An Introduction to the Rvalue Reference in C++0X

Howard E. Hinnant
Apple Inc.
ACCU 2007
Outline

• Move Semantics
  • Problem statement
  • Solution

• Perfect Forwarding
  • Problem statement
  • Solution
We would like to return “big” types by value without worrying if RVO will work or not.

Functional style programming for expensive value types.

Workaround: Pass a reference or pointer to the result.

```cpp
vector<int>
compute(...data...);

// Expensive, O(N)!
v = compute(...data...);

void
compute(vector<int>& out, ...
...data ...
);```
Code That We Wish Worked

Generic, sequence modifying code:

Iter i = first;
for (++i; i != last; ++i)
{
    Iter j = i;
    value_type tmp(*j);  // Expensive!
    for (Iter k = i; k != first && tmp < *--k; --j)
        *j = *k;  // Expensive!
    *j = tmp;  // Expensive!
}

• Consider when value_type is vector<string>
• An rvalue reference is just a new type of reference spelled “&&” instead of “&”.

```cpp
class A {}
A a;
A&& a_ref1 = A();
A& a_ref2 = a;
```
Why A New Reference Type?

• Use of a new reference type, as opposed to a keyword which means “move” or “rvalue” is desirable because the rvalue reference solves more problems than just move semantics.
  • Move semantics
  • Perfect Forwarding
  • Useful rvalue streams
  • etc.
What Exactly Is an Rvalue Reference?

- An rvalue reference is exactly like our existing reference (now known as an lvalue reference) with a few exceptions:
  - An rvalue will bind to a non-const rvalue reference.
  - Rvalue references and lvalue references are distinct types and overloadable.
Overloading References

- lvalues can bind to any kind of reference (but prefer lvalue references).
- rvalues can bind to rvalue references or a const-qualified lvalue reference.
Overloading References

- lvalues can bind to any kind of reference (but prefer lvalue references).
- rvalues can bind to rvalue references or a const-qualified lvalue reference.
Overloading References

- const lvalues can bind to any const-qualified reference.
- const rvalues can bind to any const-qualified reference.
Putting It Together
(clone_ptr is just an example)

template <class T> class clone_ptr {
private: T* ptr_; 
public: clone_ptr(const clone_ptr& p)
    : ptr_(p.ptr_ ? p.ptr_->clone() : 0) {}
   clone_ptr& operator=(const clone_ptr& p) {
    if (this != &p) {
        T* tmp = p.ptr_ ? p.ptr_->clone() : 0;
        delete ptr_; 
        ptr_ = tmp;  
    }
   return *this;
  }
};
Putting It Together
(clone_ptr is just an example)

template <class T> class clone_ptr {
private: T* ptr_
public:
    clone_ptr(const clone_ptr& p); // lvalues
    clone_ptr& operator=(const clone_ptr& p);
    clone_ptr(clone_ptr&& p) // rvalues
        : ptr_(p.ptr_) {p.ptr_ = 0;} 
    clone_ptr& operator=(clone_ptr&& p) {
        delete ptr_
        ptr_ = p.ptr_
        p.ptr_ = 0;
        return *this;
    }
};

• With move semantics
• no throw
• fast!
Example Uses of clone_ptr

typedef clone_ptr<int> CP;
CP p1;
CP p2(p1);                    // clone_ptr(const clone_ptr&)  
CP p3 = std::move(p2);        // clone_ptr(clone_ptr&&)
CP make_clone_ptr();          // function prototype
CP p4 = make_clone_ptr();     // clone_ptr(clone_ptr&&)

• Client code can copy or move clone_ptr’s around.
  • If clone_ptr’s are returned (by value) from a function, the move happens implicitly.
  • All std::move does is turn an lvalue into an rvalue (a temporary).
When base classes and class data members are involved, delegate move logic with explicit use of std::move (just as a rule of thumb).

```cpp
class A : B {
    std::string name_
public:
    A(A&& a) : B(std::move(a)), name_(std::move(a.name_)) {}
    A& operator=(A&& a) {
        B::operator=(std::move(a));
        name_ = std::move(a.name_);
        return *this;
    }
};
```
Double-move Protection

void f1(const A&);
void f1(A&&);  // can change arg
void f2(const A&);
void f2(A&&);  // can change arg

void my_func(A&& a) {
    f1(a);
    f2(a);
}

• If f1() moves from “a”, f2() is likely to have a run-time failure.
• Therefore both calls treat “a” as an lvalue.
Double-move Protection

void f1(const A&);
void f1(A&&);  // can change arg
void f2(const A&);
void f2(A&&);  // can change arg

void my_func(A&& a) {
    f1(a);       // f1(const A&)
    f2(a);       // f2(const A&)
}

• If f1() moves from “a”, f2() is likely to have a run-time failure.

• Therefore both calls treat “a” as an lvalue.
Double-move Protection

```cpp
void f1(const A&);
void f1(A&&);  // can change arg
void f2(const A&);
void f2(A&&);  // can change arg

void my_func(A&& a) {
    f1(a);        // f1(const A&)
    f2(std::move(a));  // f2(A&&)
}
```

- If a move is desired, it must be explicit.
- This prevents accidental moves.
Move-Only Types

template <class T> class unique_ptr {
private: T* ptr_;  
    unique_ptr(const unique_ptr& p);
    unique_ptr& operator=(const unique_ptr& p);

public:
    unique_ptr(unique_ptr&& p)
        : ptr_(p.ptr_) {p.ptr_ = 0;}
    unique_ptr& operator=(unique_ptr&& p) {
        delete ptr_;  
        ptr_ = p.ptr_;  
        p.ptr_ = 0;  
        return *this;  
    }
};

- A smart pointer which can \textit{only be moved}, not copied, is a simple variation of clone\_ptr.
Move-Only Types

template <class T> class clone_ptr {
private: T* ptr_;  
public:
    clone_ptr(const clone_ptr& p);
    clone_ptr& operator=(const clone_ptr& p);
    clone_ptr(clone_ptr&& p)
        : ptr_(p.ptr_) {p.ptr_ = 0;}
    clone_ptr& operator=(clone_ptr&& p) {
        delete ptr_;  
        ptr_ = p.ptr_;  
        p.ptr_ = 0;
        return *this;
    } 
};

- A smart pointer which can only be moved, not copied, is a simple variation of clone_ptr.
Move-Only Types

template <class T> class unique_ptr {
private: T* ptr_;    
unique_ptr(const unique_ptr& p);    
unique_ptr& operator=(const unique_ptr& p);
public:
unique_ptr(unique_ptr&& p)    
    : ptr_(p.ptr_) {p.ptr_ = 0;}
unique_ptr& operator=(unique_ptr&& p) {
    delete ptr_;    
    ptr_ = p.ptr_;    
    p.ptr_ = 0;    
    return *this;
}
};

• A smart pointer which can only be moved, not copied, is a simple variation of clone_ptr.
Example Uses of `unique_ptr`

typedef `unique_ptr<int>` CP;
CP p1;
CP p2(p1);  // compile time error
CP p3 = std::move(p2);  // `unique_ptr(unique_ptr&&)`
CP make_unique_ptr();  // function prototype
CP p4 = make_unique_ptr();  // `unique_ptr(unique_ptr&&)`

- Client code can move `unique_ptr`'s around.
- If `unique_ptr`'s are returned (by value) from a function, the move happens implicitly.
- Clients can not copy `unique_ptr`'s (compile time error)
Returning Types From Functions

```cpp
def make_unique_ptr() -> unique_ptr<int>:
    p = unique_ptr<int>(new int);
    *p = 2;
    return p;  # Ok, implicit cast to rvalue
```

- Where Return Value Optimization is already legal today, there will be an implicit cast to rvalue.
- Thus move-only types are easily returned from factory functions.
Returning Types From Functions

vector<int> make_vector() 
{
    vector<int> v(100000);
    v[0] = 2;
    return v;  // Ok, implicit cast to rvalue
}  // vector(vector&&) called or elided

- Types that are expensive to copy, but cheap to move, can be efficiently returned from factory functions!
- The move may still be elided via RVO.
Returning Types From Functions

complex<double> make_complex()
{
    complex<double> c(1., 2.);
    c *= c;
    return c; // Ok, implicit cast to rvalue
} // complex(const complex&) called or elided

• Types with copy constructors, but without move constructors continue to work exactly as they do today
Examples Of Move-Only Types

- fstreams, stringstream
- unique_ptr (auto_ptr replacement)
- vector<a move-only type>
- thread - proposed
- lock<Mutex>

- Any handle-type class which refers to a unique, uncopiable resource, is a good candidate for move-only.
Unique Ownership Strategies
Example

- Consider:

```cpp
class Matrix
{
    double* data_;  
    int row_;  
    int col_;  

public:
    Matrix(int row, int col);
    ~Matrix();
    Matrix(const Matrix& m);  
    Matrix& operator+=(const Matrix& m);  
    ...  
};
```

- What is the best way to implement Matrix+Matrix?
Simple Unique Ownership Example

- Use Case:
  
  ```cpp
  Matrix m1 = m2 + m3 + m4 + m5 + m6;
  ```

- The C++03 solution:

  ```cpp
  Matrix operator+(const Matrix& x, const Matrix& y) {
    Matrix r(x);
    r += y;
    return r;
  }
  ```
Simple Unique Ownership
Example - Cost

• Cost analysis

Matrix m1 = m2 + m3 + m4 + m5 + m6;
allocate row*col doubles for m2+m3 -> t1
allocate row*col doubles for t1+m4 -> t2
allocate row*col doubles for t2+m5 -> t3
allocate row*col doubles for t3+m6 -> m1
deallocate doubles for t3
deallocate doubles for t2
deallocate doubles for t1

• 4 allocations, 3 deallocations (assumes RVO).
Copy On Write - Refcounting

class Matrix
{
    MatrixImp* imp_; // refcounted ptr
public:
    Matrix(int row, int col);
    ~Matrix();
    Matrix(const Matrix& m);
    Matrix& operator+=(const Matrix& m);
    ...
};

• Assume a quality implementation:
  • Embedded refcount; construction allocates once.
  • Refcount protected by atomics.
Copy On Write - Refcounting

- Cost analysis

\[ \text{Matrix } m1 = m2 + m3 + m4 + m5 + m6; \]

allocate row*col doubles for m2+m3 -> t1
allocate row*col doubles for t1+m4 -> t2
allocate row*col doubles for t2+m5 -> t3
allocate row*col doubles for t3+m6 -> m1
deallocate doubles for t3
deallocate doubles for t2
deallocate doubles for t1

- 4 allocations, 3 deallocations.

- At least 3 atomic operations.
Rvalue Reference Based Solution

Matrix::Matrix(Matrix&& x)
    : data_(x.data_), row_(x.row_), col_(x.col_)
    {x.data_ = 0; x.row_ = x.col_ = 0;}

Matrix // this as in C++03
operator+(const Matrix& x, const Matrix& y);

Matrix
operator+(Matrix&& temp, const Matrix& y) {
    temp += y;
    return std::move(temp);
}

- Add a move constructor.
- If op+ sees a temporary, just add to it instead of creating a new temporary.
Rvalue Reference Based Solution

- Cost analysis
  
  \[
  \text{Matrix } m1 = m2 + m3 + m4 + m5 + m6;
  \]

  allocate row*col doubles for m2+m3 -> m1
Rvalue Reference Based Solution

- Cost analysis
  
  Matrix m1 = m2 + m3 + m4 + m5 + m6;

  allocate row*col doubles for m2+m3 -> t1
  t1 RVO’d (or moved) out
  t1 += m4  // t1 moved out
  t1 += m5  // t1 moved out
  t1 += m6  // t1 moved into m1

- One allocation.

- Cost analysis does not change if RVO is not applied.
Cost Summary for Matrix + Matrix

<table>
<thead>
<tr>
<th></th>
<th>Allocations</th>
<th>Deallocations</th>
<th>Outstanding</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++03</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>COW (rc)</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Rvalue-ref</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- Move semantics makes unique ownership both simple and practical.
Cost Summary for Matrix + Matrix

- **C++03:**
  - Allocations: 4
  - Deallocations: 3
  - Outstanding: 1

- **COW (rc):**
  - Allocations: 4
  - Deallocations: 3
  - Outstanding: 1

- **Rvalue-ref:**
  - Allocations: 1
  - Deallocations: 0
  - Outstanding: 1

- Move semantics makes unique ownership both simple and practical.
• Rvalue reference greatly reduces the motivation for shared ownership.
Memory Ownership Strategies

- R-value reference greatly reduces the motivation for shared ownership.

Unique Ownership

- Circular Shared Ownership

Shared Ownership

Reference interface
Iter \(i = \text{first};\)
for (++\(i; i \neq \text{last}; ++i\))
{
    Iter \(j = i;\)
    value_type tmp(std::move(*j));
    for (Iter \(k = i; k \neq \text{first} \&\& \text{tmp} < \ast--k; --j\))
        *\(j = \text{std}::\text{move}(*k);\)
    *\(j = \text{std}::\text{move}(\text{tmp});\)
}

- This fact that a value is no longer needed after copied from is communicated with std::move.
- For some value_types (such as vector) this makes the “assignment” very fast.
- For other value_types (such as int) the std::move has absolutely no affect whatsoever.
Iter i = first;
for (++i; i != last; ++i)
{
    Iter j = i;
    value_type tmp(std::move(*j));
    for (Iter k = i; k != first && tmp < *--k; --j)
        *j = std::move(*k);
    *j = std::move(tmp);
}

- This fact that a value is no longer needed after copied from is communicated with std::move.
- For some value_types (such as vector) this makes the “assignment” very fast.
- For other value_types (such as int) the std::move has absolutely no affect whatsoever.
Perfect Forwarding

• Consider writing a generic factory function for `shared_ptr<T>`, where T is unknown, as are the arguments for constructing T.

• Assume for the moment that T’s constructor takes a single argument.

```cpp
template <class T, class A1>
shared_ptr<T> make_ptr(const A1& a1)
{
    return shared_ptr<T>(new T(a1));
}
...
shared_ptr<B> p = make_ptr<B>(2);
Problem using const A1&

template <class T, class A1>
shared_ptr<T>
make_ptr(const A1& a1)
{
    return shared_ptr<T>(new T(a1));
}
...

double d = 2;
shared_ptr<B> p = make_ptr<B>(d);

• What if B constructs with a non-const reference? B::B(double&)

• Doesn’t compile because make_ptr uses const A1&.
Solution: A1 &

template <class T, class A1>
shared_ptr<T>
make_ptr(A1& a1)
{
    return shared_ptr<T>(new T(a1));
}
...

double d = 2;
shared_ptr<B> p = make_ptr<B>(d);

• Now make_ptr works with B::B(double&), but...
Problem using A1&

template <class T, class A1>
shared_ptr<T>
make_ptr(A1& a1)
{
    return shared_ptr<T>(new T(a1));
}
...
shared_ptr<B> p = make_ptr<B>(2);

• What if B constructs with a double or const double&? B::B(double)

• Doesn’t compile because make_ptr uses A1&. It won’t bind to rvalue arguments.
Lack of Ability to Write Generic Factory Functions

make_ptr(A1& a1)
make_ptr(const A1& a1)

• Neither design is sufficiently generic to cover all reasonable use cases.

• We need a way to tell make_ptr to accept any lvalue or rvalue, and forward that to T’s constructor, with the same cv-qualifications, and lvalue or rvalue status as the original argument supplied to make_ptr.
Rvalue Reference Supplies
Perfect Forwarding

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

... shared_ptr<B> p = make_ptr<B>(2);

- A1&& binds to both lvalues and rvalues.
- The cv-qualifications are deduced into the type of A1 (same as today with A1&).
template <class T, class A1>
shared_ptr<T>
make_ptr(A1&& a1)
{
    return shared_ptr<T>(
        new T(std::forward<A1>(a1)));
}

...  
shared_ptr<B> p = make_ptr<B>(2);

- Also deduced into the type of A1 is whether an lvalue or rvalue was bound to the parameter.
Rvalue Reference Supplies
Perfect Forwarding

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

...  
shared_ptr<B> p = make_ptr<B>(2);

• If an lvalue of type B was bound, A1 is B&.

• If an rvalue of type B was bound, A1 is B.
Rvalue Reference Supplies
Perfect Forwarding

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

... shared_ptr<B> p = make_ptr<B>(2);

- std::forward<B&> returns an lvalue B.
- std::forward<B> returns a rvalue B.
Rvalue Reference Supplies
Perfect Forwarding

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

...  
shared_ptr<B> p = make_ptr<B>(2);

• Thus the B constructor, called from inside of make_ptr, sees **exactly** the same argument as supplied from outside of make_ptr.
Making it Work With an Arbitrary Number of Arguments

template <class T, class ...A>
shared_ptr<T>
make_ptr(A&& a...)
{
    return shared_ptr<T>(
        new T(std::forward<A>(a)...));
}
...
shared_ptr<B1> p1 = make_ptr<B1>(2);
shared_ptr<B2> p2
    = make_ptr<B2>(3, 4.5, std::string("Hi!")));

• One simple function perfectly forwards an arbitrary number of arguments!
Move + Forward

- Perfect forwarding enables move semantics to work correctly across generic forwarding functions.

```c
struct B2 {
    B2(int, double, const string&); // copy string in
    B2(int, double, string&&);     // move string in
};
```
Move + Forward

struct B2 {
    B2(int, double, const string&);// copy string in  
    B2(int, double, string&&);     // move string in
};

- Perfect forwarding enables move semantics to work correctly across generic forwarding functions.

// constructs using B2(int, double, string&&)
shared_ptr<B2> p2
    = make_ptr<B2>(3, 4.5, string("Hi!")));
Summary

• The rvalue reference is a new reference type.
• It solves the problem of move semantics.
• It solves the problem of perfect forwarding.
• Its use is largely hidden at the highest code levels.
• Use helper functions move and forward.