.: The code I am working on is computer graphics related so I prefer inlining and that is why I used both final and inline. I kept the final specifier to hope the release build is smart enough to build the DLL by inlining it even without me directly hinting so...

PS (Linux).: I expect the same does not happen in gcc or clang as I routinely used to do these kind of things. I am not sure where this issue comes from... I prefer doing c++ on Linux or at least with some gcc, but sometimes project is different in needs.

Share Edit Follow

answered Jun 29 '20 at 18:35

prenex

24 ● 1

0

It does make sense to make virtual functions and then call them on objects rather than references or pointers. Scott Meyer recommends, in his book "effective c++", to never redefine an inherited non-virtual function. That makes sense, because when you make a class with a non-virtual function and redefine the function in a derived class, you may be sure to use it correctly yourself, but you can't be sure others will use it correctly. Also, you may at a later date use it incorrectly yoruself. So, if you make a function in a base class and you want it to be redifinable, you should make it virtual. If it makes sense to make virtual functions and call them on objects, it also makes sense to inline them.

Share Edit Follow

edited Apr 25 '12 at 22:13

answered Apr 25 '12 at 22:03

Balthazar

245 ● 1 ● 11