The `auto` Keyword: The Rest of the Story

Consider this example of using `auto`:

```
int x = int(); // x is an int, initialized to 0
assert(x == 0);

const int& crx = x; // crx is a const int& that refers to x
x = 42;
assert(crx == 42 && x == 42);

auto something = crx;
```

The crucial question is, what is the type of `something`? Since the declared type of `crx` is `const int&`, and the initializing expression for `something` is `crx`, you might think that the type of `something` is `const int&`. Not so. It turns out that the type of `something` is `int`:

```
assert(something == 42 && crx == 42 && x == 42);
// something is not const:
something = 43;
// something is not a reference to x:
assert(something == 43 && crx == 42 && x == 42);
```

Before we discuss the rationale behind this behavior, let us state the exact rules by which `auto` infers the type from an initializing expression:

When `auto` sets the type of a declared variable from its initializing expression, it proceeds as follows:

1. If the initializing expression is a reference, the reference is ignored.
2. If, after Step 1 has been performed, there is a top-level `const` and/or `volatile` qualifier, it is ignored.

There will be two small amendments to this rule, stemming from adorning `auto` with qualifiers and references, as explained below.

As you have probably noticed, the rules above look like the ones that function templates use to deduce the type of a template argument from the corresponding function argument. There is actually a small difference: `auto` can deduce the type `std::initializer_list` from a C++11-style braced list of values, whereas function template argument deduction cannot. Therefore, you may use the rule "`auto` works like function template argument deduction" as a first intuition and a mnemonic device, but you need to remember that it is not quite accurate.

Continuing with the example above, suppose we pass the `const int&` variable `crx` to a function template:

```
template<class T>
void foo(T arg);
foo(crx);
```

Then the template argument `T` resolves to `int`, not to `const int&`. So for this instantiation of `foo`, the argument `arg` is of type `int`, not `const int&`. If you want the argument `arg` of `foo` to be a `const int&`, you can achieve that either by specifying the template argument at the call site, like this:

```
foo<const int&>(crx);
```

or by declaring the function like this:

```
template<class T>
void foo(const T& arg);
```

The latter option works analogously with `auto`:

```
const auto& some_other_thing = crx;
```

Now `some_other_thing` is a `const int&`, and that is of course true regardless of whether the initializing expression is an `int`, an `int&`, a `const int`, or a `const int&`.

```
assert(some_other_thing == 42 && crx == 42 && x == 42);
some_other_thing = 43;  // error, some_other_thing is const
x = 43;
assert(some_other_thing == 43 && crx == 43 && x == 43);
```

We're now in a position to state the two aforementioned amendments to `auto`'s type deduction rules.

First Amendment
Consider this example:

```
const int c = 0;
auto& rc = c;
rc = 44; // error: const qualifier was not removed
```

If you went strictly by the rules stated earlier, `auto` would first strip the const qualifier off the type of `c`, and then the reference would be added. But that would give us a non-const reference to the const variable `c`, enabling us to modify `c`. Therefore, `auto` refrains from stripping the const qualifier in this situation. This is of course no different from what function template argument deduction does.

Second Amendment
The second amendment concerns the speical case where `auto` is adorned with an rvalue reference.

```
int i = 42;
auto&& ri_1 = i;
auto&& ri_2 = 42;
```

In both cases, the initializing expression is of type `int`. Therefore, you would, absent any special rule, assume that both `ri_1` and `ri_2` are of type `int&&`. Not so. Adorning `auto` with an rvalue reference causes its type deduction to work differently, as follows:

- If the intializing expression is an lvalue, an `&&`-adorned `auto` first performs its ordinary type deduction, then adds an lvalue reference to that.
- If the intializing expression is an rvalue, an `&&`-adorned `auto` just performs its ordinary type deduction.

To understand what the end result of that is, let's look at the example again. In the case of `ri_1`, where

```
int i = 42;
auto&& ri_1 = i;
```

`auto` first deduces the type `int`, obviously. Then it sees that `i` is an lvalue. Therefore, in its second step, `auto` adds an lvalue reference, ending up with type `int&`. Together with the adornment `&&`, that gives `int& &&`. By the reference collapsing rules that were introduced with C++11, `& &&` collapses to `&`, and therefore, the end result is `int&`. In the case of `ri_2`, where

```
auto&& ri_2 = 42;
```

`auto` again deduces the type `int`. Then it sees that `42` is an rvalue, and thus, no further processing takes place. Due to the adornment `&&`, the end result is `int&&`.

This is not the place to get into the rationale behind this second amendment. To get more information, all you need to know is that the buzzword to search for is "universal reference," a term that was coined by Scott Meyers. Lately, there has been a tendency to use the term "forwarding reference" instead, so you may want to search for that one as well.

Let's do one more example to demonstrate that `auto` drops `const` and `volatile` qualifiers only if they're at the top or right below an outermost reference:

```
int x = 42;
const int* p1 = &x;
auto p2 = p1;
*p2 = 43; // error: p2 is const int*
```

Now that we know how `auto` works, let's discuss the rationale behind the design. There is probably more than one way to argue. Here's my way to see why `auto`'s qualifier- and reference-stripping behavior is plausible: being a reference is not so much a type characteristic as it is a behavioral characteristic of a variable. The fact that the expression from which I initialize a new variable behaves like a reference does not imply that I want my new variable to behave like a reference as well. Similar reasoning can be applied to constness and volatility[1]. Therefore, `auto` does not automatically transfer these characteristics from the initializing expression to my new variable. I have the option to give these characteristics to my new variable, by using syntax like `const auto&`, but it does not happen by default.

---

[1]It is perhaps worth noting that C++11 does *not* apply this reasoning to constness and volatility when it comes to closures: when a lambda captures a const local, the copy in the closure is again const. The same is true for volatile.

---

1 2 3 4 5 6 7 8 9 10 11 12

FOLLOW ME ON twitter