6.10 — Pointers and const

BY ALEX ON JULY 16TH, 2007 | LAST MODIFIED BY ALEX ON SEPTEMBER 29TH, 2015

Pointing to const variables

So far, all of the pointers you've seen are non-const pointers to non-const values:

```
int value = 5;
int *ptr = &value;
*ptr = 6; // change value to 6
```

However, what happens if value is const?

```
const int value = 5; // value is const
int *ptr = &value; // compile error: cannot convert const int* to int*
*ptr = 6; // change value to 6
```

The above snippet won't compile -- we can't set a non-const pointer to a const variable. This makes sense: a const variable is one whose value can not be changed. Hypothetically, if we could set a non-const pointer to a const value, then we would be able to dereference the non-const pointer and change the value. That would violate the intention of const.

Pointer to const value

A **pointer to a const value** is a (non-const) pointer that points to a constant value.

To declare a pointer to a const value, use the *const* keyword before the data type:

```
const int value = 5;
const int *ptr = &value; // this is okay, ptr is pointing to a "const int"
*ptr = 6; // not allowed, we can't change a const value
```

In the above example, ptr points to a const int.

So far, so good, right? Now consider the following example:

```
int value = 5; // value is not constant
const int *ptr = &value; // this is still okay
```

A pointer to a constant variable can point to a non-constant variable (such as variable value in the example above). Think of it this way: a pointer to a constant variable treats the variable as constant when it is accessed through the pointer, regardless of whether the variable was initially defined as const or not.

Thus, the following is okay:

```
int value = 5;
const int *ptr = &value; // ptr points to a "const int"
value = 6; // the value is non-const when accessed through a non-const identifier
```

But the following is not:

```
int value = 5;
const int *ptr = &value; // ptr points to a "const int"
*ptr = 6; // ptr treats its value as const, so changing the value through ptr is not legal
```

Because a pointer to a const value is not const itself (it just points to a const value), the pointer can be redirected to point at other values:

```
int value1 = 5;
const int *ptr = &value1; // ptr points to a const int

int value2 = 6;
```

```
ptr = &value2; // okay, ptr now points at some other const int
```

Const pointers

We can also make a pointer itself constant. A **const pointer** is a pointer whose value can not be changed after initialization

To declare a const pointer, use the *const* keyword between the asterisk and the pointer name:

```
int value = 5;
int *const ptr = &value;
```

Just like a normal const variable, a const pointer must be initialized to a value upon declaration. This means a const pointer will always point to the same address. In the above case, ptr will always point to the address of value (until ptr goes out of scope and is destroyed).

```
int value1 = 5;
int value2 = 6;

int * const ptr = &value1; // okay, the const pointer is initialized to the address of value1
ptr = &value2; // not okay, once initialized, a const pointer can not be changed.
```

However, because the *value* being pointed to is still non-const, it is possible to change the value being pointed to via dereferencing the const pointer:

```
int value = 5;
int *const ptr = &value; // ptr will always point to value
*ptr = 6; // allowed, since ptr points to a non-const int
```

Const pointer to a const value

Finally, it is possible to declare a const pointer to a const value by using the *const* keyword both before the type and before the variable name:

```
int value = 5;
const int *const ptr = &value;
```

A const pointer to a const value can not be set to point to another address, nor can the value it is pointing to be changed through the pointer.

Recapping

To summarize, you only need to remember 4 rules, and they are pretty logical:

- A non-const pointer can be redirected to point to other addresses.
- A const pointer always points to the same address, and this address can not be changed.

- A pointer to a non-const value can change the value it is pointing to. These can not point to a const value.
- A pointer to a const value treats the value as const (even if it is not), and thus can not change the value it is pointing to.

Keeping the declaration syntax straight can be challenging. Just remember that the type of value the pointer points to is always on the far left:

```
int value = 5;
const int *ptr1 = &value; // ptr1 points to a "const int", so this is a pointer to a const value.
int *const ptr2 = &value; // ptr2 points to an "int", so this is a const pointer to a non-const value.
```

Conclusion

Const pointers are primarily used in function parameters (for example, when passing an array to a function) to help ensure the function doesn't inadvertently change the passed in argument. We will discuss this further in the section on functions.

6.11 -- Reference variables

Index

6.9a -- Dynamically allocating arrays

Share this:

Email Facebook 6 Twitter Google Pinterest

C++ TUTORIAL | PRINT THIS POST

35 comments to 6.10 — Pointers and const

JLight
September 18, 2009 at 2:36 am · Reply

Hi.

Firstly i am getting a huge amount out of this course, the way it is laid out means i am ready for each new concept as i find it, and i'm learning alot about programming, not just C++.

Have i got it wrong or in your last example about should you not assign the constant int nValue in line one?

Cheers V much for the time this must have taken, fantastic and fun too!

I would love more quizzes to get practicle practice though. I've been setting myself tasks, but its difficult to think of tasks that really test you cos they're usually based on what i can do, not what i should be able to do…

MJAM
January 28, 2015 at 1:10 pm · Reply

+1 to this! I am learning loads too but would really like some more practice for each task to cement my understanding of each new feature.

Paapeli
November 22, 2010 at 8:44 pm · Reply

Is there any reason I shouldn't use a pointer to a const value in defining array sizes as opposed to using dynamic memory allocation? The following code seems much less dangerous and prone to memory leaks than allocating the memory dynamically:

```
int nArraySize = 5;
const int *pnSizePointer = &nArraySize;
int anArray[*pnSizePointer];
```

Am I right?

broadwayLamb
June 13, 2011 at 5:38 pm · Reply

I am curious about this too. This is a really interesting bit of code. I tested a few things just to see what could be done with it, and I did find a potential danger in declaring a variable this way.

If you declare an array's size by dereferencing a pointer to a number X, and then later you change the size of X, then the program will believe that the size of the array has changed.

Consider this code:

```
int main()
{
/* We create a variable and pointer and use them in the suggested manner
to initialize an array of ten items. We then output the size of the array */
int arraySize = 0;
const int *sizePtr = &arraySize;

arraySize = 10;
int oneArray[*sizePtr];
cout << "First array uses " << sizeof(oneArray) << " bytes" << endl;

/* Now we reset the arraySize variable again to create a new array of 5 items.
We then print out the size of this array */

arraySize = 5;
int twoArray[*sizePtr];
cout << "Second array uses " << sizeof(twoArray) << " bytes" << endl;

/* Now has anything changed with the first array? */

cout << "After changing the arraySize variable, oneArray is now " <<
sizeof(oneArray) << " bytes" << endl;

return(0);
}
```

Here is the output:

```
First array uses 40 bytes
Second array uses 20 bytes
```

```
After changing the arraySize variable, oneArray is now 20 bytes
```

We can see the program now believes that the first array is only 5 items long, but as programmers we may be expecting that first array still be 10 items long.

Imagine the situation in reverse… we declare an array with size based on a variable, then later we raise the value of that variable dramatically. Now the array thinks it is larger that it was when it was allocated, but if you work with that extra space you could be modifying memory that belongs to other variables or code.

Alex
August 19, 2015 at 5:05 pm · Reply

This shouldn't compile, because C++ won't let you declare a fixed array with a length that's not constant at compile time. *pnSizePointer definitely isn't constant at compile time (because it doesn't point to nArraySize until it's initialized).

(Note: Some compilers, such as the one used by Code::Blocks will allow you to do this anyway)

fmunshi
December 14, 2010 at 5:44 pm · Reply

What if I have the following:

int x = 5;
const int *const ptrToX = &x

From what I understand, this is a constant pointer to a constant integer.
The pointer cannot be made to point at another address.
The pointer cannot change the value of the integer at the address it points to.
BUT, I can change the value of x like this:

x = 6;

because the integer x is itself not a const integer but the pointer is treating the variable as a const when it is accessed through the pointer.

Thanks,
Fardan

Alex
August 19, 2015 at 5:06 pm · Reply

Yup, everything you've said is correct.

Sultan
January 9, 2012 at 3:02 pm · Reply

Hi there,

I don't understand what the benefit is when we declare a pointer as constant.

Any help?

Alex
August 19, 2015 at 5:10 pm · Reply

The same thing as making non-pointer variables constant. They help ensure we don't change things when we didn't mean to.

alwaysAstudent
August 16, 2012 at 7:18 am · Reply

Please correct me if I am wrong. I think that the relation between pointers and constant can be best understood when seen from pointers' point of view.

CONSTANT POINTER : is a pointer which is constant; meaning it holds a constant value (address which it points to). that is, it constantly points to the address of a variable (assigned during declaration). the variable may not be a constant. thus the constant pointer points to one unique address and can manipulate the variable that has that address. hence,

```
int nValue = 5;
int *const pnPtr = &nValue;
*pnPtr = 6; // it is possible !!
nValue = 7; // this is also possible !! inadvertently skipped in this tutorial i guess
```

POINTER TO A CONSTANT VARIABLE :

the variable may not be a constant but still, when declared this way, the pointer treats the variable as a constant.

```
int nValue = 5;
const int *pnPtrt = &nValue; // POINTER TO A CONSTANT VARIABLE
nValue = 6; // this can be done as the variable need not be constant
```

however, *pnPtr = 7; will cause program to crash because the pointer pnPtr treats nValue as a constant.

CONSTANT POINTER TO A CONSTANT VARIABLE :

here, both the pointer as well as the variable, are constant.

```
const int nValue = 5;
const int *const pnPtr = &nValue;
```

```
nValue = 6; // not possible because nValue is a constant
*pnPtr = 6; // not possible because pnPtr is pointer to a constant ( and not because pnPtr is a constant pointer !)
```

kekie
June 9, 2014 at 5:21 pm · Reply

This seems to be a very silly naming system. Shouldn't 'a pointer to a constant variable' be called a 'read only pointer' or something?

Alex
August 19, 2015 at 5:13 pm · Reply

I agree that would be easier. But these things really are called "const pointers" and "pointer to const". I try to use the actual names for things even if they're silly so you can look up more information about them later.

That said, once you understand the basic concept, it's pretty clear what is meant by each. It's more the declaration syntax that I can never remember.

Joseph

March 17, 2015 at 7:54 am · Reply

Near the end, you wrote:

Because a pointer to a const value is a non-const pointer, the pointer can be redirected to point at other values:

```
int nValue = 5;
int nValue2 = 6;

const int *pnPtr = &nValue;
pnPtr = &nValue2; // okay
```

Shouldn’t nValue and nValue2 be the consts instead of pnPtr? Would this not be incorrect since pnPtr is a const pointer which means it cannot be changed?

Alex

August 19, 2015 at 5:23 pm · Reply

> Shouldn’t nValue and nValue2 be the consts instead of pnPtr?

No, a pointer to a const value can point at non-const values. It will treat them as const when they are accessed through the pointer.

> Would this not be incorrect since pnPtr is a const pointer which means it cannot be changed?

pnPtr is a pointer to a const value, not a const pointer. Because it is not const itself, its address can be reassigned.

Joe

April 27, 2015 at 5:39 pm · Reply

If I used a non constant pointer such as:

const int *pnPtr = &nValue;

would it not be the same thing as:

int *pnPtr =&nValue;

What is the advantage of using the first expression? Would this be necessary to point to a const int?

Grent

May 16, 2015 at 12:15 am · Reply

Not the same!

In your first expression nValue cannot be changed via the pointer, as in:

```
*pnPtr = 7;
```

Because the pointer treats the pointed-to variable as a constant, no matter if it indeed is one by itself.

As in your second expression, you can change the pointed-to variable via the pointer as shown in the code snippet above.

Todd

July 7, 2015 at 2:04 pm · Reply

I was getting confused, so I decided to write out each pertinent combination. I hope this is is helpful to someone else as well as to me.

```
int A(1);       // non-const variable
int B(2);       // non-const variable
const int C(3); // const variable

int *point1;    // non-const pointer to a non-const value
point1 = &A;    // delayed initialization ok
*point1 = 11;   // redefining A's value ok
point1 = &B;    // redirecting to B ok
point1 = &C;    // redirecting to const C NOT OKAY

int *const point2 = &A; // const pointer to a non-const value
*point2 = 21;           // redefining A's value ok
point2 = &B;            // redirecting to B NOT OKAY

int *const point22 = &C;    // const pointer to a non-const value CANNOT point to const C

const int *point3;  // non-const pointer to a const value
point3 = &A;        // delayed initialization ok
*point3 = 31;       // redefining A's value NOT OKAY
point3 = &B;        // redirecting to B ok
point3 = &C;        // redirecting to const C ok

const int *const point4 = &A;   // const pointer to a const value
*point4 = 41;                   // redefining A's value NOT OKAY
point4 = &B;                    // redirecting to B NOT OKAY

const int *const point44 = &C;  // const pointer to a const value CAN point to const C
```

Avneet
August 20, 2015 at 9:11 pm · Reply

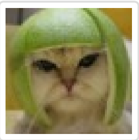
1. " If we set a non-const pointer to a const value, we could dereference the non-const pointer and change the value"

```
const int value = 5; //const value(variable)
int *ptr = &value; // non-const pointer (compile error in this line)
*ptr = 6;  // dereferencing the non-const pointer and changing the value (never compiled!)
```

So, we can't assign a const value's address to a non-const pointer. What are you trying to say above? Is it a typo (if yes, then could should be couldn't) or something that I am not understanding?

2. "Note that the pointer to a constant variable does not actually have to point to a constant variable! Instead, think of it this way: a pointer to a constant variable treats the variable as constant when it is accessed through the pointer, regardless of whether the variable was initially defined as const or not"

What do you mean by "constant variable"? Do you mean "pointer to a const value does not have to point a const variable"…



Alex
August 21, 2015 at 10:47 am · Reply

1) The statement is a hypothetical one. I'll update the statement to make it clearer I'm trying to describe why this is not allowed.
2) Any const variable like "const int value". I've reworded that section a little to try and make it clearer.

James
August 25, 2015 at 8:31 am · Reply

Thank you so much for putting together this amazing set of lessons!

What would be the proper way to destroy a constant pointer? C++ lets me delete a const pointer but then it will not allow me to set it to NULL because it's constant.

Alex
August 25, 2015 at 12:28 pm · Reply

As you've noted, C++ will let you delete a pointer to a constant value -- if you couldn't, you'd never be able to deallocate memory assigned to a const pointer. But because it's a const pointer, you can't set it to a null pointer afterward.

The only real option I see is to live with it, and be careful not to access the pointer after it has been deleted. 🙂

Advanced readers might think about using a const_cast to convert the const int* to a non-const int* so you can set the value to 0. However, modifying any value that was defined as const can have undefined behavior.

Jarvis Chen
September 19, 2015 at 2:43 pm · Reply

Consider the third line in the summary:

```
const int value = 5;
const int *ptr1 = &value; // ptr1 points to a "const int", so this is a pointer to a const value.
int *const ptr2 = &value; // ptr2 points to an "int", so this is a const pointer to a non-const value.
```

On the left, we have a const pointer (always pointing to the same address) pointing to an int (which can change), but on the right we have the address of a const int.

That's an error right?

Alex
September 19, 2015 at 3:36 pm · Reply

Yes. I've changed value to be non-const so both syntaxes compile. Thanks for pointing that out.

Mr D
September 22, 2015 at 12:06 pm · Reply

I'm not understanding this part:

```
int value1 = 5;
const int *ptr = &value1; // ptr points to a const int

int value2 = 6;
ptr = &value2; // okay, ptr now points at some other const int
```

You said "ptr points to a const int", but int value1 isn't const!

I thought maybe it somehow turns value1 into const, but this compiles no problem:

```
int main()
{
    int value1 = 5;
    const int *ptr = &value1; // ptr points to a const int
    value1 = 7;
    std::cout << value1 << "n";
    std::cout << *ptr << "n";
}
```

So what exactly is const here, i don't see it, as i can still re-assign the value of both ptr and value1!?

Alex
September 22, 2015 at 9:52 pm · Reply

ptr is a pointer to a const int. And you’re right, value1 isn’t const -- that’s okay. ptr treats whatever it is pointing at as const (when accessed through ptr), even if the actual value is not const.

That means:

```
value1 = 7; // okay, because value1 is non-const
*ptr = 7; // not okay, because ptr is pointing to a const value, and we can't change a
const value
```

You can reassign ptr to point at a different int because ptr is not itself const, it just points to a const value. And because it points to a const value, you can not change the dereferenced value of ptr.

richi_rich
September 28, 2015 at 11:30 am · Reply

“int const *p;” and “const int *p;” are the same thing

Kevin
September 29, 2015 at 6:10 am · Reply

Grammatical/spelling error

However, what happens if value is const?i <--

Thought it was a "!" at first

Alex
September 29, 2015 at 3:27 pm · Reply

Fixed, thanks!

Huy
October 10, 2015 at 8:12 pm · Reply

why "int const *p;” and “const int *p;" are same, could you please explain in more detailed ?

Alex
October 12, 2015 at 9:03 am · Reply

For the same reason that “int const x” and “const int x” are the same. The type and const specifier can be in either order.

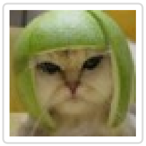
hussein
October 20, 2015 at 10:58 am · Reply

Const pointer to a const value

Finally, it is possible to declare a const pointer to a const value by using the const keyword both before the type and before the variable name:

int value = 5;
const int *const ptr = &value;

maybe you meant:?
const int value = 5;
int *const ptr = &value;



Alex
October 20, 2015 at 1:09 pm · Reply

No. Your ptr is a const pointer to a non-const value, which is then being pointed at const value. You can't have a pointer to a non-const value pointing to a const value. So this won't even compile.

The example is correct as originally written: ptr is a const pointer to a const value. The fact that variable value is non-const doesn't matter. ptr will treat it as a const when the value is accessed through ptr (it can still be accessed through variable value in a non-const manner).

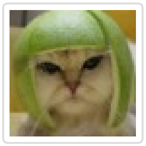
Rob G.
January 5, 2016 at 7:14 pm · Reply

"A pointer to a const value is a (non-const) pointer that points to a constant value.
To declare a pointer to a const value, use the const keyword before the data type:"

```
const int value = 5;
const int *ptr = &value; // this is okay, ptr is pointing to a "const int"
*ptr = 6; // not allowed, we can't change a const value
```

Alex shouldn't this be

```
const int value = 5;
int * ptr = &value;
*ptr = 6; // not allowed, we can't change a const value
```



Alex
January 6, 2016 at 12:48 pm · Reply

No, you can't set a non-const pointer to a const value. The compiler will complain.

Markus
March 9, 2016 at 9:20 pm · Reply

This is the best website about C++ on the internet! I really appreciate your work, you're doing a great job!