12.14 — Type deduction with pointers, references, and const

ALEX[1] JANUARY 21, 2025

In lesson 10.8 -- Type deduction for objects using the auto keyword (https://www.learncpp.com/cpp-tutorial/type-deduction-for-objects-using-the-auto-keyword/)[2], we discussed how the `auto` keyword can be used to have the compiler deduce the type of a variable from the initializer:

```
int main()
{
    int a { 5 };
    auto b { a }; // b deduced as an int

    return 0;
}
```

We also noted that by default, type deduction will drop `const` from types:

```
int main()
{
    const double a { 7.8 }; // a has type const double
    auto b { a };           // b has type double (const dropped)

    constexpr double c { 7.8 }; // c has type const double (constexpr implicitly applies const)
    auto d { c };               // d has type double (const dropped)

    return 0;
}
```

Const (or constexpr) can be reapplied by adding the `const` (or `constexpr`) qualifier to the definition of the deduced type:

```
int main()
{
    double a { 7.8 };    // a has type double
    const auto b { a };  // b has type const double (const applied)

    constexpr double c { 7.8 }; // c has type const double (constexpr implicitly applies const)
    const auto d { c };         // d is const double (const dropped, const reapplied)
    constexpr auto e { c };     // e is constexpr double (const dropped, constexpr reapplied)

    return 0;
}
```

Type deduction drops references

In addition to dropping const, type deduction will also drop references:

```
#include <string>

std::string& getRef(); // some function that returns a reference

int main()
{
    auto ref { getRef() }; // type deduced as std::string (not std::string&)

    return 0;
}
```

In the above example, variable `ref` is using type deduction. Although function `getRef()` returns a `std::string&`, the reference qualifier is dropped, so the type of `ref` is deduced as `std::string`.

Just like with dropped `const`, if you want the deduced type to be a reference, you can reapply the reference at the point of definition:

```
#include <string>

std::string& getRef(); // some function that returns a reference

int main()
{
    auto ref1 { getRef() };  // std::string (reference dropped)
    auto& ref2 { getRef() }; // std::string& (reference dropped, reference reapplied)

    return 0;
}
```

Top-level const and low-level const

A **top-level const** is a const qualifier that applies to an object itself. For example:

```
const int x;    // this const applies to x, so it is top-level
int* const ptr; // this const applies to ptr, so it is top-level
// references don't have a top-level const syntax, as they are implicitly top-level const
```

In contrast, a **low-level const** is a const qualifier that applies to the object being referenced or pointed to:

```
const int& ref; // this const applies to the object being referenced, so it is low-level
const int* ptr; // this const applies to the object being pointed to, so it is low-level
```

A reference to a const value is always a low-level const. A pointer can have a top-level, low-level, or both kinds of const:

```
const int* const ptr; // the left const is low-level, the right const is top-level
```

When we say that type deduction drops const qualifiers, it only drops top-level consts. Low-level consts are not dropped. We'll see examples of this in just a moment.

Type deduction and const references

If the initializer is a reference to const, the reference is dropped first (and then reapplied if applicable), and then any top-level const is dropped from the result.

```
#include <string>

const std::string& getConstRef(); // some function that returns a reference to const

int main()
{
    auto ref1{ getConstRef() }; // std::string (reference dropped, then top-level const dropped from result)

    return 0;
}
```

In the above example, since `getConstRef()` returns a `const std::string&`, the reference is dropped first, leaving us with a `const std::string`. This const is now a top-level const, so it is also dropped, leaving the deduced type as `std::string`.

> **Key insight**
>
> Dropping a reference may change a low-level const to a top-level const: `const std::string&` is a low-level const, but dropping the reference yields `const std::string`, which is a top-level const.

We can reapply a reference and/or const:

```
#include <string>

const std::string& getConstRef(); // some function that returns a const reference

int main()
{
    auto ref1{ getConstRef() };        // std::string (reference and top-level const dropped)
    const auto ref2{ getConstRef() };  // const std::string (reference dropped, const dropped, const reapplied)

    auto& ref3{ getConstRef() };       // const std::string& (reference dropped and reapplied, low-level const not dropped)
    const auto& ref4{ getConstRef() }; // const std::string& (reference dropped and reapplied, low-level const not dropped)

    return 0;
}
```

We covered the case for `ref1` in the prior example. For `ref2`, this is similar to the `ref1` case, except we're reapplying the `const` qualifier, so the deduced type is `const std::string`.

Things get more interesting with `ref3`. Normally the reference would be dropped first, but since we've reapplied the reference, it is not dropped. That means the type is still `const std::string&`. And since this const is a low-level const, it is not dropped. Thus the deduced type is `const std::string&`.

The `ref4` case works similarly to `ref3`, except we've reapplied the `const` qualifier as well. Since the type is already deduced as a reference to const, us reapplying `const` here is redundant. That said, using `const` here makes it explicitly clear that our result will be const (whereas in the `ref3` case, the constness of the result is implicit and not obvious).

Best practice

If you want a const reference, reapply the `const` qualifier even when it's not strictly necessary, as it makes your intent clear and helps prevent mistakes.

What about constexpr references?

Constexpr is not part of an expression's type, so it is not deduced by `auto`.

A reminder

When defining a const reference (e.g. `const int&`), the const applies to the object being referenced, not the reference itself.

When defining a constexpr reference to a const variable (e.g. `constexpr const int&`), we need to apply both `constexpr` (which applies to the reference) and `const` (which applies to the type being referenced).

This is covered in lesson [12.4 -- Lvalue references to const](https://www.learncpp.com/cpp-tutorial/lvalue-references-to-const/) (https://www.learncpp.com/cpp-tutorial/lvalue-references-to-const/)[3].

```
#include <string_view>
#include <iostream>

constexpr std::string_view hello { "Hello" };   // implicitly const

constexpr const std::string_view& getConstRef() // function is constexpr, returns a const std::string_view&
{
    return hello;
}

int main()
{
    auto ref1{ getConstRef() };                  // std::string_view (reference dropped and top-level const dropped)
    constexpr auto ref2{ getConstRef() };        // constexpr const std::string_view (reference dropped and top-level const dropped, constexpr applied, implicitly const)

    auto& ref3{ getConstRef() };                 // const std::string_view& (reference reapplied, low-level const not dropped)
    constexpr const auto& ref4{ getConstRef() }; // constexpr const std::string_view& (reference reapplied, low-level const not dropped, constexpr applied)

    return 0;
}
```

Type deduction and pointers

Unlike references, type deduction does not drop pointers:

```
#include <string>

std::string* getPtr(); // some function that returns a pointer

int main()
{
    auto ptr1{ getPtr() }; // std::string*

    return 0;
}
```

We can also use an asterisk in conjunction with pointer type deduction (`auto*`) to make it clearer that the deduced type is a pointer:

```
#include <string>

std::string* getPtr(); // some function that returns a pointer

int main()
{
    auto ptr1{ getPtr() };  // std::string*
    auto* ptr2{ getPtr() }; // std::string*

    return 0;
}
```

Key insight

The reason that references are dropped during type deduction but pointers are not dropped is because references and pointers have different semantics.

When we evaluate a reference, we're really evaluating the object being referenced. Therefore, when deducing a type, it makes sense that we should deduce the type of the thing being referenced, not the reference itself. Also, since we deduce a non-reference, it's really easy to make it a reference by using `auto&`. If type deduction were to deduce a reference instead, the syntax for removing a reference if we didn't want it is much more complicated.

On the other hand, pointers hold the address of an object. When we evaluate a pointer, we are evaluating the pointer, not the object being pointed to (if we want that, we can dereference the pointer). Therefore, it makes sense that we should deduce the type of the pointer, not the thing being pointed to.

The difference between auto and auto* Optional

When we use `auto` with a pointer type initializer, the type deduced for `auto` includes the pointer. So for `ptr1` above, the type substituted for `auto` is `std::string*`.

When we use `auto*` with a pointer type initializer, the type deduced for auto does *not* include the pointer -- the pointer is reapplied afterward after the type is deduced. So for `ptr2` above, the type substituted for `auto` is `std::string`, and then the pointer is reapplied.

In most cases, the practical effect is the same (`ptr1` and `ptr2` both deduce to `std::string*` in the above example).

However, there are a couple of difference between `auto` and `auto*` in practice. First, `auto*` must resolve to a pointer initializer, otherwise a compile error will result:

```
#include <string>

std::string* getPtr(); // some function that returns a pointer

int main()
{
    auto ptr3{ *getPtr() };      // std::string (because we dereferenced getPtr())
    auto* ptr4{ *getPtr() };     // does not compile (initializer not a pointer)

    return 0;
}
```

This makes sense: in the `ptr4` case, `auto` deduces to `std::string`, then the pointer is reapplied. Thus `ptr4` has type `std::string*`, and we can't initialize a `std::string*` with an initializer that is not a pointer.

Second, there are differences in how `auto` and `auto*` behave when we introduce `const` into the equation. We'll cover this below.

Type deduction and const pointers Optional

Since pointers aren't dropped, we don't have to worry about that. But with pointers, we have both the const pointer and the pointer to const cases to think about, and we also have `auto` vs `auto*`. Just like with references, only top-level const is dropped during pointer type deduction.

Let's start with a simple case:

```
#include <string>

std::string* getPtr(); // some function that returns a pointer

int main()
{
    const auto ptr1{ getPtr() };  // std::string* const
    auto const ptr2 { getPtr() }; // std::string* const

    const auto* ptr3{ getPtr() }; // const std::string*
    auto* const ptr4{ getPtr() }; // std::string* const

    return 0;
}
```

When we use either `auto const` or `const auto`, we're saying, "make the deduced pointer a const pointer". So in the case of `ptr1` and `ptr2`, the deduced type is `std::string*`, and then const is applied, making the final type `std::string* const`. This is similar to how `const int` and `int const` mean the same thing.

However, when we use `auto*`, the order of the const qualifier matters. A `const` on the left means "make the deduced pointer a pointer to const", whereas a `const` on the right means "make the deduced pointer type a const pointer". Thus `ptr3` ends up as a pointer to const, and `ptr4` ends up as a const pointer.

Now let's look at an example where the initializer is a const pointer to const.

```
#include <string>

int main()
{
    std::string s{};
    const std::string* const ptr { &s };

    auto ptr1{ ptr };  // const std::string*
    auto* ptr2{ ptr }; // const std::string*

    auto const ptr3{ ptr };  // const std::string* const
    const auto ptr4{ ptr };  // const std::string* const

    auto* const ptr5{ ptr }; // const std::string* const
    const auto* ptr6{ ptr }; // const std::string*

    const auto const ptr7{ ptr };  // error: const qualifer can not be applied twice
    const auto* const ptr8{ ptr }; // const std::string* const

    return 0;
}
```

The `ptr1` and `ptr2` cases are straightforward. The top-level const (the const on the pointer itself) is dropped. The low-level const on the object being pointed to is not dropped. So in both cases, the final type is `const std::string*`.

The `ptr3` and `ptr4` cases are also straightforward. The top-level const is dropped, but we're reapplying it. The low-level const on the object being pointed to is not dropped. So in both cases, the final type is `const std::string* const`.

The `ptr5` and `ptr6` cases are analogous to the cases we showed in the prior example. In both cases, the top-level const is dropped. For `ptr5`, the `auto* const` reapplies the top-level const, so the final type is `const std::string* const`. For `ptr6`, the `const auto*` applies const to the type being pointed to (which in this case was already const), so the final type is `const std::string*`.

In the `ptr7` case, we're applying the const qualifier twice, which is disallowed, and will cause a compile error.

And finally, in the `ptr8` case, we're applying const on both sides of the pointer (which is allowed since `auto*` must be a pointer type), so the resulting types is `const std::string* const`.

Best practice

If you want a const pointer, pointer to const, or const pointer to const, reapply the `const` qualifier(s) even when it's not strictly necessary, as it makes your intent clear and helps prevent mistakes.

Tip

Consider using `auto*` when deducing a pointer type. Using `auto*` in this case makes it clearer that we are deducing a pointer type, enlists the compiler's help to ensure we don't deduce a non-pointer type, and gives you more control over const.

Summary

Sorry to hear about your headache. Let's recap the most important points quickly.

Top-level vs low-level const:

- A top-level const applies to the object itself (e.g. `const int x` or `int* const ptr`).
- A low-level const applies to the object accessed through a reference or pointer (e.g. `const int& ref`, `const int* ptr`).

What type deduction deduces:

- Type deduction first drops any references (unless the deduced type is defined as a reference). For a const reference, dropping the reference will cause the (low-level) const to become a top-level const.
- Type deduction then drops any top-level const (unless the deduced type is defined as `const` or `constexpr`).
- Constexpr is not part of the type system, so is never deduced. It must always be explicitly applied to the deduced type.
- Type deduction does not drop pointers.
- Always explicitly define the deduced type as a reference, `const`, or `constexpr` (as applicable), and even if these qualifiers are redundant because they would be deduced. This helps prevent errors and makes it clear what your intent is.

Type deduction and pointers:

- When using `auto`, the deduced type will be a pointer only if the initializer is a pointer. When using `auto*`, the deduced type is always a pointer, even if the initializer is not a pointer.
- `auto const` and `const auto` both make the deduced pointer a const pointer. There is no way to explicitly specify a low-level const (pointer-to-const) using `auto`.
- `auto* const` also makes the deduced pointer a const pointer. `const auto*` makes the deduced pointer a pointer-to-const. If these are hard to remember, `int* const` is a const pointer (to int), so `auto* const` must be a const pointer. `const int*` is a pointer-to-const (int), so `const auto*` must be a pointer-to-const)
- Consider using `auto*` over `auto` when deducing a pointer type, as it allows you to explicitly reapply both the top-level and low-level const, and will error if a pointer type is not deduced.

Next lesson
12.15 std::optional

4

Back to table of contents

5

Previous lesson
12.13 In and out parameters

6

7

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/[10] are connected to your provided email address.

187 COMMENTS Newest

Diddy
June 24, 2025 9:30 pm PDT

WOW! this is really incredible

0 Reply

Wide
June 11, 2025 9:30 pm PDT

Say const one more time...

0 Reply

RSH
May 21, 2025 10:20 pm PDT

Const this const that, how bout const auto*{skiplessonptr()}

6 Reply

Zeca
April 22, 2025 6:41 pm PDT

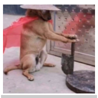
MY BRAIN WILL EXPLODE

20 Reply

smart
April 8, 2025 4:51 pm PDT

I actually enjoy that I'm not very proficient in English. I often find myself translating ambiguous phrases and exploring why something is called that, as well as how it's translated into my language. In the end, I remember everything much better.

3 Reply



LechugaPlayer
April 1, 2025 5:13 am PDT

yes

0 Reply

Mr. F

Mr. F
March 29, 2025 9:30 pm PDT

const const const const goddayum, i swear to god, if u mention const again... :v

Last edited 3 months ago by Mr. F

6 Reply

> 
>
> **Zeca**
> Reply to Mr. F [11] April 22, 2025 6:39 pm PDT
>
> xD. I was thinking the same thing. STOPPPPP THE CONST PLEEEASE
>
> 2 Reply

Aklyseus
March 13, 2025 5:00 am PDT

Is it me or there is a contradiction in the summary when it says:

`auto const` and `const auto` both make the deduced pointer a const pointer. **There is no way to explicitly specify a low-level const (pointer-to-const) using auto**.
`auto* const` also makes the deduced pointer a const pointer. **`const auto*` makes the deduced pointer a pointer-to-const.**

So in the second bullet you mention there is no way to specify a pointer-to-const, and then "`const auto*` makes the deduced pointer a pointer-to-const."

0 Reply

> 
>
> **vitrums**
> Reply to Aklyseus [12] April 5, 2025 12:48 am PDT

Both `auto const` and `const auto` semantically mean `const (type)`, i.e. make `type` const. As in `const (const int*)` or `const (int*)` resolving into `const int* const` and `int* const` accordingly. Hence there is no way to affect the low-level const by deducing with **naked** auto without an asterisk.

0 Reply

lukas m

February 15, 2025 5:35 pm PST

yeah i skip this lesson i wont need it XD HAHA

2 Reply

Will

January 8, 2025 6:22 pm PST

This lesson was rough, but I think I get the idea of it. Might be helpful to make a summary list of rules in order that may better help determine who gets what const. Maybe this is as clear as it can be made. Thanks, Alex!

2 Reply

Alex Author

Reply to Will [13] January 21, 2025 8:06 pm PST

I added a summary at the bottom.

6 Reply

Robert

Reply to Alex [14] May 27, 2025 11:03 am PDT

I adore your content Alex is so fantastic, but in this case I think that things like const-ness, const pointers, pointers to const It would have been fantastic to explain them in previous chapters using small graphics simulating memory pointers, what const-ness meant in each case. Perhaps we would have arrived at this lesson with a clearer mental map. In any case, everything you've done has been a gem.

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/type-deduction-for-objects-using-the-auto-keyword/
3. https://www.learncpp.com/cpp-tutorial/lvalue-references-to-const/
4. https://www.learncpp.com/cpp-tutorial/stdoptional/
5. https://www.learncpp.com/
6. https://www.learncpp.com/cpp-tutorial/in-and-out-parameters/
7. https://www.learncpp.com/type-deduction-with-pointers-references-and-const/
8. https://www.learncpp.com/cpp-tutorial/passing-and-returning-structs/
9. https://www.learncpp.com/cpp-tutorial/introduction-to-random-number-generation/
10. https://gravatar.com/
11. https://www.learncpp.com/cpp-tutorial/type-deduction-with-pointers-references-and-const/#comment-608889
12. https://www.learncpp.com/cpp-tutorial/type-deduction-with-pointers-references-and-const/#comment-608508
13. https://www.learncpp.com/cpp-tutorial/type-deduction-with-pointers-references-and-const/#comment-606388
14. https://www.learncpp.com/cpp-tutorial/type-deduction-with-pointers-references-and-const/#comment-606829
15. https://g.ezoic.net/privacy/learncpp.com