# Item 24: Distinguish universal references from rvalue references.

It's been said that the truth shall set you free, but under the right circumstances, a well-chosen lie can be equally liberating. This Item is such a lie. Because we're dealing with software, however, let's eschew the word "lie" and instead say that this Item comprises an "abstraction."

To declare an rvalue reference to some type `T`, you write `T&&`. It thus seems reasonable to assume that if you see "`T&&`" in source code, you're looking at an rvalue reference. Alas, it's not quite that simple:

```
void f(Widget&& param);              // rvalue reference

Widget&& var1 = Widget();            // rvalue reference

auto&& var2 = var1;                  // not rvalue reference

template<typename T>
void f(std::vector<T>&& param);      // rvalue reference

template<typename T>
void f(T&& param);                   // not rvalue reference
```

In fact, "`T&&`" has two different meanings. One is rvalue reference, of course. Such references behave exactly the way you expect: they bind only to rvalues, and their primary *raison d'être* is to identify objects that may be moved from.

The other meaning for "`T&&`" is *either* rvalue reference *or* lvalue reference. Such references look like rvalue references in the source code (i.e., "`T&&`"), but they can behave as if they were lvalue references (i.e., "`T&`"). Their dual nature permits them to bind to rvalues (like rvalue references) as well as lvalues (like lvalue references). Furthermore, they can bind to `const` or non-`const` objects, to `volatile` or non-`volatile` objects, even to objects that are both `const` and `volatile`. They can bind to virtually *anything*. Such unprecedentedly flexible references deserve a name of their own. I call them *universal references*.[1]

Universal references arise in two contexts. The most common is function template parameters, such as this example from the sample code above:

1 Item 25 explains that universal references should almost always have `std::forward` applied to them, and as this book goes to press, some members of the C++ community have started referring to universal references as *forwarding references*.

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

The second context is `auto` declarations, including this one from the sample code above:

```
auto&& var2 = var1;             // var2 is a universal reference
```

What these contexts have in common is the presence of *type deduction*. In the template `f`, the type of `param` is being deduced, and in the declaration for `var2`, `var2`'s type is being deduced. Compare that with the following examples (also from the sample code above), where type deduction is missing. If you see "`T&&`" without type deduction, you're looking at an rvalue reference:

```
void f(Widget&& param);         // no type deduction;
                                // param is an rvalue reference

Widget&& var1 = Widget();       // no type deduction;
                                // var1 is an rvalue reference
```

Because universal references are references, they must be initialized. The initializer for a universal reference determines whether it represents an rvalue reference or an lvalue reference. If the initializer is an rvalue, the universal reference corresponds to an rvalue reference. If the initializer is an lvalue, the universal reference corresponds to an lvalue reference. For universal references that are function parameters, the initializer is provided at the call site:

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

Widget w;
f(w);                   // lvalue passed to f; param's type is
                        // Widget& (i.e., an lvalue reference)

f(std::move(w));        // rvalue passed to f; param's type is
                        // Widget&& (i.e., an rvalue reference)
```

For a reference to be universal, type deduction is necessary, but it's not sufficient. The *form* of the reference declaration must also be correct, and that form is quite constrained. It must be precisely "`T&&`". Look again at this example from the sample code we saw earlier:

```
template<typename T>
void f(std::vector<T>&& param);  // param is an rvalue reference
```

When `f` is invoked, the type `T` will be deduced (unless the caller explicitly specifies it, an edge case we'll not concern ourselves with). But the form of `param`'s type declara-

tion isn't "T&&", it's "std::vector<T>&&". That rules out the possibility that param is a universal reference. param is therefore an rvalue reference, something that your compilers will be happy to confirm for you if you try to pass an lvalue to f:

```
std::vector<int> v;
f(v);                                   // error! can't bind lvalue to
                                        // rvalue reference
```

Even the simple presence of a const qualifier is enough to disqualify a reference from being universal:

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

If you're in a template and you see a function parameter of type "T&&", you might think you can assume that it's a universal reference. You can't. That's because being in a template doesn't guarantee the presence of type deduction. Consider this push_back member function in std::vector:

```
template<class T, class Allocator = allocator<T>>  // from C++
class vector {                                     // Standards
public:
  void push_back(T&& x);
  …
};
```

push_back's parameter certainly has the right form for a universal reference, but there's no type deduction in this case. That's because push_back can't exist without a particular vector instantiation for it to be part of, and the type of that instantiation fully determines the declaration for push_back. That is, saying

```
std::vector<Widget> v;
```

causes the std::vector template to be instantiated as follows:

```
class vector<Widget, allocator<Widget>> {
public:
  void push_back(Widget&& x);                  // rvalue reference
  …
};
```

Now you can see clearly that push_back employs no type deduction. This push_back for vector<T> (there are two—the function is overloaded) always declares a parameter of type rvalue-reference-to-T.

In contrast, the conceptually similar emplace_back member function in std::vector *does* employ type deduction:

```
template<class T, class Allocator = allocator<T>>  // still from
class vector {                                     // C++
public:                                            // Standards
  template <class... Args>
  void emplace_back(Args&&... args);
  …
};
```

Here, the type parameter `Args` is independent of `vector`'s type parameter `T`, so `Args` must be deduced each time `emplace_back` is called. (Okay, `Args` is really a parameter pack, not a type parameter, but for purposes of this discussion, we can treat it as if it were a type parameter.)

The fact that `emplace_back`'s type parameter is named `Args`, yet it's still a universal reference, reinforces my earlier comment that it's the *form* of a universal reference that must be "`T&&`". There's no requirement that you use the name `T`. For example, the following template takes a universal reference, because the form ("*type*`&&`") is right, and `param`'s type will be deduced (again, excluding the corner case where the caller explicitly specifies the type):

```
template<typename MyTemplateType>          // param is a
void someFunc(MyTemplateType&& param);     // universal reference
```

I remarked earlier that `auto` variables can also be universal references. To be more precise, variables declared with the type `auto&&` are universal references, because type deduction takes place and they have the correct form ("`T&&`"). `auto` universal references are not as common as universal references used for function template parameters, but they do crop up from time to time in C++11. They crop up a lot more in C++14, because C++14 lambda expressions may declare `auto&&` parameters. For example, if you wanted to write a C++14 lambda to record the time taken in an arbitrary function invocation, you could do this:

```
auto timeFuncInvocation =
  [](auto&& func, auto&&... params)                // C++14
  {
    start timer;
    std::forward<decltype(func)>(func)(            // invoke func
      std::forward<decltype(params)>(params)...    // on params
      );
    stop timer and record elapsed time;
  };
```

If your reaction to the "`std::forward<decltype(`*blah blah blah*`)>`" code inside the lambda is, "What the…?!", that probably just means you haven't yet read Item 33. Don't worry about it. The important thing in this Item is the `auto&&` parameters that

the lambda declares. `func` is a universal reference that can be bound to any callable object, lvalue or rvalue. `args` is zero or more universal references (i.e., a universal reference parameter pack) that can be bound to any number of objects of arbitrary types. The result, thanks to `auto` universal references, is that `timeFuncInvocation` can time pretty much any function execution. (For information on the difference between "any" and "pretty much any," turn to Item 30.)

Bear in mind that this entire Item—the foundation of universal references—is a lie… er, an "abstraction." The underlying truth is known as *reference collapsing*, a topic to which Item 28 is dedicated. But the truth doesn't make the abstraction any less useful. Distinguishing between rvalue references and universal references will help you read source code more accurately ("Does that `T&&` I'm looking at bind to rvalues only or to everything?"), and it will avoid ambiguities when you communicate with your colleagues ("I'm using a universal reference here, not an rvalue reference…"). It will also allow you to make sense of Items 25 and 26, which rely on the distinction. So embrace the abstraction. Revel in it. Just as Newton's laws of motion (which are technically incorrect) are typically just as useful as and easier to apply than Einstein's theory of general relativity ("the truth"), so is the notion of universal references normally preferable to working through the details of reference collapsing.

**Things to Remember**

- If a function template parameter has type `T&&` for a deduced type `T`, or if an object is declared using `auto&&`, the parameter or object is a universal reference.
- If the form of the type declaration isn't precisely *`type`*`&&`, or if type deduction does not occur, *`type`*`&&` denotes an rvalue reference.
- Universal references correspond to rvalue references if they're initialized with rvalues. They correspond to lvalue references if they're initialized with lvalues.

# Item 25: Use `std::move` on rvalue references, `std::forward` on universal references.

Rvalue references bind only to objects that are candidates for moving. If you have an rvalue reference parameter, you *know* that the object it's bound to may be moved:

```
class Widget {
  Widget(Widget&& rhs);          // rhs definitely refers to an
```