Constants and compiler optimization in C++ [duplicate]

Asked 16 years, 2 months ago Modified 11 months ago

Viewed 23k times

52

This question already has answers here:

What kind of optimization does const offer in C/C++? (6 answers)

Closed 12 months ago.

I've read all the advice on const-correctness in C++ and that it is important (in part) because it helps the compiler to optimize your code. What I've never seen is a good explanation on how the compiler uses this information to optimize the code, not even the good books go on explaining what happens behind the curtains.

For example, how does the compiler optimize a method that is declared const vs one that isn't but should be. What happens when you introduce mutable variables? Do they affect these optimizations of const methods?

c++ optimization compiler-construction compiler-optimization const-correctness

Share

Improve this question

Follow

edited Jan 10 at 1:54

Jan Schultke

38.3k ● 8 ● 87 ● 168

asked Oct 17, 2008 at 13:56

David Holm

17.9k ● 8 ● 49 ● 47

12 Answers

Sorted by: Highest score (default)

55

I think that the const keyword was primarily introduced for compilation checking of the program semantic, not for optimization.

Herb Sutter, in the GotW #81 article, explains very well why the compiler can't optimize anything when passing parameters by const reference, or when declaring const return value. The reason is that the compiler has no way to be sure that the object referenced won't be changed, even if declared const : one could use a const_cast, or some other code can have a non-const reference on the same object.

However, quoting Herb Sutter's article :

> There is [only] one case where saying "const" can really mean something, and that is when objects are made const at the point they are defined. In that case, the compiler can often

> successfully put such "really const" objects into read-only memory[...].

There is a lot more in this article, so I encourage you reading it: you'll have a better understanding of constant optimization after that.

Share Improve this answer

Follow

answered Oct 17, 2008 at 14:30

Luc Touraille

81.9k ● 16 ● 99 ● 139

36

Let's disregard methods and look only at const objects; the compiler has much more opportunity for optimization here. If an object is declared const, then (ISO/IEC 14882:2003 7.1.5.1(4)):

> Except that any class member declared mutable (7.1.1) can be modified, any attempt to modify a const object during its lifetime (3.8) results in undefined behavior.

Lets disregard objects that may have mutable members - the compiler is free to assume that the object will not be modified, therefore it can produce significant optimizations. These optimizations can include things like:

- incorporating the object's value directly into the machines instruction opcodes

- complete elimination of code that can never be reached because the const object is used in a conditional expression that is known at compile time
- loop unrolling if the const object is controlling the number of iterations of a loop

Note that this stuff applies only if the actual object is const - it does not apply to objects that are accessed through const pointers or references because those access paths can lead to objects that are not const (it's even well-defined to change objects though const pointers/references as long as the actual object is non-const and you cast away the constness of the access path to the object).

In practice, I don't think there are compilers out there that perform any significant optimizations for all kinds of const objects. but for objects that are primitive types (ints, chars, etc.) I think that compilers can be quite aggressive in optimizing the use of those items.

answered Oct 17, 2008 at 15:41

Michael Burr

340k ● 52 ● 548 ● 768

2 @AndyT: Yes it is. You are allowed to `const_cast` away the constness only for objects which are not const in the first place. – Alexandre C. Oct 30, 2013 at 7:59

7 @AlexandreC. no you can `const_cast` away the constness of any object. But attempting to modify it is undefined behaviour. The cast itself is always fine. – Simple Oct 30, 2013 at 10:08

2 @AlexandreC. it's useful to call a `const` overload from the non-`const` overload and then cast the constness of the result away. – Simple Oct 31, 2013 at 9:37

3 All of the optimization opportunities mentioned in this answer appear to be possible only in case of *compile-time* constants. They seem useless in the case of `const` declared objects whose value cannot be known by the compiler, for instance because their constructor uses non-constant arguments. – Marc van Leeuwen Sep 26, 2014 at 14:23

3 Since when is "I *think* compilers can blah blah" an answer? Many people *think* `const` provides some kind of benefit to optimisation, and loudly proclaim this all around, yet never seem to present generated code demonstrating this... Despite that, newbies believe them anyway, and the myth balloons evermore. Sigh. – underscore_d Jul 15, 2016 at 21:32

6

handwaving begins

Essentially, the earlier the data is fixed, the more the compiler can move around the actual assignment of the data, ensuring that the pipeline doesn't stall out

end handwaving

Share Improve this answer

Follow

answered Oct 17, 2008 at 14:03

Paul Nathan

40.2k ● 30 ● 120 ● 215

5

Meh. Const-correctness is more of a style / error-checking thing than an optimisation. A full-on optimising compiler will follow variable usage and can detect when a variable is effectively const or not.

Added to that, the compiler cannot rely on you telling it the truth - you could be casting away the const inside a library function it doesn't know about.

So yes, const-correctness is a worthy thing to aim for, but it doesn't tell the compiler anything it won't figure out for itself, assuming a good optimising compiler.

Share Improve this answer

Follow

answered Oct 17, 2008 at 14:14

Mike F

4

It does not optimize the function that is declared const.

It can optimize functions that *call* the function that is declared const.

```
void someType::somefunc();

void MyFunc()
{
    someType A(4);   //
    Fling(A.m_val);
```

```
    A.someFunc();
    Flong(A.m_val);
}
```

Here to call Fling, the valud A.m_val had to be loaded into a CPU register. If someFunc() is not const, the value would have to be reloaded before calling Flong(). If someFunc is const, then we can call Flong with the value that's still in the register.

Share Improve this answer

Follow

answered Oct 17, 2008 at 14:11

James Curran

103k ● 37 ● 185 ● 262

1 I don't think const helps with optimization at all. A.someFunc() could easily do const_cast<someType*>(this)->m_val = 42; or any number of other things that would change m_val (e.g. if there is a non-const pointer to A in a global variable, someFunc could change it) – Qwertie Oct 17, 2008 at 15:14

3 The compiler is allowed to assume you won't cast away const; you do so at your own risk. And, as A lives entirely within MyFunc(), the compiler can track if there's a global pointer to it. (In this example, there can't be) – James Curran Oct 17, 2008 at 15:17

3 I'm not sure this answer (and comment) is correct. As far as I know, it is completely legal to cast away constness and modify an object as long as it is not a const object. In your example, A is not const, so `somefunc` (which is not declared const, btw) is allowed to cast away constness on `this` and modify the current object. However, if A was defined as `const someType A(4)`, then modifying it by casting away constness would yeld undefined behavior. Consequently, I don't think the compiler can optimize

anything in the example you gave. – Luc Touraille Mar 11, 2011 at 21:01

It is important to make a distinction between a const object and a const handle (pointer or reference) to an object (which can be const or not). The former can potentially lead to compiler optimizations, will the latter cannot (AFAIK). – Luc Touraille Mar 11, 2011 at 21:03

2 @JamesCurran You can't modify a const object, but you can cast away const. If you decide to cast away const, it's your own responsibility to ensure that you won't end up modifying a const object. – curiousguy Dec 26, 2011 at 15:49

3

The main reason for having **methods** as const is for const correctness, not for possible compilation optimization of the method itself.

If **variables** are const they can (in theory) be optimized away. But only is the scope can be seen by the compiler. After all the compiler must allow for them to be modified with a const_cast elsewhere.

Share Improve this answer

Follow

answered Oct 17, 2008 at 14:13

Andrew Stein

13.1k ● 5 ● 38 ● 44

2

Those are all true answers, but the answers and the question seem to presume one thing: that compiler optimization actually matters.

There is only one kind of code where compiler optimization matters, that is in code that is

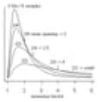
- a tight inner loop,
- in code that you compile, as opposed to a 3rd-party library,
- not containing function or method calls (even hidden ones),
- where the program counter spends a noticeable fraction of its time

If the other 99% of the code is optimized to the Nth degree, it won't make a hoot of difference, because it only matters in code where the program counter actually spends time (which you can find by sampling).

Share Improve this answer Follow

edited Dec 23, 2008 at 20:33

answered Dec 23, 2008 at 20:23



Mike Dunlavey

40.6k ● 15 ● 94 ● 138

I would be surprised if the optimizer actually puts much stock into a const declaration. There is a lot of code that

1

will end up casting const-ness away, it would be a very reckless optimizer that relied on the programmer declaration to assume when the state may change.

Share Improve this answer

Follow

answered Oct 17, 2008 at 14:15

Rob Walker

47.4k ● 15 ● 100 ● 137

1 It would be a very recless programmer to cast away constness. What would she be thinking, "naaah, the documentation doesnt really mean that it's immutable. It doesn't apply to *me*"? – gnud Oct 17, 2008 at 14:30

@gnud: Irrelevant - the compiler can't assume the programmer isn't doing it. – Mike F Oct 17, 2008 at 14:36

2 @Mike F: Extremely relevant, the compiler CAN assume the programmer isn't doing it. The standard states (7.1.6.1/4) that: "Except that any class member declared mutable can be modified, any attempt to modify a const object during its lifetime results in undefined behavior." – Nick Lewis Jul 23, 2009 at 16:39

@NickLewis Of course, but I assume the answer was talking about `const` references/pointers to objects not declared `const` and the ability to cast away `const` on those and thus modify the referent - which *is* defined, and which, along with `mutable` and casting off `const` on a non-`const` instance, is a very valid objection to the oft-repeated but never-cited popular myth that `const` gives some free pass to optimising compilers. – underscore_d Jul 16, 2016 at 12:18

const-correctness is also useful as documentation. If a function or parameter is listed as const, I don't need to

1

worry about the value changing out from under my code (unless somebody else on the team is being very naughty). I'm not sure it would be actually worth it if it wasn't built into the library, though.

Share Improve this answer

Follow

answered Dec 23, 2008 at 21:25

David Thornley

57k • 9 • 95 • 158

0

The most obvious point where `const` is a direct optimization is in passing arguments to a function. It's often important to ensure that the function doesn't modify the data so the only real choices for the function signature are these:

```
void f(Type dont_modify); // or
void f(Type const& dont_modify);
```

Of course, the real magic here is passing a reference rather than creating an (expensive) copy of the object. But if the reference weren't marked as `const`, this would weaken the semantics of this function and have negative effects (such as making error-tracking harder). Therefore, `const` enables an optimization here.

/EDIT: actually, a good compiler can analyze the control flow of the function, determine that it doesn't modify the argument and make the optimization (passing a reference rather than a copy) itself. `const` here is merely a help for the compiler. However, since C++ has some pretty

complicated semantics and such control flow analysis can be very expensive for big functions, we probably shouldn't rely on compilers for this. Does anybody have any data to back me up / prove me wrong?

/EDIT2: and yes, as soon as custom copy constructors come into play, it gets even trickier because compilers unfortunately aren't allowed to omit calling them in this situation.

Share Improve this answer

Follow

answered Oct 17, 2008 at 14:58

Konrad Rudolph

545k ● 139 ● 956 ● 1.2k

1 Either your build tools need link-time optimization, or your copy constructor (and possibly destructor) need to be visible to the compiler at the call site. Otherwise, the compiler will not be able to optimize out a copy. – Tom Dec 24, 2008 at 0:49

0

This code,

```
class Test
{
public:
  Test (int value) : m_value (value)
  {
  }

  void SetValue (int value) const
  {
    const_cast <Test&>(*this).MySetValue (value);
  }
```

```
  int Value () const
  {
    return m_value;
  }

private:
  void MySetValue (int value)
  {
    m_value = value;
  }

  int
    m_value;
};

void modify (const Test &test, int value)
{
  test.SetValue (value);
}

void main ()
{
  const Test
    test (100);

  cout << test.Value () << endl;
  modify (test, 50);
  cout << test.Value () << endl;
}
```

outputs:

```
100
50
```

which means the const declared object has been altered in a const member function. The presence of const_cast (and the mutable keyword) in the C++ language means that the const keyword can't aide the compiler in

generating optimised code. And as I pointed out in my previous posts, it can even produce unexpected results.

As a general rule:

> const != optimisation

In fact, this is a legal C++ modifier:

```
volatile const
```

Share Improve this answer

Follow

edited Jul 3, 2012 at 14:33

user142162

answered Oct 17, 2008 at 15:17

Skizz

71k ● 10 ● 74 ● 109

6 You don't understand `const`. – curiousguy Dec 16, 2011 at 22:51

2 while I agree that optimisation is frequently touted yet never demonstrated that I've seen, this post isn't evidence in favour of that. you're deliberately bombarding the compiler with demands that it must let you shoot yourself in the foot, and invoking UB in the process, then concluding that `const` is broken. no, your code is broken. `void main` is a fine cherry on the top of this proving that. – underscore_d Jul 15, 2016 at 21:27

-1

`const` helps compilers optimize mainly because it makes you write optimizable code. Unless you throw in `const_cast`.

Share Improve this answer

Follow

edited Jul 3, 2012 at 13:26

user142162

answered Oct 17, 2008 at 18:51

MSN

54.5k ● 7 ● 78 ● 107

1 This is unfortunately not true. The compiler has to assume that const_cast and mutable internals will exist elsewhere, unless it can see all of the code in one pass (header-only code). – Tom Dec 24, 2008 at 0:53