O'REILLY®

Effective Modern C++

42 SPECIFIC WAYS TO IMPROVE YOUR USE OF C++11 AND C++14

Scott Meyers

CHAPTER 1
Deducing Types

C++98 had a single set of rules for type deduction: the one for function templates. C++11 modifies that ruleset a bit and adds two more, one for `auto` and one for `decltype`. C++14 then extends the usage contexts in which `auto` and `decltype` may be employed. The increasingly widespread application of type deduction frees you from the tyranny of spelling out types that are obvious or redundant. It makes C++ software more adaptable, because changing a type at one point in the source code automatically propagates through type deduction to other locations. However, it can render code more difficult to reason about, because the types deduced by compilers may not be as apparent as you'd like.

Without a solid understanding of how type deduction operates, effective programming in modern C++ is all but impossible. There are just too many contexts where type deduction takes place: in calls to function templates, in most situations where `auto` appears, in `decltype` expressions, and, as of C++14, where the enigmatic `decltype(auto)` construct is employed.

This chapter provides the information about type deduction that every C++ developer requires. It explains how template type deduction works, how `auto` builds on that, and how `decltype` goes its own way. It even explains how you can force compilers to make the results of their type deductions visible, thus enabling you to ensure that compilers are deducing the types you want them to.

Item 1: Understand template type deduction.

When users of a complex system are ignorant of how it works, yet happy with what it does, that says a lot about the design of the system. By this measure, template type deduction in C++ is a tremendous success. Millions of programmers have passed arguments to template functions with completely satisfactory results, even though many of those programmers would be hard-pressed to give more than the haziest description of how the types used by those functions were deduced.

If that group includes you, I have good news and bad news. The good news is that type deduction for templates is the basis for one of modern C++'s most compelling features: `auto`. If you were happy with how C++98 deduced types for templates, you're set up to be happy with how C++11 deduces types for `auto`. The bad news is that when the template type deduction rules are applied in the context of `auto`, they sometimes seem less intuitive than when they're applied to templates. For that reason, it's important to truly understand the aspects of template type deduction that `auto` builds on. This Item covers what you need to know.

If you're willing to overlook a pinch of pseudocode, we can think of a function template as looking like this:

```
template<typename T>
void f(ParamType param);
```

A call can look like this:

```
f(expr);                        // call f with some expression
```

During compilation, compilers use *`expr`* to deduce two types: one for `T` and one for *`ParamType`*. These types are frequently different, because *`ParamType`* often contains adornments, e.g., `const` or reference qualifiers. For example, if the template is declared like this,

```
template<typename T>
void f(const T& param);         // ParamType is const T&
```

and we have this call,

```
int x = 0;

f(x);                           // call f with an int
```

`T` is deduced to be `int`, but *`ParamType`* is deduced to be `const int&`.

It's natural to expect that the type deduced for `T` is the same as the type of the argument passed to the function, i.e., that `T` is the type of *`expr`*. In the above example, that's the case: `x` is an `int`, and `T` is deduced to be `int`. But it doesn't always work that way. The type deduced for `T` is dependent not just on the type of *`expr`*, but also on the form of *`ParamType`*. There are three cases:

- *ParamType* is a pointer or reference type, but not a universal reference. (Universal references are described in Item 24. At this point, all you need to know is that they exist and that they're not the same as lvalue references or rvalue references.)
- *ParamType* is a universal reference.
- *ParamType* is neither a pointer nor a reference.

We therefore have three type deduction scenarios to examine. Each will be based on our general form for templates and calls to it:

```
template<typename T>
void f(ParamType param);

f(expr);                    // deduce T and ParamType from expr
```

Case 1: *ParamType* is a Reference or Pointer, but not a Universal Reference

The simplest situation is when *ParamType* is a reference type or a pointer type, but not a universal reference. In that case, type deduction works like this:

1. If *expr*'s type is a reference, ignore the reference part.
2. Then pattern-match *expr*'s type against *ParamType* to determine `T`.

For example, if this is our template,

```
template<typename T>
void f(T& param);           // param is a reference
```

and we have these variable declarations,

```
int x = 27;                 // x is an int
const int cx = x;           // cx is a const int
const int& rx = x;          // rx is a reference to x as a const int
```

the deduced types for `param` and `T` in various calls are as follows:

```
f(x);                       // T is int, param's type is int&

f(cx);                      // T is const int,
                            // param's type is const int&

f(rx);                      // T is const int,
                            // param's type is const int&
```

In the second and third calls, notice that because `cx` and `rx` designate `const` values, `T` is deduced to be `const int`, thus yielding a parameter type of `const int&`. That's important to callers. When they pass a `const` object to a reference parameter, they expect that object to remain unmodifiable, i.e., for the parameter to be a reference-to-`const`. That's why passing a `const` object to a template taking a `T&` parameter is safe: the `const`ness of the object becomes part of the type deduced for `T`.

In the third example, note that even though `rx`'s type is a reference, `T` is deduced to be a non-reference. That's because `rx`'s reference-ness is ignored during type deduction.

These examples all show lvalue reference parameters, but type deduction works exactly the same way for rvalue reference parameters. Of course, only rvalue arguments may be passed to rvalue reference parameters, but that restriction has nothing to do with type deduction.

If we change the type of `f`'s parameter from `T&` to `const T&`, things change a little, but not in any really surprising ways. The `const`ness of `cx` and `rx` continues to be respected, but because we're now assuming that `param` is a reference-to-`const`, there's no longer a need for `const` to be deduced as part of `T`:

```
template<typename T>
void f(const T& param);   // param is now a ref-to-const

int x = 27;               // as before
const int cx = x;         // as before
const int& rx = x;        // as before

f(x);                     // T is int, param's type is const int&

f(cx);                    // T is int, param's type is const int&

f(rx);                    // T is int, param's type is const int&
```

As before, `rx`'s reference-ness is ignored during type deduction.

If `param` were a pointer (or a pointer to `const`) instead of a reference, things would work essentially the same way:

```
template<typename T>
void f(T* param);         // param is now a pointer

int x = 27;               // as before
const int *px = &x;       // px is a ptr to x as a const int
```

```
f(&x);                    // T is int, param's type is int*

f(px);                    // T is const int,
                          // param's type is const int*
```

By now, you may find yourself yawning and nodding off, because C++'s type deduction rules work so naturally for reference and pointer parameters, seeing them in written form is really dull. Everything's just obvious! Which is exactly what you want in a type deduction system.

Case 2: *ParamType* is a Universal Reference

Things are less obvious for templates taking universal reference parameters. Such parameters are declared like rvalue references (i.e., in a function template taking a type parameter `T`, a universal reference's declared type is `T&&`), but they behave differently when lvalue arguments are passed in. The complete story is told in Item 24, but here's the headline version:

- If *expr* is an lvalue, both `T` and *ParamType* are deduced to be lvalue references. That's doubly unusual. First, it's the only situation in template type deduction where `T` is deduced to be a reference. Second, although *ParamType* is declared using the syntax for an rvalue reference, its deduced type is an lvalue reference.
- If *expr* is an rvalue, the "normal" (i.e., Case 1) rules apply.

For example:

```
template<typename T>
void f(T&& param);        // param is now a universal reference

int x = 27;               // as before
const int cx = x;         // as before
const int& rx = x;        // as before

f(x);                     // x is lvalue, so T is int&,
                          // param's type is also int&

f(cx);                    // cx is lvalue, so T is const int&,
                          // param's type is also const int&

f(rx);                    // rx is lvalue, so T is const int&,
                          // param's type is also const int&

f(27);                    // 27 is rvalue, so T is int,
                          // param's type is therefore int&&
```

Item 24 explains exactly why these examples play out the way they do. The key point here is that the type deduction rules for universal reference parameters are different from those for parameters that are lvalue references or rvalue references. In particular, when universal references are in use, type deduction distinguishes between lvalue arguments and rvalue arguments. That never happens for non-universal references.

Case 3: *ParamType* is Neither a Pointer nor a Reference

When *ParamType* is neither a pointer nor a reference, we're dealing with pass-by-value:

```
template<typename T>
void f(T param);           // param is now passed by value
```

That means that `param` will be a copy of whatever is passed in—a completely new object. The fact that `param` will be a new object motivates the rules that govern how `T` is deduced from *expr*:

1. As before, if *expr*'s type is a reference, ignore the reference part.
2. If, after ignoring *expr*'s reference-ness, *expr* is `const`, ignore that, too. If it's `volatile`, also ignore that. (`volatile` objects are uncommon. They're generally used only for implementing device drivers. For details, see Item 40.)

Hence:

```
int x = 27;            // as before
const int cx = x;      // as before
const int& rx = x;     // as before

f(x);                  // T's and param's types are both int

f(cx);                 // T's and param's types are again both int

f(rx);                 // T's and param's types are still both int
```

Note that even though `cx` and `rx` represent `const` values, `param` isn't `const`. That makes sense. `param` is an object that's completely independent of `cx` and `rx`—a *copy* of `cx` or `rx`. The fact that `cx` and `rx` can't be modified says nothing about whether `param` can be. That's why *expr*'s `const`ness (and `volatile`ness, if any) is ignored when deducing a type for `param`: just because *expr* can't be modified doesn't mean that a copy of it can't be.

It's important to recognize that `const` (and `volatile`) is ignored only for by-value parameters. As we've seen, for parameters that are references-to- or pointers-to-`const`, the `const`ness of *expr* is preserved during type deduction. But consider the case where *expr* is a const pointer to a const object, and *expr* is passed to a by-value param:

```
template<typename T>
void f(T param);          // param is still passed by value

const char* const ptr =   // ptr is const pointer to const object
  "Fun with pointers";

f(ptr);                   // pass arg of type const char * const
```

Here, the const to the right of the asterisk declares ptr to be const: ptr can't be made to point to a different location, nor can it be set to null. (The const to the left of the asterisk says that what ptr points to—the character string—is const, hence can't be modified.) When ptr is passed to f, the bits making up the pointer are copied into param. As such, *the pointer itself (ptr) will be passed by value*. In accord with the type deduction rule for by-value parameters, the constness of ptr will be ignored, and the type deduced for param will be const char*, i.e., a modifiable pointer to a const character string. The constness of what ptr points to is preserved during type deduction, but the constness of ptr itself is ignored when copying it to create the new pointer, param.

Array Arguments

That pretty much covers it for mainstream template type deduction, but there's a niche case that's worth knowing about. It's that array types are different from pointer types, even though they sometimes seem to be interchangeable. A primary contributor to this illusion is that, in many contexts, an array *decays* into a pointer to its first element. This decay is what permits code like this to compile:

```
const char name[] = "J. P. Briggs";  // name's type is
                                     // const char[13]

const char * ptrToName = name;       // array decays to pointer
```

Here, the const char* pointer ptrToName is being initialized with name, which is a const char[13]. These types (const char* and const char[13]) are not the same, but because of the array-to-pointer decay rule, the code compiles.

But what if an array is passed to a template taking a by-value parameter? What happens then?

```
template<typename T>
void f(T param);       // template with by-value parameter
```

```
f(name);               // what types are deduced for T and param?
```

We begin with the observation that there is no such thing as a function parameter that's an array. Yes, yes, the syntax is legal,

```
void myFunc(int param[]);
```

but the array declaration is treated as a pointer declaration, meaning that `myFunc` could equivalently be declared like this:

```
void myFunc(int* param);         // same function as above
```

This equivalence of array and pointer parameters is a bit of foliage springing from the C roots at the base of C++, and it fosters the illusion that array and pointer types are the same.

Because array parameter declarations are treated as if they were pointer parameters, the type of an array that's passed to a template function by value is deduced to be a pointer type. That means that in the call to the template `f`, its type parameter `T` is deduced to be `const char*`:

```
f(name);           // name is array, but T deduced as const char*
```

But now comes a curve ball. Although functions can't declare parameters that are truly arrays, they *can* declare parameters that are *references* to arrays! So if we modify the template `f` to take its argument by reference,

```
template<typename T>
void f(T& param);      // template with by-reference parameter
```

and we pass an array to it,

```
f(name);               // pass array to f
```

the type deduced for `T` is the actual type of the array! That type includes the size of the array, so in this example, `T` is deduced to be `const char [13]`, and the type of `f`'s parameter (a reference to this array) is `const char (&)[13]`. Yes, the syntax looks toxic, but knowing it will score you mondo points with those few souls who care.

Interestingly, the ability to declare references to arrays enables creation of a template that deduces the number of elements that an array contains:

```
// return size of an array as a compile-time constant. (The
// array parameter has no name, because we care only about
// the number of elements it contains.)
template<typename T, std::size_t N>                 // see info
constexpr std::size_t arraySize(T (&)[N]) noexcept  // below on
{                                                   // constexpr
```

```
  return N;                                          // and
}                                                    // noexcept
```

As Item 15 explains, declaring this function `constexpr` makes its result available during compilation. That makes it possible to declare, say, an array with the same number of elements as a second array whose size is computed from a braced initializer:

```
int keyVals[] = { 1, 3, 7, 9, 11, 22, 35 };      // keyVals has
                                                 // 7 elements

int mappedVals[arraySize(keyVals)];              // so does
                                                 // mappedVals
```

Of course, as a modern C++ developer, you'd naturally prefer a `std::array` to a built-in array:

```
std::array<int, arraySize(keyVals)> mappedVals;  // mappedVals'
                                                 // size is 7
```

As for `arraySize` being declared `noexcept`, that's to help compilers generate better code. For details, see Item 14.

Function Arguments

Arrays aren't the only things in C++ that can decay into pointers. Function types can decay into function pointers, and everything we've discussed regarding type deduction for arrays applies to type deduction for functions and their decay into function pointers. As a result:

```
void someFunc(int, double);   // someFunc is a function;
                              // type is void(int, double)

template<typename T>
void f1(T param);             // in f1, param passed by value

template<typename T>
void f2(T& param);            // in f2, param passed by ref

f1(someFunc);                 // param deduced as ptr-to-func;
                              // type is void (*)(int, double)

f2(someFunc);                 // param deduced as ref-to-func;
                              // type is void (&)(int, double)
```

This rarely makes any difference in practice, but if you're going to know about array-to-pointer decay, you might as well know about function-to-pointer decay, too.

So there you have it: the `auto`-related rules for template type deduction. I remarked at the outset that they're pretty straightforward, and for the most part, they are. The special treatment accorded lvalues when deducing types for universal references muddies the water a bit, however, and the decay-to-pointer rules for arrays and functions stirs up even greater turbidity. Sometimes you simply want to grab your compilers and demand, "Tell me what type you're deducing!" When that happens, turn to Item 4, because it's devoted to coaxing compilers into doing just that.

Things to Remember

- During template type deduction, arguments that are references are treated as non-references, i.e., their reference-ness is ignored.
- When deducing types for universal reference parameters, lvalue arguments get special treatment.
- When deducing types for by-value parameters, `const` and/or `volatile` arguments are treated as non-`const` and non-`volatile`.
- During template type deduction, arguments that are array or function names decay to pointers, unless they're used to initialize references.

Item 2: Understand `auto` type deduction.

If you've read Item 1 on template type deduction, you already know almost everything you need to know about `auto` type deduction, because, with only one curious exception, `auto` type deduction *is* template type deduction. But how can that be? Template type deduction involves templates and functions and parameters, but `auto` deals with none of those things.

That's true, but it doesn't matter. There's a direct mapping between template type deduction and `auto` type deduction. There is literally an algorithmic transformation from one to the other.

In Item 1, template type deduction is explained using this general function template

```
template<typename T>
void f(ParamType param);
```

and this general call:

```
f(expr);                        // call f with some expression
```

In the call to `f`, compilers use *`expr`* to deduce types for `T` and *`ParamType`*.

When a variable is declared using `auto`, `auto` plays the role of `T` in the template, and the type specifier for the variable acts as *`ParamType`*. This is easier to show than to describe, so consider this example:

```
auto x = 27;
```

Here, the type specifier for `x` is simply `auto` by itself. On the other hand, in this declaration,

```
const auto cx = x;
```

the type specifier is `const auto`. And here,

```
const auto& rx = x;
```

the type specifier is `const auto&`. To deduce types for `x`, `cx`, and `rx` in these examples, compilers act as if there were a template for each declaration as well as a call to that template with the corresponding initializing expression:

```
template<typename T>               // conceptual template for
void func_for_x(T param);          // deducing x's type

func_for_x(27);                    // conceptual call: param's
                                   // deduced type is x's type

template<typename T>               // conceptual template for
void func_for_cx(const T param);   // deducing cx's type

func_for_cx(x);                    // conceptual call: param's
                                   // deduced type is cx's type

template<typename T>               // conceptual template for
void func_for_rx(const T& param);  // deducing rx's type

func_for_rx(x);                    // conceptual call: param's
                                   // deduced type is rx's type
```

As I said, deducing types for `auto` is, with only one exception (which we'll discuss soon), the same as deducing types for templates.

Item 1 divides template type deduction into three cases, based on the characteristics of *`ParamType`*, the type specifier for `param` in the general function template. In a variable declaration using `auto`, the type specifier takes the place of *`ParamType`*, so there are three cases for that, too:

- Case 1: The type specifier is a pointer or reference, but not a universal reference.
- Case 2: The type specifier is a universal reference.

- Case 3: The type specifier is neither a pointer nor a reference.

We've already seen examples of cases 1 and 3:

```
auto x = 27;          // case 3 (x is neither ptr nor reference)

const auto cx = x;    // case 3 (cx isn't either)

const auto& rx = x;   // case 1 (rx is a non-universal ref.)
```

Case 2 works as you'd expect:

```
auto&& uref1 = x;     // x is int and lvalue,
                      // so uref1's type is int&

auto&& uref2 = cx;    // cx is const int and lvalue,
                      // so uref2's type is const int&

auto&& uref3 = 27;    // 27 is int and rvalue,
                      // so uref3's type is int&&
```

Item 1 concludes with a discussion of how array and function names decay into pointers for non-reference type specifiers. That happens in `auto` type deduction, too:

```
const char name[] =            // name's type is const char[13]
  "R. N. Briggs";

auto arr1 = name;              // arr1's type is const char*

auto& arr2 = name;             // arr2's type is
                               // const char (&)[13]


void someFunc(int, double);    // someFunc is a function;
                               // type is void(int, double)

auto func1 = someFunc;         // func1's type is
                               // void (*)(int, double)

auto& func2 = someFunc;        // func2's type is
                               // void (&)(int, double)
```

As you can see, `auto` type deduction works like template type deduction. They're essentially two sides of the same coin.

Except for the one way they differ. We'll start with the observation that if you want to declare an `int` with an initial value of 27, C++98 gives you two syntactic choices:

```
int x1 = 27;
int x2(27);
```

C++11, through its support for uniform initialization, adds these:

```
int x3 = { 27 };
int x4{ 27 };
```

All in all, four syntaxes, but only one result: an `int` with value 27.

But as Item 5 explains, there are advantages to declaring variables using `auto` instead of fixed types, so it'd be nice to replace `int` with `auto` in the above variable declarations. Straightforward textual substitution yields this code:

```
auto x1 = 27;
auto x2(27);
auto x3 = { 27 };
auto x4{ 27 };
```

These declarations all compile, but they don't have the same meaning as the ones they replace. The first two statements do, indeed, declare a variable of type `int` with value 27. The second two, however, declare a variable of type `std::initializer_list<int>` containing a single element with value 27!

```
auto x1 = 27;             // type is int, value is 27

auto x2(27);              // ditto

auto x3 = { 27 };         // type is std::initializer_list<int>,
                          // value is { 27 }

auto x4{ 27 };            // ditto
```

This is due to a special type deduction rule for `auto`. When the initializer for an `auto`-declared variable is enclosed in braces, the deduced type is a `std::initializer_list`. If such a type can't be deduced (e.g., because the values in the braced initializer are of different types), the code will be rejected:

```
auto x5 = { 1, 2, 3.0 };  // error! can't deduce T for
                          // std::initializer_list<T>
```

As the comment indicates, type deduction will fail in this case, but it's important to recognize that there are actually two kinds of type deduction taking place. One kind stems from the use of `auto`: `x5`'s type has to be deduced. Because `x5`'s initializer is in braces, `x5` must be deduced to be a `std::initializer_list`. But `std::initializer_list` is a template. Instantiations are `std::initializer_list<T>` for some type `T`, and that means that `T`'s type must also be deduced. Such deduction falls under the purview of the second kind of type deduction occurring here: template type deduction. In this example, that deduction fails, because the values in the braced initializer don't have a single type.

The treatment of braced initializers is the only way in which `auto` type deduction and template type deduction differ. When an `auto`–declared variable is initialized with a braced initializer, the deduced type is an instantiation of `std::initializer_list`. But if the corresponding template is passed the same initializer, type deduction fails, and the code is rejected:

```
auto x = { 11, 23, 9 };    // x's type is
                           // std::initializer_list<int>

template<typename T>       // template with parameter
void f(T param);           // declaration equivalent to
                           // x's declaration

f({ 11, 23, 9 });          // error! can't deduce type for T
```

However, if you specify in the template that `param` is a `std::initializer_list<T>` for some unknown `T`, template type deduction will deduce what `T` is:

```
template<typename T>
void f(std::initializer_list<T> initList);

f({ 11, 23, 9 });          // T deduced as int, and initList's
                           // type is std::initializer_list<int>
```

So the only real difference between `auto` and template type deduction is that `auto` *assumes* that a braced initializer represents a `std::initializer_list`, but template type deduction doesn't.

You might wonder why `auto` type deduction has a special rule for braced initializers, but template type deduction does not. I wonder this myself. Alas, I have not been able to find a convincing explanation. But the rule is the rule, and this means you must remember that if you declare a variable using `auto` and you initialize it with a braced initializer, the deduced type will always be `std::initializer_list`. It's especially important to bear this in mind if you embrace the philosophy of uniform initialization—of enclosing initializing values in braces as a matter of course. A classic mistake in C++11 programming is accidentally declaring a `std::initializer_list` variable when you mean to declare something else. This pitfall is one of the reasons some developers put braces around their initializers only when they have to. (When you have to is discussed in Item 7.)

For C++11, this is the full story, but for C++14, the tale continues. C++14 permits `auto` to indicate that a function's return type should be deduced (see Item 3), and C++14 lambdas may use `auto` in parameter declarations. However, these uses of `auto` employ *template type deduction*, not `auto` type deduction. So a function with an `auto` return type that returns a braced initializer won't compile:

```
auto createInitList()
{
  return { 1, 2, 3 };          // error: can't deduce type
}                              // for { 1, 2, 3 }
```

The same is true when `auto` is used in a parameter type specification in a C++14 lambda:

```
std::vector<int> v;
…

auto resetV =
  [&v](const auto& newValue) { v = newValue; };     // C++14

…

resetV({ 1, 2, 3 });           // error! can't deduce type
                               // for { 1, 2, 3 }
```

Things to Remember

- `auto` type deduction is usually the same as template type deduction, but `auto` type deduction assumes that a braced initializer represents a `std::initializer_list`, and template type deduction doesn't.
- `auto` in a function return type or a lambda parameter implies template type deduction, not `auto` type deduction.

Item 3: Understand `decltype`.

`decltype` is an odd creature. Given a name or an expression, `decltype` tells you the name's or the expression's type. Typically, what it tells you is exactly what you'd predict. Occasionally however, it provides results that leave you scratching your head and turning to reference works or online Q&A sites for revelation.

We'll begin with the typical cases—the ones harboring no surprises. In contrast to what happens during type deduction for templates and `auto` (see Items 1 and 2), `decltype` typically parrots back the exact type of the name or expression you give it:

```
const int i = 0;            // decltype(i) is const int

bool f(const Widget& w);    // decltype(w) is const Widget&
                            // decltype(f) is bool(const Widget&)

struct Point {
  int x, y;                 // decltype(Point::x) is int
};                          // decltype(Point::y) is int

Widget w;                   // decltype(w) is Widget

if (f(w)) …                 // decltype(f(w)) is bool

template<typename T>        // simplified version of std::vector
class vector {
public:
  …
  T& operator[](std::size_t index);
  …
};

vector<int> v;              // decltype(v) is vector<int>
…
if (v[0] == 0) …            // decltype(v[0]) is int&
```

See? No surprises.

In C++11, perhaps the primary use for `decltype` is declaring function templates where the function's return type depends on its parameter types. For example, suppose we'd like to write a function that takes a container that supports indexing via square brackets (i.e., the use of "`[]`") plus an index, then authenticates the user before returning the result of the indexing operation. The return type of the function should be the same as the type returned by the indexing operation.

`operator[]` on a container of objects of type `T` typically returns a `T&`. This is the case for `std::deque`, for example, and it's almost always the case for `std::vector`. For `std::vector<bool>`, however, `operator[]` does not return a `bool&`. Instead, it returns a brand new object. The whys and hows of this situation are explored in

Item 6, but what's important here is that the type returned by a container's operator[] depends on the container.

decltype makes it easy to express that. Here's a first cut at the template we'd like to write, showing the use of decltype to compute the return type. The template needs a bit of refinement, but we'll defer that for now:

```
template<typename Container, typename Index>    // works, but
auto authAndAccess(Container& c, Index i)       // requires
  -> decltype(c[i])                             // refinement
{
  authenticateUser();
  return c[i];
}
```

The use of auto before the function name has nothing to do with type deduction. Rather, it indicates that C++11's *trailing return type* syntax is being used, i.e., that the function's return type will be declared following the parameter list (after the "->"). A trailing return type has the advantage that the function's parameters can be used in the specification of the return type. In authAndAccess, for example, we specify the return type using c and i. If we were to have the return type precede the function name in the conventional fashion, c and i would be unavailable, because they would not have been declared yet.

With this declaration, authAndAccess returns whatever type operator[] returns when applied to the passed-in container, exactly as we desire.

C++11 permits return types for single-statement lambdas to be deduced, and C++14 extends this to both all lambdas and all functions, including those with multiple statements. In the case of authAndAccess, that means that in C++14 we can omit the trailing return type, leaving just the leading auto. With that form of declaration, auto *does* mean that type deduction will take place. In particular, it means that compilers will deduce the function's return type from the function's implementation:

```
template<typename Container, typename Index>    // C++14;
auto authAndAccess(Container& c, Index i)       // not quite
{                                               // correct
  authenticateUser();
  return c[i];                  // return type deduced from c[i]
}
```

Item 2 explains that for functions with an auto return type specification, compilers employ template type deduction. In this case, that's problematic. As we've discussed, operator[] for most containers-of-T returns a T&, but Item 1 explains that during template type deduction, the reference-ness of an initializing expression is ignored. Consider what that means for this client code:

```
std::deque<int> d;
…
authAndAccess(d, 5) = 10;  // authenticate user, return d[5],
                           // then assign 10 to it;
                           // this won't compile!
```

Here, `d[5]` returns an `int&`, but `auto` return type deduction for `authAndAccess` will strip off the reference, thus yielding a return type of `int`. That `int`, being the return value of a function, is an rvalue, and the code above thus attempts to assign 10 to an rvalue `int`. That's forbidden in C++, so the code won't compile.

To get `authAndAccess` to work as we'd like, we need to use `decltype` type deduction for its return type, i.e., to specify that `authAndAccess` should return exactly the same type that the expression `c[i]` returns. The guardians of C++, anticipating the need to use `decltype` type deduction rules in some cases where types are inferred, make this possible in C++14 through the `decltype(auto)` specifier. What may initially seem contradictory (`decltype` *and* `auto`?) actually makes perfect sense: `auto` specifies that the type is to be deduced, and `decltype` says that `decltype` rules should be used during the deduction. We can thus write `authAndAccess` like this:

```
template<typename Container, typename Index>   // C++14; works,
decltype(auto)                                 // but still
authAndAccess(Container& c, Index i)           // requires
{                                              // refinement
  authenticateUser();
  return c[i];
}
```

Now `authAndAccess` will truly return whatever `c[i]` returns. In particular, for the common case where `c[i]` returns a `T&`, `authAndAccess` will also return a `T&`, and in the uncommon case where `c[i]` returns an object, `authAndAccess` will return an object, too.

The use of `decltype(auto)` is not limited to function return types. It can also be convenient for declaring variables when you want to apply the `decltype` type deduction rules to the initializing expression:

```
Widget w;

const Widget& cw = w;

auto myWidget1 = cw;             // auto type deduction:
```

```
                                    // myWidget1's type is Widget

decltype(auto) myWidget2 = cw;      // decltype type deduction:
                                    // myWidget2's type is
                                    //   const Widget&
```

But two things are bothering you, I know. One is the refinement to `authAndAccess` I mentioned, but have not yet described. Let's address that now.

Look again at the declaration for the C++14 version of `authAndAccess`:

```
template<typename Container, typename Index>
decltype(auto) authAndAccess(Container& c, Index i);
```

The container is passed by lvalue-reference-to-non-`const`, because returning a reference to an element of the container permits clients to modify that container. But this means it's not possible to pass rvalue containers to this function. Rvalues can't bind to lvalue references (unless they're lvalue-references-to-`const`, which is not the case here).

Admittedly, passing an rvalue container to `authAndAccess` is an edge case. An rvalue container, being a temporary object, would typically be destroyed at the end of the statement containing the call to `authAndAccess`, and that means that a reference to an element in that container (which is typically what `authAndAccess` would return) would dangle at the end of the statement that created it. Still, it could make sense to pass a temporary object to `authAndAccess`. A client might simply want to make a copy of an element in the temporary container, for example:

```
std::deque<std::string> makeStringDeque();    // factory function

// make copy of 5th element of deque returned
// from makeStringDeque
auto s = authAndAccess(makeStringDeque(), 5);
```

Supporting such use means we need to revise the declaration for `authAndAccess` to accept both lvalues and rvalues. Overloading would work (one overload would declare an lvalue reference parameter, the other an rvalue reference parameter), but then we'd have two functions to maintain. A way to avoid that is to have `authAndAccess` employ a reference parameter that can bind to lvalues *and* rvalues, and Item 24 explains that that's exactly what universal references do. `authAndAccess` can therefore be declared like this:

```
template<typename Container, typename Index>      // c is now a
decltype(auto) authAndAccess(Container&& c,       // universal
                             Index i);            // reference
```

In this template, we don't know what type of container we're operating on, and that means we're equally ignorant of the type of index objects it uses. Employing pass-by-value for objects of an unknown type generally risks the performance hit of unnecessary copying, the behavioral problems of object slicing (see Item 41), and the sting of our coworkers' derision, but in the case of container indices, following the example of the Standard Library for index values (e.g., in `operator[]` for `std::string`, `std::vector`, and `std::deque`) seems reasonable, so we'll stick with pass-by-value for them.

However, we need to update the template's implementation to bring it into accord with Item 25's admonition to apply `std::forward` to universal references:

```
template<typename Container, typename Index>       // final
decltype(auto)                                     // C++14
authAndAccess(Container&& c, Index i)              // version
{
  authenticateUser();
  return std::forward<Container>(c)[i];
}
```

This should do everything we want, but it requires a C++14 compiler. If you don't have one, you'll need to use the C++11 version of the template. It's the same as its C++14 counterpart, except that you have to specify the return type yourself:

```
template<typename Container, typename Index>       // final
auto                                               // C++11
authAndAccess(Container&& c, Index i)              // version
-> decltype(std::forward<Container>(c)[i])
{
  authenticateUser();
  return std::forward<Container>(c)[i];
}
```

The other issue that's likely to be nagging at you is my remark at the beginning of this Item that decltype *almost* always produces the type you expect, that it *rarely* surprises. Truth be told, you're unlikely to encounter these exceptions to the rule unless you're a heavy-duty library implementer.

To *fully* understand `decltype`'s behavior, you'll have to familiarize yourself with a few special cases. Most of these are too obscure to warrant discussion in a book like this, but looking at one lends insight into `decltype` as well as its use.

Applying `decltype` to a name yields the declared type for that name. Names are lvalue expressions, but that doesn't affect `decltype`'s behavior. For lvalue expressions more complicated than names, however, `decltype` ensures that the type reported is always an lvalue reference. That is, if an lvalue expression other than a name has type `T`, `decltype` reports that type as `T&`. This seldom has any impact, because the type of most lvalue expressions inherently includes an lvalue reference qualifier. Functions returning lvalues, for example, always return lvalue references.

There is an implication of this behavior that is worth being aware of, however. In

```
int x = 0;
```

`x` is the name of a variable, so `decltype(x)` is `int`. But wrapping the name `x` in parentheses—"`(x)`"—yields an expression more complicated than a name. Being a name, `x` is an lvalue, and C++ defines the expression `(x)` to be an lvalue, too. `decltype((x))` is therefore `int&`. Putting parentheses around a name can change the type that `decltype` reports for it!

In C++11, this is little more than a curiosity, but in conjunction with C++14's support for `decltype(auto)`, it means that a seemingly trivial change in the way you write a `return` statement can affect the deduced type for a function:

```
decltype(auto) f1()
{
  int x = 0;
  …
  return x;        // decltype(x) is int, so f1 returns int
}

decltype(auto) f2()
{
  int x = 0;
  …
  return (x);      // decltype((x)) is int&, so f2 returns int&
}
```

Note that not only does `f2` have a different return type from `f1`, it's also returning a reference to a local variable! That's the kind of code that puts you on the express train to undefined behavior—a train you certainly don't want to be on.

The primary lesson is to pay very close attention when using `decltype(auto)`. Seemingly insignificant details in the expression whose type is being deduced can affect the type that `decltype(auto)` reports. To ensure that the type being deduced is the type you expect, use the techniques described in Item 4.

At the same time, don't lose sight of the bigger picture. Sure, `decltype` (both alone and in conjunction with `auto`) may occasionally yield type-deduction surprises, but that's not the normal situation. Normally, `decltype` produces the type you expect.

This is especially true when `decltype` is applied to names, because in that case, `decltype` does just what it sounds like: it reports that name's declared type.

Things to Remember

- `decltype` almost always yields the type of a variable or expression without any modifications.
- For lvalue expressions of type `T` other than names, `decltype` always reports a type of `T&`.
- C++14 supports `decltype(auto)`, which, like `auto`, deduces a type from its initializer, but it performs the type deduction using the `decltype` rules.

Item 4: Know how to view deduced types.

The choice of tools for viewing the results of type deduction is dependent on the phase of the software development process where you want the information. We'll explore three possibilities: getting type deduction information as you edit your code, getting it during compilation, and getting it at runtime.

IDE Editors

Code editors in IDEs often show the types of program entities (e.g., variables, parameters, functions, etc.) when you do something like hover your cursor over the entity. For example, given this code,

```
const int theAnswer = 42;

auto x = theAnswer;
auto y = &theAnswer;
```

an IDE editor would likely show that `x`'s deduced type was `int` and `y`'s was `const int*`.

For this to work, your code must be in a more or less compilable state, because what makes it possible for the IDE to offer this kind of information is a C++ compiler (or at least the front end of one) running inside the IDE. If that compiler can't make enough sense of your code to parse it and perform type deduction, it can't show you what types it deduced.

For simple types like `int`, information from IDEs is generally fine. As we'll see soon, however, when more complicated types are involved, the information displayed by IDEs may not be particularly helpful.

Compiler Diagnostics

An effective way to get a compiler to show a type it has deduced is to use that type in a way that leads to compilation problems. The error message reporting the problem is virtually sure to mention the type that's causing it.

Suppose, for example, we'd like to see the types that were deduced for `x` and `y` in the previous example. We first declare a class template that we *don't define*. Something like this does nicely:

```
template<typename T>       // declaration only for TD;
class TD;                  // TD == "Type Displayer"
```

Attempts to instantiate this template will elicit an error message, because there's no template definition to instantiate. To see the types for `x` and `y`, just try to instantiate `TD` with their types:

```
TD<decltype(x)> xType;     // elicit errors containing
TD<decltype(y)> yType;     // x's and y's types
```

I use variable names of the form *`variableName`*`Type`, because they tend to yield error messages that help me find the information I'm looking for. For the code above, one of my compilers issues diagnostics reading, in part, as follows (I've highlighted the type information we're after):

```
error: aggregate 'TD<int> xType' has incomplete type and
    cannot be defined
error: aggregate 'TD<const int *> yType' has incomplete type
    and cannot be defined
```

A different compiler provides the same information, but in a different form:

```
error: 'xType' uses undefined class 'TD<int>'
error: 'yType' uses undefined class 'TD<const int *>'
```

Formatting differences aside, all the compilers I've tested produce error messages with useful type information when this technique is employed.

Runtime Output

The `printf` approach to displaying type information (not that I'm recommending you use `printf`) can't be employed until runtime, but it offers full control over the formatting of the output. The challenge is to create a textual representation of the type you care about that is suitable for display. "No sweat," you're thinking, "it's `typeid` and `std::type_info::name` to the rescue." In our continuing quest to see the types deduced for `x` and `y`, you may figure we can write this:

```
std::cout << typeid(x).name() << '\n';    // display types for
std::cout << typeid(y).name() << '\n';    // x and y
```

This approach relies on the fact that invoking `typeid` on an object such as `x` or `y` yields a `std::type_info` object, and `std::type_info` has a member function, `name`, that produces a C-style string (i.e., a `const char*`) representation of the name of the type.

Calls to `std::type_info::name` are not guaranteed to return anything sensible, but implementations try to be helpful. The level of helpfulness varies. The GNU and Clang compilers report that the type of `x` is "`i`", and the type of `y` is "`PKi`", for example. These results make sense once you learn that, in output from these compilers, "`i`" means "`int`" and "`PK`" means "pointer to ~~`konst`~~ `const`." (Both compilers support a tool, `c++filt`, that decodes such "mangled" types.) Microsoft's compiler produces less cryptic output: "`int`" for `x` and "`int const *`" for `y`.

Because these results are correct for the types of `x` and `y`, you might be tempted to view the type-reporting problem as solved, but let's not be hasty. Consider a more complex example:

```
template<typename T>                 // template function to
void f(const T& param);              // be called

std::vector<Widget> createVec();     // factory function

const auto vw = createVec();         // init vw w/factory return

if (!vw.empty()) {
  f(&vw[0]);                         // call f
  …
}
```

This code, which involves a user-defined type (`Widget`), an STL container (`std::vector`), and an `auto` variable (`vw`), is more representative of the situations where you might want some visibility into the types your compilers are deducing. For example, it'd be nice to know what types are inferred for the template type parameter `T` and the function parameter `param` in `f`.

Loosing `typeid` on the problem is straightforward. Just add some code to `f` to display the types you'd like to see:

```
template<typename T>
void f(const T& param)
{
  using std::cout;
```

```
    cout << "T =     " << typeid(T).name() << '\n';     // show T

    cout << "param = " << typeid(param).name() << '\n'; // show
    …                                                   // param's
  }                                                     // type
```

Executables produced by the GNU and Clang compilers produce this output:

```
T =     PK6Widget
param = PK6Widget
```

We already know that for these compilers, `PK` means "pointer to `const`," so the only mystery is the number `6`. That's simply the number of characters in the class name that follows (`Widget`). So these compilers tell us that both `T` and `param` are of type `const Widget*`.

Microsoft's compiler concurs:

```
T =     class Widget const *
param = class Widget const *
```

Three independent compilers producing the same information suggests that the information is accurate. But look more closely. In the template `f`, `param`'s declared type is `const T&`. That being the case, doesn't it seem odd that `T` and `param` have the same type? If `T` were `int`, for example, `param`'s type should be `const int&`—not the same type at all.

Sadly, the results of `std::type_info::name` are not reliable. In this case, for example, the type that all three compilers report for `param` are incorrect. Furthermore, they're essentially *required* to be incorrect, because the specification for `std::type_info::name` mandates that the type be treated as if it had been passed to a template function as a by-value parameter. As Item 1 explains, that means that if the type is a reference, its reference-ness is ignored, and if the type after reference removal is `const` (or `volatile`), its `const`ness (or `volatile`ness) is also ignored. That's why `param`'s type—which is `const Widget * const &`—is reported as `const Widget*`. First the type's reference-ness is removed, and then the `const`ness of the resulting pointer is eliminated.

Equally sadly, the type information displayed by IDE editors is also not reliable—or at least not reliably useful. For this same example, one IDE editor I know reports `T`'s type as (I am not making this up):

```
const
std::_Simple_types<std::_Wrap_alloc<std::_Vec_base_types<Widget,
std::allocator<Widget> >::_Alloc>::value_type>::value_type *
```

The same IDE editor shows `param`'s type as:

```
const std::_Simple_types<...>::value_type *const &
```

That's less intimidating than the type for `T`, but the "`...`" in the middle is confusing until you realize that it's the IDE editor's way of saying "I'm omitting all that stuff that's part of `T`'s type." With any luck, your development environment does a better job on code like this.

If you're more inclined to rely on libraries than luck, you'll be pleased to know that where `std::type_info::name` and IDEs may fail, the Boost TypeIndex library (often written as *Boost.TypeIndex*) is designed to succeed. The library isn't part of Standard C++, but neither are IDEs or templates like `TD`. Furthermore, the fact that Boost libraries (available at *boost.com*) are cross-platform, open source, and available under a license designed to be palatable to even the most paranoid corporate legal team means that code using Boost libraries is nearly as portable as code relying on the Standard Library.

Here's how our function `f` can produce accurate type information using Boost.TypeIndex:

```
#include <boost/type_index.hpp>

template<typename T>
void f(const T& param)
{
  using std::cout;
  using boost::typeindex::type_id_with_cvr;

  // show T
  cout << "T =     "
       << type_id_with_cvr<T>().pretty_name()
       << '\n';

  // show param's type
  cout << "param = "
       << type_id_with_cvr<decltype(param)>().pretty_name()
       << '\n';
  …
}
```

The way this works is that the function template `boost::typeindex::type_id_with_cvr` takes a type argument (the type about which we want information) and *doesn't* remove `const`, `volatile`, or reference qualifiers (hence the "`with_cvr`" in the template name). The result is a `boost::typeindex::type_index` object, whose `pretty_name` member function produces a `std::string` containing a human-friendly representation of the type.

With this implementation for `f`, consider again the call that yields incorrect type information for `param` when `typeid` is used:

```
std::vector<Widget> createVec();      // factory function

const auto vw = createVec();          // init vw w/factory return

if (!vw.empty()) {
  f(&vw[0]);                          // call f
  …
}
```

Under compilers from GNU and Clang, Boost.TypeIndex produces this (accurate) output:

```
T =     Widget const*
param = Widget const* const&
```

Results under Microsoft's compiler are essentially the same:

```
T =     class Widget const *
param = class Widget const * const &
```

Such near-uniformity is nice, but it's important to remember that IDE editors, compiler error messages, and libraries like Boost.TypeIndex are merely tools you can use to help you figure out what types your compilers are deducing. All can be helpful, but at the end of the day, there's no substitute for understanding the type deduction information in Items 1–3.

Things to Remember

- Deduced types can often be seen using IDE editors, compiler error messages, and the Boost TypeIndex library.
- The results of some tools may be neither helpful nor accurate, so an understanding of C++'s type deduction rules remains essential.