Document number: N1377=02-0035

Howard E. Hinnant, hinnant@twcny.rr.com
Peter Dimov, pdimov@mmltd.net
Dave Abrahams, dave@boost-consulting.com

September 10, 2002

A Proposal to Add Move Semantics Support to the C++ Language

Introduction

This proposal seeks to add language support for move semantics to C++. This proposal will discuss various library applications of move semantics, but falls short of making specific library proposals. The library applications discussed herein are to serve as motivation for the language support. Detailed library proposals involving move semantics can be brought forth if the language support finds favor in the committee.

This proposal has the potential to introduce a fundamental new concept into the C++ language. And yet the concept is not new to C++ programmers. Move semantics in various forms has been discussed in C++ forums (most notably comp.lang.c++.moderated) for years. However sweeping, this proposal seeks to minimize changes to the existing

language. An additional goal of this proposal is to not break *any* existing (working) C++ program.

This proposal has also been designed to be 100% compatible with "perfect forwarding" as presented in N1385=020043. That is, the language changes proposed herein solve both move semantics **and** the forwarding problem.

Motivation

Move semantics is mostly about performance optimization: the ability to move an expensive object from one address in memory to another, while pilfering resources of the source in order to construct the target with minimum expense.

Move semantics already exists in the current language and library to a certain extent:

- copy constructor elision in some contexts
- auto_ptr "copy"
- list::splice
- swap on containers

All of these operations involve transferring resources from one object (location) to another (at least conceptually). What is lacking is uniform syntax and semantics to enable generic code to move arbitrary objects (just as generic code today can copy arbitrary objects). There are several places in the standard library that would greatly benefit from the ability to move objects instead of copy them (to be discussed in depth below).

Copy vs Move

C and C++ are built on copy semantics. This is a Good Thing. Move semantics is not an attempt to supplant copy semantics, nor undermine it in any way. Rather this proposal seeks to augment copy semantics. A general user defined class might be both copyable and movable, one or the other, or neither.

The difference between a copy and a move is that a copy leaves the source unchanged. A move on the other hand leaves the source in a state defined differently for each type. The state of the source may be unchanged, or it may be radically different. The only requirement is that the object remain in a self consistent state (all internal invariants are still intact). From a client code point of view, choosing move instead of copy means that you don't care what happens to the state of the source.

For PODs, move and copy are identical operations (right down to the machine instruction level).

What is needed from the language?

Why can not move semantics be a pure library proposal? Can't we just adopt a convention in the library that means move? This almost works. But it falls flat in two very important cases:

1. One of the most important applications for move semantics is to move from temporaries (rvalues). Copy constructor elision (NRVO) almost fixes this, but not quite. Sometimes elision is not possible. Other times even when NRVO is done, it is not sufficient: alternate algorithms are desirable when the source data is known to be an rvalue (e.g. string+string example to be discussed below).
2. Moving from const objects (even rvalues) must be prohibited. It is very difficult to distinguish between a const and an rvalue in the current language (not impossible ... auto_ptr pulls it off).

To help solve these problems, this paper proposes the introduction of a new type of reference that will bind to an rvalue:

```
struct A {/*...*/};

void foo(A&& i);
```

The '&&' is the token which identifies the reference as an "rvalue reference" (bindable to an rvalue) and distinguishes it from our current reference syntax.

Binding temporaries to references

Bjarne in his excellent text "The Design and Evolution of C++" discusses the motivation for prohibiting the binding of an rvalue to a non-const reference in section 3.7. The following example is shown:

```
void incr(int& rr) {rr++;}

void g()
{
    double ss = 1;
    incr(ss);
}
```

ss is not incremented, as a temporary int must be created to pass to incr(). The authors want to say right up front that we agree with this analysis 100%. Howard was even bitten by this "bug" once with an early compiler. It took him forever to track down what was going on (in that case it was an implicit conversion from float to double that created the temporary).

Having said that, we would like to add: You don't ever want to bind a temporary to a non-const reference ... except when you do.

A non-const reference is not always intended to be an "out" parameter. Consider:

```
template <class T>
class auto_ptr
{
public:
    auto_ptr(auto_ptr& a);
    ...
};
```

The "copy" constructor takes a non-const reference named "a". But the modification of "a" is not the primary goal of this function. The primary goal is to construct a new auto_ptr by pilfering "a". If "a" happens to refer to an rvalue, this *is not* a logical error!

This is what the A&& (rvalue reference) is about. Sometimes you really do want to allow a temporary to bind to a non-const reference. The new reference type introduces syntax for allowing that functionality without changing the meaning of any existing code. Thus auto_ptr can be reformulated without auto_ptr_ref. However, the rvalue reference and move semantics are about a lot more than just auto_ptr. The auto_ptr class is used here just to establish a common point of reference.

More on A&&

The rvalue reference is a new type, distinct from the current (lvalue) reference. Functions can be overloaded on A& and A&&, requiring such functions to have distinct signatures.

The most common overload set anticipated is:

```
void foo(const A& t);  // #1
void foo(A&& t);       // #2
```

The rules for overload resolution are (in addition to the current rules):

> rvalues will prefer rvalue references. lvalues will prefer lvalue references. CV qualification conversions are considered secondary relative to r/l-value conversions. rvalues can still bind to a const lvalue reference (const A&), but only if there is not a more attractive rvalue reference in the overload set. lvalues can bind to an rvalue reference, but will prefer an lvalue reference if it exists in the overload set. The rule that a more cv-qualified object can not bind to a less cv-qualified reference stands ... both for lvalue and rvalue references.

Examples:

```
struct A {};
A source();
const A const_source();
...
A a;
const A ca;
foo(a);               // binds to #1
foo(ca);              // binds to #1
foo(source());        // binds to #2
foo(const_source());  // binds to #1
```

The first foo() call prefers the lvalue reference as (lvalue)a is a better match for const A& than for A&& (lvalue -> rvalue conversion is a poorer match than A& -> const A& conversion). The second foo() call is an exact match for #1. The third foo() call is an exact match for #2. The fourth foo() call can not bind to #2 because of the disallowed const A&& -> A&& conversion. But it will bind to #1 via an rvalue->lvalue conversion.

Note that without the second foo() overload, the example code works with all calls going to #1. As move semantics are introduced, the author of foo knows that he will be attracting non-const rvalues by introducing the A&& overload and can act accordingly. Indeed, the only reason to introduce the overload is so that special action can be taken for non-const rvalues.

Even though *named* rvalue references can bind to an rvalue, they are treated as lvalues when used. For example:

```
struct A {};

void h(const A&);
void h(A&&);

void g(const A&);
void g(A&&);

void f(A&& a)
{
    g(a);  // calls g(const A&)
    h(a);  // calls h(const A&)
}
```

Although an rvalue can bind to the "a" parameter of f(), once bound, a is now treated as an lvalue. In particular, calls to the overloaded functions g() and h() resolve to the const A& (lvalue) overloads. Treating "a" as an rvalue within f would lead to error prone code: First the "move version" of g() would be called, which would likely pilfer "a", and then the pilfered "a" would be sent to the move overload of h().

In a sense, the rvalue reference is already used in C++ when dealing with the implicit object parameter as discussed in section 13.3 (Overload resolution). A temporary is allowed to bind to the implicit object parameter which is said to be of type "reference to cv X" (see 13.3.1/4). Introduction of the "rvalue reference" allows for the exception relating to temporaries and the implicit object parameter to be removed from the language (section 13.3.3.1.4 / 3). Instead, the implicit object parameter can simply be of type cv X&&. Remember: you really do not want to bind a temporary to a non-const reference ... except when you do!

move_ptr Example

An auto_ptr-like class can be introduced (lets call it move_ptr) which is movable, but not copyable. Client code might look like:

```
template <class T> move_ptr<T> source();
...
move_ptr<int> p(new int(1));
move_ptr<int> q = source<int>();
p = source<int>();
```

The move_ptr class is constructible and assignable from rvalues. However, one can not copy construct nor assign from lvalues of move_ptr. The following client code will generate compile time errors:

```
move_ptr<int> r = p;  // error:  can't copy from lvalue
p = q;                // error:  can't assign from lvalue
```

The refusal to move from lvalues using copy syntax is key to the safety of move_ptr. Later it will be shown that move_ptr can be safely put into a move-aware container such as vector or map with no chance of accidental transfer of ownership into or out of the container.

The move_ptr has an accessible constructor and assignment taking non-const rvalues, but traditional copy semantics has been disabled by making the copy constructor and copy assignment private:

```
template<class X>
class move_ptr
```

```
{
public:
    typedef X value_type;

    explicit move_ptr(X* p = 0) throw()
        : ptr_(p) {}
    move_ptr(move_ptr&& a) throw()
        : ptr_(a.release()) {}
    template<class Y> move_ptr(move_ptr<Y>&& a) throw()
        : ptr_(a.release()) {}

    ~move_ptr() throw() {delete ptr_;}

    move_ptr& operator=(move_ptr&& a) throw()
        {reset(a.release()); return *this;}
    template<class Y> move_ptr& operator=(move_ptr<Y>&& a) throw()
        {reset(a.release()); return *this;}

    X& operator*() const throw()  {return *ptr_;}
    X* operator->() const throw() {return ptr_;}
    X* get() const throw()        {return ptr_;}
    X* release() throw()
        {X* tmp = ptr_; ptr_ = 0; return tmp;}
    void reset(X* p = 0) throw()
        {if (ptr_ != p) {delete ptr_; ptr_ = p;}}
private:
    X* ptr_;

    move_ptr(const move_ptr& a);
    template<class Y> move_ptr(const move_ptr<Y>& a);
    move_ptr& operator=(const move_ptr& a);
    template<class Y> move_ptr& operator=(const move_ptr<Y>& a);
};
```

The move constructor is **not** a special member. It is like any other constructor and not like the copy constructor in this regard. Introduction of a move constructor does not supplant the copy constructor, or inhibit the implicit definition of a copy constructor. The move constructor overloads the copy constructor, whether the copy constructor is implicit or not. Similarly for the move assignment. Move constructors and move assignment are also not implicitly defined by the compiler should the class author not include them. Thus no current C++ classes are movable. The class author must explicitly provide move semantics for his class.

So far so good. The combination of the `const A&` and the `A&&` makes it easy for the everyday class designer to put copy semantics and/or move semantics into his class. Move semantics will automatically come into play when given rvalue arguments. This is perfectly safe because moving resources from an rvalue can not be noticed by the rest of the program (*nobody else has a reference to the rvalue in order to detect a difference*).

Cast to rvalue

The client of a movable object might decide that he wants to move from an object even though that object is an lvalue. There are many situations when such a need might arise. One common example is when you have a full dynamic array of objects and you want to add to it. You must allocate a larger array and *move* the objects from the old buffer to the new. The objects in the array are obviously not rvalues. And yet there is no reason to **copy** them to the new array if **moving** can be accomplished *much* faster.

Thus this paper proposes the ability to cast from an lvalue type T to an rvalue:

```
move_ptr<int> p, q;
...
p = static_cast<move_ptr<int>&&>(q);  // ok
```

The right hand side is considered to be an rvalue after the cast. Thus the move_ptr assignment works via the move assignment operator. Note that this amounts to a *request* to move, not a *demand* to move. Had move_ptr been a class that supported copy assignment, but not move assignment, the above assignment from q to p would still have worked since you can always assign from an rvalue using a copy assignment (unless the type explicitly forbids copy semantics like move_ptr).

The "request to move" semantics turn out to be very handy in generic code. One can request that a type move itself without having to know whether or not the type is really movable. If the type is movable it will move, else if the type is copyable, it will copy, else you will get a compile-time error.

A classic example of where this could come in handy is swap:

swap Example

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

If T is movable, then move construction and move assignment will be used to perform the swap. If T is not movable, but is copyable, then copy semantics will handily perform the swap. Otherwise the swap function will fail at compile time. Swapping using move semantics (when available) can produce code that is as efficient, or nearly as efficient (constant complexity) as a custom swap.

Returning A&&

The static_cast<A&&> syntax is admittedly ugly. This is not necessarily a bad thing as you want to clearly call out when you are doing something as dangerous (and useful!) as moving from an lvalue. But the cast can become so ugly as to be unreadable when the type A is a long complicated identifier.

Earlier this paper proposed that:

> Named rvalue references are treated as lvalues.

This thought will now be completed:

> Unnamed rvalue references are treated as rvalues.

An example of an unnamed rvalue reference would be returning such a type from a function. Consider:

```
template <class T>
inline
```

```
T&&
move(T&& x)
{
    return static_cast<T&&>(x);
}
```

Now calling move(x) is a synonym for casting x to an rvalue. Thus swap can be made more readable:

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

Treating unnamed rvalue references as rvalues is consistent, at least syntactically with treating the result of static_cast<A&&> as an rvalue. This behavior is also beneficial (performance wise) to the string+string example shown later.

Binding Summary

Here are several examples that serve to summarize the binding rules in one place:

```
struct A {};

void foo(const A&);  // #1
void foo(A&&);       // #2

A   source_rvalue();
A&  source_ref();
A&& source_rvalue_ref();

const A   source_const_rvalue();
const A&  source_const_ref();
const A&& source_const_rvalue_ref();

int main()
{
    A a;
    A&  ra = a;
    A&& rra = a;
    const A ca;
    const A& rca = ca;
    const A&& rrca = ca;

    foo(a);      // #1
    foo(ra);     // #1
    foo(rra);    // #1

    foo(ca);     // #1
    foo(rca);    // #1
    foo(rrca);   // #1

    foo(source_rvalue());     // #2
    foo(source_ref());        // #1
    foo(source_rvalue_ref()); // #2

    foo(source_const_rvalue());     // #1
    foo(source_const_ref());        // #1
    foo(source_const_rvalue_ref()); // #1
}
```

lvalues, both const and non-const bind to foo(const A&). Named references, both lvalue and rvalue, all bind to foo(const A&). Non-const rvalues and unnamed rvalue references bind to foo(A&&). Const rvalues and const rvalue references bind to foo(const A&). Had there been a foo(const A&&) available, they would have bound to that. The lvalue references bind to foo(const A&).

string Motivation

Like any other class, std::basic_string can be given both move and copy constructors, and move and copy assignment, just as previously discussed. For example:

```
class string
{
public:
    // copy semantics
    string(const string& s)
        : data_(new char[s.size_]), size_(s.size_)
        {memcpy(data_, s.data_, size_);}
    string& operator=(const string& s)
        {if (this != &s)
        {
            if (size_ < s.size_)
                // get sufficient data buffer
            size_ = s.size_;
            memcpy(data_, s.data_, size_);
        }
        return *this;}
    // move semantics
    string(string&& s)
        : data_(s.data_), size_(s.size_) {s.data_ = 0; s.size_ = 0;}
    string& operator=(string&& s) {swap(s); return *this;}
    // ...
private:
    char* data_;
    size_t size_;
    // ...
};
```

The move constructor and move assignment will be automatically called when given rvalues, and client code can explicitly cast to rvalue if moving from an lvalue is desired. So far so good. But there is more fun to be had. Consider:

```
string s1("12345678901234567890");
string s0 = s1 + "a" + "b" + "cd";
```

Even with NRVO implemented in operator+(string, string), and even with a short string optimization (holding say less than 20 characters), the expression to form s0 will typically allocate memory at least once per operator+() for a total of 3 accesses to the heap. The reason is that operator+(string, string) will typically look something like:

```
string
operator+(const string& x, const string& y)
{
    string result;
    result.reserve(x.size() + y.size());
    result = x;
    result += y;
    return result;
}
```

For each +operation, a new string is created with sufficient capacity for the result and that is returned. With move semantics the expression for s0 can be much more efficient, going to the heap only once (or maybe twice depending on some string implementation details). This is made possible by overloading operator+ for rvalues as one or both of the arguments. If one of the arguments is an rvalue, it is much more efficient to simply append to the rvalue, rather than make a whole new string. It is quite possible that the existing capacity in the rvalue is already sufficient so that the append operation need not go to the heap at all.

```
string&&
operator+(string&& x, const string& y)
{
    return x += y;
}

string&&
operator+(const string& x, string&& y)
{
    return y.insert(0, x);
}

string&&
operator+(string&& x, string&& y)
{
    return x += y;
}
```

Note that an rvalue reference to the rvalue argument can be returned. This is further motivation for the rule that unnamed rvalue references are treated as rvalues. The return type of string+string must be regarded as an rvalue. The rvalue overloads of string+string could return by value, but this would not be as efficient as returning by reference, even with a move constructor helping out.

This proposal has been partially implemented in Metrowerks CodeWarrior, and indeed, the code to form s0 above only goes to the heap once during s1 + "a". The temporary created in that first operation happens to have sufficient capacity for the next two concatenations.

If NRVO is not operational, move semantics further aids by move constructing s0 from the rvalue generated by the expression.

Moving from local values

A further language refinement can be made at this point. When returning a non-cv-qualified object with automatic storage from a function, there should be an implicit cast to rvalue:

```
string
operator+(const string& x, const string& y)
{
    string result;
    result.reserve(x.size() + y.size());
    result = x;
    result += y;
    return result;  // as if return static_cast<string&&>(result);
}
```

The logic resulting from this implicit cast results in an automatic hierarchy of "move semantics" from best to worst:

- If you can elide the move/copy, do so (by present language rules)
- Else if there is a move constructor, use it
- Else if there is a copy constructor, use it
- Else the program is ill formed

With this language feature in place, move/copy elision, although still important, is no longer critical. There are some functions where NRVO is allowed, but can be exceedingly difficult to implement. For example:

```
A
f(bool b)
{
    A a1, a2;
    // ...
   return b ? a1 : a2;
}
```

It is somewhere between difficult and impossible to decide whether to construct a1 or a2 in the caller's preferred location. Using A's move constructor (instead of copy constructor) to send a1 or a2 back to the caller is the best solution.

We could require that the author of operator+ explicitly request the move semantics. But what would be the point? The current language already allows for the elision of this copy, so the coder already can not rely on destruction order of the local, nor can he rely on the copy constructor being called. The auto-local is about to be conceptually destructed anyway, so it is very "rvalue-like". The move is not detectable except by measuring performance, or counting copies (which may be elided anyway).

Note that this language addition permits movable, but non-copyable objects (such as move_ptr) to be returned by value, since a move constructor is found and used (or elided) instead of the inaccessible copy constructor.

References to References

As discussed in Core Defect 106, and in the forwarding proposal, it is important to consider what happens when references to references are formed. To summarize from N1385=020043, the *reference collapsing* rules are:

- `A& & -> A&`
- `A& && -> A&`
- `A&& & -> A&`
- `A&& && -> A&&`

(cv qualifications are unioned as discussed in cw 106 and N1385=020043).

This behavior is not only critical for perfect forwarding, but for move as well. Consider for example a container such as boost::compressed_pair that is able to hold references. The move constructor for such an object will look like:

```
compressed_pair<T1, T2>::compressed_pair(compressed_pair&& x)
    : first_ (static_cast<T1&&>(x.first_)),
      second_(static_cast<T2&&>(x.second_))
{}
```

That is, each data member from the source will be cast to rvalue in order to invoke the move constructor for the target data member. But what if T1 is a reference: A& ? The static_cast<T1&&> is casting a reference type to an rvalue. The result can not be an rvalue, or it would not bind to first_, which is itself a reference. (rvalues can not bind to non-const references). But expanding out the types manually (static_cast<A& &&>(), and using the *reference collapsing* rules, the expression simplifies to static_cast<A&>. That is, when T1 is a reference type, the call behaves in a copy-like manner, and when T1 is not a reference type, the call behaves in move-like manner. Exactly what is needed!

Template Argument Deduction with A&&

This final language proposal is not strictly needed for move, but it is needed for perfect forwarding. It is briefly summarized here for completeness, and to assure the gentle reader that this behavior is 100% compatible with this move proposal.

When deducing a function template type using an lvalue argument matching to an rvalue reference, the type is deduced as an lvalue reference type. When deduction is given an rvalue argument, type deduction proceeds just as with other types. For example:

```
template <typename T>
void f(T&& t);
...
struct A {};

void g()
{
    f(A());  // calls f<A>(A&& t)
    A a;
    f(a);    // calls f<A&>(A& && t)  ->  f<A&>(A& t)
}
```

This behavior is key to "perfect forwarding". A forwarding function can exactly replicate both the cv qualifiers and the l/r-valueness of the argument it receives so that the forwarded-to function sees the exact same argument.

What is needed from the language - Summary

- A new reference type that will bind to rvalues, (an rvalue reference).
- The rvalue reference must be overloadable with the lvalue reference (the current reference).
- Must be able to cast from lvalue to rvalue.
- Named rvalue references are treated as lvalues.
- Unnamed rvalue references are treated as rvalues.
- When elision of the copy constructor is already legal for function return values, implicit cast to rvalue.
- Reference collapsing rules extended from core issue 106.
- "Augmented" template argument deduction when binding an lvalue to an rvalue reference.

With these few basic (but closely related) language tools, this proposal has already shown significant potential enhancements to both the standard library, and to code in general:

- A move_ptr smart pointer class that will move and not copy.

- A much improved swap function template.
- General syntax (for any class) for move constructors and move assignment.
- A significantly more efficient operator+ for a string class.
- Automatic moving from rvalues.
- Ability for clients to manually request "move if you can, else copy" with a consistent, uniform syntax.
- An auto_ptr design that does not need auto_ptr_ref.
- Solves the forwarding problem - perfectly!

The above summary is all that this proposal asks for. We believe that these potential benefits alone should justify this incremental (and fully backwards compatible) change in the language. But these benefits are just the tip of the iceberg. The remainder of the proposal consists of further motivation, clarifications, and alternative designs (for move).

vector Example

Like <string>, <vector> can be made movable by implementing a move constructor and move assignment. But as vector is a container for a general type T, significant optimizations can take place if T itself is movable (string doesn't have this issue because its value_type is assumed to be a POD). For pods moving and copying are the same thing.

The most significant impact move has (or can have) on vector is in the implementation of its erase and insert functions. During a vector::insert one of two things can happen:

1. The vector's capacity is exceeded: a new buffer must be allocated, and the old elements are moved/copied from the old buffer to the new buffer, possibly in different relative positions to make room for the newly inserted elements.
2. The vector's capacity is not exceeded, in which case the vector's elements are moved/copied to other parts of the buffer in order to make room for the newly inserted elements (unless the new elements are to be appended on the end).

In either case, if T can be moved instead of copied, the performance benefits are clear. Consider vector<string>. In the sufficient capacity case if copy semantics are used, then every string assignment or copy