Item 4: Know how to view deduced types.

People who want to see the types that compilers deduce usually fall into one of two camps. The first are the pragmatists. They're typically motivated by a behavioral problem in their software (i.e., they're debugging), and they're looking for insights into compilation that can help them identify the source of the problem. The second are the experimentalists. They're exploring the type deduction rules described in Items 1-3. Often, they want to confirm their predictions about the results of various type deduction scenarios ("For this code, I think compilers will deduce *this* type..."), but sometimes they simply want to answer "what if" questions. "How," they might wonder, "do the results of template type deduction change if I replace a universal reference (see Item 26) with an lvalue-reference-to-`const` parameter (i.e., replace `T&&` with `const T&` in a function template parameter list)?"

Regardless of the camp you fall into (both are legitimate), the tools you have at your disposal depend on the phase of the software development process where you'd like to see the types your compilers have inferred. 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 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.

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;
                           // see Item 3 for decltype info
```

I use variable names of the form *`variableName`*`Type`, because they tend to yield quite informative error messages. For the code above, one of my compilers issues diagnositics reading, in part, as follows. (I've highlighted the type information we're looking for.)

```
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 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 more 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 being processed 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`, its `const`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 result type 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 disturbing 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."

My understanding is that most of what's displayed here is `typedef` cruft and that once you push through the `typedef`s to get to the underlying type information, you get what you're looking for, but having to do that work pretty much eliminates any utility the display of the types in the IDE originally promised. With any luck, your IDE editor does a better job on code like this.

In my experience, compiler diagnostics are a more dependable source of information about the results of type deduction. Revising the template `f`'s implementation to instantiate the declared-but-not-defined template `TD` yields this:

```
template<typename T>
void f(const T& param)
```

```
{
  TD<T> TType;                       // elicit errors containing
  TD<decltype(param)> paramType;     // T's and param's types
  …
}
```

Each of GNU's, Clang's, and Microsoft's compilers produce error messages with the correct types for `T` and `param`. The exact message contents and formats vary, but as an example, this is what GNU's compiler issues (after minor reformatting):

```
error: 'TD<const Widget *> TType' has incomplete type
error: 'TD<const Widget * const &> paramType' has incomplete
       type
```

Beyond `typeid`

If you want accurate runtime information about deduced types, we've seen that `typeid` is not a reliable route to getting it. One way to work around that is to implement your own mechanism for mapping from a type to its displayable representation. In concept, it's not difficult: you just use type traits and template metaprogramming (see Item 9) to break a type into its various components (using type traits such as `std::is_const`, `std::is_pointer`, `std::is_lvalue_reference`, etc.), and you create a string representation of the type from textual representations of each of its parts. (You'd still be dependent on `typeid` and `std::type_info::name` to generate string representations of the names of user-defined classes, though.)

If you'd use such a facility often enough to justify the effort needed to write, debug, document, and maintain it, that's a reasonable approach. But if you're willing to live with a little platform-dependent code that's easy to implement and produces better results than those based on `typeid`, it's worth noting that many compilers support a language extension that yields a printable representation of the full signature for a function, including, for functions generated from templates, types for both template and function parameters.

For example, the GNU and Clang compilers support a construct called `__PRETTY_FUNCTION__`, and Microsoft's compiler offers `__FUNCSIG__`. These constructs represent a variable (for GNU and Clang) or a macro (for Microsoft) whose value is the signature of the containing function. If we reimplement our template `f` like this,

```
template<typename T>
void f(const T& param)
{
#if defined(__GNUC__)                              // For GNU and
  std::cout << __PRETTY_FUNCTION__ << '\n';     // Clang
#elif defined(_MSC_VER)
  std::cout << __FUNCSIG__ << '\n';             // For Microsoft
#endif

  …

}
```

and call `f` as we did before,

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

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

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

we get the following result from GNU:

```
void f(const T&) [with T = const Widget*]
```

This tells us that `T` has been deduced to be `const Widget*` (the same thing we got via `typeid`, but without the "PK" encoding and the "6" in front of the class name), but it also tells us that `f`'s parameter has type `const T&`. If we expand `T` in that formulation, we get `const Widget * const &`. That's different from what `typeid` told us, though it's the same as the type in the error message provoked by use of the declared-but-not-defined `TD` template. It's also correct.

Use of Microsoft's `__FUNCSIG__` produces this output:

```
void __cdecl f<const classWidget*>(const class Widget *const &)
```

The type inside the angle brackets is the type deduced for `T`: `const Widget*`. This, too, is what we got via `typeid`. But the type inside parentheses is the type deduced for `param`: `const Widget * const&`. That's not what `typeid` told us, though, again, it's the same as the (correct) information we'd get during compilation from use of the TD template.

Clang's function-signature-reporting facility, despite using the same name as GNU's (__PRETTY_FUNCTION__), is not as forthcoming as GNU's or Microsoft's. It yields simply:

```
void f(const Widget *const &)
```

This shows param's type directly, but it leaves it up to you to deduce that T's type must have been const Widget* (or to rely on the information provided via typeid).

IDE editors, compiler error messages, typeid, and language extensions like __PRETTY_FUNCTION__ and __FUNCSIG__ are merely tools you can use to help you figure out what types your compilers are deducing for you. 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, typeid, and language extensions such as __PRETTY_FUNCTION__ and __FUNCSIG__.
- The results of such tools may be neither helpful nor accurate, so an understanding of C++'s type deduction rules remains essential.