Type Inference

C++11 provides mechanisms for type inference which make the compiler deduce the types of expressions. I'm starting the book with type inference because it can make your code more concise and expressive. Also, a lot of examples in subsequent chapters rely on these features.

`auto`

I'm sure everyone has been annoyed at least once by having to type out types like `std::map<std::string, std::vector<int>>::iterator`. The new keyword `auto` solves this problem by inferring the type of a variable from the initializing expression:

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

> **Note:**
>
> The keyword `auto` has 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 been removed from the language.

> **Note:**
>
> 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.

`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, it can be hard to specify the type explicitly. But the compiler can choose the appropriate types once it knows the types of the arguments:

```
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
    // ...
}
```

You should be aware that variables declared with `auto` still have a static type just like any other variables, and the compiler needs to have enough information (in the form of an initializing expression) to figure the type out. This is why you can't declare arrays with `auto`, or use it for function parameters or member variables. None of the following will compile:

```
void invalid(auto i) {}

class A
{
    auto_m;
};

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

Using the keyword has a lot of advantages over specifying types explicitly:

- it reduces verbosity of the code and makes it more DRY
- it 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 types for intermediate expressions by using `auto`
- it's much easier to declare variables of undocumented types
- it allows you to store lambdas for later use. I'll talk about lambdas in the next 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`.

  Note:

  DRY stands for Don't Repeat Yourself. This principle was defined in the book called The Pragmatic Programmer by Dave Thomas and Andy Hunt. It promotes reduced repetition of information.

As a rule of thumb, you should always use `auto` unless you require a type conversion, i.e. 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.

A common objection to `auto` is that it obscures the type information and makes code hard to comprehend for a new programmer. It is a valid consideration, but the benefits outlined above outweigh it. The fact that the types of variables are less obvious is mitigated to a large degree by IntelliSense and by the improved readability achieved by not having type name duplication all over the code.

Consider that similar objections have been made for a long time about other C++ features such as operator overloading, and yet they contribute to more powerful and expressive code. `auto` is in the same camp.

An interesting thing about `auto` is that the compilers have had the ability to infer types for a long time as they had to figure out the types of template arguments. The mechanism behind `auto` is the same as that used for templates:

```
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
```

`auto` can be used to declare multiple variables, 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
```

You can add `const` or `volatile` qualifiers to your `auto` declarations, as well as turn them into a pointer, a reference or an rvalue reference (rvalue references will be discussed in a later chapter):

```
map<string, int> index;

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

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

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

For example:

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

volatile longclock = 0;
auto c = clock;     // c is not volatile

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

auto g = func;      // type of g is int(*)(double)
auto& h = func;     // type of h is int(&)(double)
```

You can use either assignment or copy initialization syntax to initialize your `auto` variables, unless the inferred type has an explicit copy constructor:

```
int i = 10;

auto a = i;
auto b(i);

struct Expl
{
    Expl() {}
    explicit Expl(constExpl&) {}
};

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

Decltype

Another 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:

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

Note that it does *not* evaluate the expression, so in the example below the value of `a` won't change outside the context of `decltype`:

```
decltype(a++) b;
```

`decltype` is considered 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:

```
vector<int> a;
decltype(a) b;
b.push_back(10);

decltype(a)::iterator iter = a.end();
```

It is particularly useful when writing templated functions where the return type depends on the template arguments:

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

The example above uses the new syntax for declaring functions with a trailing return type. This is discussed in detail below.

Because `decltype` is a type specifier, it ignores `const` and `volatile` qualifiers. If you want to get a `const` reference from `decltype`, put the expression into parentheses:

```
is_reference<decltype((i))>::value;  // evaluates to true
```

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`:

```
auto a = 10;
decltype(a++) b;
cout << a << endl;          // outputs 10 as value of a is unchanged
```

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

```
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 passed an expression with the comma operator to `decltype` in the last statement of this example. 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, which you may sometimes prefer not to happen.

Trailing return types

The keyword `auto` has a different meaning when it's used in conjunction with function declarations. C++11 introduces an alternative syntax for declaring functions, where the return type is specified *after* the parameters (hence *trailing return type*). In order to use this syntax, you have to start your function declaration with `auto`:

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

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:

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

I want `ReturnType` to be the type of `(x * y)` but I have no way of specifying it, because `x` and `y` aren't in scope yet:

```
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 come to the rescue. Putting the return type after the parameter list allows the parameters to be used to specify the return type.

Note:

There is a hack which allows you to use type names instead of the parameter names, but it's ugly and error-prone:

```
template<class X, class Y>
decltype(*(X*)(0) * *(Y*)(0))
mul2(X x, Y y)
{
    return x * y;
}
```