Move semantics and rvalue references in C++11

By Alex Allain

C++ has always produced fast programs. Unfortunately, until C++11, there has been an obstinate wart that slows down many C++ programs: the creation of temporary objects. Sometimes these temporary objects can be optimized away by the compiler (the return value optimization, for example). But this is not always the case, and it can result in expensive object copies. What do I mean?

Let's say that you have the following code:

```
#include <iostream>

using namespace std;

vector<int> doubleValues (const vector<
{
    vector<int> new_values;
    new_values.reserve(v.size());
    for (auto itr = v.begin(), end_itr
    {
        new_values.push_back( 2 * *itr
    }
    return new_values;
}

int main()
{
    vector<int> v;
    for ( int i = 0; i < 100; i++ )
    {
        v.push_back( i );
    }
    v = doubleValues( v );
}
```

If you've done a lot of high performance work in C++, sorry about the pain that brought on. If you haven't--well, let's walk through why this code is terrible C++03 code. (The rest of this tutorial will be about why it's fine C++11 code.) The problem is with the copies. When doubleValues is called, it constructs a vector, new_values, and fills it up. This alone might not be ideal performance, but if we want to keep our original vector unsullied, we need a second copy. But what happens when we hit the return statement?

The entire contents of new_values must be copied! In principle, there could be up to two copies here: one into a temporary object to be returned, and a second when the vector assignment operator runs on the line v = doubleValues(v);. The first copy may be optimized away by the compiler automatically, but there is no avoiding that the assignment to v will have to copy all the values again, which requires a new memory allocation and another iteration over the entire vector.

This example might be a little bit contrived--and of course you can find ways to avoid this kind of problem--for example, by storing and returning the vector by pointer, or by passing in a vector to be filled up. The thing is, neither of these programming styles is particularly natural. Moreover, an approach that requires returning a pointer has introduced at least one more memory allocation, and one of the design goals of C++ is to avoid memory allocations.

The worst part of this whole story is that the object returned from doubleValues is a temporary value that's no longer needed. When you have the line v = doubleValues(v), the result of doubleValues(v) is just going to get thrown away once it is copied! In theory, it should be possible to skip the whole copy and just pilfer the pointer inside the temporary vector and keep it in v. In effect, why can't we **move** the object? In C++03, the answer is that there was no way to tell if an object was a temporary or not, you had to run the same code in the assignment operator or copy constructor, no matter where the value came from, so no pilfering was possible. In C++11, the answer is--you can!

That's what rvalue references and move semantics are for! Move semantics allows you to avoid unnecessary copies when working with temporary objects that are about to evaporate, and whose resources can safely be taken from that temporary object and used by another.

Move semantics relies on a new feature of C++11, called rvalue references, which you'll want to understand to really appreciate what's going on. So first let's talk about what an rvalue is, and then what an rvalue reference is. Finally, we'll come back to move semantics and how it can be implemented with rvalue references.

Rvalues and lvalues - bitter rivals, or best of friends?

In C++, there are rvalues and lvalues. An lvalue is an expression whose address can be taken, a locator value--essentially, an lvalue provides a (semi)permanent piece of memory. You can make assignments to lvalues. For example:

```
int a;
a = 1; // here, a is an lvalue
```

You can also have lvalues that aren't variables:

```
int x;
int& getRef ()
{
        return x;
}

getRef() = 4;
```

Here, getRef returns a reference to a global variable, so it's returning a value that is stored in a permanent location. (You could literally write & getRef() if you wanted to, and it would give you the address of x.)

Rvalues are--well, rvalues are not lvalues. An expression is an rvalue if it results in a temporary object. For example:

```
int x;
int getVal ()
{
    return x;
}
getVal();
```

Here, getVal() is an rvalue--the value being returned is not a reference to x, it's just a temporary value. This gets a little bit more interesting if we use real objects instead of numbers:

```
string getName ()
{
    return "Alex";
}
getName();
```

Here, getName returns a string that is constructed inside the function. You can assign the result of getName to a variable:

```
string name = getName();
```

But you're assigning from a temporary object, not from some value that has a fixed location. getName() is an rvalue.

Detecting temporary objects with rvalue references

The important thing is that rvalues refer to temporary objects--just like the value returned from doubleValues. Wouldn't it be great if we could know, without a shadow of a doubt, that a value returned from an expression was temporary, and somehow write code that is overloaded to behave differently for temporary objects? Why, yes, yes indeed it would be. And this is what rvalue references are for. An rvalue reference is a reference that will bind only to a temporary object. What do I mean?

Prior to C++11, if you had a temporary object, you could use a "regular" or "lvalue reference" to bind it, but only if it was const:

```
const string& name = getName(); // ok
string& name = getName(); // NOT ok
```

The intuition here is that you cannot use a "mutable" reference because, if you did, you'd be able to modify some object that is about to disappear, and that would be dangerous. Notice, by the way, that holding on to a const reference to a temporary object ensures that the temporary object isn't immediately destructed. This is a nice guarantee of C++, but it is still a temporary object, so you don't want to modify it.

In C++11, however, there's a new kind of reference, an "rvalue reference", that will let you bind a mutable reference to an rvalue, but not an lvalue. In other words, rvalue references are perfect for detecting if a value is temporary object or not. Rvalue references use the && syntax instead of just &, and can be const and non-const, just like lvalue references, although you'll rarely see a const rvalue reference (as we'll see, mutable references are kind of the point):

```
const string&& name = getName(); // ok
string&& name = getName(); // also ok -
```

So far this is all well and good, but how does it help? The most important thing about lvalue references vs rvalue references is what happens when you write functions that take lvalue or rvalue references as arguments. Let's say we have two functions:

```
printReference (const String& str)
{
        cout << str;
}

printReference (String&& str)
{
        cout << str;
}
```

Now the behavior gets interesting--the printReference function taking a const lvalue reference will accept any argument that it's given, whether it be an lvalue or an rvalue, and regardless of whether the lvalue or rvalue is mutable or not. However, in the presence of the second overload, printReference taking an rvalue reference, it will be given all values *except* mutable rvalue-references. In other words, if you write:

```
string me( "alex" );
printReference(  me ); // calls the firs

printReference( getName() ); // calls th
```

Now we have a way to determine if a reference variable refers to a temporary object or to a permanent object. The rvalue reference version of the method is like the secret back door entrance to the club that you can only get into if you're a temporary object (boring club, I guess). Now that we have our method of determining if an object was a temporary or a permanent thing, how can we use it?

Move constructor and move assignment operator

The most common pattern you'll see when working with rvalue references is to create a move constructor and move assignment operator (which follows the same principles). A move constructor, like a copy constructor, takes an instance of an object as its argument and creates a new instance based on the original object. However, the move constructor can avoid memory reallocation because we know it has been provided a temporary object, so rather than copy the fields of the object, we will move them.

What does it mean to move a field of the object? If the field is a primitive type, like int, we just copy it. It gets more interesting if the field is a pointer: here, rather than allocate and initialize new memory, we can simply steal the pointer and null out the pointer in the temporary object! We know the temporary object will no longer be needed, so we can take its pointer out from under it.

Imagine that we have a simple ArrayWrapper class, like this:

```
class ArrayWrapper
```

```
{
    public:
        ArrayWrapper (int n)
            : _p_vals( new int[ n ] )
            , _size( n )
        {}
        // copy constructor
        ArrayWrapper (const ArrayWrappe
            : _p_vals( new int[ other._
            , _size( other._size )
        {
            for ( int i = 0; i < _size;
            {
                _p_vals[ i ] = other._p
            }
        }
        ~ArrayWrapper ()
        {
            delete [] _p_vals;
        }
    private:
    int *_p_vals;
    int _size;
};
```

Notice that the copy constructor has to both allocate memory and copy every value from the array, one at a time! That's a lot of work for a copy. Let's add a move constructor and gain some massive efficiency.

```
class ArrayWrapper
{
public:
    // default constructor produces a m
    ArrayWrapper ()
        : _p_vals( new int[ 64 ] )
        , _size( 64 )
    {}

    ArrayWrapper (int n)
        : _p_vals( new int[ n ] )
        , _size( n )
    {}

    // move constructor
    ArrayWrapper (ArrayWrapper&& other)
        : _p_vals( other._p_vals  )
        , _size( other._size )
    {
        other._p_vals = NULL;
        other._size = 0;
    }

    // copy constructor
    ArrayWrapper (const ArrayWrapper& o
        : _p_vals( new int[ other._size
        , _size( other._size )
    {
        for ( int i = 0; i < _size; ++i
        {
            _p_vals[ i ] = other._p_val
        }
    }
    ~ArrayWrapper ()
    {
        delete [] _p_vals;
    }

private:
    int *_p_vals;
```

```
    int _size;
};
```

Wow, the move constructor is actually simpler than the copy constructor! That's quite a feat. The main things to notice are:

1. The parameter is a non-const rvalue reference
2. other._p_vals is set to NULL

The second observation explains the first--we couldn't set other._p_vals to NULL if we'd taken a const rvalue reference. But why do we need to set other._p_vals = NULL? The reason is the destructor--when the temporary object goes out of scope, just like all other C++ objects, its destructor will run. When its destructor runs, it will free _p_vals. The same _p_vals that we just copied! If we don't set other._p_vals to NULL, the move would not really be a move--it would just be a copy that introduces a crash later on once we start using freed memory. This is the whole point of a move constructor: to avoid a copy by changing the original, temporary object!

Again, the overload rules work such that the move constructor is called only for a temporary object--and only a temporary object that can be modified. One thing this means is that if you have a function that returns a const object, it will cause the copy constructor to run instead of the move constructor--so don't write code like this:

```
const ArrayWrapper getArrayWrapper (); /
```

There's still one more situation we haven't discussed how to handle in a move constructor--when we have a field that is an object. For example, imagine that instead of having a size field, we had a metadata field that looked like this:

```
class MetaData
{
public:
    MetaData (int size, const std::stri
        : _name( name )
        , _size( size )
    {}

    // copy constructor
    MetaData (const MetaData& other)
        : _name( other._name )
        , _size( other._size )
    {}

    // move constructor
    MetaData (MetaData&& other)
        : _name( other._name )
        , _size( other._size )
    {}

    std::string getName () const { retu
    int getSize () const { return _size
    private:
    std::string _name;
    int _size;
};
```

Now our array can have a name and a size, so we might have to change the definition of ArrayWrapper like so:

```
class ArrayWrapper
{
public:
    // default constructor produces a m
    ArrayWrapper ()
        : _p_vals( new int[ 64 ] )
        , _metadata( 64, "ArrayWrapper"
    {}

    ArrayWrapper (int n)
```

```
        : _p_vals( new int[ n ] )
        , _metadata( n, "ArrayWrapper"
    {}

    // move constructor
    ArrayWrapper (ArrayWrapper&& other)
        : _p_vals( other._p_vals  )
        , _metadata( other._metadata )
    {
        other._p_vals = NULL;
    }

    // copy constructor
    ArrayWrapper (const ArrayWrapper& o
        : _p_vals( new int[ other._meta
        , _metadata( other._metadata )
    {
        for ( int i = 0; i < _metadata.
        {
            _p_vals[ i ] = other._p_val
        }
    }
    ~ArrayWrapper ()
    {
        delete [] _p_vals;
    }
private:
    int *_p_vals;
    MetaData _metadata;
};
```

Does this work? It seems very natural, doesn't it, to just call the MetaData move constructor from within the move constructor for ArrayWrapper? The problem is that this just doesn't work. The reason is simple: the value of other in the move constructor--it's an rvalue reference. But an rvalue reference is not, in fact, an rvalue. It's an lvalue, and so the copy constructor is called, not the move constructor. This is weird. I know--it's confusing. Here's the way to think about it. A rvalue is an expression that creates an object that is about to evaporate into thin air. It's on its last legs in life--or about to fulfill its life purpose. Suddenly we pass the temporary to a move constructor, and it takes on new life in the new scope. In the context where the rvalue expression was evaluated, the temporary object really is over and done with. But in our constructor, the object has a name; it will be alive for the entire duration of our function. In other words, we might use the variable other more than once in the function, and the temporary object has a defined location that truly persists for the entire function. It's an lvalue in the true sense of the term locator value, we can locate the object at a particular address that is stable for the entire duration of the function call. We might, in fact, want to use it later in the function. If a move constructor were called whenever we held an object in an rvalue reference, we might use a moved object, by accident!

```
// move constructor
ArrayWrapper (ArrayWrapper&& other)
    : _p_vals( other._p_vals  )
    , _metadata( other._metadata )
{
    // if _metadata( other._metadata ) c
    // other._metadata here would be ext
    other._p_vals = NULL;
}
```

Put a final way: both lvalue and rvalue references are lvalue expressions. The difference is that an lvalue reference must be const to hold a reference to an rvalue, whereas an rvalue reference can always hold a reference to an rvalue. It's like the difference between a pointer, and what is pointed to. The thing pointed-to came from an rvalue, but when we use rvalue reference itself, it results in an lvalue.

std::move

So what's the trick to handling this case? We need to use std::move, from <utility>--std::move is a way of saying, "ok, honest to God I know I have an lvalue, but I want it to be an rvalue." std::move does not, in and of itself,

move anything; it just turns an lvalue into an rvalue, so that you can invoke the move constructor. Our code should look like this:

```
#include <utility> // for std::move

    // move constructor
    ArrayWrapper (ArrayWrapper&& other)
        : _p_vals( other._p_vals  )
        , _metadata( std::move( other._m
    {
        other._p_vals = NULL;
    }
```

And of course we should really go back to MetaData and fix its own move constructor so that it uses std::move on the string it holds:

```
MetaData (MetaData&& other)
    : _name( std::move( other._name ) )
    : _size( other._size )
{}
```

Move assignment operator

Just as we have a move constructor, we should also have a move assignment operator. You can easily write one using the same techniques as for creating a move constructor.

Move constructors and implicitly generated constructors

As you know, in C++ when you declare any constructor, the compiler will no longer generate the default constructor for you. The same is true here: adding a move constructor to a class will require you to declare and define your own default constructor. On the other hand, declaring a move constructor does not prevent the compiler from providing an implicitly generated copy constructor, and declaring a move assignment operator does not inhibit the creation of a standard assignment operator.

How does std::move work

You might be wondering, how does one write a function like std::move? How do you get this magical property of transforming an lvalue into an rvalue reference? The answer, as you might guess, is typecasting. The actual declaration for std::move is somewhat more involved, but at its heart, it's just a static_cast to an rvalue reference. This means, actually, that you don't really *need* to use move--but you should, since it's much more clear what you mean. The fact that a cast is required is, by the way, a very good thing! It means that you cannot accidentally convert an lvalue into an rvalue, which would be dangerous since it might allow an accidental move to take place. You must explicitly use std::move (or a cast) to convert an lvalue into an rvalue reference, and an rvalue reference will never bind to an lvalue on its own.

Returning an explicit rvalue-reference from a function

Are there ever times where you should write a function that returns an rvalue reference? What does it mean to return an rvalue reference anyway? Aren't functions that return objects by value already rvalues?

Let's answer the second question first: returning an explicit rvalue reference is different than returning an object by value. Take the following simple example:

```
int x;

int getInt ()
{
    return x;
}

int && getRvalueInt ()
{
    // notice that it's fine to move a
```

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

Clearly in the first case, despite the fact that getInt() is an rvalue, there is a copy of the variable x being made. We can even see this by writing a little helper function:

```
void printAddress (const int& v) // cons
{
    cout << reinterpret_cast<const void*
}

printAddress( getInt() );
printAddress( x );
```

When you run this program, you'll see that there are two separate values printed.

On the other hand,

```
printAddress( getRvalueInt() );
printAddress( x );
```

prints the same value because we are explicitly returning an rvalue here.

So returning an rvalue reference is a different thing than not returning an rvalue reference, but this difference manifests itself most noticeably if you have a pre-existing object you are returning instead of a temporary object created in the function (where the compiler is likely to eliminate the copy for you).

Now on to the question of whether you want to do this. The answer is: probably not. In most cases, it just makes it more likely that you'll end up with a dangling reference (a case where the reference exists, but the temporary object that it refers to has been destroyed). The issue is quite similar to the danger of returning an lvalue reference--the referred-to object may no longer exist. Rvalue references cannot magically keep an object alive for you. Returning an rvalue reference would primarily make sense in very rare cases where you have a member function and need to return the result of calling std::move on a field of the class from that function--and how often are you going to do that?

Move semantics and the standard library

Going back to our original example--we were using a vector, and we don't have control over the vector class and whether or not it has a move constructor or move assignment operator. Fortunately, the standards committee is wise, and move semantics has been added to the standard library. This means that you can now efficiently return vectors, maps, strings and whatever other standard library objects you want, taking full advantage of move semantics.

Moveable objects in STL containers

In fact, the standard library goes one step further. If you enable move semantics in your own objects by creating move assignment operators and move constructors, when you store those objects in a container, the STL will automatically use std::move, automatically taking advantage of move-enabled classes to eliminate inefficient copies.

Move semantics and rvalue reference compiler support

Rvalue references are supported by GCC, the Intel compiler and MSVC.