There Ain't No Such Thing As A Universal Reference

Jonathan Wakely
Red Hat

Universal References

Scott Meyers's term
(Overload 111, isocpp.org blog, Channel 9 video)

T&& Doesn't Always Mean "Rvalue Reference"

Universal References

Scott presents a very useful model for understanding function templates such as:

```
template<typename T>
  void func( T&& arg )
  { }
```

Which can be passed lvalue or rvalue arguments:

```
int i = 0;
func( i      );     // lvalue
func( i + 1  );     // rvalue
func( "cat"s );     // rvalue
```

Universal References

```
template<typename T>
  void func( T&& )
  { }
```

> "The essence of the issue is that "&&" in a type declaration sometimes means rvalue reference, but sometimes it means **either** rvalue reference **or** lvalue reference."

Unless stated otherwise all quotes in these slides are from Meyers, 2012, 'Universal References in C++11', *Overload*, no. 111, p. 8-12.

Universal References

Scott introduces new terminology for these rvalue-or-lvalue references:

> "Such unusually flexible references deserve their own name. I call them universal references."
>
> "If a variable or parameter is declared to have type `T&&` for some **deduced type** `T`, that variable or parameter is a universal reference."

(This includes types deduced with `auto&&` because `auto` uses the same rules as template argument deduction)

We don't need no stinkin' abstractions!

Universal references are a useful model for reading and understanding template code in modern C++, everyone should read the article

But according to the C++ standard (and your compiler) ...

There ain't no such thing as a universal reference!

The standard (and compilers) only know about lvalue references and rvalue references

And Scott's requirement that there must be a deduced type involved means there are cases that aren't explained by universal references

Not a universal reference

```
template<typename T>
  class Wrapper
  {
  public:
    Wrapper( T&& t ) // not deduced
    : wrapped(std::forward<T>(t))
    { }
    // ...
  private:
    T wrapped;
  };

template<typename T>
  Wrapper<T>
  wrap( T&& t ) // deduced
  { return Wrapper<T>{std::forward<T>(t)}; }
```

Behind the curtain

There are two properties of the C++ language that explain everything covered by the term universal references, and other cases too:

- Reference collapsing
- A special case in the argument deduction rules

Reference Collapsing

Let's declare a simple typedef for a reference type:

```
typedef int&      intlref;
```

And then apply some transformations:

```
typedef intlref&  intlref2;
typedef intlref&& intlref3;
```

We still have the same type:

```
is_same<intlref, intlref2> // true, int&
is_same<intlref, intlref3> // true, int&
```

Reference Collapsing

Using a typedef for an rvalue-reference type:

```
typedef int&&      intrref;
```

Then applying the same transformations:

```
typedef intrref&  intlref4;
typedef intrref&& intrref2;
```

We don't always get the same type:

```
is_same<intrref, intlref4> // false, int&
is_same<intrref, intrref2> // true, int&&
```

Reference Collapsing

We can use C++11 alias templates to perform these same transformations:

```
template<typename T> using ref = T&;
template<typename T> using refref = T&&;

is_same<ref<int>, int&>
is_same<ref<int&>, int&>
is_same<ref<int&&>, int&>
is_same<refref<int>, int&&>
is_same<refref<int&>, int&>
is_same<refref<int&&>, int&&>
```

Reference Collapsing

These core language rules for reference collapsing are modelled by transformation traits in the standard library:

```
add_lvalue_reference<int>::type    ⇒ int&
add_lvalue_reference<int&>::type   ⇒ int&
add_lvalue_reference<int&&>::type  ⇒ int&

add_rvalue_reference<int>::type    ⇒ int&&
add_rvalue_reference<int&>::type   ⇒ int&
add_rvalue_reference<int&&>::type  ⇒ int&&
```

This property of C++ is why `std::move` is declared to return `remove_reference<T>::type&&` and not simply `T&&`, you can't just append && to a type and expect to get an rvalue-reference

Reference Collapsing

The exact same rules we've seen for reference collapsing in typedefs also apply in other contexts, such as function parameters:

```
void func(intlref&);   void func(ref<int&>);

void func(intlref&&);  void func(refref<int&>);
```

All those declarations are the same function as:

```
void func(int&);
```

Explicit Template Arguments

Using this function template:

```
template<typename T>
  void func(T&&) { }
```

We can substitute the type T using an explicit template argument:

```
func<int>(1);   // func<int>(int&&)

int i;
func<int&>(i);  // func<int&>(int&)

func<int&&>(1); // func<int&&>(int&&)
```

Not a universal reference

```
template<typename T>
  class Wrapper
  {
  public:
    Wrapper( T&& t ) // not deduced, but collapses
    : wrapped(std::forward<T>(t))
    { }
    // ...
  private:
    T wrapped;
  };

template<typename T>
  Wrapper<T>
  wrap( T&& t ) // deduced
  { return Wrapper<T>{std::forward<T>(t)}; }
```

Template Argument Deduction

Using the same function template:

```
template<typename T>
  void func(T&&) { }
```

We are now ready to understand what happens when the type `T` is deduced:

```
func(1);   // func<int>(int&&)

int i;
func(i);   // func<int&>(int&)
```

Deduction of lvalue-refs

```
template<typename T>
  void func(T&&) { }

 int i;
 func(i);   // func<int&>(int&)
```

The key thing to notice here, which is not at all obvious, is that

A function template with a parameter like `T&&` allows the template parameter `T` to be *deduced as an lvalue reference type.*

Deduction of lvalue-refs

This all-important rule about deducing lvalue reference types is hard to find in the standard and quite understated:

> "[...] If P is an rvalue reference to a cv-unqualified template parameter and the argument is an lvalue, the type "lvalue reference to A" is used in place of A for type deduction. [...]"
>
> ISO/IEC 14882:2011, Programming Languages — C++, [temp.deduct.call] § 14.8.2.1 ¶3

Universal References

First there is a mountain
- then there is no mountain
- then there is
— Zen proverb

Although it's valuable to understand what your code is actually doing, universal reference are a useful model for reading and understanding C++11 code, even if they aren't really there!

These slides are available at https://gitorious.org/wakelyaccu/accu2014

You can download and view the HTML version of these slides with commands like:

```
git clone https://git.gitorious.org/wakelyaccu/accu2014.git
firefox accu2014/tanstaaur.html
```