6.10 — Pointers and const

BY ALEX ON JULY 16TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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 a non-const pointer that is pointing to a "con
*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 valu
int *const ptr2 = &value; // ptr2 points to an "int", so this is a const pointer to a non-const
const int *const ptr3 = &value; // ptr3 points to a "const int", so this is a const pointer to
```

Conclusion

Pointers to const values 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

C++ TUTORIAL | PRINT THIS POST

106 comments to 6.10 — Pointers and const

« Older Comments 1 2

Ged
November 24, 2019 at 6:54 am · Reply

I commented the line which works and which doesn't. Just wanted to ask. When we initialize a variable with some data type we use braces. But when we want to assign to it a new value or in this case a new address, we cannot use braces again?

```
1  #include <iostream>
2
3  int main()
4  {
5      int value1{ 5 };
6      const int* ptr{ &value1 };
7      std::cout << ptr << '\n';
```

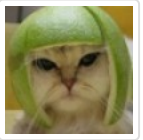
```
    std::cout << *ptr << '\n';
    std::cout << "--------------------------\n";
    int value2{ 6 };
    ptr{ &value2 }; // This doesn't work
        ptr = &value2; // This works
    std::cout << ptr << '\n';
    std::cout << *ptr << '\n';
    return 0;
}
```

nascardriver
November 24, 2019 at 6:59 am · Reply

Correct. We use braces for initialization (Or parentheses or equals, but braces provide higher type-safety). Initialization can only occur once. You can assign as many times as you like, with an equals-sign.

hero
November 16, 2019 at 5:55 am · Reply

do you also have something about "this->"?



Alex
November 16, 2019 at 6:00 pm · Reply

Yes, **https://www.learncpp.com/cpp-tutorial/8-8-the-hidden-this-pointer/**

hersel99
November 3, 2019 at 9:40 am · Reply

Hi! I was trying to see what happen if I access a const int with a pointer to a non-const int.
And I end up with a great idea using increment or decrement on my pointer (I know it's dangerous but I'm curious and want to know) as you can see bellow.
However things doesn't go as I expected it.

```
int a {3};
const int b {5};
int *p {&a};
--p; //Trick to acces address of constant int `b`

*p = 6; // This should change const int `b` to 5 (Mua, ha, ha!)

std::cout << &b << " " << *(&b) << '\n';    // 5
std::cout << p << " " << *p << '\n';        // 6?
std::cout << (p == &b);                     // true
```

This was compiled in codeblocks, gcc 6.3, debian.
So why my pointer `p` points to a value 6? Where does this "magic" number comes from?

Also I do have a gravatar image but seems like it isn't working right now.

nascardriver
November 4, 2019 at 3:35 am · Reply

Hi!

> I do have a gravatar image but seems like it isn't working right now
Your email is correct, you should have a cat avatar. If your avatar still doesn't show up the next time you post a comment, mention it again.

> Where does this "magic" number comes from?
You're setting it in line 6.

Line 6 causes undefined behavior. For one, because `p` is an invalid pointer, then again because you're trying to modify a `const` variable.
Your compiler probably replaced `*(&b)` in line 8 with 5. It's allowed to do that, because `b` is const, so its value can never change anyway. Line 9 then accesses `b` location in memory, which you changed to 6.

Andy
October 29, 2019 at 12:24 am · Reply

Hi,
A very good tutorial, with good advice on best practice.
One thing I was wondering about with respect to const pointers, would it be worthwhile recommending that pointers to dynamically allocated objects should be const?
This would prevent one of the memory leak errors where the pointer is assigned to a different address, eg:

```
1 int value = 5;
2 const int *ptr = new int; // allocate memory
3 ptr = &value; // compile error, old address not lost, memory leak does not occur
```

But a disadvantage of this is that would not allow the pointer to be assigned to nullptr after the object has been deleted

nascardriver
October 29, 2019 at 2:58 am · Reply

Hi

Your `ptr` points to a const `int`, but the pointer itself is non-const.

```
1 const int* ptr{ new int{} };
2 *ptr = 3; // Illegal
3 ptr = nullptr; // Legal
4
5 int* const ptr{ new int{} };
6 *ptr = 3; // Legal
7 ptr = nullptr; // Illegal
```

I wouldn't say this is too useful. There'll still be pointers that you can't initialize with a new expression, so they can't be const.
Fortunately this isn't too big of a problem, because C++ has smart pointers (Covered later). With them you don't have to worry about leaks.

Andy
October 30, 2019 at 1:36 am · Reply

Thanks for correcting my mistake, I realised it after going through a later chapter.
And thanks for the explanation, I will look forward to smart pointers...

hellmet
October 21, 2019 at 9:17 am · Reply

In recap, I think it would be better if it was a table, like so

```
Legend..
  p is a pointer
  v is value
  CC = 'can change', abbreviated for space constraints :P
    Replace 'int' with any other type required

        p, v types        CC p?  CC v?    CC v with p?    Declaration of p
nonconst p to nonconst v  Yes    Yes     Yes             int *p
nonconst p to    const v  Yes    No      No              const int *p
   const p to nonconst v  No     Yes     Yes             int *const p
   const p to    const v  No     No      No              const int *const p
```

Hope that's correct!

Behroo
February 28, 2019 at 10:40 am · Reply

Hi!
I've tried this code :

```
#include "pch.h"
#include <iostream>

void showValue(int *ptr)
{
    std::cout << ptr << " Is a pointer that's pointing "
        << "to a non-const value " << *ptr << '\n';
}

void showValue(const int *ptr)
{
    std::cout << ptr << " Is a pointer that's poiting "
        << "to a const Or non-const value " << *ptr << '\n';
}

void showValue(const int *const ptr)
{
    std::cout << ptr << " Is a const pointer that's poiting "
        << "to a const Or non-const value " << *ptr << '\n';
}

int main()
{
    int value1{ 10 };
    const int value2{ 12 };
    int *ptr1{ &value1 };
    const int *ptr2{ &value1 };
    const int *ptr3{ &value2 };
    const int *const ptr4{ &value1 };
    const int *const ptr5{ &value2 };

    showValue(ptr1);
    showValue(ptr2);
    showValue(ptr3);
    showValue(ptr4);
    showValue(ptr5);
}
```

I was expecting to run and each function call finds its path but I got these errors :

Error	C2264	'showValue': error in function definition or declaration; function not called	33
Error	C2264	'showValue': error in function definition or declaration; function not called	32
Error	C2264	'showValue': error in function definition or declaration; function not called	31
Error	C2264	'showValue': error in function definition or declaration; function not called	30
Error	C2084	function 'void showValue(const int *)' already has a body	15

can you explain me why did that happen ?

nascardriver
March 2, 2019 at 4:33 am · Reply

Hi Behroo!

The last @showValue is ambiguous. Here's a smaller example that's hopefully easier to understand:

```
1  void fn(int i)
2  {}
3
4  void fn(const int i)
5  {}
6
7  fn(3); // Which @fn is supposed to be called?
```

Shrushti Vora
March 29, 2019 at 3:35 am · Reply

Hi,
My question is just an extension to the question asked by Behroo. I fixed the part of the code that was giving the errors.

```
1  #include <iostream>
2
3  void showValue(int *ptr)
4  {
5      std::cout << ptr << " is a pointer that's pointing to a non-const value " <<
6  }
7
8  //! Part of the original code which is giving errors
9  //! void showValue(const int *ptr)
10 //! {
11 //!     std::cout << ptr << " is a pointer that's pointing to a const or non-cons
12 //! }
13
14 void showValue(const int *const ptr)
15 {
16     std::cout << ptr << " is a const pointer that's pointing to a const or non-co
17               << *ptr << '\n';
18 }
19
20 int main()
21 {
22     int value1{10};
23     const int value2{12};
24     int *ptr1{&value1};
25     const int *ptr2{&value1};
26     const int *ptr3{&value2};
27     const int *const ptr4{&value1};
28     const int *const ptr5{&value2};
```

```
29
30      showValue(ptr1);
31      showValue(ptr2);
32      showValue(ptr3);
33      showValue(ptr4);
34      showValue(ptr5);
35
36      return 0;
37  }
```

My question is that when the function showValue is called for

```
1  showValue(ptr2);
2  showValue(ptr3);
```

I would expect the compiler would either complaint since none of the above-defined functions have matching arguments. But instead, this is what I got as output on my machine:

0x7ffeea785728 is a pointer that's pointing to a non-const value 10
0x7ffeea785728 is a const pointer that's pointing to a const or non-const value 10
0x7ffeea785724 is a const pointer that's pointing to a const or non-const value 12
0x7ffeea785728 is a const pointer that's pointing to a const or non-const value 10
0x7ffeea785724 is a const pointer that's pointing to a const or non-const value 12

Can you explain why it called the second function since it only takes a const pointer to a const int?

nascardriver
March 29, 2019 at 3:55 am · Reply

const parameters can be initialized with const and non-const types. The const that's not part of the pointer-type declaration only affects the contents of the function, but not the call.

```
1  void fn(const int i);
```

can be called with a const or non-const int. The const only means that @i cannot be modified inside of @fn.

The same goes for pointers

```
1  void fn(const int *const p);
```

The first const means that the pointer cannot be used to modify the pointed-to object. This is a restriction the caller has to follow.
The second const only means that @p cannot be modified inside of @fn. This doesn't affect the caller, since they won't see modifications to @p anyway.

Ian
February 1, 2019 at 10:30 am · Reply

Hi,

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

Should line two here not be "this is okay, ptr is a const pointer...

Just trying to correct for future generations...

nascardriver
February 2, 2019 at 4:13 am · Reply

Hi Ian!

Nope, it's correct as it is.
The pointer itself is not const.

```
// This is allowed, as is any other reassignment.
ptr = nullptr;
```

const pointers use this syntax

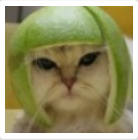
```
// Assuming @value is non-const
int *const ptr{ &value };

// Not allowed
ptr = nullptr;

// Allowed
*ptr = 6;
```

Raindrop7
January 26, 2019 at 6:00 pm · Reply

good explanation. would you add multiple indirection pointers? I mean pointers to pointers.



Alex
January 27, 2019 at 5:03 pm · Reply

Those are covered later in this chapter.

Nguyen
October 16, 2018 at 9:23 pm · Reply

Hi,

```

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

A pointer(ptr) to a constant variable(???) can point to a non-constant variable(such as variable value in the example above).

I am confused. What is the constant variable?

Thanks, Have a great day

nascardriver
October 17, 2018 at 6:22 am · Reply

@ptr is non-const
@value is non-const
But @ptr treats @value as a const int.

```
value = 9; // Legal
*ptr = 9; // Illegal
```

```
3 ptr = nullptr; // Legal
```

Nguyen
January 8, 2019 at 11:44 pm · Reply

But the following is not:

```
1
2 int value = 5;
3 const int *ptr = &value; // ptr points to a "const int"
4 *ptr = 6; // ptr treats its value as const, so changing the value through ptr is n
```

Sorry, the comments in line 4 confuse me.
To me, the variable value is different from &value. So, I am not sure if "its value" should be understood as &value or the variable value in "ptr treats its value as const". Another thing is that ptr is not the same as *ptr, so "changing the value through ptr is not legal" confuses me. It would make more sense to me if it was said "change the value through *ptr instead of ptr is not legal".

Piyush
January 19, 2019 at 2:18 am · Reply

```
1  int value{5};
2  const int *ptr =&value; //This line means that when ptr will be dereferenced
3  //So you can do this.
4  value=6;
5  //But not this.
6  *ptr=6;
7  //Why because as soon as ptr is dereferenced the compiler treats it as a cons
8  //So basically you can't change the value of the variable "value" in the abov
9  *ptr=SomeValue; // :(
10 //But you can change it using value itself.
11 value=someValue; // :)
```

Bouke285
February 22, 2018 at 10:10 am · Reply

This isn't directly related to this lesson, but more general. I've heard the practice, "Declaration is initialization." Which I believe means any time you declare a member, you should initialize it to some value, even if not necessary.

Is this a good practice, to initialize all variables to 0 or similar when you don't have an actual value to initialize to, even when it will be re-assigned inside a loop a couple of lines down the execution path (as an example)?

I just notice that I've been having some trouble forgetting to initialize values to 0 when it is required, such as with a sum variable. Not all compilers will crash, but some will since this is undefined.

Thank you!

nascardriver
February 23, 2018 at 4:07 am · Reply

Hi Bouke!

Always initialize your variables! It appears you already figured why.

Alex
February 26, 2018 at 11:35 am · Reply

It's a good rule of thumb. It may be slightly inefficient in certain cases, but better inefficient than wrong. :)

Sameer Verma
January 15, 2018 at 11:45 pm · 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:

```
1 int value = 5;     //Shouldn't it be "const int value = 5" to declare the value as constant?
2 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.

Alex
January 17, 2018 at 6:08 pm · Reply

It's not necessary. A pointer to a const value can point to a non-const value -- that value is considered const when accessed through the pointer.

It's the same thing with references. A const reference can reference a non-const value.

aditya rawat
October 21, 2017 at 5:31 am · Reply

hey Alex,
pointers point to the address of a variable they are pointed to, so any change in the pointers will lead to a change in the original variable.

now consider this program

#include<iostream>
using namespace std;
int main()
{
  int a=22;//a is assigned a value 22
  const int *ptr=&a;//ok, const ptr points to the address of a.
  cout<<*ptr*2<<endl;//gives 44
  cout<<a;// but why does the value of original variable does not change.
}

i know the question may be silly, but i didn't seem to to understand the function of pointer in this particular case

Alex
October 21, 2017 at 12:47 pm · Reply

Be careful when you say "any change in the pointers", because you can change what address a pointer points to, and you can dereference the pointer to change the value it points to.

In your example, you set ptr to point to variable a's address. Then you access a's value through *ptr, and multiply that value by 2, and print it. This doesn't affect a's value -- it's just a result that's calculated and thrown away.

If you'd done this instead:

```
*ptr = *ptr * 2; // assuming ptr wasn't const
```

That would actually change a's value (through the pointer)

Lamont Peterson
September 8, 2017 at 6:43 am · Reply

Alex,

If I'm understanding the intent of this example snippet properly:

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
```

... then, shouldn't both of those variable declarations (for value1 & value2) be "const int"?

Alex
September 9, 2017 at 8:14 pm · Reply

They can be, but it's not necessary. A pointer to a const value can point at either a const value or a non-const value. In either case, the value will be treated as const when accessed through the pointer.

Lamont Peterson
September 11, 2017 at 12:32 pm · Reply

Alex,

Good point. Definitely worth noting.

I think that while in the context of the lesson, the way it read for that example lead me to believe that the goal was for the pointer to be pointing to const int values (as in, it should be used only with type "const int" variables) but the pointer itself is NOT const (as in, the pointer can be changed to point somewhere else). Thus, the ambiguity which led to my question. Perhaps I was taking you too literally?

Perhaps both points ought be mentioned in this lesson?

haihui li
August 31, 2017 at 7:25 pm · Reply

Hi,Alex
For beginners, it is difficult to distinguish between variables that are pointer types or normal variables.

Alex
August 31, 2017 at 10:34 pm · Reply

Do you think so? Variables that are pointer types are declared with an asterisk, while normal variables are not...

haihui li
September 1, 2017 at 1:11 am · Reply

thanks for your reply.If you add CONST, there will be chemical reactions between them ◦just like□int *const ptr = ...□The asterisk is far from ptr□but it is pointer type.Although I understand now,This problem bothering me for a long time ◦So□I think the way of thinking is more important than its grammar.Thanks again.

Finn St John
August 28, 2017 at 10:33 am · Reply

Hi Alex,

you say: A pointer to a const value is a (non-const) pointer that points to a constant value.

You clearly say that the pointer SHOULD NOT BE constant but in you very first example after that, the pointer you initialize IS a constant.

If this a mistake or am i missing something?

Alex
August 28, 2017 at 5:15 pm · Reply

In that example, ptr is not a const pointer -- it's a non-const pointer pointing to a "const int".

tf
July 7, 2017 at 3:45 pm · Reply

Hello,
I was wondering how pointers interact with constexpr (variables getting evaluated at compile-time):

```
1  int main()
2  {
3      constexpr int value1{ 1 };
4      constexpr int value2{ 2 };
5      constexpr int value3{ 3 };
6      constexpr int value4{ 4 };
7      constexpr int value5{ 5 };
8
9      int *ptr1 = &value1;                    //not okay, since we point with a non const poi
10     const int *ptr2 = &value2;               //okay, since we point at a constexpr variabl
11     constexpr int *ptr3 = &value3;               //not okay, since it can't convert "const
12     const int *const ptr4 = &value4;          //okay, pointing a constant pointer at a co
13     constexpr const int *ptr5 = &value5;          //okay, evaluates similar to ptr4?
14
```

```
15
16      return 0;
17  }
```

VS17 output:

```
1>------ Build started: Project: learncpp, Configuration: Debug Win32 ------
1>learncpp.cpp
1>learncpp.cpp(9): error C2440: 'initializing': cannot convert from 'const int *' to 'int *'
1>learncpp.cpp(9): note: Conversion loses qualifiers
1>learncpp.cpp(11): error C2440: 'initializing': cannot convert from 'const int *' to 'int *
1>learncpp.cpp(11): note: Conversion loses qualifiers
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```

I'm confused about ptr3 and ptr5. From my understanding ptr3 should not work, because the memory for value3 gets assigned at runtime and not at compile time, meaning that pointers can't be constexpr, since the compiler can't predict what memory address the OS will assign. Yet it only complains about being able to convert to a *const pointer (since it seems to be missing const). Isn't const implied by constexpr?

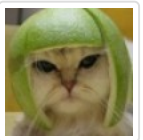
VS2017 seems to treat "constexpr const int *ptr" exactly the same to "const int const* ptr"?

When I run the same stuff in Clang 4.0 (clang++ prog.cc -Wall -Wextra -std=gnu++1z)

```
Start
prog.cc:9:7: error: cannot initialize a variable of type 'int *' with an rvalue of type 'co
        int *ptr1 = &value1;
             ^      ~~~~~~~
prog.cc:11:17: error: cannot initialize a variable of type 'int *const' with an rvalue of t
        constexpr int *ptr3 = &value3;
                       ^      ~~~~~~~
prog.cc:13:23: error: constexpr variable 'ptr5' must be initialized by a constant expressio
        constexpr const int *ptr5 = &value5;
                             ^      ~~~~~~~
prog.cc:13:23: note: pointer to 'value5' is not a constant expression
prog.cc:7:16: note: declared here
        constexpr int value5{ 5 };
                      ^
3 errors generated.
1
Finish
```

It complains about "constexpr const int *ptr5 = &value5;" not being a constant expression, which makes sense to me since the compiler does not know what the memory address will be at compile time.

Is this just a quirk in VS2017 or is it undefined behavior?



Alex
July 7, 2017 at 11:33 pm · Reply

I think this is a quirk of VS2017. Since the address of a variable isn't known at compile time, I wouldn't expect you to be able to set a constexpr pointer to a constexpr value. But this is pushing my knowledge of the subject, so it's very possible I could be wrong.

justAnotherConfusedStudent
June 28, 2017 at 3:22 pm · Reply

So I'm confused. When you initialize a pointer (not a const pointer) and have it point to a const int, can the pointer be redirected to point at other, non-const values?

Alex
June 29, 2017 at 4:02 pm · Reply

You can't make a non-const pointer point at a const int.

Rex Lucas
November 30, 2017 at 6:20 am · Reply

Alex
In the light of your fundamental statement at the top of this page:
"A pointer to a const value is a (non-const) pointer that points to a constant value."

I guess I am missing some subtle point in your reply to "justAnotherConfusedStudent". Are you able to elaborate, please?

Alex
December 2, 2017 at 5:27 pm · Reply

I think I misread his question.

A non-const pointer to a const value can be redirected.
A const pointer to a non-const value can not be redirected.

Basically, the constness of the pointer determines whether the pointer can have the address it holds changed, just like the constness of a variable determines whether that variables value can be changed.

jenifer
June 7, 2017 at 8:58 am · Reply

how can i deal with pointers to static int?

Alex
June 7, 2017 at 2:19 pm · Reply

A pointer to a static int works the same way as a pointer to a non-static int.

jenifer
June 7, 2017 at 9:10 pm · Reply

What about the life time of the pointer to static int? Is it same as the static int, if so i can use it to return by address in a function.

Alex
June 8, 2017 at 11:16 am · Reply

The lifetime of a static int is until the end of the program, so yes, you could return a pointer to a local static variable and be fine.

That said, I can't think of a good use case for actually doing this -- if the caller needs access to the static int's value, why wouldn't you return by value instead?

Mehul
April 26, 2017 at 2:34 am · Reply

Hi Alex

```
I have eritten below code and i am really confused how it is working

    int k;
    const int j = 0;

    int *p = (int *)&j;

    *p = 25;

    cout << &j << endl;
    cout << p << endl;

    cout << j << endl;
    cout << *p << endl;

    cin >> k;
```

O/P
0x7ffff9bf6f80
0x7ffff9bf6f80
0
25

Queries
Here pointer is pointing to a constant variable, And still modifying the values
And strange thing is j remains 0 and *p is only updated to 25

Can you please help me on understanding this

Alex
April 26, 2017 at 12:19 pm · Reply

You're casting away a const, which you should never do unless you're sure the original value was non-const. This will result in undefined behavior, which is what you're seeing.

Mehul
April 29, 2017 at 5:27 am · Reply

Thanks Alex

« Older Comments 1 2