Move Semantics, Rvalue References, and Perfect Forwarding

Scott Meyers, Ph.D.
Software Development Consultant

smeyers@aristeia.com
http://www.aristeia.com/

Voice: 503/638-6028
Fax: 503/974-1887

C++0x Warning

Some examples show C++0x features unrelated to move semantics.

I'm sorry about that.

But not that sorry :-)

© 2010 Scott Meyers, all rights reserved.
Abridgement Warning

A thorough treatment of this topic requires 3-4 hours.

We have 90 minutes.

Some details have been sacrificed :-)

Move Support

C++ sometimes performs unnecessary copying:

```cpp
typedef std::vector<T> TVec;
TVec createTVec(); // factory function
TVec vt;
...
vt = createTVec(); // copy return value object to vt,
// then destroy return value object
```

![Diagram](image.png)
Move Support

Moving values would be cheaper:

```cpp
TVec vt;
...
vt = createTVec(); // move data in return value object
// to vt, then destroy return value
// object
```

---

Move Support

Appending to a full vector causes much copying before the append:

```cpp
std::vector<T> vt;
...
vt.push_back(T object); // assume vt lacks
// unused capacity
```
Move Support

Again, moving would be more efficient:

```cpp
std::vector<T> vt;
...
vt.push_back(T object); // assume vt lacks // unused capacity
```

Other vector and deque operations could similarly benefit.

- insert, emplace, resize, erase, etc.

---

Move Support

Still another example:

```cpp
template<typename T> // straightforward std::swap impl.
void swap(T& a, T& b)
{
    T tmp(a); // copy a to tmp (⇒ 2 copies of a)
    a = b; // copy b to a (⇒ 2 copies of b)
    b = tmp; // copy tmp to b (⇒ 2 copies of tmp)
    // destroy tmp
}
```

---
Move Support

```cpp
template<typename T> // straightforward std::swap impl.
void swap(T& a, T& b) {
    T tmp(std::move(a)); // move a’s data to tmp
    a = std::move(b); // move b’s data to a
    b = std::move(tmp); // move tmp’s data to b
    // destroy (eviscerated) tmp
}
```

Move Support

Moving most important when:

- Object has data in separate memory (e.g., on heap).
- Copying is deep.

Moving copies only object memory.

- Copying copies object memory + separate memory.

Consider copying/moving A to B:

Moving never slower than copying, and often faster.
Performance Data

Consider these use cases again:

\[ vt = \text{createTVec}(); \quad \text{// return/assignment} \]
\[ vt.\text{push}\_\text{back}(T\ \text{object}); \quad \text{// push\_back} \]

Copy-vs-move performance differences notable:

<table>
<thead>
<tr>
<th></th>
<th>Copy</th>
<th>Move</th>
<th>Copy</th>
<th>Move</th>
<th>Copy</th>
<th>Move</th>
<th>Copy</th>
<th>Move</th>
<th>Copy</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Move Support

Lets C++ recognize move opportunities and take advantage of them.

- How recognize them?
- How take advantage of them?
Lvalues and Rvalues

Lvalues are generally things you can take the address of:
- Named objects.
- Lvalue references.
  - More on this term in a moment.

Rvalues are generally things you can’t take the address of.
- Typically unnamed temporary objects.

Examples:

```cpp
int x, *pInt;            // x, pInt, *pInt are lvalues
std::size_t f(const char str); // str is lvalue, f's return is rvalue
f("Hello");            // temp string created for call
                      // is rvalue
std::vector<int> vi;    // vi is lvalue
...                     // vi[5] is lvalue
```

- Recall that `vector<T>::operator[]` returns `T&`.

Moving and Lvalues

Value movement generally not safe when the source is an lvalue.
- The lvalue object continues to exist, may be referred to later:

```
TVec vt1;
...
TVec vt2(vt1); // author expects vt1 to be
               // copied to vt2, not moved!
...use vt1...  // value of vt1 here should be
               // same as above
```
Moving and Rvalues

Value movement is safe when the source is an rvalue.

- Temporaries go away at statement’s end.
- No way to tell if their value has been modified.

```cpp
TVec vt1;
vt1 = createTVec(); // rvalue source: move okay
auto vt2 { createTVec() }; // rvalue source: move okay
vt1 = vt2; // lvalue source: copy needed
auto vt3(vt2); // lvalue source: copy needed

std::size_t f(std::string str); // as before
f("Hello"); // rvalue (temp) source: move okay
std::string s("C++0x");
f(s); // lvalue source: copy needed
```

Rvalue References

C++0x introduces **rvalue references**.

- Syntax: `T&&`

“Normal” references now known as **lvalue references**.

Rvalue references behave similarly to lvalue references.

- Must be initialized, can’t be rebound, etc.

Rvalue references identify objects that may be moved from.
Reference Binding Rules

Important for overloading resolution.
As always:
- Lvalues may bind to lvalue references.
- Rvalues may bind to lvalue references to const.
In addition:
- Rvalues may bind to rvalue references to non-const.
- Lvalues may not bind to rvalue references.
  ➔ Otherwise lvalues could be accidentally modified.

Rvalue References

Examples:

```cpp
void f1(const TVec&); // takes const lvalue ref
TVec vt;
f1(vt);           // fine (as always)
f1(createTVec()); // fine (as always)

void f2(const TVec&); // #1: takes const lvalue ref
void f2(TVec&&);     // #2: takes non-const rvalue ref
f2(vt);             // lvalue ⇒ #1
f2(createTVec());   // both viable, non-const rvalue ⇒ #2

void f3(const TVec&); // #1: takes const rvalue ref
void f3(TVec&&);     // #2: takes non-const rvalue ref
f3(vt);             // error! lvalue
f3(createTVec());   // both viable, non-const rvalue ⇒ #2
```
Distinguishing Copying from Moving

Overloading exposes move-instead-of-copy opportunities:

```cpp
class Widget {
public:
  Widget(const Widget&); // copy constructor
  Widget(Widget&&); // move constructor
  Widget& operator=(const Widget&); // copy assignment op
  Widget& operator=(Widget&&); // move assignment op
  …
};

Widget createWidget(); // factory function
Widget w1;
Widget w2 = w1; // lvalue src ⇒ copy req’d
w2 = createWidget(); // rvalue src ⇒ move okay
w1 = w2; // lvalue src ⇒ copy req’d
```

Implementing Move Semantics

Move operations take source’s value, but leave source in valid state:

```cpp
class Widget {
public:
  Widget(Widget&& rhs) : pds(rhs.pds) // take source’s value
    { rhs.pds = nullptr; } // leave source in valid state
  Widget& operator=(Widget&& rhs)
  {
    delete pds; // get rid of current value
    pds = rhs.pds;
    rhs.pds = nullptr;
    return *this;
  }
  …
private:
  struct DataStructure;
  DataStructure *pds;
};
```

Easy for built-in types (e.g., pointers). Trickier for UDTs…
Implementing Move Semantics

Part of C++0x’s `string` type:

```cpp
string::string(const string&); // copy constructor
string::string(string&&); // move constructor
```

An incorrect move constructor:

```cpp
class Widget {
  private:
    std::string s;
  public:
    Widget(Widget&& rhs) // move constructor
    : s(rhs.s) // compiles, but copies!
    { … }
    ...
};
```

- `rhs.s` an lvalue, because it has a name.
- Lvalueness/rvalueness orthogonal to type!
- Ints can be lvalues or rvalues, and rvalue references can, too.
- `s` initialized by `string`’s copy constructor.

Implementing Move Semantics

Another example:

```cpp
class WidgetBase {
  public:
    WidgetBase(const WidgetBase&); // copy ctor
    WidgetBase(WidgetBase&&); // move ctor
    ...
};
class Widget: public WidgetBase {
  public:
    Widget(Widget&& rhs) // move ctor
    : WidgetBase(rhs) // copies!
    { … }
    ...
};
```

- `rhs` is an lvalue, because it has a name.
- Its declaration as `Widget&&` not relevant!
Explicit Move Requests

To request a move on an lvalue, use `std::move`:

```cpp
class WidgetBase { ... };  
class Widget: public WidgetBase {
public:
    Widget(Widget&& rhs) // move constructor
        : WidgetBase(std::move(rhs)), // request move
            s(std::move(rhs.s)) // request move
    { ... }

    Widget& operator=(Widget&& rhs) // move assignment
    {
        WidgetBase::operator=(std::move(rhs)); // request move
        s = std::move(rhs.s); // request move
        return *this;
    }
    ...
};
```

`std::move` turns lvalues into rvalues.

- The overloading rules do the rest.

---

Why move Rather Than Cast?

`std::move` uses implicit type deduction. Consider:

```cpp
template<typename It>
void someAlgorithm(It begin, It end)
{
    // permit move from *begin to temp, static_cast version
    auto temp1 = static_cast<typename std::iterator_traits<It>::value_type&&>(*begin);

    // same thing, C-style cast version
    auto temp2 = (typename std::iterator_traits<It>::value_type&&)*begin;

    // same thing, std::move version
    auto temp3 = std::move(*begin);
    ...
}
```

What would you rather type?
Implementing `std::move`

`std::move` is simple – in concept:

```cpp
template<typename T>
T&& move(MagicReferenceType obj) // return as an rvalue whatever
{                               // is passed in; must work with
  return obj;                   // both lvalue/rvalues
}
```

Between concept and implementation lie arcane language rules.
- Fundamentally, `std::move` simply casts to a `T&&`.

---

Gratuitous Animal Photo

Leaf-Cutter Ants

Move is an Optimization of Copy

Move requests for copyable types w/o move support yield copies:

```cpp
class Widget { // class w/o move support
class Gadget { // class with move support
public:
    Widget(const Widget&); // copy ctor
    Gadget(Gadget&& rhs) // move ctor
        : w(std::move(rhs.w)) // request to move w's value
        { … } 
private:
    Widget w; // lacks move support
};
```

rhs.w is copied to w:

- `std::move(rhs.w)` returns an rvalue of type `Widget`.
- That rvalue is passed to `Widget`'s copy constructor.

---

Move is an Optimization of Copy

If `Widget` adds move support:

```cpp
class Widget { // as before
public:
    Widget(const Widget&); // copy ctor
    Widget(Widget&&); // move ctor
};
class Gadget { // as before
public:
    Gadget(Gadget&& rhs) // move ctor
        : w(std::move(rhs.w)) // request to move w's value
        { … } 
private:
    Widget w; // lacks move support
};
```

rhs.w is now moved to w:

- `std::move(rhs.w)` still returns an rvalue of type `Widget`.
- That rvalue now passed to `Widget`'s move constructor.
  - Via normal overloading resolution.
Move is an Optimization of Copy

Implications:

- Giving classes move support can improve performance even for move-unaware code.
  - Copy requests for rvalues may silently become moves.
- Move requests safe for types w/o explicit move support.
  - Such types perform copies instead.
    - E.g., all built-in types.

In short:

- Give classes move support.
- Use `std::move` for lvalues that may safely be moved from.

Implicitly-Generated Move Operations

Move constructor and move `operator=` are “special:”

- Generated by compilers under appropriate conditions.

Conditions:

- All data members and base classes are movable.
  - Implicit move operations move everything.
  - Most types qualify:
    - All built-in types (move ≡ copy).
    - Most standard library types (e.g., all containers).
- Generated operations likely to maintain class invariants.
  - No user-declared copy or move operations.
  - Custom semantics for any ⇒ default semantics inappropriate.
  - Move is an optimization of copy.
  - No user-declared destructor.
  - Often indicates presence of implicit class invariant.
Beyond Move Construction/Assignment

Move support useful for other functions, e.g., setters:

```cpp
class Widget {
public:
 ...
void setName(const std::string& newName)
{ name = newName; } // copy param
void setName(std::string&& newName)
{ name = std::move(newName); } // move param
void setCoords(const std::vector<int>& newCoords)
{ coordinates = newCoords; } // copy param
void setCoords(std::vector<int>&& newCoords)
{ coordinates = std::move(newCoords); } // move param
...
private:
 std::string name;
 std::vector<int> coordinates;
};
```

Construction and Perfect Forwarding

Constructors often copy parameters to data members:

```cpp
class Widget {
public:
 Widget(const std::string& n, const std::vector<int>& c)
 : name(n), // copy n to name
  coordinates(c) // copy c to coordinates
 {}  
...
private:
 std::string name;
 std::vector<int> coordinates;
};
```
Construction and Perfect Forwarding

Moves for rvalue arguments would be preferable:

```cpp
std::string lookupName(int id);
int widgetID;
...
std::vector<int> tempVec;  // used only for Widget ctor
...
Widget w(lookupName(widgetID), // rvalues args, but Widget
        std::move(tempVec));  // ctor copies to members
```

Overloading `Widget` ctor for lvalue/rvalue combos ⇒ 4 functions.

- Generally, `n` parameters requires `2^n` overloads.
  - Impractical for large `n`.
  - Boring/repetitive/error-prone for smaller `n`.

Goal: one function that “does the right thing:”

- Copies lvalue args, moves rvalue args.

Solution is a **perfect forwarding** ctor:

```cpp
class Widget {
public:
    template<typename T1, typename T2>
    Widget(T1&& n, T2&& c)
        : name(std::forward<T1>(n)),  // forward n to string ctor
          coordinates(std::forward<T2>(c))  // forward c to vector ctor
    {}  
...
private:
    std::string name;
    std::vector<int> coordinates;
};
```

Scot Meyers, Software Development Consultant © 2010 Scot Meyers, all rights reserved.
Construction and Perfect Forwarding

Once again:

- A templated ctor forwarding $T\&\&$ params to members:

```cpp
class Widget {
public:
    template<typename T1, typename T2>
    Widget(T1&& n, T2&& c)
    : name(std::forward<T1>(n)), // forward n to string ctor
      coordinates(std::forward<T2>(c)) // forward c to vector ctor
    {};

private:
    std::string name;
    std::vector<int> coordinates;
};
```

Effect:

- Lvalue arg passed to $n \Rightarrow$ std::string ctor receives lvalue.
- Rvalue arg passed to $n \Rightarrow$ std::string ctor receives rvalue.
- Similarly for $c$ and std::vector ctor.

Perfect Forwarding Beyond Construction

Useful for more than just construction, e.g., for setters:

```cpp
class Widget { // revised
public: // example
    ... // example
    template<typename T>
    void setName(T&& newName) // forward
    { name = std::forward<T>(newName); } // newName

    template<typename T>
    void setCoords(T&& newCoords) // forward
    { coordinates = std::forward<T>(newCoords); } // newCoords
    ...

private:
    std::string name;
    std::vector<int> coordinates;
};
```
Perfect Forwarding Beyond Construction

Despite `T&&` parameter, code fully type-safe:

- Type compatibility verified upon instantiation.
  - E.g., only `std::string`-compatible types valid in `setName`.

More flexible than a typed parameter.

- Accepts/forwards all compatible parameter types.
  - E.g., `std::string`, `char*`, `const char*` for `setName`.

---

Perfect Forwarding Beyond Construction

Flexibility can be removed via `static_assert`:

```cpp
template<typename T>
void setName(T&& newName)
{
    static_assert(std::is_same<typename std::decay<T>::type,
                                std::string
                                >::value,
                  "T must be a [const] std::string"
    );
    name = std::forward<T>(newName);
}
```
Perfect Forwarding

- Applicable only to function templates.
- Preserves arguments' lvalueness/rvalueness/constness when forwarding them to other functions.
- Implemented via std::forward.

Further Information

  - Details somewhat outdated per March 2009 rule changes.
  - Good explanations of std::move/std::forward implementations.
- “GCC C++0x Features Exploration,” Dean Michael Berris, C++ Soup!, 15 March 2009.
- “Howard’s STL / Move Semantics Benchmark,” Howard Hinnant, C++Next, 13 October 2010.
  - Move-based speedup for std::vector<std::set<int>> w/gcc 4.0.1.
  - Reader comments give data for newer gccs, other compilers.
Further Information

- “Your Next Assignment…,” Dave Abrahams, C++Next, 28 September 2009.
  - Correctness and performance issues for move operator=.
  - Exception safety issues for move operations.
  - Describes rules governing implicit move operations.
- “Class invariants and implicit move constructors (C++0x),” comp.lang.c++, thread initiated 14 August 2010.

Further Information

  - Popularized the idea of move semantics.
  - Discusses perfect forwarding.
  - Arguments that can’t be perfectly forwarded.
Licensing Information

Scott Meyers licenses materials for this and other training courses for commercial or personal use. Details:

- **Commercial use**: [http://aristeia.com/Licensing/licensing.html](http://aristeia.com/Licensing/licensing.html)
- **Personal use**: [http://aristeia.com/Licensing/personalUse.html](http://aristeia.com/Licensing/personalUse.html)

Courses currently available for personal use include:

About Scott Meyers

Scott is a trainer and consultant on the design and implementation of software systems, typically in C++. His web site, [http://www.aristeia.com/](http://www.aristeia.com/) provides information on:

- Training and consulting services
- Books, articles, other publications
- Upcoming presentations
- Professional activities blog