4.8 — The auto keyword

BY ALEX ON JULY 30TH, 2015 | LAST MODIFIED BY ALEX ON MAY 31ST, 2017

Auto prior to C++11

Prior to C++11, the auto keyword was probably the least used keyword in C++. In lesson **4.1a -- Local variables and local scope**, you learned that local variables have automatic duration (they're created at the point of definition, and destroyed when the block they are part of is exited).

The auto keyword was a way to explicitly specify that a variable should have automatic duration:

```
int main()
{
    auto int foo(5); // explicitly specify that foo should have automatic duration

    return 0;
}
```

However, since all variables in modern C++ default to automatic duration unless otherwise specified, the auto keyword was superfluous, and thus obsolete.

Type inference in C++11

In C++11, the meaning of the auto keyword has changed, and it is now a useful addition to your C++ vocabulary.

Consider the following statement:

```
double d = 5.0;
```

If C++ already knows 5.0 is a double literal, why do we have to explicitly specify that d is actually a double? Wouldn't it be nice if we could tell a variable to just assume the proper type based on the value we're initializing it with?

Starting with C++11, the auto keyword does just that. When initializing a variable, the auto keyword can be used in place of the variable type to tell the compiler to infer the variable's type from the initializer's type. This is called **type inference** (also sometimes called type deduction).

For example:

```
auto d = 5.0; // 5.0 is a double literal, so d will be type double
auto i = 1 + 2; // 1 + 2 evaluates to an integer, so i will be type int
```

This even works with the return values from functions:

```
int add(int x, int y)
{
    return x + y;
}

int main()
{
    auto sum = add(5, 6); // add() returns an int, so sum will be type int
    return 0;
}
```

Note that this only works when initializing a variable upon creation. Variables created without initialization values can not use this feature (as C++ has no context from which to deduce the type).

While using auto in place of fundamental data types only saves a few (if any) keystrokes, in future lessons we will see examples where the types get complex and lengthy. In those cases, using auto can be very nice.

The auto keyword can't be used with function parameters

Many new programmers try something like this:

```cpp
#include <iostream>

void addAndPrint(auto x, auto y)
{
    std::cout << x + y;
}
```

This won't work, because the compiler can't infer types for function parameters x and y at compile time.

If you're looking to create functions that work with a variety of different types, you should be using function templates, not type inference. This restriction may be lifted in future versions of C++ (with auto acting as a shorthand way to create function templates), but as of C++14 this is not supported. The one exception is for lambda expressions, which is an advanced C++ topic.

Type inference for functions in C++14

In C++14, the auto keyword was extended to be able to auto-deduce a function's return type. Consider:

```cpp
auto add(int x, int y)
{
    return x + y;
}
```

Since x + y evaluates to an integer, the compiler will deduce this function should have a return type of int.

While this may seem neat, we recommend that this syntax be avoided for functions. The return type of a function is of great use in helping to document for the caller what a function is expected to return. When a specific type isn't specified, the caller may misinterpret what type the function will return, which can lead to inadvertent errors.

Interested readers may wonder why using auto when initializing variables is okay, but not recommended for function return types. A good rule of thumb is that auto is okay to use when defining a variable, because the object the variable is inferring a type from is right there, on the right side of the statement. However, with functions, that is not the case -- there's no context to help indicate what type the function returns. A user would actually have to dig into the function body itself to determine what type the function returned. It's much less intuitive, and therefore more error prone.

Trailing return type syntax in C++11

C++11 also added the ability to use a **trailing return syntax**, where the return type is specified after the rest of the function prototype.

Consider the following function declaration:

```cpp
int add(int x, int y);
```

In C++11, this could be equivalently written as:

```cpp
auto add(int x, int y) -> int;
```

In this case, auto does not perform type inference -- it is just part of the syntax to use a trailing return type.

Why would you want to use this?

One nice thing is that it makes all of your function names line up:

```cpp
auto add(int x, int y) -> int;
auto divide(double x, double y) -> double;
auto printSomething() -> void;
auto calculateThis(int x, double d) -> std::string;
```

But it is of more use when combined with some advanced C++ features, such as classes and the decltype keyword. We'll talk more about the other auto uses when we cover the decltype keyword.

For now, we recommend the continued use of the traditional function return syntax.

Summary

Starting with C++11, the auto keyword can be used in place of a variable's type when doing an initialization in order to perform type inference.