12.14 — Type deduction with pointers, references, and const

In lesson 10.8 -- Type deduction for objects using the auto keyword, we discussed how the `auto` keyword can be used to have the compiler deduce the type of a variable from the initializer:

```
int getVal(); // some function that returns an int by value

int main()
{
    auto val { getVal() }; // val deduced as type int

    return 0;
}
```

We also noted that by default, type deduction will drop `const` (and `constexpr`) qualifiers:

```
const double foo()
{
    return 5.6;
}

int main()
{
    const double cd{ 7.8 };

    auto x{ cd };    // double (const dropped)
    auto y{ foo() }; // double (const dropped)

    return 0;
}
```

Const (or constexpr) can be (re)applied by adding the `const` (or `constexpr`) qualifier in the definition:

```
const double foo()
{
    return 5.6;
}

int main()
{
    constexpr double cd{ 7.8 };

    const auto x{ foo() };  // const double (const dropped, const reapplied)
    constexpr auto y{ cd }; // constexpr double (constexpr dropped, constexpr
reapplied)
    const auto z { cd };    // const double (constexpr dropped, const applied)

    return 0;
}
```

Type deduction drops references

In addition to dropping const qualifiers, 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 the dropped `const` qualifier, 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 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
```

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 (or constexpr), 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 either or both of these:

```
#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
reapplied)

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

    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?

These work the same way as const references:

```
#include <string_view>

constexpr std::string_view hello { "Hello" };

constexpr const std::string_view& getConstRef()
{
    return hello;
}

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

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

```
#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;
}
```

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 whatever the deduced type is const". 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 type 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, reapply the `const` qualifier even when it's not strictly necessary, as it makes your intent clear and helps prevent mistakes.