Universal References in C++11 -- Scott Meyers

By Blog Staff | Nov 1, 2012 01:07 PM | Tags: intermediate advanced

FEATURES
Current ISO C++ status
Upcoming ISO C++ meetings
Upcoming C++ conferences
Compiler conformance status

CATEGORIES
News
Product News
Articles & Books
Video & On-Demand
Events
Training
Standardization

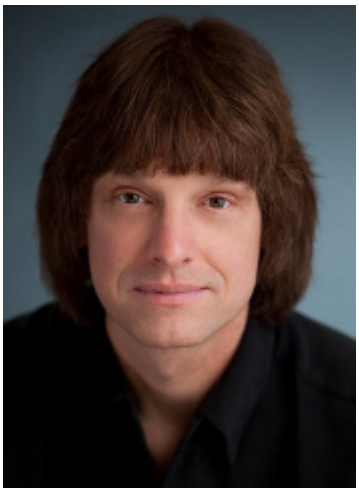
TAGS
basics intermediate advanced experimental

SUBSCRIBE
All Posts
All Comments
News
Product News
Articles & Books
Video & On-Demand
Events
Training

Universal References in C++11

T&& Doesn't Always Mean "Rvalue Reference"

by Scott Meyers



Related materials:

- A video of Scott's C&B talk based on this material is available on Channel 9.
- A black-and-white PDF version of this article is available in Overload 111.

Perhaps the most significant new feature in C++11 is rvalue references; they're the foundation on which move semantics and perfect forwarding

Standardization

ARCHIVES
October 2020
September 2020
August 2020
July 2020
June 2020
May 2020
April 2020
March 2020
February 2020
January 2020
December 2019
November 2019

UPCOMING EVENTS
Meeting C++ 2020
Nov 12-14, Berlin | ONLINE EVENT

code::dive 2020
Nov 18, ONLINE EVENT

TWITTER TIMELINE
Tweets by isocpp

are built. (If you're unfamiliar with the basics of rvalue references, move semantics, or perfect forwarding, you may wish to read Thomas Becker's overview before continuing.)

Syntactically, rvalue references are declared like "normal" references (now known as *lvalue references*), except you use two ampersands instead of one. This function takes a parameter of type rvalue-reference-to-`Widget`:

```
void f(Widget&& param);
```

Given that rvalue references are declared using "&&", it seems reasonable to assume that the presence of "&&" in a type declaration indicates an rvalue reference. That is not the case:

```
Widget&& var1 = someWidget;      // here, "&&" means rvalue reference

auto&& var2 = var1;              // here, "&&" does not mean rvalue refe

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

template<typename T>
void f(T&& param);               // here, "&&"does not mean rvalue refer
```

In this article, I describe the two meanings of "&&" in type declarations, explain how to tell them apart, and introduce new terminology that makes it possible to unambiguously communicate which meaning of "&&" is intended. Distinguishing the different meanings is important, because if you think "rvalue reference" whenever you see "&&" in a type declaration, you'll misread a lot of C++11 code.

The essence of the issue is that "&&" in a type declaration sometimes means rvalue reference, but sometimes it means *either* rvalue reference *or* lvalue reference. As such, some occurrences of "&&" in source code may actually have the meaning of "&", i.e., have the syntactic *appearance* of an rvalue reference ("&&"), but the *meaning* of an lvalue reference ("&"). References where this is possible are more flexible than either lvalue references or rvalue references. Rvalue references may bind only to rvalues, for example, and lvalue references, in addition to being able to bind to lvalues, may bind to rvalues only under restricted circumstances.[1] In contrast, references declared with "&&" that may be

either lvalue references or rvalue references may bind to *anything*. Such unusually flexible references deserve their own name. I call them *universal references*.

The details of when "`&&`" indicates a universal reference (i.e., when "`&&`" in source code might actually mean "`&`") are tricky, so I'm going to postpone coverage of the minutiae until later. For now, let's focus on the following rule of thumb, because that is what you need to remember during day-to-day programming:

> If a variable or parameter is declared to have type **`T&&`** for some **deduced type** `T`, that variable or parameter is a *universal reference*.

The requirement that type deduction be involved limits the situations where universal references can be found. In practice, almost all universal references are parameters to function templates. Because the type deduction rules for `auto`-declared variables are essentially the same as for templates, it's also possible to have `auto`-declared universal references. These are uncommon in production code, but I show some in this article, because they are less verbose in examples than templates. In the Nitty Gritty Details section of this article, I explain that it's also possible for universal references to arise in conjunction with uses of `typedef` and `decltype`, but until we get down to the nitty gritty details, I'm going to proceed as if universal references pertained only to function template parameters and `auto`-declared variables.

The constraint that the form of a universal reference be `T&&` is more significant than it may appear, but I'll defer examination of that until a bit later. For now, please simply make a mental note of the requirement.

Like all references, universal references must be initialized, and it is a universal reference's initializer that determines whether it represents an lvalue reference or an rvalue reference:

- If the expression initializing a universal reference is an lvalue, the universal reference becomes an lvalue reference.
- If the expression initializing the universal reference is an rvalue, the universal reference becomes an rvalue reference.

This information is useful only if you are able to distinguish lvalues from rvalues. A precise definition for these terms is difficult to develop (the C++11 standard generally specifies whether an expression is an lvalue or an rvalue on a case-by-case basis), but in practice, the following suffices:

- If you can take the address of an expression, the expression is an lvalue.
- If the type of an expression is an lvalue reference (e.g., `T&` or `const T&`, etc.), that expression is an lvalue.
- Otherwise, the expression is an rvalue. Conceptually (and typically also in fact), rvalues correspond to temporary objects, such as those returned from functions or created through implicit type conversions. Most literal values (e.g., `10` and `5.3`) are also rvalues.

Consider again the following code from the beginning of this article:

```
Widget&& var1 = someWidget;
auto&& var2 = var1;
```

You can take the address of `var1`, so `var1` is an lvalue. `var2`'s type declaration of `auto&&` makes it a universal reference, and because it's being initialized with `var1` (an lvalue), `var2` becomes an lvalue reference. A casual reading of the source code could lead you to believe that `var2` was an rvalue reference; the "&&" in its declaration certainly suggests that conclusion. But because it is a universal reference being initialized with an lvalue, `var2` becomes an lvalue reference. It's as if `var2` were declared like this:

```
Widget& var2 = var1;
```

As noted above, if an expression has type lvalue reference, it's an lvalue. Consider this example:

```
std::vector<int> v;
...
auto&& val = v[0];              // val becomes an lvalue reference (see
```

`val` is a universal reference, and it's being initialized with `v[0]`, i.e., with the result of a call to `std::vector<int>::operator[]`. That

function returns an lvalue reference to an element of the vector.[2] Because all lvalue references are lvalues, and because this lvalue is used to initialize `val`, `val` becomes an lvalue reference, even though it's declared with what looks like an rvalue reference.

I remarked that universal references are most common as parameters in template functions. Consider again this template from the beginning of this article:

```
template<typename T>
void f(T&& param);              // "&&" might mean rvalue reference
```

Given this call to f,

```
f(10);                          // 10 is an rvalue
```

`param` is initialized with the literal `10`, which, because you can't take its address, is an rvalue. That means that in the call to `f`, the universal reference `param` is initialized with an rvalue, so `param` becomes an rvalue reference – in particular, `int&&`.

On the other hand, if `f` is called like this,

```
int x = 10;
f(x);                           // x is an lvalue
```

`param` is initialized with the variable `x`, which, because you can take its address, is an lvalue. That means that in this call to `f`, the universal reference param is initialized with an lvalue, and param therefore becomes an lvalue reference – `int&`, to be precise.

The comment next to the declaration of `f` should now be clear: whether `param`'s type is an lvalue reference or an rvalue reference depends on what is passed when `f` is called. Sometimes param becomes an lvalue reference, and sometimes it becomes an rvalue reference. `param` really is a *universal reference*.

Remember that "&&" indicates a universal reference *only where type deduction takes place*. Where there's no type deduction, there's no universal reference. In such cases, "&&" in type declarations always means rvalue reference. Hence:

```
template<typename T>
void f(T&& param);               // deduced parameter type ⇒ type deduct
                                 // && ≡ universal reference

template<typename T>
class Widget {
    ...
    Widget(Widget&& rhs);        // fully specified parameter type ⇒ no
    ...                          // && ≡ rvalue reference
};

template<typename T1>
class Gadget {
    ...
    template<typename T2>
    Gadget(T2&& rhs);            // deduced parameter type ⇒ type deduct
    ...                          // && ≡ universal reference
};

void f(Widget&& param);          // fully specified parameter type ⇒ no
                                 // && ≡ rvalue reference
```

There's nothing surprising about these examples. In each case, if you see `T&&` (where `T` is a template parameter), there's type deduction, so you're looking at a universal reference. And if you see "`&&`" after a particular type name (e.g., `Widget&&`), you're looking at an rvalue reference.

I stated that the form of the reference declaration must be "`T&&`" in order for the reference to be universal. That's an important caveat. Look again at this declaration from the beginning of this article:

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

Here, we have both type deduction and a "`&&`"-declared function parameter, but the form of the parameter declaration is not "`T&&`", it's "`std::vector<t>&&`". As a result, the parameter is a normal rvalue reference, not a universal reference. Universal references can only occur in the form "`T&&`"! Even the simple addition of a `const` qualifier is enough to disable the interpretation of "`&&`" as a universal reference:

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

Now, "`T&&`" is simply the required *form* for a universal reference. It doesn't mean you have to use the name `T` for your template parameter:

```
template<typename MyTemplateParamType>
void f(MyTemplateParamType&& param);  // "&&" means universal reference
```

Sometimes you can see `T&&` in a function template declaration where `T` is a template parameter, yet there's still no type deduction. Consider this `push_back` function in `std::vector`:[3]

```
template <class T, class Allocator = allocator<T> >
class vector {
public:
    ...
    void push_back(T&& x);       // fully specified parameter type ⇒ no
    ...                          // && ≡ rvalue reference
};
```

Here, `T` is a template parameter, and `push_back` takes a `T&&`, yet the parameter is not a universal reference! How can that be?

The answer becomes apparent if we look at how `push_back` would be declared outside the class. I'm going to pretend that `std::vector`'s `Allocator` parameter doesn't exist, because it's irrelevant to the discussion, and it just clutters up the code. With that in mind, here's the declaration for this version of `std::vector::push_back`:

```
template <class T>
void vector<T>::push_back(T&& x);
```

`push_back` can't exist without the class `std::vector<T>` that contains it. But if we have a class `std::vector<T>`, we already know what `T` is, so there's no need to deduce it.

An example will help. If I write

```
Widget makeWidget();             // factory function for Widget
std::vector<Widget> vw;
...

Widget w;
vw.push_back(makeWidget());      // create Widget from factory, add it t
```

my use of `push_back` will cause the compiler to instantiate that function for the class `std::vector<Widget>`. The declaration for that `push_back` looks like this:

```
void std::vector<Widget>::push_back(Widget&& x);
```

See? Once we know that the class is `std::vector<Widget>`, the type of `push_back`'s parameter is fully determined: it's `Widget&&`.

There's no role here for type deduction.

Contrast that with `std::vector`'s `emplace_back`, which is declared like this:

```
template <class T, class Allocator = allocator<T> >
class vector {
public:
    ...
    template <class... Args>
    void emplace_back(Args&&... args); // deduced parameter types ⇒ type
    ...                                // && ≡ universal references
};
```

Don't let the fact that `emplace_back` takes a variable number of arguments (as indicated by the ellipses in the declarations for `Args` and `args`) distract you from the fact that a type for each of those arguments must be deduced. The function template parameter `Args` is independent of the class template parameter `T`, so even if we know that the class is, say, `std::vector<Widget>`, that doesn't tell us the type(s) taken by `emplace_back`. The out-of-class declaration for `emplace_back` for `std::vector<Widget>` makes that clear (I'm continuing to ignore the existence of the `Allocator` parameter):

```
template<class... Args>
void std::vector<Widget>::emplace_back(Args&&... args);
```

Clearly, knowing that the class is `std::vector<Widget>` doesn't eliminate the need for the compiler to deduce the type(s) passed to `emplace_back`. As a result, `std::vector::emplace_back`'s parameters are universal references, unlike the parameter to the version of `std::vector::push_back` we examined, which is an rvalue reference.

A final point is worth bearing in mind: the lvalueness or rvalueness of an expression is independent of its type. Consider the type `int`. There are lvalues of type `int` (e.g., variables declared to be `int`s), and there are rvalues of type `int` (e.g., literals like `10`). It's the same for user-defined types like `Widget`. A `Widget` object can be an lvalue (e.g., a `Widget` variable) or an rvalue (e.g., an object returned from a `Widget`-creating factory function). The type of an expression does not tell you whether it is an lvalue or an rvalue.

Because the lvalueness or rvalueness of an expression is independent of its type, it's possible to have *lvalues* whose type is *rvalue reference*, and it's also possible to have *rvalues* of the type *rvalue reference*:

```
Widget makeWidget();                        // factory function for Widge

Widget&& var1 = makeWidget()                // var1 is an lvalue, but
                                            // its type is rvalue referen

Widget var2 = static_cast<Widget&&>(var1); // the cast expression yields
                                            // its type is rvalue referen
```

The conventional way to turn lvalues (such as `var1`) into rvalues is to use `std::move` on them, so `var2` could be defined like this:

```
Widget var2 = std::move(var1);              // equivalent to above
```

I initially showed the code with `static_cast` only to make explicit that the type of the expression was an rvalue reference (`Widget&&`).

Named variables and parameters of rvalue reference type are lvalues. (You can take their addresses.) Consider again the `Widget` and `Gadget` templates from earlier:

```
template<typename T>
class Widget {
    ...
    Widget(Widget&& rhs);           // rhs's type is rvalue reference,
    ...                             // but rhs itself is an lvalue
};

template<typename T1>
class Gadget {
    ...
    template <typename T2>
    Gadget(T2&& rhs);               // rhs is a universal reference whose t
    ...                             // eventually become an rvalue referenc
};                                  // an lvalue reference, but rhs itself
```

In `Widget`'s constructor, `rhs` is an rvalue reference, so we know it's bound to an rvalue (i.e., an rvalue was passed to it), but `rhs` itself is an lvalue, so we have to convert it back to an rvalue if we want to take advantage of the rvalueness of what it's bound to. Our motivation for this is generally to use it as the source of a move operation, and that's why the way to convert an lvalue to an rvalue is to use `std::move`. Similarly, `rhs` in `Gadget`'s constructor is a universal reference, so it might be bound to an lvalue or to an rvalue, but regardless of what it's

bound to, `rhs` itself is an lvalue. If it's bound to an rvalue and we want to take advantage of the rvalueness of what it's bound to, we have to convert `rhs` back into an rvalue. If it's bound to an lvalue, of course, we don't want to treat it like an rvalue. This ambiguity regarding the lvalueness and rvalueness of what a universal reference is bound to is the motivation for `std::forward`: to take a universal reference lvalue and convert it into an rvalue only if the expression it's bound to is an rvalue. The name of the function ("`forward`") is an acknowledgment that our desire to perform such a conversion is virtually always to preserve the calling argument's lvalueness or rvalueness when passing – *forwarding* – it to another function.

But `std::move` and `std::forward` are not the focus of this article. The fact that "&&" in type declarations may or may not declare an rvalue reference is. To avoid diluting that focus, I'll refer you to the references in the Further Information section for information on `std::move` and `std::forward`.

Nitty Gritty Details

The true core of the issue is that some constructs in C++11 give rise to references to references, and references to references are not permitted in C++. If source code explicitly contains a reference to a reference, the code is invalid:

```
Widget w1;
...
Widget& & w2 = w1;              // error! No such thing as "reference t
```

There are cases, however, where references to references arise as a result of type manipulations that take place during compilation, and in such cases, rejecting the code would be problematic. We know this from experience with the initial standard for C++, i.e., C++98/C++03.

During type deduction for a template parameter that is a universal reference, lvalues and rvalues of the same type are deduced to have slightly different types. In particular, lvalues of type `T` are deduced to be of type `T&` (i.e., lvalue reference to `T`), while rvalues of type `T` are

deduced to be simply of type T. (Note that while lvalues are deduced to be lvalue references, rvalues are not deduced to be rvalue references!) Consider what happens when a template function taking a universal reference is invoked with an rvalue and with an lvalue:

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

...

int x;

...

f(10);                          // invoke f on rvalue
f(x);                           // invoke f on lvalue
```

In the call to f with the rvalue 10, T is deduced to be int, and the instantiated f looks like this:

```
void f(int&& param);            // f instantiated from rvalue
```

That's fine. In the call to f with the lvalue x, however, T is deduced to be int&, and f's instantiation contains a reference to a reference:

```
void f(int& && param);          // initial instantiation of f with lval
```

Because of the reference-to-reference, this instantiated code is *prima facie* invalid, but the source code– "f(x)" – is completely reasonable. To avoid rejecting it, C++11 performs "reference collapsing" when references to references arise in contexts such as template instantiation.

Because there are two kinds of references (lvalue references and rvalue references), there are four possible reference-reference combinations: lvalue reference to lvalue reference, lvalue reference to rvalue reference, rvalue reference to lvalue reference, and rvalue reference to rvalue reference. There are only two reference-collapsing rules:

- An rvalue reference to an rvalue reference becomes ("collapses into") an rvalue reference.
- All other references to references (i.e., all combinations involving an lvalue reference) collapse into an lvalue reference.

Applying these rules to the instantiation of f on an lvalue yields the following valid code, which is how the compiler treats the call:

```
void f(int& param);              // instantiation of f with lvalue after
```

This demonstrates the precise mechanism by which a universal reference can, after type deduction and reference collapsing, become an lvalue reference. The truth is that a universal reference is really just an rvalue reference in a reference-collapsing context.

Things get subtler when deducing the type for a variable that is itself a reference. In that case, the reference part of the type is ignored. For example, given

```
int x;

...

int&& r1 = 10;                   // r1's type is int&&

int& r2 = x;                     // r2's type is int&
```

the type for both `r1` and `r2` is considered to be `int` in a call to the template `f`. This reference-stripping behavior is independent of the rule that, during type deduction for universal references, lvalues are deduced to be of type `T&` and rvalues of type `T`, so given these calls,

```
f(r1);

f(r2);
```

the deduced type for both `r1` and `r2` is `int&`. Why? First the reference parts of `r1`'s and `r2`'s types are stripped off (yielding `int` in both cases), then, because each is an lvalue, each is treated as `int&` during type deduction for the universal reference parameter in the call to `f`.

Reference collapsing occurs, as I've noted, in "contexts such as template instantiation." A second such context is the definition of `auto` variables. Type deduction for `auto` variables that are universal references is essentially identical to type deduction for function template parameters that are universal references, so lvalues of type `T` are deduced to have type `T&`, and rvalues of type `T` are deduced to have type `T`. Consider again this example from the beginning of this article:

```
Widget&& var1 = someWidget;      // var1 is of type Widget&& (no use of

auto&& var2 = var1;              // var2 is of type Widget& (see below)
```

`var1` is of type `Widget&&`, but its reference-ness is ignored during type deduction in the initialization of `var2`; it's considered to be of type `Widget`. Because it's an lvalue being used to initialize a universal reference (`var2`), its deduced type is `Widget&`. Substituting `Widget&` for `auto` in the definition for `var2` yields the following invalid code,

```
Widget& && var2 = var1;          // note reference-to-reference
```

which, after reference collapsing, becomes

```
Widget& var2 = var1;             // var2 is of type Widget&
```

A third reference-collapsing context is `typedef` formation and use. Given this class template,

```
template<typename T>
class Widget {
    typedef T& LvalueRefType;
    ...
};
```

and this use of the template,

```
Widget<int&> w;
```

the instantiated class would contain this (invalid) typedef:

```
typedef int& & LvalueRefType;
```

Reference-collapsing reduces it to this legitimate code:

```
typedef int& LvalueRefType;
```

If we then use this `typedef` in a context where references are applied to it, e.g.,

```
void f(Widget<int&>::LvalueRefType&& param);
```

the following invalid code is produced after expansion of the `typedef`,

```
void f(int& && param);
```

but reference-collapsing kicks in, so `f`'s ultimate declaration is this:

```
void f(int& param);
```

The final context in which reference-collapsing takes place is the use of `decltype`. As is the case with templates and `auto`, `decltype` performs type deduction on expressions that yield types that are either `T` or `T&`, and `decltype` then applies C++11's reference-collapsing rules. Alas, the type-deduction rules employed by `decltype` are not the same as those used during template or `auto` type deduction. The details are too arcane for coverage here (the Further Information section provides pointers to, er, further information), but a noteworthy difference is that `decltype`, given a named variable of non-reference type, deduces the type `T` (i.e., a non-reference type), while under the same conditions, templates and `auto` deduce the type `T&`. Another important difference is that `decltype`'s type deduction depends only on the `decltype` expression; the type of the initializing expression (if any) is ignored. Ergo:

```
Widget w1, w2;

auto&& v1 = w1;                 // v1 is an auto-based universal refere
                                // initialized with an lvalue, so v1 be
                                // lvalue reference referring to w1.

decltype(w1)&& v2 = w2;         // v2 is a decltype-based universal ref
                                // decltype(w1) is Widget, so v2 become
                                // w2 is an lvalue, and it's not legal
                                // rvalue reference with an lvalue, so
                                // this code does not compile.
```

Summary

In a type declaration, "&&" indicates either an rvalue reference or a *universal reference* – a reference that may resolve to either an lvalue reference or an rvalue reference. Universal references always have the form `T&&` for some deduced type `T`.

Reference collapsing is the mechanism that leads to universal references (which are really just rvalue references in situations where reference-collapsing takes place) sometimes resolving to lvalue references and sometimes to rvalue references. It occurs in specified contexts where references to references may arise during compilation. Those contexts

are template type deduction, `auto` type deduction, `typedef` formation and use, and `decltype` expressions.

Acknowledgments

Draft versions of this article were reviewed by Cassio Neri, Michal Mocny, Howard Hinnant, Andrei Alexandrescu, Stephan T. Lavavej, Roger Orr, Chris Oldwood, Jonathan Wakely, and Anthony Williams. Their comments contributed to substantial improvements in the content of the article as well as in its presentation.

Notes

[1] I discuss rvalues and their counterpart, lvalues, later in this article. The restriction on lvalue references binding to rvalues is that such binding is permitted only when the lvalue reference is declared as a reference-to-`const`, i.e., a `const T&`.

[2] I'm ignoring the possibility of bounds violations. They yield undefined behavior.

[3] `std::vector::push_back` is overloaded. The version shown is the only one that interests us in this article.

Further Information

C++11, Wikipedia.

Overview of the New C++ (C++11), Scott Meyers, Artima Press, last updated January 2012.

C++ Rvalue References Explained, Thomas Becker, last updated September 2011.

decltype, Wikipedia.

"A Note About decltype," Andrew Koenig, Dr. Dobb's, 27 July 2011.

Share this Article

Add a Comment

Comments are closed.

Comments (18)

0
1

Scott Meyers said on **Nov 18, 2012 06:29 PM:**

@Richardo Costa: Correct, but in order to know that, you have to recognize that "&&" in that example denotes an rvalue reference.

0
0

Thomas Becker said on **Nov 27, 2012 11:25 PM:**

I'd like to point out one small thing, just to make sure there are no misunderstandings. The article uses the expression "take the address of" several times, as a quick test to see if something is an lvalue: if you can take it's address, then it must be an lvalue. In this context, "taking the address" is to be read as "taking the address via the & operator". That is relevant because in C++, it is perfectly possible (though not exactly advisable!) to *obtain* the address of an rvalue, by means other than the & operator. Here's some code:

```
class Thing {
public:
  Thing* getThis(){
    return this;
  }
};
```

```
Thing t;
Thing foo() {
  return t;
}

int main(int argc, char **argv) {

  // Not good at all, but legal C++ (compiles without warning
  Thing* p = foo().getThis();
}
```

0

0

Paul Preney said on **Dec 2, 2012 04:00 PM:**

@Thomas Becker: Cool point, however, I feel I should add something to your point since it involves playing games with pointers and references: the two are NOT the same since their underlying representations can be very different.

While one can retrieve the address of a reference in some cases without the address-of operator, any time one explicitly acquires an address of a referent he/she will likely "force the hand" of the compiler in terms of its freedom to represent the referent. Specifically, I am referring to C++11's §8.3.2 para. 4 where it states:

[blockquote]It is unspecified whether or not a reference requires storage.[/blockquote]

As long as something remains a l- or r-value reference and never has to make its internal representation explicit, then the compiler is safe to represent the referent in whatever ways it sees fit (e.g., in a register, as a compiler data structure construct that can be potentially optimized away at compile-time, etc.) [b]due to this paragraph in the standard[/b]. The moment one [b]explicitly demands its address[/b], however, the compiler must then [b]map[/b] it to a [b]suitable-for-runtime address[/b] (i.e., some type of

pointer). A consequence is that the compiler now has potentially [b]fewer choices[/b] available to it with respect to [b]how it represents the referent[/b]. This is probably most especially true with respect the ability for the compiler to perform optimizations.

This was part of the reason why I wrote what I did in my blog post [url="http://preney.ca/paul/archives/1051"]References and Pointers[/url]. I humbly suggest that [b]unless one actually <u>needs</u> to use pointers[/b] (e.g., dynamic memory allocation, invalid memory addresses/nullptr) [b]modern C++ code should use references --not pointers--[/b] provided, upon initialization, the referent is a valid object.

0
0

gast128 said on **Dec 14, 2012 10:03 AM:**

Good article, but I really wonder how many programmers will use this. The rules are just too complicate to remember. I probably understand 80% of what was written in this article now. Next week I have a ton of other matters to discuss and think about and probably only 10% is left of this knowledge. A programming language should help the programmer and not the other way around. It was originally meant for move semantics and perfect forwarding and really couldn't this be solved simpler?

0
0

Ben Hekster said on **Apr 7, 2013 09:48 AM:**

It seems to me that the reference collapsing rules that explain this behavior are very nearly analogous to 'const collapsing' rules. One might just as well say that "T doesn't always mean non-const." When you see a declaration of "template <typename T> void f(T)", T

may be const or non-const; but somehow we don't need the concept of 'universal constness' to explain that?

I don't know if this site allows links, but I've tried to explain my point in more detail on http://www.hekster.org/Professional/Universal-References.html .

0

0

Scott Meyers said on **Apr 10, 2013 10:54 PM:**

@Ben Hekster: I apologize for the delayed reply. I didn't get any notification that you'd left a comment here.

I'm introducing the notion of a universal reference, because I think it makes things easier to understand. As I try to make clear in my talks and my articles on this topic, everything derives from the reference-collapsing rules, so if you've memorized those and can apply them unconsciously, there is no need for the concept of a universal reference. In my experience, however, many people struggle to understand why a move constructor or a move assignment operator works only with rvalues, yet functions like make_shared and std::vector<T>::emplace_back accept both lvalues and rvalues, even though their parameters are declared in exactly the same way. I believe that introducing the idea of universal references makes this fairly easy to understand.

More importantly, I've found that other guidelines for effective C++11 use naturally fall out of the distinction between rvalue references and universal references. When do you use std::move versus std::forward, for example? You use std::move with rvalue references,

and you use std::forward with universal references. What's a good guideline to follow when overloading rvalue references? Well, overloading on lvalue and rvalue references is fine, but overloading on universal references is generally a very bad idea.

My understanding is that during work on C++0x, the committee recognized the distinction between rvalue and universal references, and they considered using different syntax for them. Ultimately, they rejected this idea, which I personally think is unfortunate.

0
0

Todd Scott said on **Jun 27, 2013 02:44 PM:**

Great article. Two clarifying questions...

For the example where push_back takes a T&& but it is not a universal reference, this is true because the std::vector is implemented in such a way that the type cannot be a reference (e.g. std::vector<Widget&> will not compile), right? So, if std::vector were somehow re-written to allow for reference types it would be a universal reference even if the declaration of push_back were unchanged. (I want to verify that there isn't anything else going on in this case.)

You indicate that a universal reference must take the form "T&&" and that even adding const will make this an rvalue reference. Is this because the rule is that a universal reference always takes the form "T&&", period, that's the rule. Or, is there something else going on because of the const?

Thank you for posting these articles.

Best Regards,
Todd

0
0

Greg Marr said on **Jun 28, 2013 10:29 AM:**

"For the example where push_back takes a T&& but it is not a universal reference, this is true because the std::vector is implemented in such a way that the type cannot be a reference (e.g. std::vector<Widget&> will not compile), right?"

No, it is because T is not being deduced at the call site. For std::vector<Widget>, push_back is not push_back(T &&), it is push_back(Widget &&), and Widget && is not T &&. You need to do the T substitution right away, and then look at the parameter type. The push_back function is not a template function, it is a member function of a template class.

If it were template<typename U> push_back(U &&), then it would be a universal reference, as push_back is now a template function, and so the parameter type is deduced.

0
0

Todd Scott said on **Jun 28, 2013 11:11 AM:**

@Gregg

"... it is because T is not being deduced at the call site."

Ok, that is the key I just didn't absorb (though on re-reading the article I now find it quite clear on this point). So, if it were possible to create a standard

vector using references like std::vector<Widget&>, then the member function push_back would have a declaration of push_back(Widget& widget) which would make it always an l-value reference for that class, and thus not a universal reference.

Thank you for taking the time to clarify that for me. (I'm realizing that a number of the points being made here are just now fully "gelling" for me.)

1
0

linas said on **Jul 18, 2013 09:18 PM:**

Hi, I have some trouble.

```
int intfun()
{
  return 0;
}

template<typename T>
void IsUniversal(T&& t)
{
}

template<typename T>
struct Test
{
 template<typename P>
 Test(P&& t)
 {

 }
};
in vs2010:
IsUniversal(intfun); // OK.The name of a global function is lv

Test<char> t1(intfun); // The name of a global function is lvc

Test<char> t2((int(&)())intfun); // P=int (__cdecl *)(void), P
```

How to understand these case?

0
0

Branko Radosavljevic said on **Oct 20, 2013 09:34 PM:**

Hi Scott,

In this article you wrote "if the type of an expression is an lvalue reference (e.g., T& or const T&, etc.), that expression is an lvalue," and later "the lvalueness or rvalueness of an expression is independent of its type." Shouldn't you add a qualifier to the latter statement, something like "(unless the type is lvalue reference, in which case the expression is necessarily an lvalue)"?

I do think your concept of universal reference is useful for understanding, and agree with you that it's unfortunate that different syntax wasn't used. I saw your talk last week in Chicago at the CBOE; it was very clear and well-paced. Thanks!

0

0

jeff wu said on **Feb 27, 2014 11:22 PM:**

```
int x = 10;
auto&& y = std::move(x);
```

As what said in this article, std::move(x) returns a RRef, so auto&& y should be a RRef, but actually it is a LRef.

0

0

vlakov said on **Apr 22, 2014 02:53 PM:**

@jeff wu:
gcc 4.8.2;
> g++ -std=c++1y testp16.cpp -o a
> ./a

```
int x = 10;
auto&& y = std::move(x);
if(  std::is_same<decltype(y), int&&>::value )
  cout << "&&" << endl;
if(  std::is_same<decltype(y), int&>::value )
  cout << "&" << endl;
```

Result is: && - rvalue reference.

0

0

MihailNaydenov said on **Feb 10, 2015 01:11 AM:**

Thank You, for the article.
It would be nice to include information about

```
const
```

however!

I was a bit surprised when moving from const& (using std::move) failed to actually do any move.
I know it makes sense and is probably the right thing, but this points out the fact that std::move is more then "just a cast", and although it is an explicit user request, it might get ignored.

Compilers should warn when moving from const! This goes silent now (latest gcc, clang), but can have dramatic effect obviously.

0

0

Scott Meyers said on **Feb 10, 2015 12:21 PM:**

@MihailNaydenov: In [i] [url="http://www.jdoqocy.com/click-7708709-11290546?url=http://shop.oreilly.com/product/0636920033707.do?cmp=af-code-books-video-product_cj_0636920033707_%zp&"]Effective Modern C++[/url][/i], I explain in Item 23 ("Understand std::move and std::forward") that applying std::move to a const object yields a const rvalue. When an attempt is made to copy that rvalue, it's copied by the copy constructor or copy assignment operator, not the move constructor or move assignment operator, because the move functions require non-const

arguments. In fact, std::move [i]is[/i] just a cast, and that's why applying std::move to a const object doesn't remove the constness.

I agree that it would have been a good idea to discuss this in the article posted here, but I wrote that article over two years ago, and my understanding of how to explain the topic was less developed then than it is now. I think you'd find the treatment in [i]Effective Modern C++[/i] more comprehensive.

0
0

kamikaze said on **Jul 1, 2015 05:00 AM:**

I find the statement that f(x) is reasonable code shocking. To me it seems a violation of the programmers intention. I would want compilation to fail, if I had written that code.

In FreeBSD we'd call that a POLA* violation.

*principal of least astonishment

0
0

Florian Muecke said on **Oct 2, 2015 02:59 PM:**

Herb Sutter suggested the term 'forward reference' instead of 'universal reference' in his talk at cppcon 14 because it better matches its usage.

0
0

Akshay Kalaria said on **May 14, 2016 06:09 AM:**

Excellent content. Two making clear concerns...

For the example where push_back requires a T&& but it is not a worldwide referrals, this is real because the std::vector is applied in such a way that the kind cannot be a referrals (e.g. std::vector<Widget&> will

not compile), right? So, if std::vector were somehow re-written to allow for referrals kinds it would be a worldwide referrals even if the announcement of push_back were the same. (I want to ensure that there isn't anything else going on in this situation.)

You indicate that a worldwide referrals must take the kind "T&&" and that even including const will create this an rvalue referrals. Is this because the concept is that a worldwide referrals always requires the kind "T&&", interval, that's the concept. Or, is there something else going on because of the const?

Thank you for publishing these content content.

« Prev Next »

Back to Top