12.9 — Pointers and const

ALEX[1] FEBRUARY 12, 2025

Consider the following code snippet:

```
int main()
{
    int x { 5 };
    int* ptr { &x }; // ptr is a normal (non-const) pointer

    int y { 6 };
    ptr = &y; // we can point at another value

    *ptr = 7; // we can change the value at the address being held

    return 0;
}
```

With normal (non-const) pointers, we can change both what the pointer is pointing at (by assigning the pointer a new address to hold) or change the value at the address being held (by assigning a new value to the dereferenced pointer).

However, what happens if the value we want to point at is const?

```
int main()
{
    const int x { 5 }; // x is now const
    int* ptr { &x };   // compile error: cannot convert from const int* to int*

    return 0;
}
```

The above snippet won't compile -- we can't set a normal pointer to point at a const variable. This makes sense: a const variable is one whose value cannot be changed. Allowing the programmer to set a non-const pointer to a const value would allow the programmer to dereference the pointer and change the value. That would violate the const-ness of the variable.

Pointer to const value

A **pointer to a const value** (sometimes called a `pointer to const` for short) 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 pointer's data type:

```cpp
int main()
{
    const int x{ 5 };
    const int* ptr { &x }; // okay: ptr is pointing to a "const int"

    *ptr = 6; // not allowed: we can't change a const value

    return 0;
}
```

In the above example, `ptr` points to a `const int`. Because the data type being pointed to is const, the value being pointed to can't be changed.

However, because a pointer to const is not const itself (it just points to a const value), we can change what the pointer is pointing at by assigning the pointer a new address:

```cpp
int main()
{
    const int x{ 5 };
    const int* ptr { &x }; // ptr points to const int x

    const int y{ 6 };
    ptr = &y; // okay: ptr now points at const int y

    return 0;
}
```

Just like a reference to const, a pointer to const can point to non-const variables too. A pointer to const treats the value being pointed to as constant, regardless of whether the object at that address was initially defined as const or not:

```cpp
int main()
{
    int x{ 5 }; // non-const
    const int* ptr { &x }; // ptr points to a "const int"

    *ptr = 6;  // not allowed: ptr points to a "const int" so we can't change the value through ptr
    x = 6; // allowed: the value is still non-const when accessed through non-const identifier x

    return 0;
}
```

Const pointers

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

To declare a const pointer, use the `const` keyword after the asterisk in the pointer declaration:

```cpp
int main()
{
    int x{ 5 };
    int* const ptr { &x }; // const after the asterisk means this is a const pointer

    return 0;
}
```

In the above case, `ptr` is a const pointer to a (non-const) int value.

Just like a normal const variable, a const pointer must be initialized upon definition, and this value can’t be changed via assignment:

```
int main()
{
    int x{ 5 };
    int y{ 6 };

    int* const ptr { &x }; // okay: the const pointer is initialized to the address of
x
    ptr = &y; // error: once initialized, a const pointer can not be changed.

    return 0;
}
```

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

```
int main()
{
    int x{ 5 };
    int* const ptr { &x }; // ptr will always point to x

    *ptr = 6; // okay: the value being pointed to is non-const

    return 0;
}
```

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 after the asterisk:

```
int main()
{
    int value { 5 };
    const int* const ptr { &value }; // a const pointer to a const value

    return 0;
}
```

A const pointer to a const value can not have its address changed, nor can the value it is pointing to be changed through the pointer. It can only be dereferenced to get the value it is pointing at.

Pointer and const recap

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

- A non-const pointer (e.g. `int* ptr`) can be assigned another address to change what it is pointing at.
- A const pointer (e.g. `int* const ptr`) always points to the same address, and this address can not be changed.
- A pointer to a non-const value (e.g. `int* ptr`) can change the value it is pointing to. These can not point to a const value.

- A pointer to a const value (e.g. `const int* ptr`) treats the value as const when accessed through the pointer, and thus can not change the value it is pointing to. These can be pointed to const or non-const l-values (but not r-values, which don't have an address).

Keeping the declaration syntax straight can be a bit challenging:

- A `const` before the asterisk (e.g. `const int* ptr`) is associated with the type being pointed to. Therefore, this is a pointer to a const value, and the value cannot be modified through the pointer.
- A `const` after the asterisk (e.g. `int* const ptr`) is associated with the pointer itself. Therefore, this pointer cannot be assigned a new address.

```
int main()
{
    int v{ 5 };

    int* ptr0 { &v };             // points to an "int" but is not const itself.  We can modify the value or the address.
    const int* ptr1 { &v };       // points to a "const int" but is not const itself.  We can only modify the address.
    int* const ptr2 { &v };       // points to an "int" and is const itself.   We can only modify the value.
    const int* const ptr3 { &v }; // points to a "const int" and is const itself.  We can't modify the value nor the address.

    // if the const is on the left side of the *, the const belongs to the value
    // if the const is on the right side of the *, the const belongs to the pointer

    return 0;
}
```

Next lesson
12.10 Pass by address

2

Back to table of contents

3

Previous lesson
12.8 Null pointers

4

5

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[8] are connected to your provided email address.

210 COMMENTS Newest

YFN
June 11, 2025 9:14 am PDT

Wouldn't a const pointer be dangerous? You wouldn't be able to change it to `nullptr`, and so it might end up dangling with no way to tell, right?

0 Reply

RURU
Reply to YFN[9] June 21, 2025 11:22 am PDT

no since with the const pointer we have to initialize it at the time of declaration itself and we cant define a const pointer later on so there shouldnt be the case of dangling pointer .

0 Reply

YFN
Reply to RURU[10] June 24, 2025 9:46 am PDT

But the creation of a dangling pointer doesn't arise from redefining the pointer; it arises from the location in memory that the pointer is storing ceasing to be used by any variable or object. I don't see how we could protect against that happening.

1 Reply

johnooooooooo
May 3, 2025 7:17 am PDT

```
int main()
{
int x{ 10 };
int* ptrToX{ &x }; //normal pointer
```

```
*ptrToX = 100; //allowed, x is non const

int y{ 100 };
ptrToX = &y; //allowed, ptrToX is non const

const int constX{ 1000 };

const int* ptrToConst{ &constX }; //pointer to const variable
*ptrToConst = 2000; //not allowed, points to a const variable

ptrToConst = &x; //allowed, pointer itself is not actually const

int* const Pointer{ &x }; //const pointer, cant change where it points
*Pointer = 23455; //allowed, pointer is const but variabale is not
Pointer = &y; //not allowed, the pointer is const

const int finalRef{ 2345 };
const int* const SolidPtr{ &finalRef };
```

`*SolidPtr = 2000; // not allowed, points to a const variable`
`SolidPtr = &x; //not allowed, the pointer is const`
`}`

0 Reply

Choco
April 18, 2025 5:37 pm PDT

when do i know where to use each of them mfs

1 Reply

Kania
March 5, 2025 6:56 am PST

And here I was worrying for no reason that I won't be able to understand pointers because everyone was calling them hard. In the end pointers were nothing difficult(or the people didn't had access to this site)

9 Reply

Leni
March 2, 2025 8:43 pm PST

Understanding the distinction between pointers to const values and const pointers is crucial for managing data safely in C++, but how can we effectively decide which type of pointer to use in different programming scenarios?

0 Reply

Badger Patcher
Reply to Leni [11] March 3, 2025 1:00 pm PST

I guess doing actual programming will help.

1 Reply

ice-shroom
February 12, 2025 1:10 am PST

May I ask a clarifying question?
That is, in the case of:

`const int* ptr1 {&v };`

- I can change the address of the pointer, but I cannot assign a new value to this address, because it refers to a constant variable,

and in the case of:

`int*const ptr2 {&v};`

I can't change the address of the pointer, but can I assign a new value to this address?

1 Reply

Alex Author
Reply to ice-shroom [12] February 12, 2025 11:41 am PST

Yep. I made some tweaks to the bottom section to try to make this clearer.

0 Reply

PooperShooter
December 31, 2024 9:05 am PST

interesting

0 Reply

Karl
May 18, 2024 12:49 am PDT

For those new to pointers, it is often helpful to read pointer/reference declarations/definitions from right to left:

```
int x{0};
const int* const cpci{&x}; // cpci is a const pointer to an integer treated as constant
```

This rule works for any of the simple cases covered in this lesson. Believe it or not, compound types can get more complex, and the right-to-left reading must be supplemented with an inner-to-outer rule to remain meaningful.

22 Reply

NordicPeace

Reply to Karl 13 December 17, 2024 2:18 am PST

thanks!

0 Reply

Sanji

Reply to Karl 13 August 12, 2024 3:15 am PDT

have been doing this for a while and it always works!! but what do you mean by the inner-to-outer rule in this context?

0 Reply

Alex Author

Reply to Sanji 14 August 14, 2024 11:19 am PDT

Probably this: https://c-faq.com/decl/spiral.anderson.html

9 Reply

newb

Reply to Karl 13 June 30, 2024 8:15 am PDT

```
int x{0};
const int* const cpci{&x}; // cpci is a const pointer to an integer treated as constant
```

Doesn't 'x' in this case need to be constant too?
edit- nvm I was thinking about this backwards.

Last edited 1 year ago by newb

0 Reply

Chayim

December 31, 2023 10:27 pm PST

If a pointer to a constant is only pointing to a constant but not a constant itself, so why the need to even declare it as `int`? why not simply `*ptr &var` because it's only a pointer to an address, and when dereferencing it it will be automatically an `int` or `constant` whatever is in the address.

Last edited 1 year ago by Chayim

1 Reply

Copernicus

Reply to Chayim 15 May 19, 2025 6:40 am PDT

C++ is a strong typed language, all objects must have a type.

0 Reply

BlackPete

Reply to Chayim 15 April 14, 2024 5:46 pm PDT

As you said, it's just a memory address. How do you know what type should be used when you dereferenced it?

1 Reply

Karl

Reply to BlackPete 16 May 18, 2024 1:21 am PDT

It is not just a memory address. As Alex stated in an earlier lesson, the address operator (&) itself doesn't simply return a memory address, but rather it returns a pointer containing the address as a value. Every pointer must also have a type to which it points (an int, double, etc.), which is why pointers are known as "compound" types.

To answer the original question in greater depth, every value for a variable is ultimately stored as 1's and 0's. The meaning of the 1's and 0's depends on the type of the value. The compiler must know the type of the value stored in order to know how to (encode/decode) the value (to/from) binary as the value is (stored/retrieved). Furthermore, a memory address generally represents one specific "bucket" or "word" of memory, but different types consume more or less memory, and the address of an object of these types generally is the address of the first "word" used. Without knowing the type to which the pointer is pointing, the compiler has no means of knowing how many consecutive "buckets" or "words" of memory to use when dereferencing the pointer. And, as already stated, even if it did know how much memory to "look at," it wouldn't know how to interpret the underlying 1's and 0's.

When a value is encoded into binary, that's the only thing stored - there is no extra information added to serve as an indicator that the value is a certain type. There are many reasons for this:

1. Having to additionally store information about the type of the value is expensive - it's best to just let the compiler keep track of types.
2. This becomes burdensome to any programmer defining new custom types, because the programmer would then have to add additional information on how the type info is encoded into the stored values in a unique and meaningful way. That would be a nightmare.

0 Reply

Alex Author

Reply to Chayim 15 January 2, 2024 2:44 pm PST

Because all objects must have a type.

5 Reply

Timo

July 23, 2023 8:35 am PDT

```
1  int main()
2  {
3      int value { 5 };
4
5      int* ptr0 { &value };           // ptr0 points to an "int" and is not const itself, so this is a normal pointer.
6      const int* ptr1 { &value };     // ptr1 points to a "const int", but is not const itself, so this is a pointer to a const value.
7      int* const ptr2 { &value };     // ptr2 points to an "int", but is const itself, so this is a const pointer (to a non-const value).
8      const int* const ptr3 { &value }; // ptr3 points to an "const int", and it is const itself, so this is a const pointer to a const value.
9
10     return 0;
11 }
```

Check last comment in the code: " ptr3 points to an "const int", and it is const itself, so this is a const pointer to a const value."

You made a typo and it should be "ptr3 points to a "const int and ... "

Last edited 1 year ago by Timo

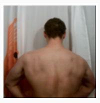
0 Reply

Alex Author

Reply to Timo 17 July 23, 2023 11:14 pm PDT

Eer, seems correct as written.

0 Reply



Timo

Reply to Alex 18 July 24, 2023 10:23 am PDT

i'm confused. Why do you write as comment then on line6 " ptr1 points to a "const int"? and on line 8 - " ptr3 points to an "const int"? English isn't my first language tho so I might be wrong...

0 Reply

Alex Author

Reply to Timo 19 July 28, 2023 5:05 pm PDT

I fixed some inconsistencies in the comments. Is it better now?

0 Reply

Tot

Reply to Alex [20] January 8, 2024 10:56 am PST

Ohh, I see what Timo means. He's talking about the fact you used "a" and "an" inconsistently.

For the second comment, you use "a" ("a "const int""), but for the last comment, you use "an" ("an "const int"").

It should be "**a** "const int"". lol

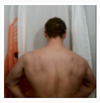
1 Reply

Alex Author

Reply to Tot [21] January 8, 2024 4:07 pm PST

Missed that one. Fixed.

1 Reply



Timo

Reply to Alex [20] July 29, 2023 1:07 am PDT

This is much better. Thx Alex

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/pass-by-address/
3. https://www.learncpp.com/
4. https://www.learncpp.com/cpp-tutorial/null-pointers/
5. https://www.learncpp.com/pointers-and-const/
6. https://www.learncpp.com/cpp-tutorial/dynamic-memory-allocation-with-new-and-delete/
7. https://www.learncpp.com/cpp-tutorial/lvalue-references/
8. https://gravatar.com/
9. https://www.learncpp.com/cpp-tutorial/pointers-and-const/#comment-610845
10. https://www.learncpp.com/cpp-tutorial/pointers-and-const/#comment-611083
11. https://www.learncpp.com/cpp-tutorial/pointers-and-const/#comment-608203
12. https://www.learncpp.com/cpp-tutorial/pointers-and-const/#comment-607647
13. https://www.learncpp.com/cpp-tutorial/pointers-and-const/#comment-597217
14. https://www.learncpp.com/cpp-tutorial/pointers-and-const/#comment-600800
15. https://www.learncpp.com/cpp-tutorial/pointers-and-const/#comment-591571
16. https://www.learncpp.com/cpp-tutorial/pointers-and-const/#comment-595783

17. https://www.learncpp.com/cpp-tutorial/pointers-and-const/#comment-584309
18. https://www.learncpp.com/cpp-tutorial/pointers-and-const/#comment-584404
19. https://www.learncpp.com/cpp-tutorial/pointers-and-const/#comment-584425
20. https://www.learncpp.com/cpp-tutorial/pointers-and-const/#comment-584680
21. https://www.learncpp.com/cpp-tutorial/pointers-and-const/#comment-591960
22. https://g.ezoic.net/privacy/learncpp.com