

Move Semantics in C++0x

Scott Meyers, Ph.D.
 Software Development Consultant

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

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

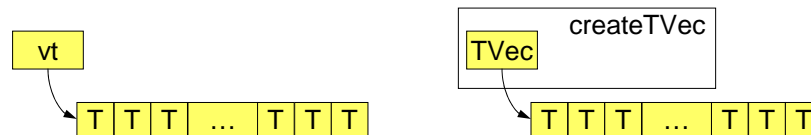
Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

© 2010 Scott Meyers, all rights reserved.
 Last Revised: 5/3/10

Copying vs. Moving

C++ sometimes performs unnecessary copying:

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



Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

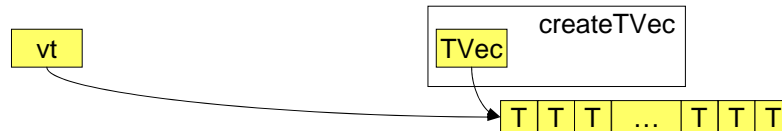
© 2010 Scott Meyers, all rights reserved.
 Slide 2

Copying vs. Moving

Moving values would be cheaper:

```

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

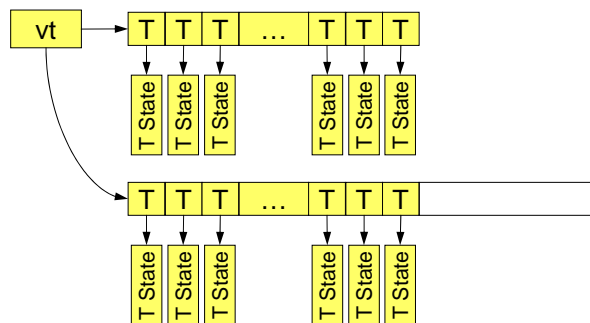


Copying vs. Moving

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

```

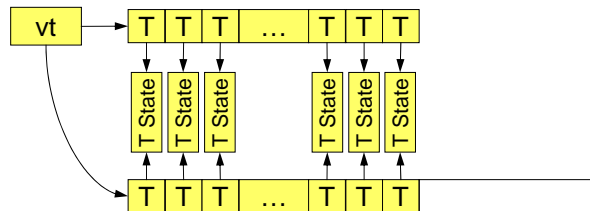
std::vector<T> vt;
...
vt.push_back(some T object); // assume vt lacks
                             // unused capacity
    
```



Copying vs. Moving

Again, moving would be more efficient:

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



Other vector and deque operations could similarly benefit.

- insert, resize, erase, etc.

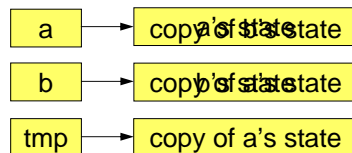
Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

© 2010 Scott Meyers, all rights reserved.
 Slide 5

Copying vs. Moving

Still another example:

```
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
```



Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

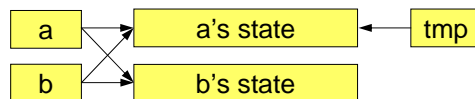
© 2010 Scott Meyers, all rights reserved.
 Slide 6

Copying vs. Moving

```

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

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:

```
int x, *pInt;           // x, pInt, *pInt are lvalues
std::size_t f(std::string 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] = 0;            // 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 "pilfered."

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.

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 “pilfered.”

Rvalue References

Examples:

```
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
```

Rvalue References and const

C++ remains const-correct:

- const lvalues/rvalues bind only to references-to-const.

But rvalue-references-to-const are essentially useless.

- Rvalue references designed for two specific problems:
 - ➔ Move semantics
 - ➔ Perfect forwarding
- C++0x language rules carefully crafted for these needs.
 - ➔ rvalue-refs-to-const not considered in these rules.
- const T&&s are legal, but not designed to be useful.
 - ➔ Doesn't mean uses won't be discovered :-)

Rvalue References and const

Implications:

- Don't declare const T&& parameters.
 - ➔ You wouldn't be able to move from them, anyway.
 - ➔ Hence this (from a prior slide) makes no sense:

```
void f3(const TVec&&);           // legal, rarely reasonable
```

- Avoid creating const rvalues.
 - ➔ They can't bind to T&& parameters.
 - ➔ E.g., avoid const function return types:
 - ◆ This is a change from C++98.

```
class Rational { ... };
```

```
const Rational operator+( const Rational&, // legal, but
                          const Rational&); // poor design
```

```
Rational operator+(const Rational&, // good design
                  const Rational&);
```

Distinguishing Copying from Moving

Overloading exposes move-instead-of-copy opportunities:

```
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:

```
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;     // take source's value
        rhs.pds = nullptr; // leave source in valid state
        return *this;
    }
    ...
private:
    struct DataStructure;
    DataStructure *pds;     // assume *pds is on heap
};
```

Easy for built-in types (e.g., pointers). Trickier for UDTs...

Implementing Move Semantics

Widget's move operator= fails given move-to-self:

```
Widget w;
w = std::move(w);           // undefined behavior!
```

It may be harder to recognize, of course:

```
Widget *pw1, *pw2;
...
*pw1 = std::move(*pw2);    // undefined if pw1 == pw2
```

C++0x is likely to condone this.

- In contrast to copy operator=.

A fix is simple, if you are inclined to implment it:

```
Widget& Widget::operator=(Widget&& rhs)
{
    if (this == &rhs) return *this;
    ...
}
```

Implementing Move Semantics

Part of C++0x's string type:

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

An incorrect move constructor:

```
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.
 - ➔ s initialized by string's *copy* constructor.

Implementing Move Semantics

Another example:

```
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&&` is not relevant!

Explicit Move Requests

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

```
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:

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

    // same thing, C-style cast version
    auto temp2 = (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:

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

Between concept and implementation lie arcane language rules.

Reference Collapsing in Templates

In C++98, given

```
template<typename T>
void f(T& param);

int x;

f<int&>(x);           // T is int&
```

f is initially instantiated as

```
void f(int& & param); // reference to reference
```

C++98's reference-collapsing rule says

- $T\& \& \Rightarrow T\&$

so f's instantiation is actually:

```
void f(int& param); // after reference collapsing
```

Reference Collapsing in Templates

C++0x's rules take rvalue references into account:

- $T\& \& \Rightarrow T\&$ // from C++98
- $T\&\& \& \Rightarrow T\&$ // new for C++0x
- $T\& \&\& \Rightarrow T\&$ // new for C++0x
- $T\&\& \&\& \Rightarrow T\&\&$ // new for C++0x

Summary:

- Reference collapsing involving a $\&$ is always $T\&$.
- Reference collapsing involving only $\&\&$ is $T\&\&$.

std::move's Return Type

To guarantee an rvalue return type, std::move does this:

```
template<typename T>
typename std::remove_reference<T>::type&&
move(MagicReferenceType obj)
{
    return obj;
}
```

- Recall that a T& return type would be an lvalue!

Hence:

```
int x;
std::move<int&>(x); // calls remove_reference<int&>::type&& std::move(/*...*/)
// => int&& std::move(/*...*/)
```

- Without remove_reference, move<int&> would return int&.

std::move's Parameter Type

It must be a non-const reference, because we want to move its value.

An lvalue reference doesn't work.

- Rvalues can't bind to them:

```
TVec createTVec();           // as before
TVec&& std::move(TVec& obj); // possible move
                             // instantiation
std::move(createTVec());    // error!
```

std::move's Parameter Type

An rvalue reference doesn't, either.

- Lvalues can't bind to them.

```
TVec&& std::move(TVec&& obj);           // possible move
                                        // instantiation
```

```
TVec vt;
```

```
std::move(vt);                          // error!
```

What std::move needs:

- For lvalue arguments, a parameter type of T&.
- For rvalue arguments, a parameter type of T&&.

Why Not Just Overload?

Overloading could solve the problem:

```
template<typename T>
typename std::remove_reference<T>::type&& move(T& lvalue)
{ return static_cast<T&&>(lvalue); }
```

```
template<typename T>
typename std::remove_reference<T>::type&& move(T&& rvalue)
{ return static_cast<T&&>(rvalue); }
```

But the perfect forwarding problem would remain:

- How forward n arguments to another function?
 - ➔ We'd need 2^n overloads!

Rvalue references aimed at both std::move and perfect forwarding.

T&& Parameter Deduction in Templates

Given

```
template<typename T>
void f(T&& param);           // note non-const rvalue reference
```

T's deduced type depends on what's passed to param:

- **Lvalue** ⇒ T is an lvalue reference (T&)
- **Rvalue** ⇒ T is a non-reference (T)

In conjunction with reference collapsing:

```
int x;
f(x);                       // lvalue: generates f<int&>(int& &&),
                           // calls f<int&>(int&)

f(10);                      // rvalue: generates/calls f<int>(int&&)

TVec vt;
f(vt);                      // lvalue: generates f<TVec&>(TVec& &&),
                           // calls f<TVec&>(TVec&)

f(createTVec());           // rvalue: generates/calls f<TVec>(TVec&&)
```

Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

© 2010 Scott Meyers, all rights reserved.

Slide 31

Implementing std::move

std::move's parameter is thus T&&:

```
template<typename T>
typename std::remove_reference<T>::type&&
move(T&& obj)
{
    return obj;
}
```

This is almost correct. Problem:

- obj is an lvalue. (It has a name.)
- move's return type is an rvalue reference.
- Lvalues can't bind to rvalue references.

Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

© 2010 Scott Meyers, all rights reserved.

Slide 32

Implementing std::move

A cast eliminates the problem:

```
template<typename T>
typename std::remove_reference<T>::type&&
move(T&& obj)
{
    using Return Type =
        typename std::remove_reference<T>::type&&;
    return static_cast<Return Type>(obj);
}
```

This is a correct implementation.

T&& Parameters in Templates

Note that function templates with a T&& parameter need not generate functions taking a T&& parameter!

```
template<typename T>
void f(T&& param);           // as before

int x;
f(x);                       // still calls f<int>(int&),
                           // i.e., f(int&)

f(10);                      // still calls f<int>(int&&),
                           // i.e., f(int&&)

TVec vt;
f(vt);                      // still calls f<TVec>(TVec&),
                           // i.e., f(TVec&)

f(createTVec());           // still calls f<TVec>(TVec&&),
                           // i.e., f(TVec&&)
```

T&& Parameters in Templates

T&&-taking function templates should be read as “takes anything:”

```
template<typename T>
void f(T&& param);           // takes anything: lvalue or rvalue,
                           // const or non-const
```

- Lvalues can't bind to rvalue references, but `param` may bind to an lvalue.
 - ➔ After instantiation, `param`'s type may be `T&`, not `T&&`.
- Important for perfect forwarding (described shortly).

T&& as a “takes anything” parameter applies only to templates!

- For functions, a `&&` parameter binds only to non-const rvalues:


```
void f(Widget&& param);     // takes only non-const rvalues
```

Move is an Optimization of Copy

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

```
class Widget {                // class w/o move support
public:
  Widget(const Widget&);      // copy ctor
};

class Gadget {                // class with move support
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 *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:

```
class Widget {
public:
    Widget(const Widget&);           // copy ctor
    Widget(Widget&&);               // move ctor
};

class Gadget {                     // as before
public:
    Gadget(Gadget&& rhs)
    : w(std::move(rhs.w)) { ... }  // as before
private:
    Widget w;
};
```

`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.

Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

© 2010 Scott Meyers, all rights reserved.
 Slide 37

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.

Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

© 2010 Scott Meyers, all rights reserved.
 Slide 38

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).
- Corresponding copy operation isn't user-declared.
 - ➔ User-declared copy semantics ⇒ user-declared move semantics.
 - ◆ Move is an optimization of copy.

Implicitly-Generated Move Operations

Examples:

```
class Widget1 { // copyable & movable type
private:
    std::u16string name; // copyable/movable type
    long long value; // copyable/movable type
public:
    explicit Widget1(std::u16string n);
}; // implicit copy/move ctor;
// implicit copy/move operator=

class Widget2 { // copy constructible but not
private: // move constructible type;
    std::u16string name; // both copy/move op= supported
    long long value;
public:
    explicit Widget2(std::u16string n);
    Widget2(const Widget2& rhs); // user-declared copy ctor
}; // => no implicit move ctor;
// implicit copy/move operator=
```

Custom Moving ⇒ Custom Copying

Declaring a move operation prevents generation of the corresponding copy operation.

- ➔ User-declared move semantics ⇒ user-declared copy semantics.
- ◆ Move is an optimization of copy.

```
class Widget3 {                                // movable type; not copyable
private:
    std::u16string name;
    long long value;
public:
    explicit Widget3(std::u16string n);
    Widget3(Widget3&& rhs);                    // user-declared move ctor
                                              // ⇒ no implicit copy ctor;
    Widget3&                                  // user-declared move op=
    operator=(Widget3&& rhs);                 // ⇒ no implicit copy op=
};
```

Move Operations

- May exist even if copy operations don't.
 - ➔ E.g., `thread` and `unique_ptr` moveable, but not copyable.
 - ◆ "Move-only types"
- Types should provide when moving cheaper than copying.
 - ➔ Libraries use moves whenever possible (e.g., STL, Boost, etc.).

Further Information

Draft C++0x:

- [Programming Languages – C++](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf), Pete Becker (Ed.), 2010-03-26, <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf>.

C++0x in General:

- [“C++0x,” Wikipedia](#).
- [C++0x - the next ISO C++ standard](http://www.research.att.com/~bs/C++0xFAQ.html), Bjarne Stroustrup, <http://www.research.att.com/~bs/C++0xFAQ.html>.
- [“C++0X: The New Face of Standard C++,”](http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=216) Danny Kalev, [informIT.com](http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=216), <http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=216>.
 ➔ Click “Guide Contents,” scroll to C++0X section, select topic.

Further Information

C++0x in General:

- [“C++0X with Scott Meyers,”](#) *Software Engineering Radio* (Audio), 5 April 2010.
- [“An Overview of the Coming C++ \(C++0x\) Standard,”](#) Matt Austern and Lawrence Crowl, *Google Tech Talk* (Video), 31 October 2008.
- [“Overview: C++ Gets an Overhaul,”](#) Danny Kalev, *DevX.com*, 18 August 2008.
- [“C++0x language feature checklist,”](http://www.codeguru.com/forum/showthread.php?t=4668939) VC10 Slow Chat, *CodeGuru.com*, December 2008, <http://www.codeguru.com/forum/showthread.php?t=4668939>.
 ➔ Discusses C++0x support in VC10.

Further Information

C++0x in General:

- [“Standard Library Changes in C++0x,”](#) *Van’s House*, 12 January 2009, <http://blogs.msdn.com/xiangfan/archive/2009/01/12/standard-library-changes-in-c-0x.aspx>.
- [“The State of the Language: An Interview with Bjarne Stroustrup,”](#) Danny Kalev, *DevX.com*, 15 August 2008.
- [C++0xCompilerSupport](#), *Apache C++ Standard Library Wiki*, <http://wiki.apache.org/stdcxx/C++0xCompilerSupport>.
 - ➔ Summarizes C++0x feature availability for various compilers.
- [“C++0x: Ausblick auf den neuen C++-Standard,”](#) Bernhard Merkle, *heise Developer*, 11 November 2008.

Further Information

Rvalue references, move semantics, perfect forwarding:

- [“A Brief Introduction to Rvalue References,”](#) Howard E. Hinnant *et al.*, *The C++ Source*, 10 March 2008.
 - ➔ Details somewhat outdated per March 2009 rule changes.
- [C++ Rvalue References Explained](#), Thomas Becker, June 2009, http://thbecker.net/articles/rvalue_references/section_01.html.
 - ➔ Good explanations of `std::move`/`std::forward` implementations.
- [“Rvalue References: C++0x Features in VC10, Part 2,”](#) Stephan T. Lavavej, *Visual C++ Team Blog*, 3 February 2009.
- [“GCC C++0x Features Exploration,”](#) Dean Michael Berris, *C++ Soup!*, 15 March 2009.

Further Information

Rvalue references, move semantics, perfect forwarding:

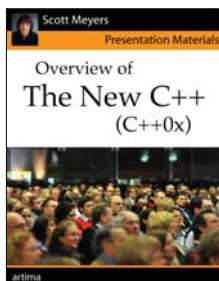
- [“Making Your Next Move,”](#) Dave Abrahams, *C++Next*, 17 September 2009.
- [“Your Next Assignment...,”](#) Dave Abrahams, *C++Next*, 28 September 2009.
 - ➔ Correctness and performance issues for move operator=s.
- [“Exceptionally Moving!,”](#) Dave Abrahams, *C++Next*, 5 Oct. 2009.
 - ➔ Exception safety issues for move operations.
- [“Onward, Forward!,”](#) Dave Abrahams, *C++Next*, 7 Dec. 2009.
 - ➔ Discusses perfect forwarding.
- [“Perfect Forwarding Failure Cases,”](#) comp.std.c++ discussion initiated 16 January 2010, <http://tinyurl.com/ygvm8kc>.
 - ➔ 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>
- **Personal use:** <http://aristeia.com/Licensing/personalUse.html>

Available courses include:



Please Note

Scott Meyers offers consulting services on the design and implementation of software systems. For details, visit his web site:

<http://www.aristeia.com/>

Scott maintains a low-volume mailing list about his professional publications and activities. Read about this list at:

<http://www.aristeia.com/MailingList/>