What use are const pointers (as opposed to pointers to const objects)? [duplicate]

Asked 16 years, 2 months ago Modified 1 year, 7 months ago Viewed 22k times

35

This question already has answers here:

What's the point of const pointers? (17 answers)

Closed 12 months ago.

I've often used pointers to const objects, like so...

```
const int *p;
```

That simply means that you can't change the integer that `p` is pointing at through `p`. But I've also seen reference to const pointers, declared like this...

```
int* const p;
```

As I understand it, that means that the pointer variable *itself* is constant -- you can change the integer it points at all day long, but you can't make it point at something else.

What possible use would that have?

c++ c constants

Share

Improve this question

Follow

edited Oct 20, 2008 at 21:23

Cristián Romo

10k ● 12 ● 52 ● 50

asked Oct 20, 2008 at 21:07

Head Geek

39.8k ● 22 ● 79 ● 89

11 Answers

Sorted by: Highest score (default)

31

When you're designing C programs for embedded systems, or special purpose programs that need to refer to the same memory (multi-processor applications sharing memory) then you need constant pointers.

For instance, I have a 32 bit MIPs processor that has a little LCD attached to it. I have to write my LCD data to a specific port in memory, which then gets sent to the LCD controller.

I could #define that number, but then I also have to cast it as a pointer, and the C compiler doesn't have as many options when I do that.

Further, I might need it to be volatile, which can also be cast, but it's easier and clearer to use the syntax provided - a const pointer to a volatile memory location.

For PC programs, an example would be: If you design DOS VGA games (there are tutorials online which are fun to go through to learn basic low level graphics) then you need to write to the VGA memory, which might be referenced as an offset from a const pointer.

Share

Improve this answer

Follow

edited May 5, 2023 at 9:33

cigien

60k • 11 • 80 • 121

answered Oct 20, 2008 at 21:16

Adam Davis

93.5k • 60 • 271 • 333

3 Minor nitpick: for memory mapped devices you certainly (not 'might') need to have the item marked as volatile - otherwise you couldn't be sure if or when the compiler would actually emit the read or write operation. – Michael Burr Oct 20, 2008 at 21:46

2 You 'might' need volatile is correct. You can use `barrier()` and other semantics depending on the device. It is true that you need to handle the values carefully. Ie, cache/no-cache, barrier, etc. It depends on the type of device and **volatile** is not always the best option. – artless-noise-bye-due2AI Feb 15, 2013 at 16:12

@MichaelBurr: I agree with you that the `volatile` should certainly be in there; on the other hand, many compiler vendors' headers don't seem to bother--a fact which can be somewhat annoying (since even if the compilers usually do the right thing, they may ignore attempts to read a register but do nothing with the result). – supercat Apr 12, 2013 at 21:59

30

It allows you to protect the pointer from being changed. This means you can protect assumptions you make based on the pointer never changing or from unintentional modification, for example:

```
int* const p = &i;

...

p++;     /* Compiler error, oops you meant */
(*p)++;  /* Increment the number */
```

Share Improve this answer Follow

answered Oct 20, 2008 at 21:19

Andrew Johnson

1 I have never seen that before and I agree that it is a very useful trick. – Rob Oct 20, 2008 at 21:30

2 It's not a trick :). I try to use const where possible with function arguments so that it is clear that the function won't modify the string or structure being passed in. – Andrew Johnson Oct 20, 2008 at 21:35

The confusing thing is that const often appears twice in a declaration, and it may actually be in a few different positions. Commonly it might appear as something like `const uint8 *const value`. This declares that the pointer and the value it points to are not modifiable (but can be if casted). – James Wald Jun 7, 2014 at 8:52

8

another example: if you know where it was initialized, you can avoid future NULL checks. The compiler guarantees you that the pointer never changed (to NULL)…

Share Improve this answer Follow

answered Oct 20, 2008 at 21:16

Benedikt Waldvogel
12.8k ● 8 ● 54 ● 61

1 In C. In C++, in your particular case (i.e. non-NULL pointers), you use references instead. – paercebal Oct 21, 2008 at 9:29

6

In any non-const C++ member function, the `this` pointer is of type `C * const`, where `C` is the class type -- you can change what it points to (i.e. its members), but you can't change it to point to a different instance of a `C`. For `const` member functions, `this` is of type `const C * const`. There are also (rarely encountered) `volatile` and `const volatile` member functions, for which `this` also has the `volatile` qualifier.

Share Improve this answer Follow

answered Oct 20, 2008 at 21:16

Adam Rosenfield
399k ● 101 ● 522 ● 597

5

One use is in low-level (device driver or embedded) code where you need to reference a specific address that's mapped to an input/output device like a hardware pin. Some languages allow you to link variables at specific addresses (e.g. Ada has `use at`). In C the most idiomatic way to do this is to declare a constant pointer. Note that such usages should also have the `volatile` qualifier.

Other times it's just defensive coding. If you have a pointer that *shouldn't* change it's wise to declare it such that it *cannot* change. This will allow the compiler (and lint tools) to detect erroneous attempts to modify it.

Share
Improve this answer
Follow

edited Oct 20, 2008 at 21:51

answered Oct 20, 2008 at 21:43

Michael Carman
30.8k ● 10 ● 79 ● 124

4

I've always used them when I wanted to avoid unintended modification to the pointer (such as pointer arithmetic, or inside a function). You can also use them for Singleton patterns.

'this' is a hardcoded constant pointer.

Share Improve this answer Follow

answered Oct 20, 2008 at 21:17

Tom Ritter
101k ● 31 ● 142 ● 174

3

Same as a "const int" ... if the compiler knows it's not going to change, it can be optimization assumptions based on that.

```
struct MyClass
{
    char* const ptr;
    MyClass(char* str) :ptr(str) {}

    void SomeFunc(MyOtherClass moc)
    {
         for(int i=0; i < 100; ++i)
         {
                 printf("%c", ptr[i]);
                 moc.SomeOtherFunc(this);
         }
    }
}
```

Now, the compiler could do quite a bit to optimize that loop --- provided it knows that SomeOtherFunc() does not change the value of ptr. With the const, the compiler knows that, and can make the assumptions. Without it, the compiler has to assume that SomeOtherFunc will change ptr.

Share
Improve this answer
Follow

edited Oct 20, 2008 at 21:22

answered Oct 20, 2008 at 21:10

James Curran
103k ● 37 ● 185 ● 262

Won't SomeOtherFunc have to be declared with a const pointer argument for this to compile? And won't that also be how the compiler would know that SomeOtherFunc won't change the pointer? So declaring the local pointer as const doesn't seem to help. – Andrew Johnson Oct 20, 2008 at 21:38

Andrew: I think you're confusing ptr with this – Leon Timmermans Oct 20, 2008 at 22:46

@Andrew: No. SomeOtherFunc is completely allowed to change any other part of MyClass object. – James Curran Oct 21, 2008 at 11:04

How does `const` matter for optimisation? Any clever compiler can determine whether you modify a variable, and optimise if it can based on that, whether or not you *said* you would. – underscore_d May 23, 2017 at 10:27

1 @underscore_d 1) `ptr` is a public member. It's very hard for a compiler to track the usage of a public variable across modules. Many won't even try, and will just treat it as a `volatile` 2) Compilers were dumber 9 years ago. – James Curran May 23, 2017 at 13:36

2

I have seen some OLE code where you there was an object passed in from outside the code and to work with it, you had to access the specific memory that it passed in. So we used const pointers to make sure that functions always manipulated the values than came in through the OLE interface.

Share Improve this answer Follow

answered Oct 20, 2008 at 21:16

epotter

7,799 ● 7 ● 65 ● 89

1

Several good reasons have been given as answers to this questions (memory-mapped devices and just plain old defensive coding), but I'd be willing to bet that most instances where you see this it's actually an error and that the intent was to have to item be a pointer-to-const.

I certainly have no data to back up this hunch, but I'd still make the bet.

Share Improve this answer Follow

answered Oct 20, 2008 at 21:38

Michael Burr

340k ● 52 ● 548 ● 769

i use it a lot for optimization. – DavidG Oct 21, 2008 at 17:44

@DavidG How, precisely, does `const` matter for optimisation? Any clever compiler can determine whether you modify a variable, and optimise if it can based on that, whether or not you *said* you would. – underscore_d May 23, 2017 at 10:24

it did back in 2008 on some devices when I wrote this comment... but the world has changed a lot since then, and now compilers have improved a lot. – DavidG May 24, 2017 at 11:31

0

Think of **type*** and **const type*** as types themselves. Then, you can see why you might want to have a const of those types.

Share Improve this answer Follow

answered Oct 20, 2008 at 22:19

Shadow2531

12.2k ● 5 ● 38 ● 51

yeah, and that type is basically an int. – DavidG Oct 21, 2008 at 17:45

-4

always think of a pointer as an int. this means that

```
object* var;
```

actually can be thought of as

```
int var;
```

so, a const pointer simply means that:

```
const object* var;
```

becomes

```
const int var;
```

and hence u can't change the address that the pointer points too, and thats all. To prevent data change, u must make it a pointer to a const object.

Share Improve this answer Follow

answered Oct 20, 2008 at 21:30

DavidG
1,797 ● 4 ● 21 ● 33

Pointers are **not** `int`s, in multiple ways. Also, the question is very clearly stated as being about `const` *pointers*, not *pointers to* `const` *objects*. But you got them the wrong way around anyway. – underscore_d May 23, 2017 at 10:26