C++ auto and decltype Explained

By Thomas Becker about me contact

Last updated: May 2013

Contents

1. Introduction
2. The `auto` Keyword: The Basics
3. The `auto` Keyword: The Rest of the Story
4. When `auto` Is Not a Luxury
5. The `decltype` keyword: The Basics
6. How `decltype` Deduces the Type of an Expression: Case 1
7. How `decltype` Deduces the Type of an Expression: Case 2
8. An Example to Put You on Guard
9. Miscellaneous Properties of `decltype`
10. Summary and Epilogue
11. Acknowledgments

Introduction

Most of the language features that were introduced with the C++11 Standard are easy to learn, and the benefit that they provide is quite obvious. Looking at them doesn't make you think, "Oh no, now I'm going to have to learn this." Instead, your immediate reaction is, "Oh great, I've always wanted that." In some cases, like lambdas, the syntax is a bit hairy, but that's of course not much of a problem. For me, the one big exception to all this were rvalue references. I had a really hard time wrapping my head around those. Therefore, I did what I always do when I need to understand something: I scraped up all the information I could get my hands on, then wrote an article on the subject. It turned out that a lot of people were having the same problem, and the article helped quite a few of them. Great, everybody's happy.

A while later, sometime in 2012, I noticed that there was another feature, or rather, a pair of features, in C++11 that I had not fully understood, namely, the `auto` and `decltype` keywords. With `auto` and `decltype`, unlike rvalue references, the problem is not that they are difficult to grasp. On the contrary, the problem is that the idea is deceptively easy, yet there are hidden subtleties that can trip you up.

Let's start with a good look at the `auto` keyword.

The `auto` Keyword: The Basics

Consider the following code snippet:

```
std::vector<int> vect;
// ...

for(
  std::vector<int>::const_iterator cit = vect.begin();
  cit != vect.end();
  ++cit
  )
{
  std::cout << *cit;
}
```

Perfectly fine, we all write code like this or similar all the time. Now let us consider this instead:

```
std::vector<int> vect;
// ...

for(
  std::vector<int>::iterator it = vect.begin();
  it != vect.end();
  ++it
  )
{
  std::cin >> *it;
}
```

When writing this code, you should be getting slightly annoyed. The variable `it` is of the exact same type as the expression `vect.begin()` with which it is being initialized, and the compiler knows what that type is. Therefore, for us to have to write `std::vector<int>::iterator` is redundant. The compiler should use this as the default for the type of the variable `it`. And that is exactly the issue that the `auto` keyword addresses. In C++11, we can write:

```
std::vector<int> vect;
// ...

for(auto it = vect.begin(); it != vect.end(); ++it)
{
  std::cin >> *it;
}
```

The compiler will use the type of the initializing expression `vect.begin()`, which is `std::vector<int>::iterator`, as the type of the variable `it`.

This sounds easy enough, but it's not quite the full story on `auto`. Read on.

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.

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);
```

The possibility of adding a reference to the `auto` keyword as shown in the example above requires a small amendment to `auto`'s const-stripping behavior. 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.

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 the 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.

When `auto` Is Not a Luxury

In the examples that we have seen so far, the use of the `auto` keyword is a convenience that saves us some unnecessary thinking and typing and makes the code look less cluttered. `auto` also allows us to do things that aren't possible without it[1]. This happens whenever you want to declare a variable whose type is the same as the type of some expression that involves templated variables. For example, suppose you're writing a function template with two arguments whose precondition is that the product of the two arguments is defined. Inside the function, you want to declare a variable to hold the product of the two arguments:

```
template<typename T, typename S>
void foo(T lhs, S rhs) {
  auto prod = lhs * rhs;
  //...
}
```

In standard C++ prior to C++11, this could not be done at all, because the type of the product is something that the compiler must infer every time the function template is instantiated. For example, if both `lhs` and `rhs` are `int`s, then the type of the product is `int`. If one of the two is an `int` and the other is a `double`, then the `int` gets promoted to a `double`, and the type of the product is `double`. Infinitely many more examples are possible using user-defined types and multiplication operators.

---

[1]Strictly speaking, `auto` is never really necessary: in theory, it can always be replaced with the more powerful `decltype`. For example, the line

```
auto prod = lhs * rhs;
```

in the code example above could be replaced with

```
decltype(lhs * rhs) prod = lhs * rhs;
```

The latter is clearly less elegant than the former. Moreover, we'll see in the next few sections that the way `decltype` deduces the type of an expression is different from the way `auto` does. If you really wanted to shun `auto` altogether and use `decltype` instead, you'd have to account for these differences using things like `remove_reference` in all the right places. That would become tedious and error-prone.

The `decltype` Keyword: The Basics

Let's take another look at the example of the previous section, where we wrote a function template with two arguments whose precondition was that the product of the two arguments was defined. Inside the function, we declared a variable that could hold the product of the two arguments:

```
template<typename T, typename S>
void foo(T lhs, S rhs) {
  auto prod = lhs * rhs;
  //...
}
```

Now suppose that instead of declaring a variable whose type is that of the product of the arguments, we want to make a typedef for that type. Since the compiler knows what the type is, we should be able to do that. Before C++11, there was no official way of doing it. But some compilers had an extension keyword `typeof` for that purpose:

```
template<typename T, typename S>
void foo(T lhs, S rhs) {
  // Pre-C++11 compiler extension, now obsolete
  typedef typeof(lhs * rhs) product_type;
  //...
}
```

The purpose of `decltype` is to provide a standardized version of `typeof`. Since the result is not identical to the existing compiler extension `typeof` (and there were probably conflicting versions out there, I'm not sure), the term `typeof` could not be used. Instead, the previously unused term `decltype` was chosen. So in C++11, we can now write

```
template<typename T, typename S>
void foo(T lhs, S rhs) {
  typedef decltype(lhs * rhs) product_type;
  //...
}
```

Another situation where `decltype` comes in handy is when we want the return type of a function to be something that needs to be deduced from an expression. For example, let us try and modify the example above in such a way that it returns the product of its arguments. Naively, you would perhaps try this:

```
template<typename T, typename S>
// Does not compile: lhs and rhs are not in scope
decltype(lhs * rhs) multiply(T lhs, S rhs) {
```

```
  return lhs * rhs;
}
```

This won't compile because `lhs` and `rhs` are not in scope preceding the function name. To fix this, C++11 introduces what's called the *trailing return type* syntax:

```
template<typename T, typename S>
auto multiply(T lhs, S rhs) -> decltype(lhs * rhs) {
  return lhs * rhs;
}
```

This will compile and make the type of the product the function's return type. This is nothing to be intimidated by. It's just a bit of syntactic trickery to allow the compiler to grab the type of the expression `lhs * rhs` and make it the function's return type.

Note that the keyword `auto` that is used here is not the `auto` that we discussed in the previous three sections. The use of `auto` in this context just continues the time-honored practice of C and C++ to use the same lexical token for multiple purposes. Actually, I personally am a bit uncomfortable with the use of `auto` in this context. I'll tell you why later, after we have learned more about `decltype`.

When I had gotten this far in my studies of C++11, I was convinced that the following was a true statement, which it is not:

<wrong>
`decltype` deduces the type of an expression just like `auto` does. The difference is that `decltype` is applicable in a wider variety of contexts.
</wrong>

Read on to see just how wrong this is.

How `decltype` Deduces the Type of an Expression: Case 1

The way `decltype` goes about determining the type of an expression is different from what `auto` does. It is based on a case distinction. There are two cases. The first case is when the expression whose type is to be determined is a plain variable or function parameter, like `x`, or a class member access, like `p->m_x`. In that case, `decltype` lives up to its name: it determines the type of the expression to be the *declared type*, that is, the type that we find in the source code at the point of declaration. When I say "find in the source code," that may of course involve going through some levels of indirection, like resolving typedefs or performing template instantiations. But apart from that, we are talking about a situation where the type of the epxression is lexically present in the source code.

> If `expr` is a is a plain, unparenthesized variable, function parameter, or class member access, then `decltype(expr)` is the type of that variable, function parameter, or class member as declared in the source code.

Here are some concrete examples. For each example, we also indicate what `auto` would do (`decltype` is blue, `auto` is green).

```
struct S {
  S(){m_x = 42;}
  int m_x;
};

int x;
const int cx = 42;
const int& crx = x;
const S* p = new S();

// x is declared as an int: x_type is int.
//
typedef decltype(x) x_type;

// auto also deduces the type as int: a is an int.
//
auto a = x;

// cx is declared as const int: cx_type is const int.
//
typedef decltype(cx) cx_type;

// auto drops the const qualifier: b is int.
```

```
//
auto b = cx;

// crx is declared as const int&: crx_type is const int&.
//
typedef decltype(crx) crx_type;

// auto drops the reference and the const qualifier: c is an int.
//
auto c = crx;

// S::m_x is declared as int: m_x_type is int
//
// Note that p->m_x cannot be assigned to. It is effectively
// constant because p is a pointer to const. But decltype goes
// by the declared type, which is int.
//
typedef decltype(p->m_x) m_x_type;

// auto sees that p->m_x is const, but it drops the const
// qualifier. Therefore, d is an int.
//
auto d = p->m_x;
```

How `decltype` Deduces the Type of an Expression: Case 2

Since `decltype`'s case distinction has only two cases, the second case is "everything else," meaning everything that is not a plain, unparenthesized variable, function parameter, or class member access. Typical examples are expressions involving operators, such as `x * y`. For ease of terminology, I will refer to expressions that fall under Case 2 as *complex expressions*, as opposed to the *simple expressions* of Case 1. A trivial way to produce such a complex expression is to take a simple expression and throw parentheses around it, as in `(x)`. So what does `decltype` do with such a complex expression? The exact formulation of the rule uses the terms lvalue, xvalue, and prvalue, so we must make sure we understand those first. The terms xvalue and prvalue define a partitioning of the set of all rvalues into two subsets. Therefore, the first step is to understand the terms lvalue and rvalue. That's actually much more difficult in C++ than it used to be in C. You can find a reasonable working definition in the introduction to my article about rvalue references. So now that we know what lvalues and rvalues are, how is the set of rvalues partitioned into xvalues and prvalues? Here's the definition:

- An rvalue is an xvalue if it is one of the following:
  1. A function call where the function's return value is declared as an rvalue reference. An example would be `std::move(x)`.
  2. A cast to an rvalue reference. An example would be `static_cast<A&&>(a)`.
  3. A member access of an xvalue. Example: `(static_cast<A&&>(a)).m_x`.
- All other rvalues are prvalues.

We are now in a position to describe how `decltype` deduces the type of a complex expression.

> Let `expr` be an expression that is *not* a plain, unparenthesized variable, function parameter, or class member access. Let `T` be the type of `expr`. If `expr` is an lvalue, then `decltype(expr)` is `T&`. If `expr` is an xvalue, then `decltype(expr)` is `T&&`. Otherwise, `expr` is a prvalue, and `decltype(expr)` is `T`.

As you can see, this differs significantly from the way `auto` works, and it also differs from the way `decltype` works in Case 1, for things like plain variables. The examples below illustrate these differences.

Let us begin by going through the list of simple expressions that we used in the previous section for Case 1 of `decltype`'s definition. This time, we'll throw a set of parentheses around each of these simple expressions to turn them into complex expressions, to which Case 2 applies. To emphasize the differences, we'll also repeat what `decltype` does with the original simple expression, and what `auto` does. `decltype` on complex expressions is brick, `decltype` on simple expressions is blue, and `auto` is green. Note that `auto` never cares whether the expression is in parentheses or not.

```
struct S {
  S(){m_x = 42;}
  int m_x;
};

int x;
const int cx = 42;
const int& crx = x;
const S* p = new S();

// (x) has type int, and decltype adds references to lvalues.
// Therefore, x_with_parens_type is int&.
//
typedef decltype((x)) x_with_parens_type;

// x is declared as an int: x_type is int.
//
typedef decltype(x) x_type;

// auto also deduces the type as int: a_p and a are ints.
//
auto a_p = (x);
auto a = x;

// The type of (cx) is const int. Since (cx) is an lvalue,
// decltype adds a reference to that: cx_with_parens_type
// is const int&.
//
typedef decltype((cx)) cx_with_parens_type;

// cx is declared as const int: cx_type is const int.
//
typedef decltype(cx) cx_type;

// auto drops the const qualifier: b_p and b are ints.
//
auto b_p = (cx);
auto b = cx;

// The type of (crx) is const int&[1], and it is an lvalue.
// decltype adds a reference. By the C++11 reference
// collapsing rules, that makes no difference. Hence,
// crx_with_parens_type is const int&.
//
typedef decltype((crx)) crx_with_parens_type;

// crx is declared as const int&: crx_type is const int&.
//
typedef decltype(crx) crx_type;
```

```
// auto drops the reference and the const qualifier: c_p and c
// are ints.
//
auto c_p = (crx);
auto c = crx;

// S::m_x is declared as int. Since p is a pointer to const,
// the type of (p->m_x) is const int. Since (p->m_x) is an
// lvalue, decltype adds a reference to that. Therefore,
// m_x_with_parens_type is const int&.
//
typedef decltype((p->m_x)) m_x_with_parens_type;

// S::m_x is declared as int: m_x_type is int.
//
typedef decltype(p->m_x) m_x_type;

// auto sees that p->m_x is const, but it drops the const
// qualifier. Therefore, d_p and d are ints.
//
auto d_p = (p->m_x);
auto d = p->m_x;
```

Now let's do some examples of expressions that really are complex, in the sense that they involve operators or function calls. We'll begin with some function and unary operator calls.

```
const S foo();
const int& foobar();
std::vector<int> vect = {42, 43};

// foo() is declared as returning const S. The type of foo()
// is const S. Since foo() is a prvalue, decltype does not
// add a reference. Therefore, foo_type is const S.
//
// Note: we had to use the user-defined type S here instead of int,
// because C++ does not allow us to return a basic type as const.
// (Ok, it does allow it, but the const would be ignored.)
//
typedef decltype(foo()) foo_type;

// auto drops the const qualifier: a is an S.
//
auto a = foo();

// The type of foobar() is const int&[1], and it is an lvalue.
// Therefore, decltype adds a reference. By the C++11 reference
// collapsing rules, that makes no difference. Therefore,
// foobar_type is const int&.
//
typedef decltype(foobar()) foobar_type;

// auto drops the reference and the const qualifier: b is
// an int.
//
```

```
auto b = foobar();

// The type of vect.begin() is std::vector<int>::iterator.
// Since vect.begin() is a prvalue, no reference
// is added. Therefore, iterator_type is
// std::vector<int>::iterator.
//
typedef decltype(vect.begin()) iterator_type;

// auto also deduces the type as std::vector<int>::iterator,
// so iter has type std::vector<int>::iterator.
//
auto iter = vect.begin();

// std::vector<int>'s operator[] is declared to have return
// type int&. Therefore, the type of the expression vect[0]
// is int&[1]. Since vect[0] is an lvalue, decltype adds a
// reference. By the C++11 reference collapsing rules,
// that makes no difference. Therefore, first_element has
// type int&.
//
decltype(vect[0]) first_element = vect[0];

// second_element has type int, because auto drops the reference.
//
auto second_element = vect[1];
```

In the last example above, the first element of the vector can be modified through the reference `first_element`. The second element cannot be modified through `second_element`, because the latter is not a reference. This demonstrates how an incomplete understanding of the workings of `auto` and `decltype` could lead to coding errors that don't show up until runtime.

Finally, here are some examples of binary and ternary operators:

```
int x = 0;
int y = 0;
const int cx = 42;
const int cy = 43;
double d1 = 3.14;
double d2 = 2.72;

// The type of the product is int, and the product
// is a prvalue. Therefore, prod_xy_type is an int.
//
typedef decltype(x * y) prod_xy_type;

// auto also deduces the type as int: a is an int.
//
auto a = x * y;

// The type of the product is int (not const int!),
// and the product is a prvalue. Therefore, prod_cxcy_type
// is an int.
```

```
//
typedef decltype(cx * cy) prod_cxcy_type;

// same for auto: b is an int.
//
auto b = cx * cy;

// The type of the expresson is double, and the expression
// is an lvalue. Therefore, a reference is added, and
// cond_type is double&.
//
typedef decltype(d1 < d2 ? d1 : d2) cond_type;

// The type of the expression is double, so c is a double.
//
auto c = d1 < d2 ? d1 : d2;

// The type of the expresson is double. The expression
// is a prvalue, because in order to accomodate the
// promotion of x to a double, a temporary has to be
// created. Therefore, no reference is added, and
// cond_type_mixed is double.
//
typedef decltype(x < d2 ? x : d2) cond_type_mixed;

// The type of the expression is double, so d is a double.
//
auto d = x < d2 ? x : d2;
```

Note that in the last example, you couldn't have just written

```
auto d = std:min(x, dbl); // error: ambiguous template parameter
```

because `std::min` requires its arguments to be of the same type. More on that in the next section.

---

[1]There is an alternate way to derive the same result for examples like this. The C++ Standard contains the following clause (5/5): "If an expression initially has the type 'reference to T' (8.3.2, 8.5.3), the type is adjusted to T prior to any further analysis." One may argue that applying `decltype` constitutes "further analysis," and therefore, the types of our expressions `(crx)`, `foobar()`, and `vect[0]` have already been stripped of the reference. However, since `decltype` adds a reference in all these cases, the end result is the same whether or not one believes that 5/5 applies.

An Example to Put You on Guard

Taking our cue from one of the examples of the previous section, let us assume that we're working on some numerical computations using floating point arithmetic, and we frequently need the min and max of two numbers that may come in as different types, such as `int` and `double`. This precludes the direct use of `std::min`, since the latter requires its arguments to be of the same type:

```
int i = 42;
double d = 42.1;
auto a = std::min(i, d); // error: ambiguous template parameter
```

If we wanted to use `std::min`, we would have to write

```
auto a = std::min(static_cast<double>(i), d);
```

This gets old pretty quickly, especially in a day and age where we want even our C++ code to look like Python. So rather naturally, we would want a version of `min` and `max` that deals with the different types.

In 2001, Andrei Alexandrescu wrote an article on implementing min and max in C++, responding to a challenge posed by Scott Meyers. If you have read the article, or any one of the related discussions on the Web that have happened since, then you know that going for full-blown generic versions of mixed-type `min` and `max` functions is not a good idea. Instead, we should aim for something that is meant to be used for our specific purpose only. We probably already have a bunch of utilities in a namespace called something like `floating_point_utilities`. Now we want to put functions `fpmin` and `fpmax` in there that allow us to write

```
using namespace floating_point_utilities;
auto a = fpmin(i, d);
```

Since the new functions are in our own namespace, we could of course call them `min` and `max`. Personally, I prefer to give things unique names whenever possible so I can use `using namespace` liberally. Also, the names `fpmin` and `fpmax` are a good reminder of the specific purpose of these functions.

The generic functions `min` and `max` take their arguments by const reference. That's because being generic, they must be concerned with the cost of copying large objects. For our `fpmin` and `fpmax`, that is not a consideration. Moreover, taking arguments by `const double&` and `const int&` would look odd in a world of numerical functions, where arguments are always taken by value. Therefore, we let our `fpmin` and `fpmax` take their arguments by value until the profiler tells us otherwise, which is not likely to happen.

All this being said, how hard can it be to write those little three-liners? Very hard. If you're not on your toes about the subtleties of `decltype`, you may end up writing the following, and you would not be the first one to do so:

```
<horrible>
template<typename T, typename S>
auto fpmin(T x, S y) -> decltype(x < y ? x : y) {
  return x < y ? x : y;
}
</horrible>
```

According to our discussion in the previous section, the type

```
decltype(x < y ? x : y)
```

may or may not be a reference. If the types of `T` and `S` are the same, then it is a reference. If they are a mixture like `int` and `double`, it is not. In the former case, our `fpmin` function as defined above returns a reference to a local variable (a parameter in this case). You will probably agree that returning a reference to a local variable or a temporary ranks high among the worst and most embarrassing things a C++ programmer can do. Depending on the circumstances, it may or may not be caught via a compiler warning. Here's the correct version of `fpmin`:

```
// Min function intended for basic numeric types. The arguments
// may be of different type, in which case the one with lower
// precision gets promoted.
//
template<typename T, typename S>
auto fpmin(T x, S y) ->
  typename std::remove_reference<decltype(x < y ? x : y)>::type {
  return x < y ? x : y;
}
```

Now is a good time to tell you, as I promised earlier, why I am not a big fan of the use of the lexical token `auto` in the trailing return type syntax, as seen above on our `fpmin` function. As we know by now, the way that the other `auto`, the one that is used when declaring and initializing a variable, deduces the type of an expression is substantially different from the way `decltype` works. *In the context of trailing return type syntax, only `decltype` matters.* Perhaps I'm overly sensitive, but the use of the lexical token `auto` in this context leads my mind astray, towards the reference-dropping semantics of the other `auto`. That mental association is dangerous, as evidenced by the `fpmin` example.

Miscellaneous Properties of `decltype`

An important property of `decltype` is that its operand never gets evaluated. For example, you can use an out-of-bounds element access to a vector as the operand of `decltype` with impunity:

```
std::vector<int> vect;
assert(vect.empty());
typedef decltype(vect[0]) integer;
```

Another property of `decltype` that is worth pointing out is that when `decltype(expr)` is the name of a plain user defined type (not a reference or pointer, not a basic or function type), then `decltype(expr)` is also a class name. This means that you can access nested types directly:

```
template<typename R>
class SomeFunctor {
public:
  typedef R result_type;
  result_type operator()() {
    return R();
  }
  SomeFunctor(){}
};

SomeFunctor<int> func;
typedef decltype(func)::result_type integer; // access nested type
```

You can even use `decltype(expr)` to specify a base class:

```
auto foo = [](){return 42;};

class DerivedFunctor : public decltype(foo)
{
  public:
    MyFunctor(): decltype(foo)(foo) {}

  // ...
};
```

Summary and Epilogue

We have learned that C++11 has three different ways of making the type of an expression available to the programmer:

- `auto` can be used to set the type of a newly declared variable from its initializing expression. It removes the reference, if any, from the expression's type, then removes topmost const and volatile qualifiers.
- `decltype` can be used in a wider variety of contexts, such as typedefs, function return types, and even in places where a class name is expected. There are two different ways in which `decltype(expr)` can work, depending on the nature of `expr`:
  - If `expr` is something as simple as a plain variable whose type can be looked up in the source code, then `decltype(expr)` is that type.
  - Otherwise, `decltype(expr)` is the type of `expr` with an extra lvalue reference added for lvalues and an extra rvalue reference added for xvalues.

If you focus on the the way references are treated in the above, you see that `auto` always drops an outermost reference, whereas `decltype` either preserves it or even adds one, depending on the circumstances.

At the end of Section 3, I gave you what I think is a plausible explanation of the rationale behind the semantics of `auto`. The question arises, "What about the rationale behind the semantics of `decltype`? Why does it work the way it does, specifically with respect to references?" As with `auto`, there is probably more than one way to argue. From what I have read and heard, the final specification of `decltype` represents a compromise between two different possible points of view. One point of view is that decltype should be a way to retrieve and reuse the type of a variable as declared in the source code: I declared a variable `x` of type `T` at some point in my code. Now I want to use that type for some other purpose, like making a typedef, or specify the return type of a function. I don't want to repeat myself, so give me a way to recover the declared type of my variable, exactly the way I originally wrote it. If that declared type has a reference and/or a `const` or `volatile` qualifier on it that I don't want, I'll remove that myself, thank you very much. This is what Case 1 of the specification of `decltype` does.

The above way of looking at `decltype` says nothing about what should happen when `decltype` is applied to more complex expressions. From what we've said so far, `decltype` might as well be undefined for complex expressions. This is where a different point of view comes in, a point of view that originates in the needs of library writers. They often find themselves in a situation where the return type of a function needs to be the type of some expression, typically something that depends on template parameters. They want to be able to write

```
auto foo([...]) -> decltype(expr) {
```

```
  [...]
  return expr;
}
```

where *expr* could be any expression. Moreover, if *expr* is an lvalue and ist not local to the function, they want to return a refefence, so the returned expression can be assigned to. This is particularly important if the function `foo` is some kind of forwarding function. For that to always work properly, there is really no choice but to give `decltype` that reference-adding semantics. The following example, which was provided by Stephan Lavavej, demonstrates that:

```
template <typename T>
auto array_access(T& array, size_t pos) -> decltype(array[pos]) {
  return array[pos];
}

std::vector<int> vect = {42, 43, 44};
int* p = &vect[0];

array_access(vect, 2) = 45;
array_access(p, 2) = 46;
```

The last line, where the pointer `p` is used to access the element, could not be made to work without having `decltype` add a reference: the type of `p[2]` is `int`, any way you turn it. There is no reference in sight here. But `p[2]` is an lvalue, and by letting `decltype` add references to lvalues, we can get the desired effect. All this comes with a caveat: the reference that `decltype` adds is not always what you want. It happens frequently that you need to remove it with `remove_reference`. We saw an example of that in Section 8.

Here are links to some more articles on `auto` and `decltype` that you may find helpful:

1. Wikipedia article "decltype"
2. Using C++11 auto and decltype - Code Synthesis
3. Koenig, Andrew. "A Note About decltype". Dr. Dobb's Journal, July 27, 2011.
4. Visual C++ Team Blog entry "decltype"

Acknowledgments

I wish to thank Scott Meyers for taking the time to read several drafts of this article and helping me with many valuable suggestions and corrections. Stephan Lavavej provided the example in Section 10 that made me understand why `decltype` is defined the way it is. Dilip Ranganathan, with his curiosity, enthusiasm, and attention to detail, has been a great help in more ways than I can remember. All remaining deficiencies are mine.