Ruminations

Mulling over topics I find interesting

When Should You Use const_cast?

Posted on 06/10/2011 by Aaron Ballman

C++ provides an explicit casting mechanism called const_cast, and yet the question pops up: when would I ever use this? You can always assign a non-const value to a const value without requiring a cast operation. So you don't need to use const_cast to *add* constness to a value. It must mean you use const_cast to remove constness from a value! Or not…

It turns out that there are actually two answers to the question. But to understand the answers, you need to understand a bit about the language.

In C and C++, there are two *cv-modifiers* that you can attach to a value: volatile and const. cv-modifiers are used to modify the semantics for accessing and mutating a value, and they are mutually exclusive. The volatile modifier tells the compiler "access to this value cannot be rearrange by the optimizer under any circumstance", and you use it in embedded programming or hand-tweaked optimizations more than anything. The const modifier tells the compiler "this object should appear to be immutable to an outside observer", and you use it to control the way values are accessed. For instance, if a method accepts a const parameter, then the caller can expect the object not to be mutated by the function being called. Or if a class method is const, then the caller can expect the underlying object not to be mutated when calling the function.

Back to the original question of when to use const_cast. You can use it to "cast away" constness or volatility. But only one of those operations is guaranteed to be safe in all circumstances.

When casting away volatility, you are telling the compiler "even though this is a volatile variable whose access cannot be rearranged, I want you to treat it as a non-volatile." Or you can add volatile behavior to a variable by using const_cast to add the volatility.

When casting away constness, you are venturing into possibly dangerous waters. The only legitimate scenario for mutating a constant object is when you are positive the original value is mutable. If the original object is not mutable, then the behavior is undefined. As a simple example:

```
const int a = 10;
const int *b = &a;

int c = 20;
const int *d = &c;

int *bad = const_cast< int * >( b );
*bad = 12;  // Invalid; undefined behavior!

int *good = const_cast< int * >( d );
*good = 42; // Valid!
```

The assignment to "good" is acceptable because d points to a non-const value. The assignment to "bad" is undefined because b points to a const value.

So the only time it is safe to cast away constness is when you can be assured that the underlying object being referred to is non-const to begin with. However, that's a pretty rare occurrence in practice. So while const_cast can be used safely, I would strongly discourage its use in day to day coding. If you find yourself using it for some reason (or using a C-style cast to remove constness), you should strongly question the reason why you are doing it and whether it's safe.

The only situation I've run into where this may be required is when writing const-correct code which uses a 3rd party library that is not const-correct. In this case, you may find yourself with a const pointer to an object that needs to be passed as a non-const pointer to the 3rd party library. If you are careful, you can use const_cast to allow you to retain the const-correctness of your code while still using the library. But you should try to bottleneck those situations to make them as few as possible.

tl;dr: const_cast is likely something you should never use. If you do use it, understand the dangers!

This entry was posted in C/C++ and tagged const, const correctness, undefined behavior. Bookmark the permalink.

3 Responses to *When Should You Use const_cast?*

David Smith *says:*
12/31/2013 at 1:30 pm

Nice write-up.

In your example code, you use the comment "Illegal", which would imply to me that the code itself would not compile, but it does compile. I'd use a more descriptive comment, like from:

*bad = 12; // Illegal!

to:

*bad = 12; // Compiles, but is bad practice and otherwise wrong

Cheers and thank you for your nice article.

Aaron Ballman *says:*
01/03/2014 at 1:02 pm

@David — thank you for the feedback! I've modified the code to be a bit more clear, and also removed "legality" from the wording, since that can be a bit political of a phrasing.

TK *says:*
11/11/2014 at 12:43 am

Great article! Thanks!

Ruminations