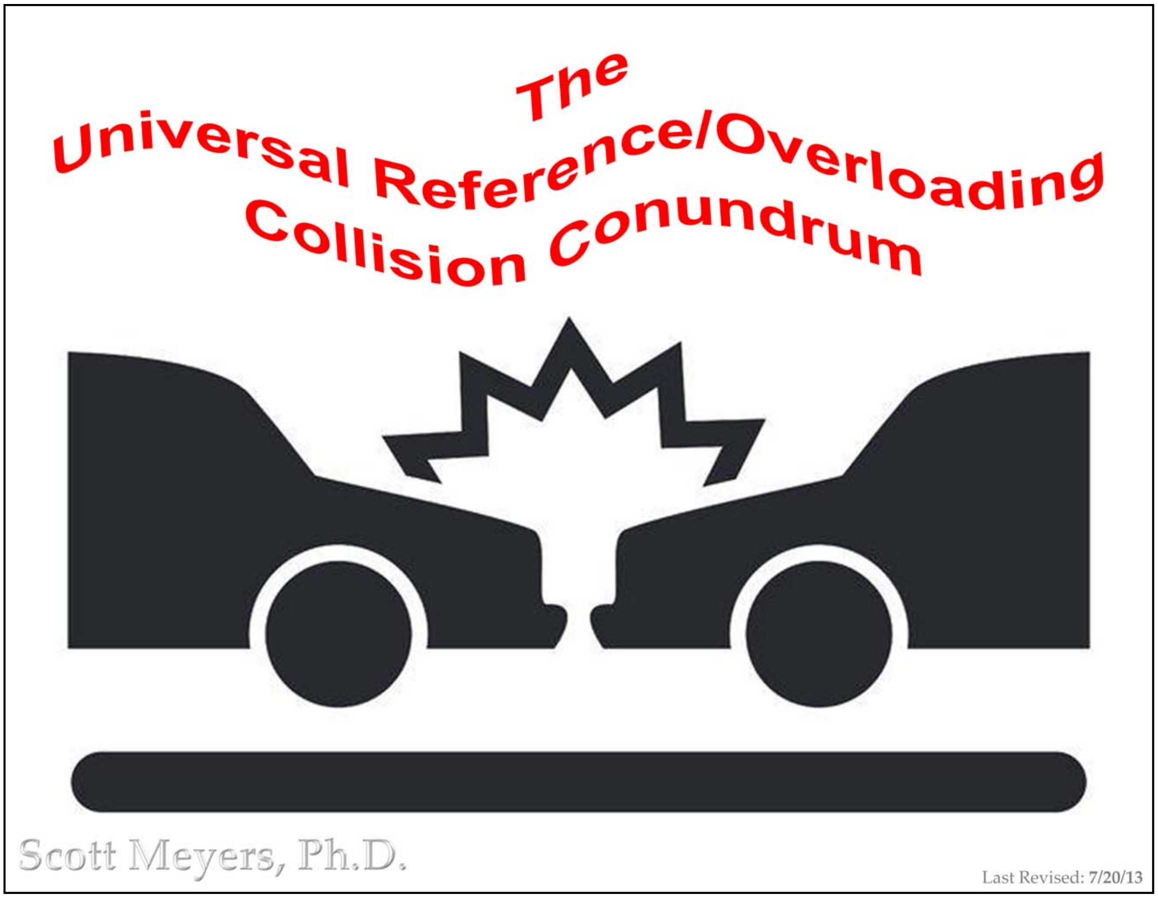


The materials shown here differ from those I used in my presentation at the Northwest C++ Users' Group. Compared to the materials I presented, these materials correct a variety of technical errors whose presence became apparent in the talk. To the best of my knowledge, these materials are correct. If you find what you believe to be errors, please report them to me: smeyers@aristeia.com.

Image credit: http://www.123rf.com/photo_12124395_vector-illustration-of-single-isolated-car-crash-icon.html.

std::move and std::forward

These functions simply cast:

- `std::move(`*`expr`*`)` casts *`expr`* to an rvalue.
  ➨ **Nothing gets moved.**

`std::move`:
Unconditional cast

- Given

```
template<typename T>
void f(T&& param)
{
  …std::forward<T>(param)…
}
```

`std::forward`:
Conditional cast

`std::forward<T>(param)` casts `param` to an rvalue only if argument passed to `param` was an rvalue:

```
Widget w;

f(w);             // inside f, std::forward<T>(param) is noop

f(std::move(w));  // inside f, std::forward<T>(param) casts
                  // param to rvalue
```

➨**Nothing gets forwarded.**

Slide is animated.

Universal References (URefs)

Types of form `T&&` for some deduced type `T`:

- Function template type parameters.
- `auto` variables.

Crucial characteristic:

- **Lvalue initializer ⇒ `T&&` becomes `T&`.**
- **Rvalue initializer ⇒ `T&&` remains `T&&`.**

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

Widget w;
const Widget cw;

f(w);                    // param's type is Widget&
f(cw);                   // param's type is const Widget&

f(std::move(w));         // param's type is Widget&&
f(std::move(cw));        // param's type is const Widget&&
```

Technically rvalue references (RRefs) in reference-collapsing contexts.

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

Universal References

No type deduction ⇒ no universal references:

- Caller explicitly specifies type for template parameter:

```
f<Widget>(w);              // param's type is Widget&&
                           // (w is lvalue ⇒ code won't compile)
```

- Caller passes braced initializer list:

```
f({ 1, 2, 3 });            // error! no type deduced for
                           // "{ 1, 2, 3 }"
```

➡ With `auto`, type deduction would succeed:

```
auto x = { 1, 2, 3 };  // std::initializer_list<int>
```


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

Avoid Overloading on Universal References

Overloading + URefs almost always an error.

- Makes no sense: URefs handle (almost) *everything*.
  - Lvalues, rvalues, `const`s, non-`const`s, `volatile`s, non-`volatile`s, etc.
  - They're *universal* references!
- Counterintuitive behavior.
  - As we'll see.

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

LRefs and RRefs and Overloading

Completely reasonable:

```
class NormalClass {
public:
  void doWork(const Widget& param);  // handle lvalues and const
                                     // rvalues
  void doWork(Widget&& param);       // handle non-const rvalues
};

NormalClass nc;
Widget w;
const Widget cw;

nc.doWork(w);                    // doWork(const Widget&)
nc.doWork(std::move(w));         // doWork(Widget&&)

nc.doWork(cw);                   // doWork(const Widget&)
nc.doWork(std::move(cw));        // doWork(const Widget&)
```

`const` rvalues considered for completeness only.

- `const` objects can't be moved from.

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

URefs and Overloading

Exhibits counterintuitive behavior:

```
class MessedUp {
public:
  template<typename T>          // goal: handle lvalues
  void doWork(const T& param);  // reality: handle const lvalues

  template<typename T>          // goal: handle rvalues
  void doWork(T&& param);       // reality: handle everything
};                              //          except const lvalues


MessedUp m;
Widget w;
const Widget cw;

m.doWork(w);                    // doWork(T&&)
m.doWork(std::move(w));         // doWork(T&&)

m.doWork(cw);                   // doWork(const T&)
m.doWork(std::move(cw));        // doWork(T&&)
```

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

Slide 7

In the first two calls to `doWork`, the universal reference overload (`T&&`) is preferred, because it can instantiate to an exact match. Choosing the `const T&` overload would require adding `const`.

In the third call, both templates instantiate to take a parameter of type `const Widget&`, but the `const T&` template is more specialized, so, per 14.5.6.2, it's preferred.

In the final call, the universal reference overload would instantiate to take a `const Widget&&`, and the `const T&` overload would instantiate to take a `const Widget&`. Because the expression being passed is an rvalue, 13.3.3.2/3 bullet 1 sub-bullet 4 dictates that it preferentially bind to the instantiation taking the rvalue reference, i.e., to the instantiation arising from the universal reference overload.

URefs and Construction

A class initializable with a name or an ID mapping to a name:

```
std::string nameFromID(int ID);

class Person {
public:
  Person(const std::string& n): name(n) {}      // from name

  Person(int ID): name(nameFromID(ID)) {}       // from ID

private:
  std::string name;
};


std::string jkr("J. K. Rowling");
Person p1(jkr);                           // fine

Person p2("John Grisham");                // fine, but copies temp
                                          // (an rvalue) into name

Person p3(44245);                         // fine

Person p4(nameFromID(44245));             // fine, but copies rvalue
```


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

URefs and Construction

Perfect forwarding constructor is more efficient:

```
class Person {
public:
  template<typename NameT>
  Person(NameT&& n)                              // now takes URef
  : name(std::forward<NameT>(n)) {}

  Person(int ID): name(nameFromID(ID)) {}        // as before

private:
  std::string name;
};


std::string jkr("J. K. Rowling");
Person p1(jkr);                      // fine (same as before)

Person p2("John Grisham");           // fine, now initializes
                                     // name from string literal

Person p3(44245);                    // fine (same as before)

Person p4(nameFromID(44245));        // fine, now moves rvalue
                                     // into name
```

Seems okay, but overloading on URef worrisome.

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

URefs and Construction

With good reason:

```
class Person {                      // as on previous slide
public:
  template<typename NameT>
  Person(NameT&& n): name(std::forward<NameT>(n)) {}

  Person(int ID): name(nameFromID(ID)) {}

private:
  std::string name;
};

int idBlock();                     // find block holding ID
std::size_t offset;                // offset into block for name
…
Person p1(idBlock() + 22);         // fine

Person p2(idBlock() + offset);     // error! tries to initialize
                                   // name with a number!
```

- `int` + `std::size_t` ⇒ unsigned type!
  - Matches URef exactly, but `int` only with conversion.

Per 4.5 and 4.7/1-2, signed `int`→`unsigned int` is a conversion, not a promotion.

URefs and Construction

URef overloading really is the problem:

```
class Person {
public:
  template<typename NameT>                  // best match for
  Person(NameT&& n)                         // all types except
  : name(std::forward<NameT>(n)) {}         // non-const int

  Person(int ID)                            // best match for
  : name(nameFromID(ID)) {}                 // non-const int only

  …
};
```

Function templates taking URefs are greediest functions in C++.

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

Reining in URefs

Curbing greed possible, but not pretty:

```
class Person {
public:
  template<typename NameT,                                   // disable
           typename = typename std::enable_if<               // ctor
                        !std::is_integral<NameT>::value      // for
                                         >::type>            // integral
  Person(NameT&& n)                                          // types
  : name(std::forward<NameT>(n)) {}

  Person(int ID)                                             // as
  : name(nameFromID(ID)) {}                                  // before

  …
};
```


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

Special Member Functions

Special member functions may be automatically generated:

- **Constructors:** default, copy, move
- **Assignment operators:** copy, move
- **Destructor**

Highlighted functions (normally) take one parameter.

```
class Widget {
public:
  …
  Widget(const Widget&);               // may be auto-generated
  Widget(Widget&&);                    // may be auto-generated

  Widget& operator=(const Widget&);    // may be auto-generated
  Widget& operator=(Widget&&);         // may be auto-generated
  …
};
```


Scott Meyers, Software Development Consultant
http://aristeia.com/
© 2013 Scott Meyers, all rights reserved.

URefs, Overloading, and Special Member Functions

Consider class with "universal" copy/moving templates:

```
class Widget {
public:
  …
  template<typename T>
  Widget(T&&);                      // "universal" copy/move ctor

  template<typename T>
  Widget& operator=(T&&);           // "universal" copy/move op=
  …
};
```

Seems to handle all argument types:

- Lvalues + rvalues
- `const` + non-`const`
- `volatile` + non-`volatile`

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

Slide 14

A more accurate name for the "universal" copy/move ctor would be "universal converting ctor"

URefs, Overloading, and Special Member Functions

Not that simple.

- Overloading on URefs may be present.

Templates don't suppress generation of special member functions.

- Such functions generated ⇒ overloading on URefs.

```
class Widget {
public:
  …
  template<typename T>
  Widget(T&&);                      // "universal" copy/move ctor
  Widget(const Widget&);            // generated copy ctor
  Widget(Widget&&);                 // generated move ctor

  template<typename T>
  Widget& operator=(T&&);           // "universal" copy/move op=
  Widget& operator=(const Widget&); // generated copy op=
  Widget& operator=(Widget&&);      // generated move op=
  …
};
```


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

URefs, Overloading, and Special Member Functions

Resulting behavior can again surprise:

```
class Widget {
public:
  …
  template<typename T>
  Widget(T&&);                       // "universal" copy/move ctor
  Widget(const Widget&);             // generated copy ctor
  Widget(Widget&&);                  // generated move ctor
  …
};

Widget w;
const Widget cw;

Widget copyLvalue(w);                    // "universal" copy ctor
Widget copyRvalue(std::move(w));         // generated move ctor
Widget copyConstLvalue(cw);              // generated copy ctor
Widget copyConstRvalue(std::move(cw));   // "universal" move ctor
```

`operator=` behaves analogously.

These special member functions are not generated in all cases, but there are cases where they are generated.

URefs and Overloading

Story similar for non-member templates:

```
template<typename T>            // handles non-volatile
void doWork(const T& param);    // const lvalues only

template<typename T>
void doWork(T&& param);         // handles everything else


Widget w;
const Widget cw;

doWork(w);                      // doWork(T&&)
doWork(std::move(w));           // doWork(T&&)

doWork(cw);                     // doWork(const T&)
doWork(std::move(cw));          // doWork(T&&)
```


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

Problem Scenario Summary

URef overloading tends to arise when:

- Lvalues/rvalues need to be distinguished when forwarding.
  - ➡ Implies URef parameter.
- Some types get special treatment.
  - ➡ Implies non-URef parameter.

Problem can be dodged in several ways.

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

Tag Dispatching

Replace overloading with tag dispatching.

- Have single URef-taking function forward to overloads w/tags.

```
class Person {
public:
  template<typename T>                   // delegating ctor
  Person(T&& param)                      // take URef
  : Person(std::forward<T>(param),       // forward param
           std::is_integral<T>())        // tag
  {}                                     // ctor body

private:
  template<typename T>                   // ctor for
  Person(T&& param, std::true_type)      // integral types
  : name(nameFromID(std::forward<T>(param))) {}

  template<typename T>
  Person(T&& param, std::false_type)     // ctor for non-
  : name(std::forward<T>(param)) {}      // integral types

  std::string name;
};
```

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

Slide 19

Because `nameFromID` takes an `int`, there is no technical need to use `std::forward` on its argument, but in view of my advice to use `std::forward` on URefs, it's a good habit to get into, even when it's not required, IMO.

`std::true_type` and `std::false_type` are motivated by the need for *types*; `true` and `false` are *values*.

This approach due to reader comments on my 10/11/12 blog post, "Parameter Types in Constructors."

Tag Dispatching

Net effect:

- Integral args forwarded to private ctor calling `nameFromID`.

```
int idBlock();                    // as before
std::size_t offset;               // as before
…
Person p1(idBlock() + 22);        // fine (passes int)
Person p2(idBlock() + offset);    // fine (passes std::size_t)
```


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

Tag Dispatching

- Non-integral args forwarded to private ctor, then to `std::string`.

```
std::string jkr("J. K. Rowling");     // as before
Person p3(jkr);                       // copies jkr into name
Person p4(nameFromID(idBlock()+22));  // moves rvalue temp
                                      // into name
Person p5("John Grisham");            // initializes name
                                      // from string literal
Person p6(3.583);                     // error! can't init
                                      // std::string w/double
```

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

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

Tag Dispatching

Special copy/move member functions tricky.

- URef overloads don't suppress them.
  - ➡ URef versions inherently lead to overloading on URefs!
- **Think twice**: do you *need* a special copy/move function taking a URef?

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

Tag Dispatching

If so, have copy ops invoke URef versions:

```
class Person {
public:
  template<typename T>             // "universal" copy/move
  Person(T&& rhs);                 // ctor (uses tag dispatch)

  Person(const Person& rhs)        // "normal" copy ctor
  : Person(std::move(rhs)) {}      // delegates to template

  …
};
```

- `std::move(rhs)` yields `const` rvalue:
  - ➡ Rvalues preferably bind to URef than to LRef-to-`const`.
  - ➡ `const` prevents copy ctor param from being moved from.
    - Important, because caller's arg was `const`!

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

Slide 23

The universal ctor uses its argument to initialize its `name` data member—a `std::string`. The copy ctor gets a `Person` parameter, which is not compatible with a `std::string`. As a result, this code makes no sense. It's only purpose is to show how a copy constructor can be made to forward to a universal reference constructor.

The caller's argument to the copy constructor must have been `const`, because a non-`const` argument would have exact-matched the URef template.

Tag Dispatching

Copy assignment operator analogous:

```
class Person {
public:
  template<typename T>          // "universal" copy/move op=
  Person& operator=(T&& rhs);   // (uses tag dispatch)

  Person& operator=(const Person& rhs)        // "normal" op=
  { return operator=(std::move(rhs)); }       // calls template

  …
};
```

No need to write move operations.

- Declaring copy ops prevents their generation.

Assuming that assignment affects only the `Person`'s `name` data member, this code make no more sense than the code for the copy constructor on the previous slide. Again, the only reason for showing this code is to demonstrate how to force all assignment operations (regardless of argument) to go through the template taking a universal reference.

Tag Dispatching

All together:

```
class Person {
public:
  template<typename T>
  Person(T&& rhs)
  : Person(std::forward<T>(rhs), std::is_integral<T>()) {}
  Person(const Person& rhs): Person(std::move(rhs)) {}
  template<typename T>
  Person& operator=(T&& rhs)
  { return assign(std::forward<T>(rhs), std::is_integral<T>()); }
  Person& operator=(const Person& rhs)
  { return operator=(std::move(rhs)); }
private:
  template<typename T> Person(T&& rhs, std::true_type)          // integral
  { … }                                                         // args
  template<typename T> Person(T&& rhs, std::false_type)         // other
  { … }                                                         // arg types
  template<typename T> Person& assign(T&& rhs, std::true_type)  // integral
  { … }                                                         // args
  template<typename T> Person& assign(T&& rhs, std::false_type) // other
  { … }                                                         // arg types
};
```


Slide is animated.

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

Pass by Value

Replace URef with by-value parameter + `std::move`.

```
class Person {
public:
  Person(std::string n)                          // now by value
  : name(std::move(n)) {}                        // now std::move

  Person(int ID): name(nameFromID(ID)) {}   // as before

private:
  std::string name;
};
```


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

Pass by Value

Result:

```
Person p1(idBlock() + 22);           // uses nameFromID
Person p2(idBlock() + offset);       // uses nameFromID
Person p3(jkr);                      // copies jkr into n,
                                     // moves n into name
Person p4(nameFromID(idBlock()+22)); // moves rvalue temp into n,
                                     // moves n into name
Person p5("John Grisham");           // initializes n from
                                     // string literal,
                                     // moves n into name
Person p6(3.583);                    // uses nameFromID
                                     // (due to double → int
                                     // conversion)
```


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

Pass by Value

Observations:

- By-value param always constructed.
  - ➡ Approach probably costlier than tag dispatch.
- Param always `std::move`d into destination.
  - ➡ Not all types are cheap to move.
- Arg type incompatibility detected sooner than with tag dispatching.

```
Widget w;        // not a valid type for init'ing a Person

Person p(w);     // by-value approach reports error in call
                 // to Person ctor. Tag dispatching approach
                 // reports error in call to ctor of data
                 // member "name" (3 levels down).
```

- Avoids perfect forwarding failure cases.
  - ➡ See Further Information.

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

Slide 28

Regarding error detection and URefs, in the tag dispatch example, the call to the constructor for `name` is three levels down: after calls to the `Person` public constructor and a `Person` private constructor.

Pass by LRef-to-const

Pass by `const T&`.

- Appropriate when parameter need only be read.
  - I.e., when perfect forwarding not required.

```
template<typename T>                    // take anything as
void logAndPrint(const T& param)        // read-only value
{
  makeLogEntry(param);
  std::cout << param;
}
```


The original problem statement said that perfect forwarding was a requirement, so this approach doesn't really correspond to the problem, but it's still worth mentioning. Also worth noting is that perfect forwarding is not a read-only operation, because forwarded non-`const` values may be modified.

Overloading Guideline

- **Overloading on RRef + LRef:** typically OK.
- **Overloading on a URef:** typically not OK.

Remember `push_back` versus `emplace_back`:

```
template<class T, class Allocator=allocator<T>>
class vector {
public:
  …
  void push_back(const T& x);         // LRef (copy lvalues)
  void push_back(T&& x);              // RRef (move rvalues)

  template<class... Args>
  void emplace_back(Args&&... args);  // URef (forward
  …                                   // everything)
};
```


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

Guideline

Avoid overloading on universal references.

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

Further Information

Universal references:

- "Universal References in C++11," Scott Meyers.
  - Print: *Overload*, October 2012.
  - Video: *Channel 9*, 9 October 2012.
  - Online: *ISOcpp.org*, 1 November 2012.
- "Universal references and std::initializer_list," user2523017, *stackoverflow*, 26 June 2013.
  - Explores overload behavior given braced initializer list arguments.
    - Template type deduction fails for braced initializer lists.
- "Perfect Forwarding Failure Cases," `comp.std.c++` discussion initiated 16 January 2010.
  - When passing arguments to universal references goes awry.

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

Further Information

Overloading and universal references:

- "Parameter Types in Constructors," Scott Meyers, *The View from Aristeia (Blog)*, 11 October 2012.
- "Copying Constructors in C++11," Scott Meyers, *The View from Aristeia (Blog)*, 13 October 2012.
- "Overloading the broken universal reference 'T&&'," *Musing Mortoray (Blog)*, 3 June 2013.
  - ➡ Another example of the tag dispatching approach.
- "Overload resolution with universal references," jleahy, *stackoverflow*, 22 May 2013.
  - ➡ Accepted solution based on `std::enable_if`.

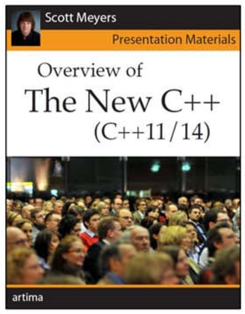
Scott Meyers, Software Development Consultant
http://aristeia.com/

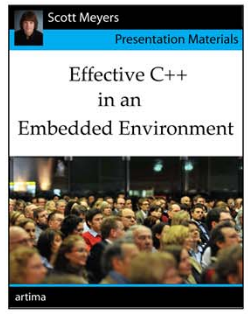
Licensing Information

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

Courses currently available for personal use include:





Scott Meyers, Software Development Consultant
http://aristeia.com/
© 2013 Scott Meyers, all rights reserved.

About Scott Meyers

Scott offers training and consulting services on the design and implementation of C++ software systems. His web site,

http://aristeia.com/

provides information on:

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

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