Universal References in C++11

Scott Meyers, Ph.D.
Software Development Consultant
http://aristeia.com
smeyers@aristeia.com

Last Revised: 8/7/12

Prerequisites

- Copyable vs. movable objects
- Lvalues vs. rvalues
- std::move(*expr*)
  - Casts *expr* to an rvalue
- Basics of lvalue references and rvalue references
  - Syntax (*type*& versus *type*&&)
  - Binding rules:
    - Lvalue references: lvalues and rvalues
    - Rvalue references: rvalues only
- Syntax for variadic templates

type&& ≠ Rvalue Reference

Crux:

- Rvalue reference ⇒ *type&&*
- *type&&* ⇏ rvalue reference

This is a lie.

A useful lie.

For most purposes, the truth is less helpful.

- We'll cover it at the end.

The Double Life of *type&&*

```
void f(Widget&& param);                      // rvalue reference
Widget&& var1 = Widget();                    // rvalue reference
auto&& var2 = var1;                          // not rvalue reference
template<typename T>
void f(std::vector<int>&& param);            // rvalue reference
template<typename T>
void f(T&& param);                           // not rvalue reference
```

The Double Life of *type&&*

In "*type&&*", "&&" means either:

- Rvalue reference.
  - Binds rvalues only.
  - Facilitates moves.
- ***Universal reference***
  - Rvalue reference *or* lvalue reference.
    - Syntactically *type&&*, but semantically *type&* *or* *type&&*.
  - Binds lvalues *and* rvalues, const and non-const—*everything*!
  - May facilitate copies, may facilitate moves.

New Terminology!

Shorthands for this Talk

- LRef = Lvalue reference
- RRef = Rvalue reference
- URef = Universal reference

In a Nutshell

URefs possible in four contexts:

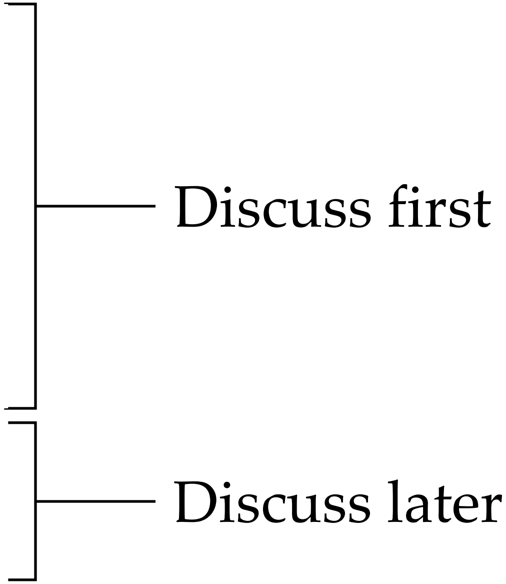
- **Function template parameters:**

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

- **auto declarations:**

```
auto&& var = ... ;
```

- **typedef declarations**
- **decltype expressions**



Discuss first

Discuss later

In a Nutshell

If a variable or parameter has declared type

T&&

for some

deduced type T,

it's a universal reference.

Otherwise it's whatever it looks like :-)

In a Nutshell

URefs require initializers:

- Initializer for URef is lvalue ⇒ URef *becomes* LRef.
- Initializer for URef is rvalue ⇒ URef *becomes* RRef.

Photo: Sid Mosdell

Examples

```
template<typename T>
void f(T&& param);         // URef: proper syntax + deduced type

Widget w;

f(w);                      // w is lvalue ⇒ URef becomes LRef;
                           // f(Widget&) instantiated

f(std::move(w));           // std::move yields rvalue ⇒
                           // URef becomes RRef;
                           // f(Widget&&) instantiated

f(Widget());               // Widget() yields rvalue ⇒
                           // URef becomes RRef;
                           // f(Widget&&) instantiated
```

Examples

```
std::vector<int> v;
...
auto&& val = 10;                 // 10 is rvalue ⇒ URef becomes RRef;
                                 // val's type is int&&

auto&& element = v[5];           // v[5] returns int& + LRefs are lvalues ⇒
                                 // v[5] is lvalue ⇒ URef becomes LRef;
                                 // element's type is int&
```

auto&&?

The foundation of range-based for. Per C++11's § 6.5.4,

```
for ( for-range-declaration : expression ) statement
```

equivalent to

```
{
  auto && __range = range-init;
  for ( auto __begin = begin-expr, __end = end-expr;
        __begin != __end;
        ++__begin ) {
    for-range-declaration = *__begin;
    statement
  }
}
```

¬Type Deduction ⇒ ¬URef

```
void f(Widget&& w);                // undeduced type ⇒ RRef

template<typename T>
void f(T&& param);                 // deduced type    ⇒ URef

template<typename T>
class Gadget1 {
  Gadget1(Gadget1&& rhs);          // undeduced type ⇒ RRef
};

template<typename T1>
class Gadget2 {
  template<typename T2>
  Gadget2(T2&& rhs);               // deduced type    ⇒ URef
};
```

¬Type Deduction ⇒ ¬URef

Not all T&&s in templates are URefs:

```
template<class T,                                // from
         class Allocator=allocator<T>>           // C++11
class vector {                                   // standard
public:
  ...
  void push_back(T&& x);        // RRef! T comes from vector<T>,
  ...                           // not arg passed to push_back
};
```

push_back vs. emplace_back

Contrast with emplace_back:

```
template<class T,
            class Allocator=allocator<T>>
class vector {
public:
  ...
  template<class... Args>
  void emplace_back(Args&&... args);        // URef! Args deduced
  ...
};
```

push_back vs. emplace_back

Note overloading (and lack thereof):

```
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)
  …
};
```

In templates, URefs essentially *forwarding references*.

URefs and Overloading

Overloading + URefs almost always an error.

- Makes no sense: URefs handle *everything*.
  - Lvalues, rvalues, `const`s, non-`const`s, `volatile`s, non-`volatile`s, etc.
  - They're *universal* references!
- Counterintuitive behavior.
  - See next page.

URefs and Overloading

```
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&&)
```

URefs and Overloading

Same story 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&&)
```

(RRef→URef) ⇒ (std::move→std::forward)

Typical function bodies with overloading:

```
void doWork(const Widget& param)                // copy
{
  ops and exprs using param
}
void doWork(Widget&& param)                     // move
{
  ops and exprs using std::move(param)
}
```

Typical function implementation with URef:

```
template<typename T>
void doWork(T&& param)                          // forward ⇒
{                                               // copy and move
  ops and exprs using std::forward<T>(param)
}
```

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);                             es)
  void push_back(T&& x);                                 ues)

  template<class... Args>
  void emplace_back(Args&&... args);                      ng)
  ...
};
```

RRef.
Overloading OK.
Use std::move.

URef.
Overloading rarely OK.
Use std::forward.

Syntax Matters

URefs *must* have form ***type&&***.

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

template<typename T>
void f(const T&& param);           // RRef

template<typename T>
void f(std::vector<T>&& param);    // RRef

Widget w;

auto&& v1 = w;                     // URef ⇒ LRef

const auto&& v2 = w;               // RRef; code won't compile
```

Syntax Matters

Realm of artistic freedom:

- Parameter name
- White space :-)

```
template<typename ParamType>
void f(ParamType&& param);                    // URef

template<typename ParamType>
void f(ParamType  && param);                  // URef

auto  && I_did_it_my_way = 44;                // URef
```

Gratuitous Animal Photo

Australian Brushturkey

- Gender of offspring (somewhat) determined by nest temperature

Photo: John Harrison

The Truth

Declaring references to references is illegal:

```
Widget w;
...
Widget& & rrw = w;              // error!
```

But refs-to-refs arise during type deduction and evaluation.

Type Deduction for URefs

Function templates taking URefs employ special type deduction:

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

- Lvalue arg to f ⇒ T an LRef (T&)
- Rvalue arg to f ⇒ T just T (a non-reference)

```
Widget w;

f(w);                                  // T is Widget&

f(std::move(w));                       // T is Widget (not Widget&&!)
```

Ref-To-Ref Generation

Lvalues thus yield a ref-to-ref:

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

Widget w;

f(w);            // generates void f<Widget&>(Widget& && param);
```

Reference Collapsing

Generated refs-to-refs undergo *reference collapsing*:

- `T& &` ⇒ `T&`
- `T&& &` ⇒ `T&`
- `T& &&` ⇒ `T&`
- `T&& &&` ⇒ `T&&`

IOW,

- RRef-to-RRef ⇒ RRef
- LRef-to-*anything* ⇒ LRef
  - ➡Stephan T. Lavavej: “Lvalue references are infectious.”

type&& Really *Does* Mean RRef!

URefs are simply RRefs in ref-collapsing contexts.

```
template<typename T>             // as before
void f(T&& param);               // acts like a URef, is an RRef

Widget w;

f(w);                            // f(Widget& &&) ⇒
                                 // f(Widget&)

f(std::move(w));                 // f(Widget&&)
```

auto

Another ref-collapsing context.

- Uses template type deduction rules (plus a bit more)

```
Widget w;
...
auto&& v1 = w;              // lvalue initializer, v1's type is Widget&
auto&& v2 = std::move(w);  // rvalue initializer, v2's type is Widget&&
```

Declarations involving typedefs

Also a ref-collapsing context:

```
template<typename T>
class Widget {
  typedef T& LvalueRefType;
  ...
};

Widget<int&> w;                          // Widget<int&>::LvalueRefType
                                         // is int& & ⇒ int&

typedef Widget&& RRtoW;

RRtoW& v1 = w;                           // v1's type is Widget&

const RRtoW& v2 = std::move(w);          // v2's type is const Widget&

RRtoW&& v3 = std::move(w);               // v3's type is Widget&&
```

References and Type Deduction

In template/`auto` type deduction, refs become non-refs before lvalue/rvalue analysis.

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

Widget w;
...
Widget& lrw = w;                  // lrw’s type is Widget&
Widget&& rrw = std::move(w);      // rrw’s type is Widget&&
f(lrw);                           // f<Widget&>(Widget&)
f(rrw);                           // f<Widget&>(Widget&)
f(std::move(rrw));                // f<Widget>(Widget&&)
```

decltype

`decltype(expr)` yields `T&` or `T`, and ref-collapsing applies.

- Sounds like templates and `auto`.
- Isn't.
  - ➡ Type evaluation rules are different:
    - ◆ `decltype(`*id*`)` ⇒ *id*'s declared type
    - ◆ `decltype(`*non-id lvalue expr*`)` ⇒ *expr*'s type; LRef (`T&`)
    - ◆ `decltype(`*non-id rvalue expr*`)` ⇒ *expr*'s type; non-ref (`T`)

```
Widget w;
decltype(w)&& r1 = std::move(w);    // r1's type is Widget&&
decltype((w))&& r2 = std::move(w);  // r2's type is Widget&
```

Truth ≡ Important, but Lie ≡ Useful

Distinguishing URefs from RRefs improves:

- Code comprehension
  - Avoids *type*&& ⇒ RRef error
- Communication among developers
  - Say/write RRef only when it can't be an LRef

```
template<typename T>
void f(T&& param);                    // URef, not RRef

for (auto&& i = factory()) ...        // URef, not RRef

typedef Gadget::TMP::type&& GTType;   // URef, not RRef

decltype(w)&& v = std::move(w);       // URef, not RRef
```

Summary

- *type&&* ⇏ rvalue reference
- *type&&* syntax + type deduction ⇒ *universal reference*.
- Whether URef becomes LRef or RRef depends on initializer.
  - Lvalue ⇒ LRef, Rvalue ⇒ RRef.
- Not all T&&s in templates are universal references.
  - ¬Type deduction ⇒ ¬URef.
- Overloading with URefs almost always wrong.
- Foundation is type deduction and reference collapsing.
  - Applies to function templates, auto, typedef, and decltype.

Further Information

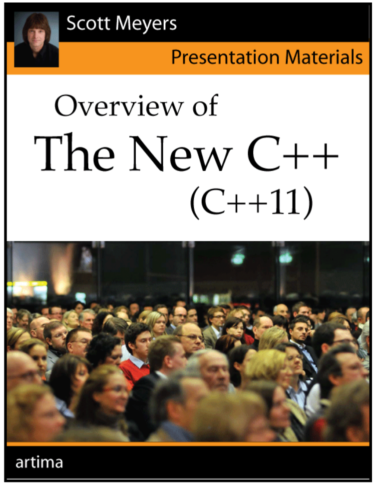
- "Universal References in C++11," Scott Meyers, forthcoming.
- "A Note About decltype," Andrew Koenig, *Dr. Dobb's*, 27 July 2011.
  - ➡Offers insight into `decltype`'s type-evaluation rules.
- "Preventing non-const lvalues from resolving to rvalue reference instead of const lvalue reference," *stackoverflow*, 13 October 2011.
  - ➡Comment from Howard Hinnant explains why "`&&`" has two meanings.

Licensing Information

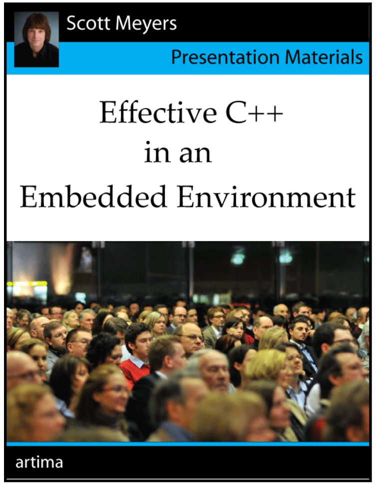
Scott Meyers licenses materials for this and other training courses for commercial or personal use. Details:

- 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
Presentation Materials
Overview of
The New C++
(C++11)
artima



Scott Meyers
Presentation Materials
Effective C++
in an
Embedded Environment
artima

About Scott Meyers

Scott is a trainer and consultant 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