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

Howard E. Hinnant
Apple Inc.
BoostCon 2007

Outline

- Move Semantics
  - Problem statement
  - Solution
- Perfect Forwarding
  - Problem statement
  - Solution

Code That We Wish Worked

- 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.

```
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!
}
```

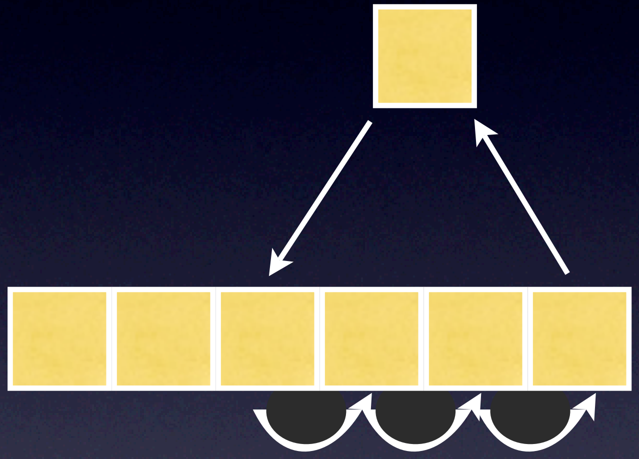
- Consider when `value_type` is `vector<string>`

Observation

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

- In each of the above expensive copies, the source value is never used again after the copy.

Observation



- In each of the above expensive copies, the source value is never used again after the copy.

Observation

```
Iter i = first;
for (++i; i != last; ++i)
{
   Iter j = i;
   value_type tmp(*j);  // *j not read again
   for (Iter k = i; k != first && tmp < *--k; --j)
      *j = *k;  // *k not read again
   *j = tmp;    // tmp not read again
}
```

- In each of the above expensive copies, the source value is never used again after the copy.

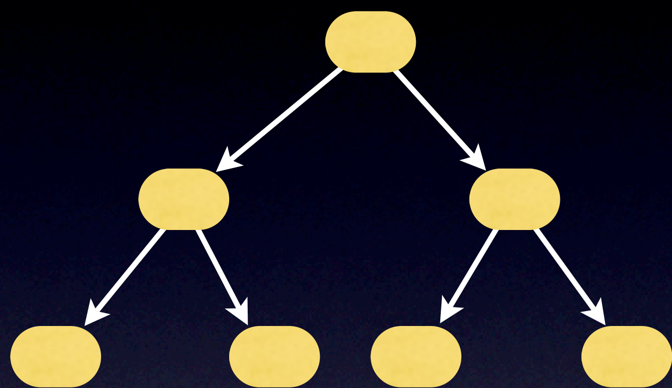
Doesn't Reference Counting Solve this Problem?

- Copy constructing and assignment using reference counting is very fast.
  - Just increment/decrement the reference count.
  - What could be faster?

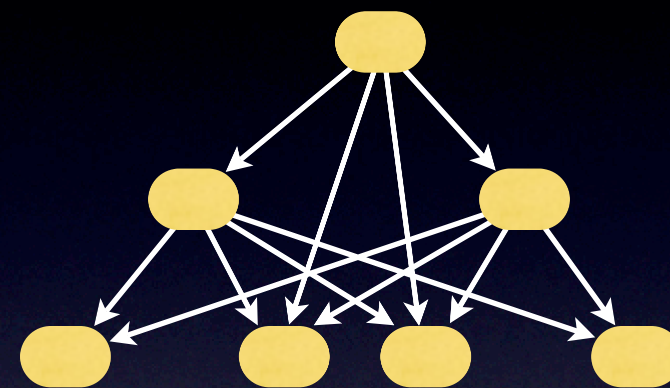
Problems With Reference Counting

- Reference counting creates shared ownership semantics which can be more complicated to reason about than sole-ownership.
- Reference counting generally requires atomic operations in a multi-threaded environment (a potential performance penalty on multiprocessor architectures).

Shared Ownership vs Sole Ownership



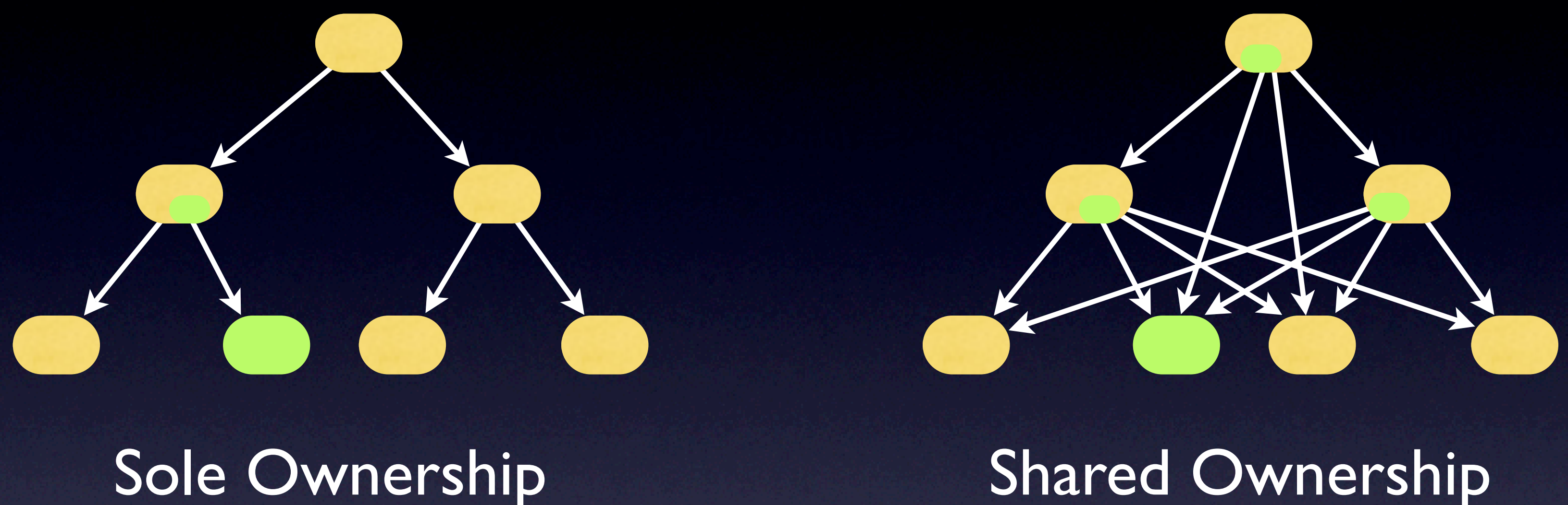
Sole Ownership



Shared Ownership

- Shared ownership is only valuable when items are immutable (const) or when reference semantics is desired (changing one copy updates all copies).
- Cheap assignment and function return is no longer a sufficient reason for the more complex shared ownership model.

Shared Ownership vs Sole Ownership



- Shared ownership is only valuable when items are immutable (const) or when reference semantics is desired (changing one copy updates all copies).
- Cheap assignment and function return is no longer a sufficient reason for the more complex shared ownership model.

vector Move Assignment

(actual syntax)

```
template <class T, class A>
class vector {
public:                                      // handles:
    vector& operator=(const vector& v);   // lvalues
    vector& operator=(vector&& v);        // rvalues
};
```

- The “&&” is an rvalue reference.
  - It attracts non-const rvalues when overloaded with const v&.

What Is An Rvalue Reference

```
class A {};
A a;
A&& a_ref1 = A();
A&  a_ref2 = a;
```

- An rvalue reference is just a new type of reference spelled “&&” instead of “&”.

Why A New Reference Type?

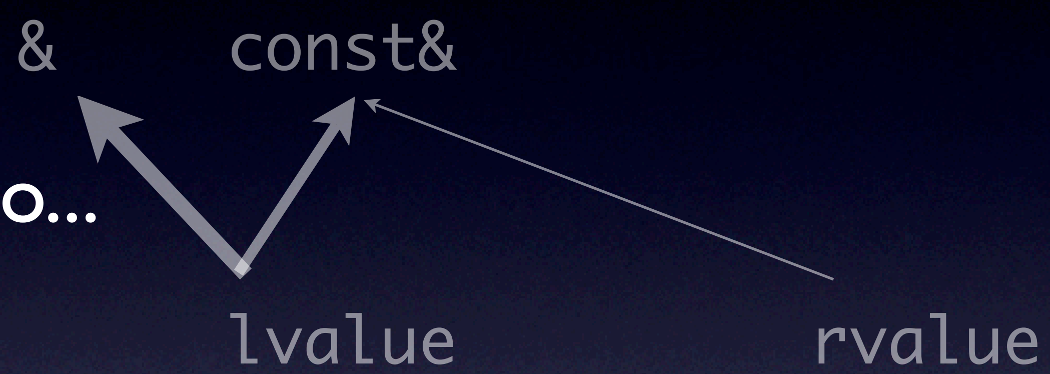
- Use of a new reference type, as opposed to a keyword which means “move” or “rvalue” is desirable because the rvalue reference solves more problems than just move semantics.
  - Move semantics
  - Perfect Forwarding
  - Useful rvalue streams
  - etc.

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.

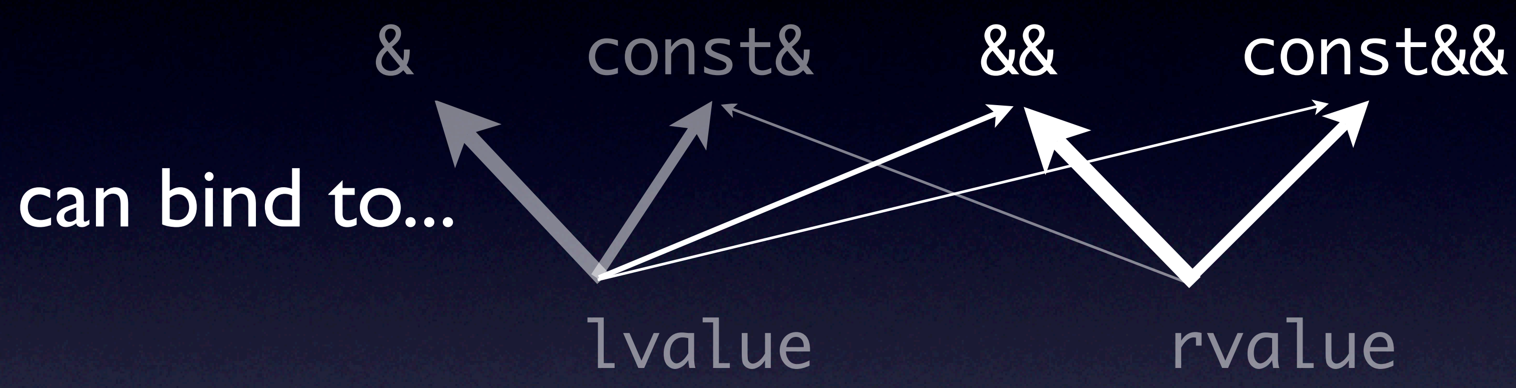
Overloading References

can bind to...



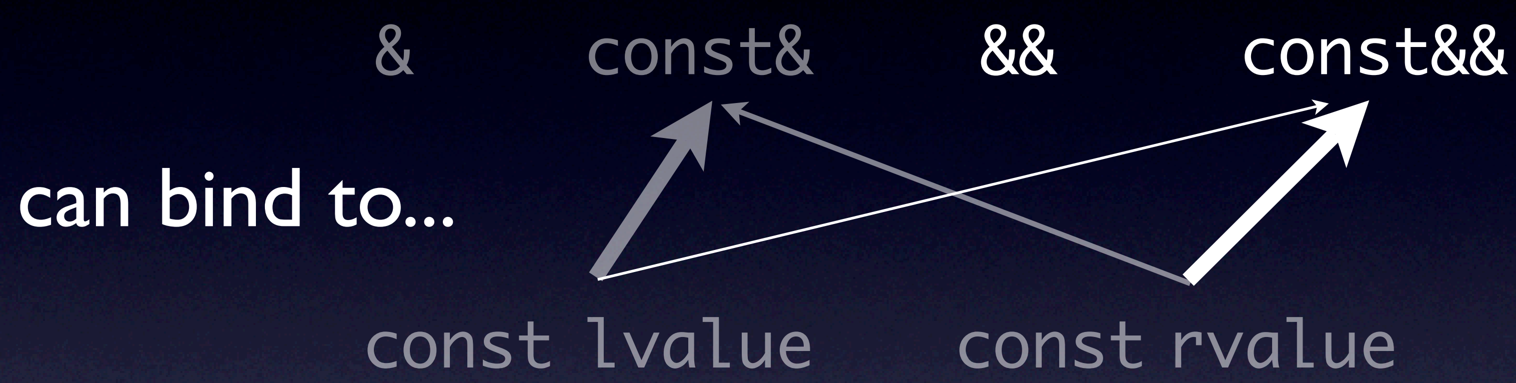
- lvalues can bind to any kind of reference (but prefer lvalue refereneces).
- rvalues can bind to rvalue references or a const-qualified lvalue reference.

Overloading References



- lvalues can bind to any kind of reference (but prefer lvalue refereneces).
- 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.

Overloading References

```
void f(const A&);  // 1
void f(A&&);       // 2
```

Overloading References

```
void f(const A&);  // 1
void f(A&&);       // 2
```

```
A a;
```

- `f(a);                    // 1`

Overloading References

```
void f(const A&);  // 1
void f(A&&);       // 2
```

```
A a;
const A ca;
```

- `f(a);                // 1`
- `f(ca);               // 1`

Overloading References

```
void f(const A&);  // 1
void f(A&&);       // 2
```

```
A a;
const A ca;
const A const_source();
```

- `f(a);                 // 1`
- `f(ca);                // 1`
- `f(const_source());    // 1`

Overloading References

```
void f(const A&);  // 1
void f(A&&);       // 2
```

```
A a;
const A ca;
const A const_source();
A source();
```

- `f(a);                  // 1`
- `f(ca);                 // 1`
- `f(const_source());     // 1`
- `f(source());           // 2`

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;
    }
};
```

- Today's copyable clone_ptr

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!*

The Key to Move Semantics

- Temporaries have no name, and can bind to only one reference.
- If you know a reference refers to a temporary, then you know it is safe to modify that temporary without the rest of your program knowing about it.
- Clients who use std::move are making a promise that you can pretend that value is a temporary.

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).

Writing Move Members

```
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;}
};
```

- When base classes and class data members are involved, delegate move logic with explicit use of std::move (just as a rule of thumb).

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

```
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 *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

```
unique_ptr<int> make_unique_ptr()
{
    unique_ptr<int> p(new int);
    *p = 2;
    return p;  // Ok, implicit cast to rvalue
} // unique_ptr(unique_ptr&&) called or elided
```

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

- Any handle-type class which refers to a unique, uncopyable resource, is a good candidate for move-only.

Containers Of Move-Only Types

```
vector<uniue_ptr<int>> vp;
vp.push_back(unique_ptr<int>(new int(0)));
*vp.front() = 1;
vector<ofstream> vf;
vf.push_back(ofstream("first file"));
vf[0] << "some data";
```

- Containers may now contain move-only types.
- Elements must be moved in or out, not copied.
- The containers are not copyable, just move-only.
- vector<unique_ptr<T>> has the same overhead as vector<T*>.

Unique Ownership Strategies Example

- Consider:

```
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:

```
Matrix m1 = m2 + m3 + m4 + m5 + m6;
```

- The C++03 solution:

```
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

```
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.

Why COW Doesn't Help Here

- COW fails to optimize when the reason you're copying is because you want to modify the copy.

```
Matrix
operator+(const Matrix& x, const Matrix& y)
{
    Matrix r(x);  // Lazy copy
    r += y;       // Copy forced here
    return r;
}
```

Copy On Write - Garbage Collected

```
class Matrix
{
    MatrixImp* imp_;  // Let it leak!
public:
    Matrix(int row, int col);
    Matrix(const Matrix& m);
    Matrix& operator+=(const Matrix& m);
    ...
};
```

- Garbage collecting libraries exist for C++.

Copy On Write - Garbage Collected

- 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
```

- 4 allocations.
- 3 allocations likely need be collected later.

Copy On Write - Garbage Collected

- Garbage collection doesn’t correct COW’s weakness in the copy/modify pattern.
- Deallocations are not “saved”, they are merely delayed until some time later.

Alternative GC-Based Solution

- Instead of COW in the Matrix copy constructor, return a leaked Matrix by reference in operator+:

```
Matrix&
operator+(const Matrix& x, const Matrix& y)
{
    Matrix* r = new Matrix(x);
    *r += y;
    return *r;
}
```

Alternative GC-Based Solution

- Cost analysis

```
Matrix m1 = m2 + m3 + m4 + m5 + m6;
```

```
allocate 3 words for t1 in m2+m3
allocate row*col doubles for m2+m3 -> t1
allocate 3 words for t2 in t1+m4
allocate row*col doubles for t1+m4 -> t2
allocate 3 words for t3 in t2+m5
allocate row*col doubles for t2+m5 -> t3
allocate 3 words for t4 in t3+m6
allocate row*col doubles for t3+m6 -> t4
allocate row*col doubles for m1
```

- 9 allocations.

Alternative GC-Based Solution

- Sometimes when people know they have garbage collection, they tend to leak with abandon. Often this can degrade performance instead of improve it.
- Garbage collection doesn't mean "never having to worry about memory."

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

```
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

	Allocations	Deallocations	Outstanding
• C++03:	4	3	1
• COW (rc)	4	3	1
• COW (gc)	4	0	4
• GC-op+	9	0	9
• Rvalue-ref	1	0	1

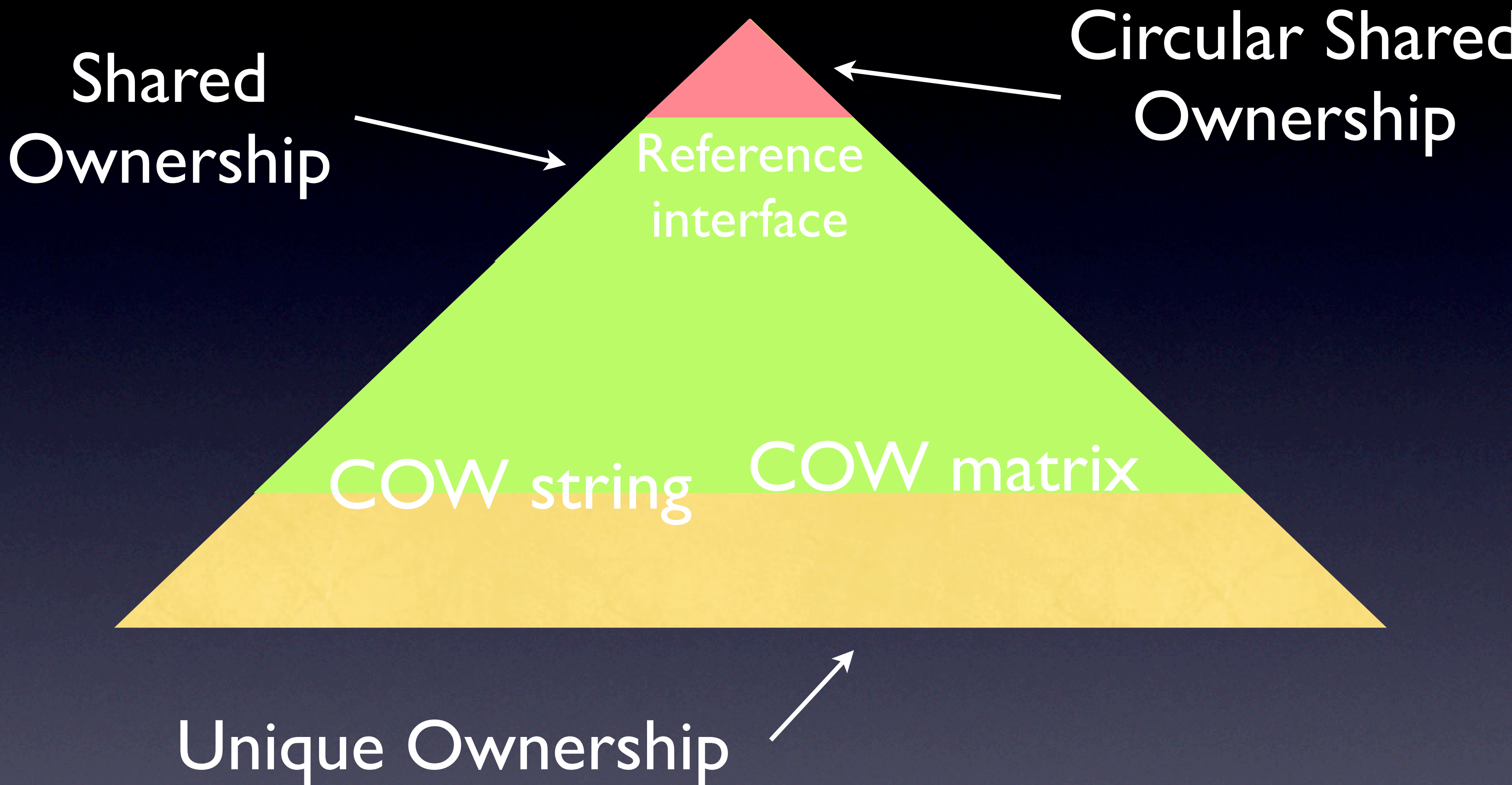
- Move semantics makes unique ownership both simple and practical.

Cost Summary for Matrix + Matrix

	Allocations	Deallocations	Outstanding
• C++03:	4	3	1
• COW (rc)	4	3	1
• COW (gc)	4	0	4
• GC-op+	9	0	9
• Rvalue-ref	1	0	1

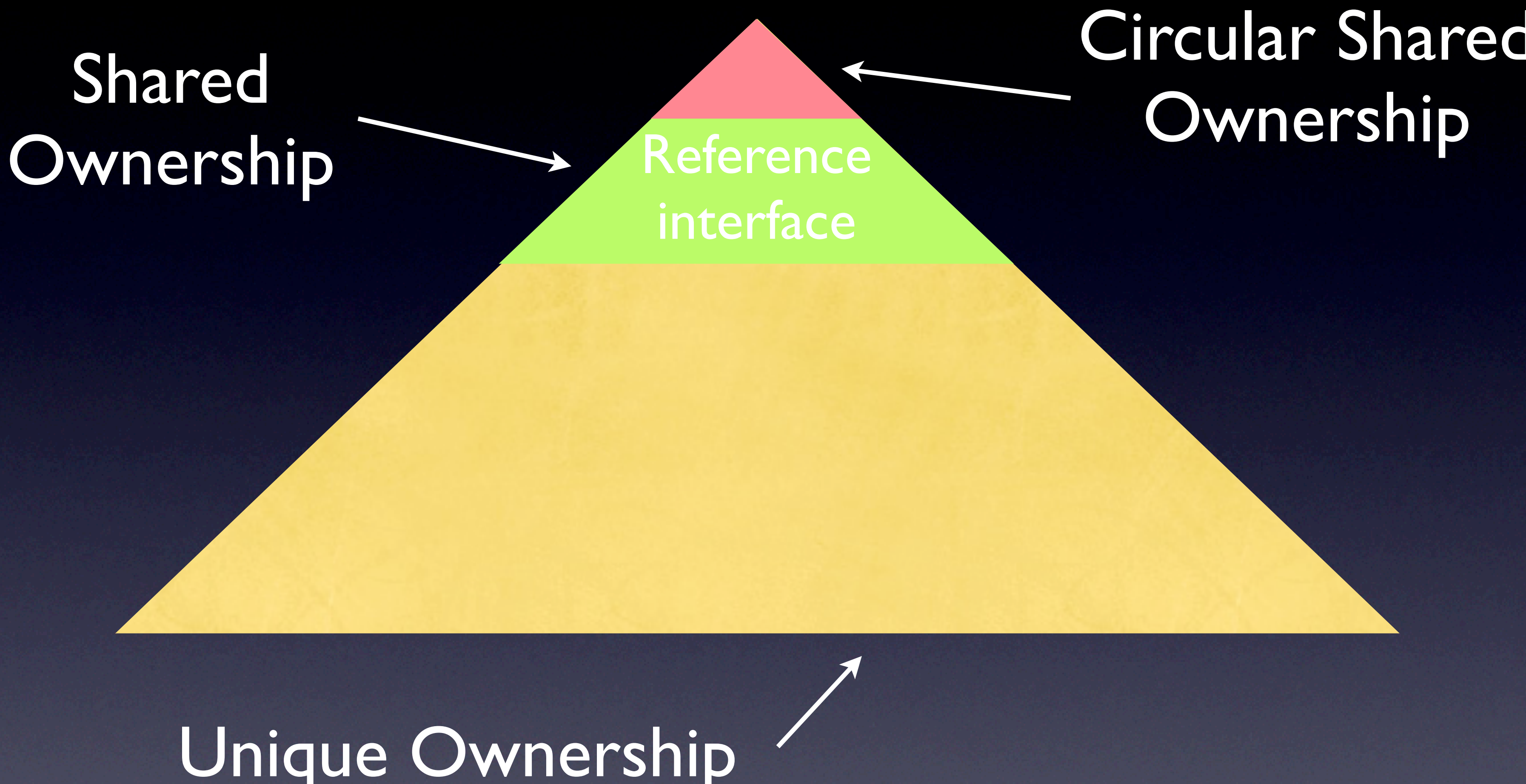
- Move semantics makes unique ownership both simple and practical.

Memory Ownership Strategies



- Rvalue reference greatly reduces the motivation for shared ownership.

Memory Ownership Strategies



- Rvalue reference greatly reduces the motivation for shared ownership.

Generic Algorithm Solution

```
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.

Generic Algorithm Solution

```
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.

Generic Containers

- I will be using vector (again) just as an example:
  - Move constructor and move assignment make container itself cheaply movable.
  - But what if the container's value_type is cheaply moveable?

Container Moving Interface

```
void push_back(T&& x);
iterator insert(iterator p, T&& x);
```

- In addition to a move constructor and move assignment operator, a container can *move single* objects into itself.

```
vector<string> v;
v.push_back(“first string”); //moved in
string s(“second string”);
v.push_back(s);              // copied in
v.push_back(std::move(s));   // moved in
```

Moving push_back Implementation

```
void push_back(T&& x) {
    if (size() < capacity()) {
        alloc().construct(end(), std::move(x));
        ...
    } else
      ...
}
```

- Use std::move to invoke x's move constructor instead of x's copy constructor.
- If the type does not have a move constructor, its copy constructor is used.

Moving push_back Implementation

- There is nothing magic about allocator::construct: It simply forwards to placement new, as if:

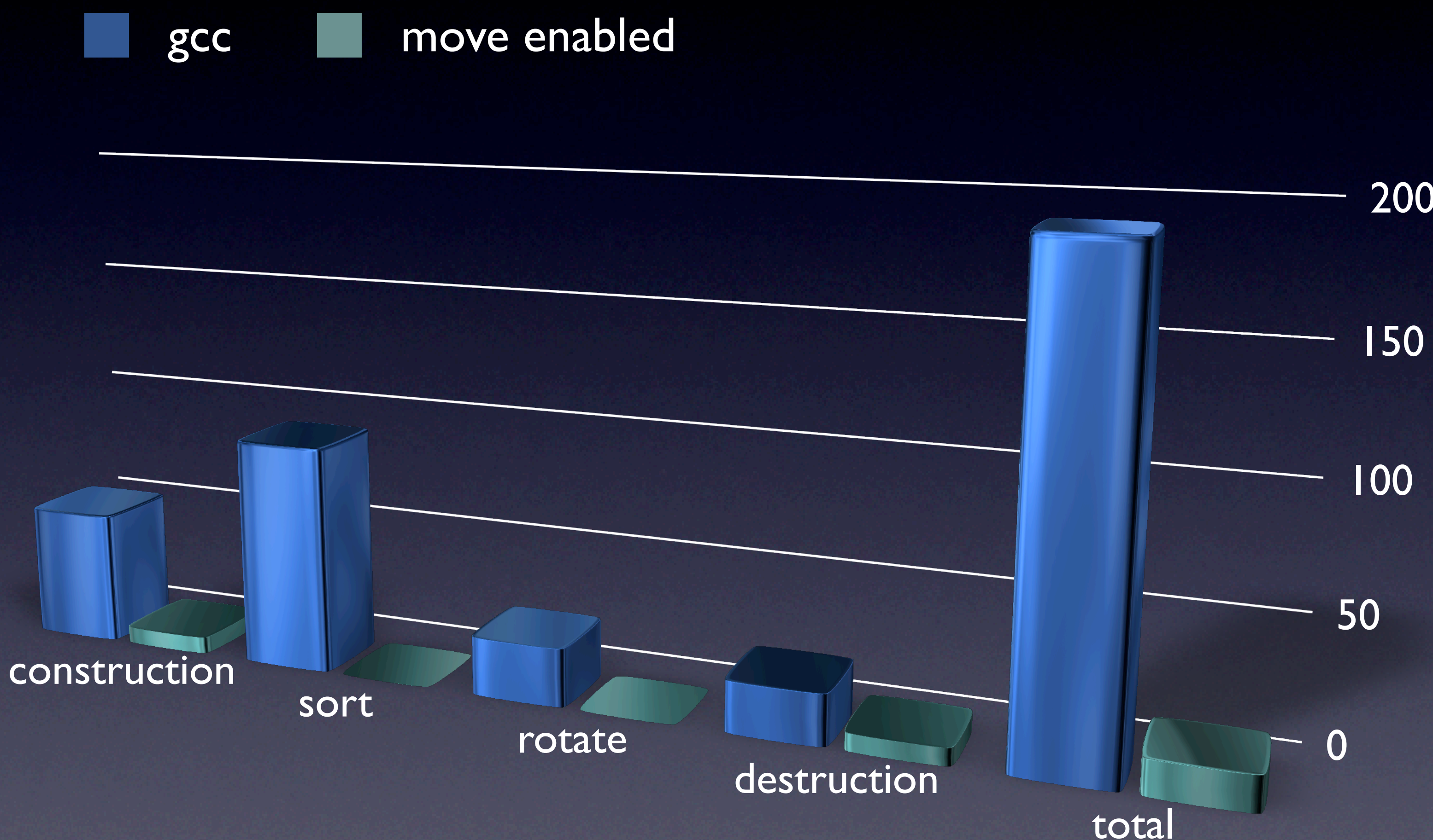
```
::new (p) T(std::move(x));
```

- If T has a move constructor, it is used. Else if T has a copy constructor, it is used. Else the placement new will fail to compile.
  - All std::move(x) does is turn x into an rvalue.

How Much Faster Can Move Be?

- A benchmark has been created which simply constructs, sorts, rotates and destructs a vector<set<int>>.
  - http://home.twcny.rr.com/hinnant/cpp_extensions/STL_benchmarks.html
- The benchmark doesn't explicitly contain std::move or rvalue reference.
  - Run it today and compare your best machine against my 1GHz laptop.

How Much Faster Can Move Be?



- The difference results from simply avoiding unnecessary trips to the heap.

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.

```
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&).

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);
```

- 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

```
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.

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!”));
```

So What Does std::move Actually Do?

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

- Accept an rvalue or lvalue and return it as an rvalue.
- The remove_reference defeats the unwanted special template deduction for lvalues.

And std::forward?

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

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

- forward<A&> returns A& (an lvalue)
  - References collapse A& && -> A&
- forward<A> returns A&& (an rvalue).

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.