Type Inference

The first thing I'm going to talk about is type inference. C++11 provides mechanisms which make the compiler deduce the types of expressions.

These features allow you to make your code more concise, more flexible, and to remove some of the redundancy in the form of repeated type names.

There are actually two different constructs, each with its own keyword: `auto` and `decltype`.

As you will see, there is some overlap between them but generally they are intended for different uses. `auto` is limited to declaring variables, while `decltype` is a more general tool.

auto

I'm sure you have been annoyed at least once by having to type out types like this:

```
std::map<std::string, std::vector<int>>::iterator
```

And this isn't even a particularly bad example. The verbosity of type names, particularly when templates are involved, can be quite a drag.

The new keyword `auto` solves this problem by inferring the type of a variable from the initializing expression:

```
auto a = 5;
auto plane = JetPlane("Boeing 737");
cout << plane.model();
for (auto i = plane.engines().begin(); i != plane.engines().end(); ++i)
    i->set_power_level(Engine::max_power_level);
```

Variables declared with `auto` still have a static type just like any other variables. The compiler needs to have enough information to figure the type out - and this is provided by the initializing expression.

In this example, a is going to be an int, and plane is an instance of the JetPlane class. The loop variable i is going to be an iterator type. As you can see, working with STL iterators becomes quite a bit nicer this way.

To be precise, the keyword auto isn't really a new keyword. It has actually been in C++ since the beginning, and used to mean "this is a local variable". However, that meaning has proven to be of no use and has now been removed from the language.

Some things are still manual

Because you need to provide sufficient type information to auto, it can't be used everywhere. None of these examples will compile:

```
void invalid(auto i) {}      // error

class A
{
    auto _m;                 // error
};

int main()
{
    auto arr[10];            // error
}
```

You can't use auto for function parameters or member variables, or declare arrays with auto.

More than syntactic sugar

auto isn't merely syntactic sugar, however. In some situations it can be not just inconvenient, but actually difficult or impossible to write C++11 code without auto. For example, when the code in a template depends on the types of the arguments, as in this example, it can be hard to specify the type explicitly.

```cpp
template<typename X, typename Y>
void multiply(const X& x, const Y& y)
{
    auto result = x * y; // without auto, it can be hard to specify the type
                         // of result
    // ...
}
```

The compiler, on the other hand, can choose the appropriate types once it knows the types of the arguments.

Why else do we need it?

So why else would you use `auto` in your code? Let's look at the full list of reasons. Using this keyword has many advantages over specifying types explicitly:

- It reduces verbosity of the code and makes it more DRY. DRY stands for Don't Repeat Yourself. This very useful principle was defined in the book called The Pragmatic Programmer by Dave Thomas and Andy Hunt.
- It also promotes a higher level of abstraction and coding to interfaces.

- It helps future-proof the code as it allows types to change without necessarily requiring code that uses them to change.
- It allows for easier refactoring, e.g. changing class names or function return types.
- It allows template code to be simpler because now you can avoid specifying some of the intermediate expression types by using `auto`.
- It's much easier to declare variables of undocumented types.
- It allows you to store lambda expressions for later use. I'll talk about lambdas in a later chapter, but the key thing here is that you can't know the type of lambda expressions so you can't declare a variable to assign a lambda expression to without `auto`.

In line with the recommendation from Scott Meyers, I suggest always using `auto` unless you require a type conversion, that is, when the type inferred by `auto` wouldn't be the type you want. A type conversion may be required, for example, when you get some proxy type from a template and you want it to decay.

Objection, Your Honor!

At this point you might be disagreeing. A common objection to `auto` is that it obscures type information and makes code hard to comprehend for a new programmer. It is a valid consideration, but I believe the benefits I've outlined outweigh it.

The fact that the types of variables are less obvious is mitigated to a large degree by code introspection tools in the modern editors, and by the improved readability achieved by reducing type name duplication throughout the code.

It's also worth considering that similar objections have been made for a long time about other C++ features such as operator overloading, and yet these features contribute to more powerful and expressive code. I believe that `auto` is in the same camp.

Diving in

Now that you're hopefully convinced of the merits of `auto`, let's look at the details of using it.

Using `auto`, you can declare multiple variables in the same statement, as long as all the initializing expressions result in the same deduced type:

```
auto a = 5.0, b = 10.0;
auto i = 1.0, *ptr = &a, &ref = b;   // regular variable, pointer and reference
                                     // declarations
auto j = 10, str = "error";          // error: initializing expressions of
                                     // different types
```

The first line defines `a` and `b` as doubles. The second line defines a regular variable, a pointer and a reference. The third line will cause a compilation error because the initializing expressions have different types.

See how on the second line I had to add pointer and reference qualifiers? `auto` infers the unadorned type, so if you want a reference or a pointer, you have to specify it explicitly:

```
map<string, int> index;

const auto j = index;
auto& ref = index;
```

```
const auto& cref = index;
auto* ptr = &index;
```

Similarly, you can add `const` and `volatile` qualifiers if you need them.

If you aren't declaring a reference, some type transformations are applied automatically:

- `const` and `volatile` specifiers are removed from the initializing type
- arrays and functions are turned into pointers

Let's look at a couple of examples:

```
const vector<int> values;
auto a = values;         // type of a is vector<int>
auto& b = values;        // type of b is const vector<int>&
```

Here I have a `const` vector called `values`. The type of `a` is going to be `vector<int>` without the `const`. `b` is going to be a `const` reference to the vector.

If I have a `volatile` variable, and I assign it to an `auto` variable, `volatile` isn't included in the inferred type:

```
volatile long clock = 0;
auto c = clock;     // c is not volatile
```

Next, suppose I have an array:

```
JetPlane fleet[10];
auto e = fleet;     // type of e is JetPlane*
auto& f = fleet;    // type of f is JetPlane(&)[10] - a reference
```

When I assign it to a plain `auto` variable `e`, `e` is going to be a pointer. If I add a reference specifier as I've done with `f`, then I'm going to get a reference to the array.

Finally, I can use `auto` with functions:

```cpp
int func(double) { return 10; }
auto g = func;       // type of g is int(*)(double)
auto& h = func;      // type of h is int(&)(double)
```

Similarly to arrays, a plain `auto` declaration gives me a pointer to the function, and if I want a reference, I need to specify it explicitly.

You can use either assignment or copy initialization syntax to initialize your `auto` variables:

```cpp
int i = 10;

auto a = i;
auto b(i);
```

However, generally you should prefer using the assignment syntax due to the way `auto` interacts with `initializer_list`. I'll talk about `initializer_list` later in the book.

One exception to this rule is if the inferred type has an explicit copy constructor:

```cpp
struct Expl
{
    Expl() {}
    explicit Expl(const Expl&) {}
};

Expl e;
auto c = e;  // illegal
```

In this situation you can only use copy initialization.

A little bit of `auto` history

This is all you need to know to use `auto`, and I just want to mention a couple of interesting things before moving on. `auto` happens to have a long history. According to Bjarne Stroustrup, he had the `auto` feature working all the way back in 1984, but had to remove it because of C compatibility issues. Those issues went away when C++98 and C99 started requiring every variable and function to be defined with an explicit type.

Even in the absence of `auto`, compilers have had the ability to infer types for a long time as they had to figure out the types of template arguments.

```
template<typename T>
void f(T t)
{}

f(expr);            // T is deduced from expr
auto var = expr;    // type of var is deduced from expr, same as above
```

The mechanism behind the type inference performed for `auto` declarations is the same as that used for templates.

Let's now take a look at the second mechanism for type inference, 'decltype'.

decltype

The second construct that deals with inferring types is the `decltype` specifier. You pass an expression to it, and it figures out the type of the expression:

```
auto i = 10;
cout << typeid(decltype(i + 1.0)).name() << endl; // outputs "double"
```

In this example, `decltype` infers the type of expression to be `double`, so the `typeid.name` outputs `double`.

Compared to `auto`, `decltype` is a more general purpose construct. While `auto` can only be used in definitions, `decltype` is meant to be a type specifier, so the intention is for you to be able to use `decltype(expr)` in place of a type name anywhere. For example, in the following bit of code, I'm using `decltype(a)` instead of `vector<int>`:

```
vector<int> a;
decltype(a) b;
b.push_back(10);
decltype(a)::iterator iter = a.end();
```

This characteristic of `decltype` is particularly useful when writing templated functions where the return type depends on the template arguments:

```cpp
template<typename X, typename Y>
auto multiply(X x, Y y) -> decltype(x * y)
{
    return x * y;
}
```

This function definition might look strange to you because uses the new syntax for declaring functions with a trailing return type. I'll explain this syntax in a few minutes. For now, the point is that in C++11 I have the ability to define a function template with a return type which is determined by its template arguments.

Side effects

`decltype` does *not* evaluate the expression that's given to it. So, for example, here the value of a won't change outside the context of `decltype`:

```cpp
auto a = 10;
decltype(a++) b;
cout << a << endl;          // outputs 10
```

Nonetheless, even though the expression isn't evaluated, `decltype` can still have a potential side effect due to template instantiation. Consider this example:

```cpp
template <int I>
struct Num
{
    static const auto c = I;
    decltype(I) _member;
    Num() : _member(c) {}
};
int i;
decltype(Num<1>::c, i) var = i;      // type of var is int&
```

I've passed an expression which uses the comma operator to `decltype`. The comma operator returns the last argument, so `var` will have the same base type as `i`, but the compiler will still need to instantiate the `Num` template to make sure the expression is valid. This template instantiation is a side effect.

`declval`

The expression passed to `decltype` has to be valid. This means I can have a problem when a class with a private constructor is involved:

```
class A
{
private:
    A();
};

cout << typeid(decltype(A())).name() << endl;  // doesn't compile: A() is private
```

My class `A` has a private constructor, and I attempt to use the constructor call in `decltype`. Even though the expression isn't evaluated, and the type I want is clearly `A`, this doesn't compile, because the expression isn't valid.

This is where `declval` can come to my rescue. `declval` is a standard template which is provided for just such situations:

```
cout << typeid(decltype(declval<A>())).name() << endl;  // OK
```

With the addition of `declval`, my `decltype` expression now compiles. Keep in mind that `declval<A>` actually yields an *rvalue reference* to `A`, which is a new type of reference added in C++11. I will discuss rvalue references in detail when I talk about *move semantics*, another new concept in the language.

One more thing to keep in mind is that `declval` can only be used in an unevaluated operand, for example with `decltype`. If you attempt to use it where it has to be evaluated, you will get a compilation error.

`auto`, `decltype` - how about both at once?

When `auto` and `decltype` are combined in a function definition, you end up with something altogether different. The keyword `auto` can be used in function declarations and definitions to indicate a trailing return type, that is, the new syntax variation where the return type is specified after the parameters. It looks like this:

```
template<typename X, typename Y>
auto multiply(X x, Y y) -> decltype(x * y)
{
    return x * y;
}
```

You've already seen this definition previously when I was talking about `decltype`.

So why is this a useful feature? The reason for introducing this syntax is that it allows you to declare templated functions where the return type depends on the type of the arguments. With the old syntax, there was a scoping problem. Consider this function template definition:

```
template<typename X, typename Y>
ReturnType multiply(X x, Y y)
{
    return x * y;
}
```

I want `ReturnType` to be the type of the result of `(x * y)` but I have no way of specifying it, because `x` and `y` aren't in scope yet. Because of this, I can't use `decltype` like this:

```
template<typename X, typename Y>
decltype(x * y) multiply(X x, Y y)    // x and y in decltype aren't in scope yet!
{
    return x * y;
}
```

This is where trailing return types are useful. Moving the return type after the parameter list allows me to use the parameters in the `decltype` expression, which gives me the return type.

Other than saying "use trailing return type syntax when needed", I won't give you a guideline on when to use it. Some people suggest using the new syntax everywhere because of the stylistic and naming advantages it offers, but it's definitely not prevalent at this point.

That's all I had to say about type inference, so let's move on to the next major feature, lambda expressions.