http://thbecker.net/articles/rvalue_references/section_01.html

C++ Rvalue References Explained

By Thomas Becker   Last updated: March 2013

Contents

Introduction

Rvalue references are a feature of C++ that was added with the C++11 standard. What makes rvalue references a bit difficult to grasp is that when you first look at them, it is not clear what their purpose is or what problems they solve. Therefore, I will not jump right in and explain what rvalue references are. Instead, I will start with the problems that are to be solved and then show how rvalue references provide the solution. That way, the definition of rvalue references will appear plausible and natural to you.

Rvalue references solve at least two problems:

1. **Implementing move semantics**
2. **Perfect forwarding**

If you are not familiar with these problems, do not worry. Both of them will be explained in detail below. We'll start with move semantics. But before we're ready to go, I need to remind you of what lvalues and rvalues are in C++. Giving a rigorous definition is surprisingly difficult, but the explanation below is good enough for the purpose at hand.

The original definition of lvalues and rvalues from the earliest days of C is as follows: An *lvalue* is an expression `e` that may appear on the left or on the right hand side of an assignment, whereas an *rvalue* is an expression that can only appear on the right hand side of an assignment. For example,

```
int a = 42;
int b = 43;

// a and b are both l-values:
a = b; // ok
b = a; // ok
a = a * b; // ok

// a * b is an rvalue:
int c = a * b; // ok, rvalue on right hand side of assignment
a * b = 42; // error, rvalue on left hand side of assignment
```

In C++, this is still useful as a first, intuitive approach to lvalues and rvalues. However, C++ with its user-defined types has introduced some subtleties regarding modifiability and assignability that cause this definition to be incorrect. There is no need for us to go further into this. Here is an alternate definition which, although it can still be argued with, will put you in a position to tackle rvalue references: An *lvalue* is an expression that refers to a memory location and allows us to take the address of that memory location via the `&` operator. An *rvalue* is an expression that is not an lvalue. Examples are:

```
// lvalues:
//
int i = 42;
i = 43; // ok, i is an lvalue
int* p = &i; // ok, i is an lvalue
int& foo();
foo() = 42; // ok, foo() is an lvalue
int* p1 = &foo(); // ok, foo() is an lvalue

// rvalues:
//
int foobar();
int j = 0;
j = foobar(); // ok, foobar() is an rvalue
int* p2 = &foobar(); // error, cannot take the address of an rvalue
j = 42; // ok, 42 is an rvalue
```

If you are interested in a rigorous definition of rvalues and lvalues, a good place to start is Mikael Kilpeläinen's ACCU article on the subject

Move Semantics

Suppose `X` is a class that holds a pointer or handle to some resource, say, `m_pResource`. By a resource, I mean anything that takes considerable effort to construct, clone, or destruct. A good example is `std::vector`, which holds a collection of objects that live in an array of allocated memory. Then, logically, the copy assignment operator for `X` looks like this:

```
X& X::operator=(X const & rhs)
```

```
{
  // [...]
  // Make a clone of what rhs.m_pResource refers to.
  // Destruct the resource that m_pResource refers to.
  // Attach the clone to m_pResource.
  // [...]
}
```

Similar reasoning applies to the copy constructor. Now suppose `X` is used as follows:

```
X foo();
X x;
// perhaps use x in various ways
x = foo();
```

The last line above

- clones the resource from the temporary returned by `foo`,
- destructs the resource held by `x` and replaces it with the clone,
- destructs the temporary and thereby releases its resource.

Rather obviously, it would be ok, and much more efficient, to swap resource pointers (handles) between `x` and the temporary, and then let the temporary's destructor destruct `x`'s original resource. In other words, in the special case where the right hand side of the assignment is an rvalue, we want the copy assignment operator to act like this:

```
// [...]
// swap m_pResource and rhs.m_pResource
// [...]
```

This is called *move semantics*. With C++11, this conditional behavior can be achieved via an overload:

```
X& X::operator=(<mystery type> rhs)
{
  // [...]
  // swap this->m_pResource and rhs.m_pResource
  // [...]
}
```

Since we're defining an overload of the copy assignment operator, our "mystery type" must essentially be a reference: we certainly want the right hand side to be passed to us by reference. Moreover, we expect the following behavior of the mystery type: when there is a choice between two overloads where one is an ordinary reference and the other is the mystery type, then rvalues must prefer the mystery type, while lvalues must prefer the ordinary reference.

If you now substitute "rvalue reference" for "mystery type" in the above, you're essentially looking at the definition of rvalue references.

Rvalue References

If `X` is any type, then `X&&` is called an *rvalue reference* to `X`. For better distinction, the ordinary reference `X&` is now also called an *lvalue reference*.

An rvalue reference is a type that behaves much like the ordinary reference `X&`, with several exceptions. The most important one is that when it comes to function overload resolution, lvalues prefer old-style lvalue references, whereas rvalues prefer the new rvalue references:

```
void foo(X& x); // lvalue reference overload
void foo(X&& x); // rvalue reference overload

X x;
X foobar();

foo(x); // argument is lvalue: calls foo(X&)
foo(foobar()); // argument is rvalue: calls foo(X&&)
```

So the gist of it is:

> Rvalue references allow a function to branch at compile time (via overload resolution) on the condition "Am I being called on an lvalue or an rvalue?"

It is true that you can overload *any* function in this manner, as shown above. But in the overwhelming majority of cases, this kind of overload should occur only for copy constructors and assignment operators, for the purpose of achieving move semantics:

```
X& X::operator=(X const & rhs); // classical implementation
X& X::operator=(X&& rhs)
{
  // Move semantics: exchange content between this and rhs
  return *this;
}
```

Implementing an rvalue reference overload for the copy constructor is similar.

Caveat: As it happens so often in C++, what looks just right at first glance is still a little shy of perfect. It turns out that in some cases, the simple exchange of content between `this` and `rhs` in the implementation of the copy assignment operator above is not quite good enough. We'll come back to this in Section 4, "Forcing Move Semantics" below.

Note: If you implement

```
void foo(X&);
```

but not

```
void foo(X&&);
```

then of course the behavior is unchanged: `foo` can be called on l-values, but not on r-values. If you implement

```
void foo(X const &);
```

but not

```
void foo(X&&);
```

then again, the behavior is unchanged: `foo` can be called on l-values and r-values, but it is not possible to make it distinguish between l-values and r-values. That is possible only by implementing

```
void foo(X&&);
```

as well. Finally, if you implement

```
void foo(X&&);
```

but neither one of

```
void foo(X&);
```

and

```
void foo(X const &);
```

then, according to the final version of C++11, `foo` can be called on r-values, but trying to call it on an l-value will trigger a compile error.

Forcing Move Semantics

As we all know, the First Amendment to the C++ Standard states: *"The committee shall make no rule that prevents C++ programmers from shooting themselves in the foot."* Speaking less facetiously, when it comes to choosing between giving programmers more control and saving them from their own carelessness, C++ tends to err on the side of giving more control. Being true to that spirit, C++11 allows you to use move semantics not just on rvalues, but, at your discretion, on lvalues as well. A good example is the std library function `swap`. As before, let X be a class for which we have overloaded the copy constructor and copy assignment operator to achieve move semantics on rvalues.

```
template<class T>
void swap(T& a, T& b)
{
  T tmp(a);
  a = b;
  b = tmp;
}

X a, b;
swap(a, b);
```

There are no rvalues here. Hence, all three lines in `swap` use non-move semantics. But we know that move semantics would be fine: wherever a variable occurs as the source of a copy construction or assignment, that variable is either not used again at all, or else it is used only as the target of an assignment.

In C++11, there is an std library function called `std::move` that comes to our rescue. It is a function that turns its argument into an rvalue without doing anything else. Therefore, in C++11, the std library function `swap` looks like this:

```
template<class T>
void swap(T& a, T& b)
{
  T tmp(std::move(a));
  a = std::move(b);
  b = std::move(tmp);
}

X a, b;
swap(a, b);
```

Now all three lines in `swap` use move semantics. Note that for those types that do not implement move semantics (that is, do not overload their copy constructor and assignment operator with an rvalue reference version), the new `swap` behaves just like the old one.

`std::move` is a very simple function. Unfortunately, though, I cannot show you the implementation yet. We'll come back to it later.

Using `std::move` wherever we can, as shown in the `swap` function above, gives us the following important benefits:

- For those types that implement move semantics, many standard algorithms and operations will use move semantics and thus experience a potentially significant performance gain. An important example is inplace sorting: inplace sorting algorithms do hardly anything else but swap elements, and this swapping will now take advantage of move semantics for all types that provide it.
- The STL often requires copyability of certain types, e.g., types that can be used as container elements. Upon close inspection, it turns out that in many cases, moveability is enough. Therefore, we can now use types that are moveable but not copyable (`unique_pointer` comes to mind) in many places where previously, they were not allowed. For example, these types can now be used as STL container elements.

Now that we know about `std::move`, we are in a position to see why the implementation of the rvalue reference overload of the copy assignment operator that I showed earlier is still a bit problematic. Consider a simple assignment between variables, like this:

```
a = b;
```

What do you expect to happen here? You expect the object held by `a` to be replaced by a copy of `b`, and in the course of this replacement, you expect the object formerly held by `a` to be destructed. Now consider the line

```
a = std::move(b);
```

If move semantics are implemented as a simple swap, then the effect of this is that the objects held by `a` and `b` are being exchanged between `a` and `b`. Nothing is being destructed yet. The object formerly held by `a` will of course be destructed eventually, namely, when `b` goes out of scope. Unless, of course, `b` becomes the target of a move, in which case the object formerly held by `a` gets passed on again. Therefore, as far as the implementer of the copy assignment operator is concerned, it is not known when the object formerly held by `a` will be destructed.

So in a sense, we have drifted into the netherworld of non-deterministic destruction here: a variable has been assigned to, but the object formerly held by that variable is still out there somewhere. That's fine as long as the destruction of that object does not have any side effects that are visible to the outside world. But sometimes destructors do have such side effects. An example would be the release of a lock inside a destructor. Therefore, any part of an object's destruction that has side effects should be performed explicitly in the rvalue reference overload of the copy assignment operator:

```
X& X::operator=(X&& rhs)
{

  // Perform a cleanup that takes care of at least those parts of the
  // destructor that have side effects. Be sure to leave the object
  // in a destructible and assignable state.

  // Move semantics: exchange content between this and rhs

  return *this;
}
```

Is an Rvalue Reference an Rvalue?

As before, let X be a class for which we have overloaded the copy constructor and copy assignment operator to implement move semantics. Now consider:

```
void foo(X&& x)
{
  X anotherX = x;
  // ...
}
```

The interesting question is: which overload of `X`'s copy constructor gets called in the body of `foo`? Here, `x` is a variable that is declared as an rvalue reference, that is, a reference which preferably and typically (although not necessarily!) refers to an rvalue. Therefore, it is quite plausible to expect that `x` itself should also bind like an rvalue, that is,

```
X(X&& rhs);
```

should be called. In other words, one might expect that anything that is declared as an rvalue reference is itself an rvalue. The designers of rvalue references have chosen a solution that is a bit more subtle than that:

Things that are declared as rvalue reference can be lvalues or rvalues. The distinguishing criterion is: *if it has a name*, then it is an lvalue. Otherwise, it is an rvalue.

In the example above, the thing that is declared as an rvalue reference has a name, and therefore, it is an lvalue:

```
void foo(X&& x)
{
  X anotherX = x; // calls X(X const & rhs)
}
```

Here is an example of something that is declared as an rvalue reference and does not have a name, and is therefore an rvalue:

```
X&& goo();
X x = goo(); // calls X(X&& rhs) because the thing on
             // the right hand side has no name
```

And here's the rationale behind the design: Allowing move semantics to be applied tacitly to something that has a name, as in

```
   X anotherX = x;
  // x is still in scope!
```

would be dangerously confusing and error-prone because the thing from which we just moved, that is, the thing that we just pilfered, is still accessible on subsequent lines of code. But the whole point of move semantics was to apply it only where it "doesn't matter," in the sense that the thing from which we move dies and goes away right after the moving. Hence the rule, "If it has a name, then it's an lvalue."

So then what about the other part, "If it does not have a name, then it's an rvalue?" Looking at the `goo` example above, it is technically possible, though not very likely, that the expression `goo()` in the second line of the example refers to something that is still accessible after it has been moved from. But recall from the previous section: sometimes that's what we want! We want to be able to force move semantics on lvalues at our discretion, and it is precisely the rule, "If it does not have a name, then it's an rvalue" that allows us to achieve that in a controlled manner. That's how the function `std::move` works. Although it is still too early to show you the exact implementation, we just got a step closer to understanding `std::move`. It passes its argument right through by reference, doing nothing with it at all, and its result type is rvalue reference. So the expression

```
std::move(x)
```

is declared as an rvalue reference and does not have a name. Hence, it is an rvalue. Thus, `std::move` "turns its argument into an rvalue even if it isn't," and it achieves that by "hiding the name."

Here is an example that shows how important it is to be aware of the if-it-has-a-name rule. Suppose you have written a class `Base`, and you have implemented move semantics by overloading Base's copy constructor and assignment operator:

```
Base(Base const & rhs); // non-move semantics
Base(Base&& rhs); // move semantics
```

Now you write a class Derived that is derived from Base. In order to assure that move semantics is applied to the Base part of your Derived objects, you must overload Derived's copy constructor and assignment operator as well. Let's look at the copy constructor. The copy assignment operator is handled analogously. The version for lvalues is straightforward:

```
Derived(Derived const & rhs)
  : Base(rhs)
{
  // Derived-specific stuff
}
```

The version for rvalues has a big fat subtlety. Here's what someone who is not aware of the if-it-has-a-name rule might have done:

```
Derived(Derived&& rhs)
  : Base(rhs) // wrong: rhs is an lvalue
{
  // Derived-specific stuff
}
```

If we were to code it like that, the non-moving version of Base's copy constructor would be called, because rhs, having a name, is an lvalue. What we want to be called is Base's moving copy constructor, and the way to get that is to write

```
Derived(Derived&& rhs)
  : Base(std::move(rhs)) // good, calls Base(Base&& rhs)
{
  // Derived-specific stuff
}
```

Move Semantics and Compiler Optimizations

Consider the following function definition:

```
X foo()
{
  X x;
  // perhaps do something to x
  return x;
}
```

Now suppose that as before, X is a class for which we have overloaded the copy constructor and copy assignment operator to implement move semantics. If you take the function definition above at face value, you may be tempted to say, wait a minute, there is a value copy happening here from x to the location of foo's return value. Let me make sure we're using move semantics instead:

```
X foo()
{
  X x;
  // perhaps do something to x
  return std::move(x); // making it worse!
}
```

Unfortunately, that would make things worse rather than better. Any modern compiler will apply *return value optimization* to the original function definition. In other words, rather than constructing an `X` locally and then copying it out, the compiler would construct the `X` object directly at the location of `foo`'s return value. Rather obviously, that's even better than move semantics.

So as you can see, in order to really use rvalue references and move semantics in an optimal way, you need to fully understand and take into account today's compilers' "special effects" such as return value optimization and copy elision. For a detailed discussion see Items 25 and 41 of Scott Meyers' book "Effective Modern C++". It gets get pretty subtle, but hey, we chose C++ as our language of choice for a reason, right? We made our beds, so now let's lie in them.

Perfect Forwarding: The Problem

The other problem besides move semantics that rvalue references were designed to solve is the perfect forwarding problem. Consider the following simple factory function:

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg arg)
{
  return shared_ptr<T>(new T(arg));
}
```

Obviously, the intent here is to forward the argument `arg` from the factory function to `T`'s constructor. Ideally, as far as `arg` is concerned, everything should behave just as if the factory function weren't there and the constructor were called directly in the client code: perfect forwarding. The code above fails miserably at that: it introduces an extra call by value, which is particularly bad if the constructor takes its argument by reference.

The most common solution, chosen e.g. by `boost::bind`, is to let the outer function take the argument by reference:

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg& arg)
{
  return shared_ptr<T>(new T(arg));
}
```

That's better, but not perfect. The problem is that now, the factory function cannot be called on rvalues:

```
factory<X>(hoo()); // error if hoo returns by value
factory<X>(41); // error
```

This can be fixed by providing an overload which takes its argument by const reference:

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg const & arg)
{
  return shared_ptr<T>(new T(arg));
}
```

There are two problems with this approach. Firstly, if `factory` had not one, but several arguments, you would have to provide overloads for all combinations of non-const and const reference for the various arguments. Thus, the solution scales extremely poorly to functions with several arguments.

Secondly, this kind of forwarding is less than perfect because it blocks out move semantics: the argument of the constructor of `T` in the body of `factory` is an lvalue. Therefore, move semantics can never happen even if it would without the wrapping function.

It turns out that rvalue references can be used to solve both these problems. They make it possible to achieve truly perfect forwarding without the use of overloads. In order to understand how, we need to look at two more rules for rvalue references.

Perfect Forwarding: The Solution

The first of the remaining two rules for rvalue references affects old-style lvalue references as well. Recall that in pre-11 C++, it was not allowed to take a reference to a reference: something like `A& &` would cause a compile error. C++11, by contrast, introduces the following *reference collapsing rules*[1]:

- `A& &` becomes `A&`
- `A& &&` becomes `A&`
- `A&& &` becomes `A&`
- `A&& &&` becomes `A&&`

Secondly, there is a special template argument deduction rule for function templates that take an argument by rvalue reference to a template argument:

```
template<typename T>
void foo(T&&);
```

Here, the following apply:

1. When foo is called on an lvalue of type `A`, then `T` resolves to `A&` and hence, by the reference collapsing rules above, the argument type effectively becomes `A&`.

2. When foo is called on an rvalue of type `A`, then `T` resolves to `A`, and hence the argument type becomes `A&&`.

Given these rules, we can now use rvalue references to solve the perfect forwarding problem as set forth in the previous section. Here's what the solution looks like:

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg&& arg)
{
  return shared_ptr<T>(new T(std::forward<Arg>(arg)));
}
```

where `std::forward` is defined as follows:

```
template<class S>
S&& forward(typename remove_reference<S>::type& a) noexcept
{
  return static_cast<S&&>(a);
}
```

(Don't pay attention to the `noexcept` keyword for now. It lets the compiler know, for certain optimization purposes, that this function will never throw an exception. We'll come back to it in Section 9.) To see how the code above achieves perfect forwarding, we will discuss separately what happens when our factory function gets called on lvalues and rvalues. Let `A` and `X` be types. Suppose first that `factory<A>` is called on an lvalue of type `X`:

```
X x;
factory<A>(x);
```

Then, by the special template deduction rule stated above, `factory`'s template argument `Arg` resolves to `X&`. Therefore, the compiler will create the following instantiations of `factory` and `std::forward`:

```
shared_ptr<A> factory(X& && arg)
{
  return shared_ptr<A>(new A(std::forward<X&>(arg)));
}

X& && forward(remove_reference<X&>::type& a) noexcept
{
  return static_cast<X& &&>(a);
}
```

After evaluating the `remove_reference` and applying the reference collapsing rules, this becomes:

```
shared_ptr<A> factory(X& arg)
{
  return shared_ptr<A>(new A(std::forward<X&>(arg)));
}

X& std::forward(X& a)
{
  return static_cast<X&>(a);
}
```
This is certainly perfect forwarding for lvalues: the argument `arg` of the factory function gets passed on to `A`'s constructor through two levels of indirection, both by old-fashioned lvalue reference.

Next, suppose that `factory<A>` is called on an rvalue of type `X`:

```
X foo();
factory<A>(foo());
```
Then, again by the special template deduction rule stated above, `factory`'s template argument `Arg` resolves to `X`. Therefore, the compiler will now create the following function template instantiations:
```
shared_ptr<A> factory(X&& arg)
{
  return shared_ptr<A>(new A(std::forward<X>(arg)));
}

X&& forward(X& a) noexcept
{
  return static_cast<X&&>(a);
}
```
This is indeed perfect forwarding for rvalues: the argument of the factory function gets passed on to `A`'s constructor through two levels of indirection, both by reference. Moreover, `A`'s constructor sees as its argument an expression that is declared as an rvalue reference and does not have a name. By the no-name rule, such a thing is an rvalue. Therefore, `A`'s constructor gets called on an rvalue. This means that the forwarding preserves any move semantics that would have taken place if the factory wrapper were not present.

It is perhaps worth noting that the preservation of move semantics is in fact the *only* purpose of `std::forward` in this context. Without the use of `std::forward`, everything would work quite nicely, except that `A`'s constructor would always see as its argument something that has a name, and such a thing is an lvalue. Another way of putting this is to say that `std::forward`'s purpose is to forward the information whether at the call site, the wrapper saw an lvalue or an rvalue.

If you want to dig a little deeper for extra credit, ask yourself this question: why is the `remove_reference` in the definition of `std::forward` needed? The answer is, it is not really needed at all. If you use just `S&` instead of `remove_reference<S>::type&` in the definition of `std::forward`, you can repeat the case distinction above to convince yourself that perfect forwarding still works just fine. However, it works fine only as long as we explicitly specify `Arg` as the template argument of `std::forward`. The purpose of the `remove_reference` in the definition of `std::forward` is to *force* us to do so.

Rejoice. We're almost done. It only remains to look at the implementation of `std::move`. Remember, the purpose of `std::move` is to pass its argument right through by reference and make it bind like an rvalue. Here's the implementation:

```
template<class T>
typename remove_reference<T>::type&&
std::move(T&& a) noexcept
{
  typedef typename remove_reference<T>::type&& RvalRef;
  return static_cast<RvalRef>(a);
}
```

Suppose that we call `std::move` on an lvalue of type `X`:

```
X x;
std::move(x);
```

By the new special template deduction rule, the template argument `T` will resolve to `X&`. Therefore, what the compiler ends up instantiating is

```
typename remove_reference<X&>::type&&
std::move(X& && a) noexcept
{
  typedef typename remove_reference<X&>::type&& RvalRef;
  return static_cast<RvalRef>(a);
}
```

After evaluating the `remove_reference` and applying the new reference collapsing rules, this becomes

```
X&& std::move(X& a) noexcept
{
  return static_cast<X&&>(a);
}
```

That does the job: our lvalue `x` will bind to the lvalue reference that is the argument type, and the function passes it right through, turning it into an unnamed rvalue reference.

I leave it to you to convince yourself that `std::move` actually works fine when called on an rvalue. But then you may want to skip that: why would anybody want to call `std::move` on an rvalue, when its only purpose is to turn things into rvalues? Also, you have probably noticed by now that instead of

```
std::move(x);
```

you could just as well write

```
static_cast<X&&>(x);
```

However, `std::move` is strongly preferred because it is more expressive.

---

[1]The reference collapsing rules stated here are correct, but not entirely complete. I have omitted a small detail that is not relevant in our context, namely, the disappearance of const and volatile qualifiers during reference collapsing. Scott Meyer's "Effective Modern C++" modification history and errata page has the full

explanation.

Rvalue References And Exceptions

Normally, when you develop software in C++, it is your choice whether you want to pay attention to exception safety, or to use exceptions at all in your code. Rvalue references are a bit different in this regard. When you overload the copy constructor and the copy assignment operator of a class for the sake of move semantics, it is very much recommended that you do the following:

1. Strive to write your overloads in such a way that they cannot throw exceptions. That is often trivial, because move semantics typically do no more than exchange pointers and resource handles between two objects.
2. If you succeeded in not throwing exceptions from your overloads, then make sure to advertise that fact using the new `noexcept` keyword.

If you don't do both of these things, then there is at least one very common situation where your move semantics will not be applied despite the fact that you would very much expect it: when an `std::vector` gets resized, you certainly want move semantics to happen when the existing elements of your vector are being relocated to the new memory block. But that won't happen unless both of 1. and 2. above are satisfied.

You don't really have to understand the reasons for this behavior. Observing the recommendations 1. and 2. above is easy enough, and it's all you really need to know. However, having an understanding that runs a little deeper than absolutely necessary has never hurt anybody. I recommend reading Item 14 (and all the other items, for that matter) of Scott Meyers' book “Effective Modern C++.”

The Case of the Implicit Move

At one point during the (often complex and controversial) discussion of rvalue references, the Standard Committee decided that move constructors and move assignment operators, that is, rvalue reference overloads for copy constructors and copy assignment operators, should be generated by the compiler when not provided by the user. This seems like a natural and plausible thing to request, considering that the compiler has always done the exact same thing for ordinary copy constructors and copy assignment operators. In August of 2010, Scott Meyers posted a message on comp.lang.c++ in which he explained how compiler-generated move constructors can break existing code in a rather serious way.

The committee then decided that this was indeed cause for alarm, and it restricted the automatic generation of move constructors and move assignment operators in such a way that it is much less likely, though not impossible, for existing code to break. The final state of affairs is described in detail in Item 17 of Scott Meyers' book “Effective Modern C++.”

The issue of implicitly moving remained controversial all the way up to the finalization of the Standard (see e.g. this paper by Dave Abrahams). In an ironic twist of fate, the only reason why the committee considered implicit move in the first place was as an attempt to resolve the issue with rvalue references and exceptions as mentioned in Section 9. This problem was subsequently solved in a more satisfactory way via the new `noexcept` keyword. Had the `noexcept` solution been found just a few months earlier, implicit move may have never seen the light of day. Oh well, so it goes.

Ok, that's it, the whole story on rvalue references. As you can see, the benefits are considerable. The details are gory. As a C++ professional, you will have to understand these details. Otherwise, you have given up on fully understanding the central tool of your trade. You can take solace, though, in the thought that in your day-to-day programming, you will only have to remember three things about rvalue references:

1. By overloading a function like this:
2. `void foo(X& x); // lvalue reference overload`
3. `void foo(X&& x); // rvalue reference overload`

   you can branch at compile time on the condition "is `foo` being called on an lvalue or an rvalue?" The primary (and for all practical purposes, the only) application of that is to overload the copy constructor and copy assignment operator of a class for the sake of implementing move semantics. If and when you do that, make sure to pay attention to exception handling, and use the new `noexcept` keyword as much as you can.

4. `std::move` turns its argument into an rvalue.

5. `std::forward` allows you to achieve perfect forwarding if you use it exactly as shown in the factory function example in Section 8.

Enjoy!

Acknowledgments and Further Reading

I am indebted to Thomas Witt for the insight and information on rvalue references that he shared with me. Thanks to Valentin David for reading the article carefully and providing valuable corrections and insights. Many more readers have helped to improve the article. Thanks everybody for their contributions, and please do keep sending feedback. All remaining inaccuracies and deficiencies are mine.

I very much recommend reading the article by Howard E. Hinnant, Bjarne Stroustrup, and Bronek Kozicki on the subject of C++ rvalue references at the C++ Source. This article has more and better examples than mine does, and it has an extensive list of links to proposals and technical papers which you will find interesting. As a tradeoff, the article does not go into all the details the way I do; for example, it does not explicitly state the new reference collapsing rules or the special template argument deduction rule for rvalue references.

Finally, your primary source of reference for all things C++11 and C++14 should be Scott Meyers' book “Effective Modern C++.” While my presentation in this article gave you sort of a sharp focus on rvalue references, Scott Meyers' book paints the entire picture of modern C++, whithout ever cutting corners on details.