Document number: N2027=06-0097

Howard E. Hinnant
Bjarne Stroustrup
Bronek Kozicki
2006-06-12

A Brief Introduction to Rvalue References

Abstract

Rvalue references is a small technical extension to the C++ language. Rvalue references allow programmers to avoid logically unnecessary copying and to provide perfect forwarding functions. They are primarily meant to aid in the design of higer performance and more robust libraries.

Contents

Introduction

This document gives a quick tour of the new C++ language feature *rvalue reference*. It is a brief tutorial, rather than a complete reference. For details, see these references.

The rvalue reference

An *rvalue reference* is a compound type very similar to C++'s traditional reference. To better distinguish these two types, we refer to a traditional C++ reference as an *lvalue reference*. When the term *reference* is used, it refers to both kinds of reference: lvalue reference and rvalue reference.

An lvalue reference is formed by placing an `&` after some type.

```
A a;
A& a_ref1 = a;  // an lvalue reference
```

An rvalue reference is formed by placing an `&&` after some type.

```
A a;
A&& a_ref2 = a;  // an rvalue reference
```

An rvalue reference behaves just like an lvalue reference except that it *can* bind to a temporary (an rvalue), whereas you can not bind a (non const) lvalue reference to an rvalue.

```
A&  a_ref3 = A();  // Error!
A&& a_ref4 = A();  // Ok
```

Question: Why on Earth would we want to do this?!

It turns out that the combination of rvalue references and lvalue references is just what is needed to easily code *move semantics*. The rvalue reference can also be used to achieve *perfect forwarding*, a heretofore

unsolved problem in C++. From a casual programmer's perspective, what we get from rvalue references is more general and better performing libraries.

Move Semantics

Eliminating spurious copies

Copying can be expensive. For example, for `std::vectors`, `v2=v1` typically involves a function call, a memory allocation, and a loop. This is of course acceptable where we actually need two copies of a `vector`, but in many cases, we don't: We often copy a `vector` from one place to another, just to proceed to overwrite the old copy. Consider:

```
template <class T> swap(T& a, T& b)
{
    T tmp(a);   // now we have two copies of a
    a = b;      // now we have two copies of b
    b = tmp;    // now we have two copies of tmp (aka a)
}
```

But, we didn't want to have *any* copies of `a` or `b`, we just wanted to swap them. Let's try again:

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

This `move()` gives its target the value of its argument, but is not obliged to preserve the value of its source. So, for a `vector`, `move()` could reasonably be expected to leave its argument as a zero-capacity `vector` to avoid having to copy all the elements. In other words, `move` is a potentially destructive read.

In this particular case, we could have optimized `swap` by a specialization. However, we can't specialize every function that copies a large object just before it deletes or overwrites it. That would be unmanageable.

The first task of rvalue references is to allow us to implement `move()` without verbosity, or rutime overhead.

`move`

The `move` function really does very little work. All `move` does is accept either an lvalue or rvalue argument, and return it as an rvalue *without* triggering a copy construction:

```
template <class T>
typename remove_reference<T>::type&&
move(T&& a)
{
    return a;
}
```

It is now up to client code to overload key functions on whether their argument is an lvalue or rvalue (e.g. copy constructor and assignment operator). When the argument is an lvalue, the argument must be copied from. When it is an rvalue, it can safely be moved from.

Overloading on lvalue / rvalue

Consider a simple *handle* class that owns a resource and also provides copy semantics (copy constructor and assignment). For example a `clone_ptr` might own a pointer, and call `clone()` on it for copying purposes:

```
template <class T>
class clone_ptr
{
```

```
private:
    T* ptr;
public:
    // construction
    explicit clone_ptr(T* p = 0) : ptr(p) {}

    // destruction
    ~clone_ptr() {delete ptr;}

    // copy semantics
    clone_ptr(const clone_ptr& p)
        : ptr(p.ptr ? p.ptr->clone() : 0) {}

    clone_ptr& operator=(const clone_ptr& p)
    {
        if (this != &p)
        {
            delete ptr;
            ptr = p.ptr ? p.ptr->clone() : 0;
        }
        return *this;
    }

    // move semantics
    clone_ptr(clone_ptr&& p)
        : ptr(p.ptr) {p.ptr = 0;}

    clone_ptr& operator=(clone_ptr&& p)
    {
        std::swap(ptr, p.ptr);
        return *this;
    }

    // Other operations
    T& operator*() const {return *ptr;}
    // ...
};
```

Except for the highlighted *move semantics* section above, `clone_ptr` is code that you might find in today's books on C++. Clients of `clone_ptr` might use it like so:

```
clone_ptr<base> p1(new derived);
// ...
clone_ptr<base> p2 = p1;  // p2 and p1 each own their own pointer
```

Note that copy constructing or assigning a `clone_ptr` is a relatively expensive operation. However when the source of the copy is known to be an rvalue, one can avoid the potentially expensive `clone()` operation by pilfering the rvalue's pointer (no one will notice!). The *move constructor* above does exactly that, leaving the rvalue in a default constructed state. The *move assignment* operator simply swaps state with the rvalue.

Now when code tries to copy an rvalue `clone_ptr`, or if that code explicitly gives permission to consider the source of the copy an rvalue (using `std::move`), the operation will execute *much* faster.

```
clone_ptr<base> p1(new derived);
// ...
clone_ptr<base> p2 = std::move(p1);  // p2 now owns the pointer intead of p1
```

For classes made up of other classes (via either containment or inheritance), the move constructor and move assignment can easily be coded using the `std::move` function:

```
class Derived
    : public Base
{
    std::vector<int> vec;
    std::string name;
    // ...
public:
```

```
        // ...
        // move semantics
        Derived(Derived&& x)              // rvalues bind here
            : Base(std::move(x)),
              vec(std::move(x.vec)),
              name(std::move(x.name)) { }

        Derived& operator=(Derived&& x)   // rvalues bind here
        {
            Base::operator=(std::move(x));
            vec  = std::move(x.vec);
            name = std::move(x.name);
            return *this;
        }
        // ...
    };
```

Each subobject will now be treated as an rvalue when binding to the subobject's constructors and assignment operators. `std::vector` and `std::string` have move operations coded (just like our eariler `clone_ptr` example) which will completely avoid the tremendously more expensive copy operations.

Note above that the argument `x` is treated as an lvalue internal to the move functions, even though it is declared as an rvalue reference parameter. That's why it is necessary to say `move(x)` instead of just `x` when passing down to the base class. This is a key safety feature of move semantics designed to prevent accidently moving twice from some named variable. All moves occur only from rvalues, or with an explicit cast to rvalue such as using `std::move`. *If you have a name for the variable, it is an lvalue.*

Question: What about types that don't own resources? (E.g. `std::complex`?)

No work needs to be done in that case. The copy constructor is already optimal when copying from rvalues.

Movable but Non-Copyable Types

Some types are not amenable to copy semantics but can still be made movable. For example:

- `fstream`
- `unique_ptr` (non-shared, non-copyable ownership)
- A type representing a thread of execution

By making such types movable (though still non-copyable) their utility is tremendously increased. Movable but non-copyable types can be returned by value from factory functions:

```
ifstream find_and_open_data_file(/* ... */);
...
ifstream data_file = find_and_open_data_file(/* ... */);  // No copies!
```

In the above example, the underlying file handle is passed from object to object, as long as the source `ifstream` is an rvalue. At all times, there is still only one underlying file handle, and only one `ifstream` owns it at a time.

Movable but non-copyable types can also safely be put into standard containers. If the container needs to "copy" an element internally (e.g. `vector` reallocation) it will move the element instead of copy it.

```
vector<unique_ptr<base>> v1, v2;
v1.push_back(unique_ptr<base>(new derived()));  // ok, moving, not copying
...
v2 = v1;              // Compile time error.  This is not a copyable type.
v2 = move(v1);        // Move ok.  Ownership of pointers transferred to v2.
```

Many standard algorithms benefit from moving elements of the sequence as opposed to copying them. This not only provides better performance (like the improved `std::swap` implementation described above), but also allows these algorithms to operate on movable but non-copyable types. For example the following code sorts a `vector<unique_ptr<T>>` based on comparing the pointed-to types:

```
struct indirect_less
{
    template <class T>
    bool operator()(const T& x, const T& y)
        {return *x < *y;}
};
...
std::vector<std::unique_ptr<A>> v;
...
std::sort(v.begin(), v.end(), indirect_less());
```

As `sort` moves the `unique_ptr`'s around, it will use `swap` (which no longer requires `Copyability`) or move construction / move assignment. Thus during the entire algorithm, the invariant that each item is owned and referenced by one and only one smart pointer is maintained. If the algorithm were to attempt a copy (say by programming mistake) a compile time error would result.

Perfect Forwarding

Consider writing a generic factory function that returns a `std::shared_ptr` for a newly constructed generic type. Factory functions such as this are valuable for encapsulating and localizing the allocation of resources. Obviously, the factory function must accept exactly the same sets of arguments as the constructors of the type of objects constructed. Today this might be coded as:

```
template <class T>
std::shared_ptr<T>
factory()   // no argument version
{
    return std::shared_ptr<T>(new T);
}

template <class T, class A1>
std::shared_ptr<T>
factory(const A1& a1)   // one argument version
{
    return std::shared_ptr<T>(new T(a1));
}

// all the other versions
```

In the interest of brevity, we will focus on just the one-parameter version. For example:

```
std::shared_ptr<A> p = factory<A>(5);
```

Question: What if `T`'s constructor takes a parameter by non-const reference?

In that case, we get a compile-time error as the const-qualifed argument of the `factory` function will not bind to the non-const parameter of `T`'s constructor.

To solve that problem, we could use non-const parameters in our factory functions:

```
template <class T, class A1>
std::shared_ptr<T>
factory(A1& a1)
{
    return std::shared_ptr<T>(new T(a1));
}
```

This is much better. If a const-qualified type is passed to the `factory`, the const will be deduced into the template parameter (`A1` for example) and then properly forwarded to `T`'s constructor. Similarly, if a non-const argument is given to `factory`, it will be correctly forwarded to `T`'s constructor as a non-const. Indeed, this is precisely how forwarding applications are coded today (e.g. `std::bind`).

However, consider:

```
std::shared_ptr<A> p = factory<A>(5);   // error
A* q = new A(5);                        // ok
```

This example worked with our first version of `factory`, but now it's broken: The `"5"` causes the `factory` template argument to be deduced as `int&` and subsequently will not bind to the rvalue `"5"`. Neither solution so far is right. Each breaks reasonable and common code.

Question: What about overloading on every combination of `AI&` and `const AI&`?

This would allow us to handle all examples, but at a cost of an exponential explosion: For our two-parameter case, this would require 4 overloads. For a three-parameter `factory` we would need 8 additional overloads. For a four-parameter `factory` we would need 16, and so on. This is not a scalable solution.

Rvalue references offer a simple, scalable solution to this problem:

```
template <class T, class A1>
std::shared_ptr<T>
factory(A1&& a1)
{
    return std::shared_ptr<T>(new T(std::forward<A1>(a1)));
}
```

Now rvalue arguments can bind to the `factory` parameters. If the argument is const, that fact gets deduced into the `factory` template parameter type.

Question: What is that `forward` function in our solution?

Like `move`, `forward` is a simple standard library function used to express our intent directly and explicitly, rather than through potentially cryptic uses of references. We want to forward the argument `a1`, so we simply say so.

Here, `forward` preserves the lvalue/rvalue-ness of the argument that was passed to `factory`. If an rvalue is passed to `factory`, then an rvalue will be passed to `T`'s constructor with the help of the `forward` function. Similarly, if an lvalue is passed to `factory`, it is forwarded to `T`'s constructor as an lvalue.

The definition of `forward` looks like this:

```
template <class T>
struct identity
{
    typedef T type;
};

template <class T>
T&& forward(typename identity<T>::type&& a)
{
    return a;
}
```

References

As one of the main goals of this paper is brevity, there are details missing from the above description. But the above content represents 95% of the knowledge with a fraction of the reading.

For further details on the motivation of move semantics, such as performance tests, details of movable but non-copyable types, and many other details please see N1377.

For a very thorough treatment of the forwarding problem, please see N1385.

For further applications of the rvalue reference (besides move semantics and perfect forwarding), please see N1690.

For proposed wording for the language changes required to standardize the rvalue reference, please see N1952.

For a summary of the impact the rvalue reference will have on the standard library, please see N1771.

For proposed wording for the library changes required to take advantage of the rvalue reference, please see:

- N1856
- N1857
- N1858
- N1859
- N1860
- N1861
- N1862

For a proposal to extend the rvalue reference to the implicit object parameter (`this`), please see N1821.