Rvalue References, Forwarding, Move Semantics

A Generic Factory Function

```
template <class T, class A1>
std::auto_ptr<T> factory(A1 const& x1)
{
   return std::auto_ptr<T>( new T(x1) );
}

class widget
{
 public:
     widget(int size);
     ...
};
// yay!
std::auto_ptr<widget> w = factory<widget>(10);
```

First Problem

```
class traced : boost::noncopyable
{
 public:
    traced(std::ostream& log) : log(log)
    {
        log << "constructed" << std::endl;
    }
    ~traced()
    {
        log << "destroyed" << std::endl;
    }
    // ...
 private:
    ostream& log;
};
// boo!
std::auto_ptr<traced> m = factory<traced>(std::cout);
```


Fix: Add an Overload

```
template <class T, class A1>
std::auto_ptr<T> factory(A1& x1)
{
   return std::auto_ptr<T>( new T(x1) );
}

template <class T, class A1>
std::auto_ptr<T> factory(A1 const& x1)
{
   return std::auto_ptr<T>( new T(x1) );
}

// yay!
std::auto_ptr<traced> m = factory<traced>(std::cout);
std::auto_ptr<widget> w = factory<widget>(10);
```

First Problem

```
class traced : boost::noncopyable
{
 public:
    traced(std::ostream& log) : log(log)
    {
        log << "constructed" << std::endl;
    }
    ~traced()
    {
        log << "destroyed" << std::endl;
    }
    // ...
 private:
    ostream& log;
};
// boo!
std::auto_ptr<traced> m = factory<traced>(std::cout);
```


Handling Two Arguments

```
template <
  class T, class A1, class A2
>
std::auto_ptr<T>
factory(A1& x1, A2& x2)
{
   return std::auto_ptr<T>(
     new T(x1, x2) );
}

template <
  class T, class A1, class A2
>
std::auto_ptr<T>
factory(A1 const& x1, A2& x2)
{
   return std::auto_ptr<T>(
     new T(x1, x2) );
}
```

```
template <
  class T, class A1, class A2
>
std::auto_ptr<T>
factory(A1& x1, A2 const& x2)
{
   return std::auto_ptr<T>(
     new T(x1, x2) );
}

template <
  class T, class A1, class A2
>
std::auto_ptr<T>
factory(A1 const&x1, A2 const&x2)
{
   return std::auto_ptr<T>(
     new T(x1, x2) );
}
```

Handling *N* Arguments

Uh oh...

Handling N arguments requires $\underline{2^N}$ overloads!

N Overloads for *N* Args (Almost)

```
template <class T, class A1>
std::auto_ptr<T> factory(A1& x1)
{
   return std::auto_ptr<T>( new T(x1) );
}

// yay!
std::auto_ptr<traced> m = factory<traced>(std::cout);

// yay!
int const x = 10;
std::auto_ptr<widget> w = factory<widget>(x);

// boo!
std::auto_ptr<widget> w = factory<widget>(10);
```

What's The Problem?

- 8.5.3 paragraph 5 (paraphrasing):
  A non-const reference shall not be bound to an rvalue

- Quoting non-normative text
  [Example:
  double& rd2 = 2.0; // error: not an lvalue and reference not const
  ]

- There is a gap in the type system (expressiveness, not safety).

3.10 Lvalues and Rvalues

1. Every expression is either an lvalue or an rvalue.
2. An lvalue refers to an object or function. Some rvalue expressions – those of class or cv-qualified class type – also refer to objects.
3. [Note: some built-in operators and function calls yield lvalues. [Example: if `E` is an expression of pointer type, then `*E` is an lvalue expression referring to the object or function to which `E` points. As another example, the function

   ```
   int& f();
   ```

   yields an lvalue, so the call `f()` is an lvalue expression.]]
4. [Note: some built-in operators expect lvalue operands. [Example: built-in assignment operators all expect their left hand operands to be lvalues.]...]

14.3 [Note: a reference can be thought of as a name of an object.]

Lvalues and Rvalues (Summary)

- All expressions that are names or that are references are lvalues.
- Everything else is an rvalue

```
int x;
int f();
int& g();

x;    // L
10;   // R
f();  // R
g();  // L
```


Why This Rule? – Killer Example

B. Stroustrup, *The **D**esign **and** **E**volution of C++* (“D&E”), section 3.7

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

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

We Need an Über-Reference!

- The rvalue reference fills that role
- Spelled: X&&
- Binds to both lvalues and rvalues

```
template <class T, class A1, class A2>
std::auto_ptr<T> factory(A1&& x1, A2&& x2)
{
   return std::auto_ptr<T>(new T(x1, x2));
}
```

- This code isn't quite right yet (more later)

Move Semantics

Sometimes you don't need to copy.

– B. Stroustrup

Simple Examples

```
std::string to_string(int x)
{
   std::string x
      = boost::lexical_cast<std::string>(x);
   return x;
}

char first(std::string x)            // copy
{
   std::sort(x.begin(), x.end());
   return *x.begin();
}

std::string greet("hello, world!");
char y = first(greet);

char z = first(to_string(10));
```


Worse Examples

```
std::vector<std::string> explode(std::string const& x)
{
    std::vector<std::string> result;

    std::transform(
        x.begin(), x.end(), std::back_inserter(result),
        to_string);

    return result;
}

void insert_in_order(   // Precondition: v is sorted
    std::vector<std::string>& v, std::string const&s)
{
    v.insert(lower_bound(v.begin(), v.end(), s), s);
}
```

Making `std::string` Moveable

```
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 );
   {
   // move constructor
   string(string&& s)
     : data_(s.data), size_(s.size)
   { s.data = 0; s.size = 0; }

   // move assignment
   string& operator=(string&& s)
   { swap(s); return *this; }

 private:
    char* data; size_t size;
};
```


Making `std::vector` move-aware

```
// This is the usual insert (no moving)
iterator insert(iterator pos, value_type const& x)
{
    if (capacity() > size())
    {
        if (pos == end())
           construct(x, end());
        else
        {
           construct(*end() - 1, end());
           copy_backward(pos, end() - 1, pos + 1);
           *pos = x;
        }
    }
    else // ...reallocate and copy the buffer...
}
```

1. Handle the "x is an Rvalue" Case

```
// This is an overload
iterator insert(iterator pos, value_type&& x)
{
    if (capacity() > size())
    {
        if (pos == end())
           construct(x, end());
        else
        {
           construct(*end() - 1, end());
           copy_backward(pos, end() - 1, pos + 1);
           *pos = x;
        }
    }
    else // ...reallocate and copy the buffer...
}
```


2. Explicit Moving in `construct`

```
// The old way
template <class T>
void construct(T const& x, void* p)
{ new (p) T(x); }

// Overload for move
template <class T>
void construct(T&& x, void* p)
{ new (p) T( std::move(x) ); }

// "Cast to rvalue – I know it's safe to move"
template <class T>
typename remove_reference<T>::type&&
move(T&& t)
{
   return t;
}
```

3. Explicit Moving in `vector`

```
// Do this to both overloads
iterator insert(iterator pos, value_type&& x)
{
    if (capacity() > size())
    {
        if (pos == end())
           construct(x, end());
        else
        {
           construct(std::move(*end()-1), end());
           move_backward(pos, end() - 1, pos + 1);
           *pos = x;
        }
    }
    else // ...reallocate and move the buffer...
}
```

4/24/2005 copyright 2005 David Abrahams 21

`move_ptr` / `unique_ptr`

- Why don't we allow `std::auto_ptr` in standard containers?
- Why is `move_ptr` okay in standard containers?
- Why might this be better than using `shared_ptr`?

4/24/2005 copyright 2005 David Abrahams 22

Move is a Pure Optimization

- Making types movable is entirely optional
- Library can be transparently upgraded to use move
- No performance decrease for existing code
- Probably a performance boost, even if you don't make your types movable.
- Libraries use "move if available, else copy" strategy: no moveability detection required.
- A class with move assign but no move construct (or vice-versa) is just fine

Arbitrarily Huge Performance Boost

- Just make a sufficiently gnarly data structure:

```
std::map<
    std::set<std::string>,
    std::vector<int>
>
```

Moving and Forwarding

- Our last factory isn't perfect:

```
template <class T, class A1, class A2>
std::auto_ptr<T> factory(A1&& x1, A2&& x2)
{
   return std::auto_ptr<T>(new T(x1, x2));
}

struct Foo
{
   foo(std::string x, std::vector<int> y);
};

factory<Foo>(
   std::string(“MoveIt”), std::vector<int>(10, 0));
```


Perfect Forwarding

- Need a way to know and restore rvalueness of argument
- Rvalueness captured in A1 and A2

```
template <class T, class A1, class A2>
std::auto_ptr<T> factory(A1&& x1, A2&& x2)
{
   return std::auto_ptr<T>(
      new T(
         std::forward<A1>(x1),
         std::forward<A2>(x2)
      )
   );
}
```

Extended Reference-Collapsing Rules

- A& & → A& by CWG 106
- A& && → A& for perfect forwarding
- A&& & → A& for completeness
- A&& && → A&& for perfect forwarding

Thank you, Peter Dimov!

Definition of `std::forward`

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

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

Library-Only Move Solutions

- History:
  - `std::auto_ptr`
  - MOJO (Alexajdrescu)
  - Proposed Boost.Move (Abrahams)
- Close, but no cigar:
  - Writing movable classes is arcane and verbose (conversion operator, two copies of lvalue copy/assign)
  - Ugly: tricks on the type system to do something the compiler already knows about
  - All optimizations lost at forwarding boundaries (tr1::bind, factory, bind1st, bind2nd, vector::push_back)

Move and Inheritance

```
struct Base
{
    Base(Base&& b);
};

struct Derived
  : Base
{
    Derived(Derived&& d)
      : Base(std::move(d)) {}
};
```