Forwarding References

Herb Sutter, Bjarne Stroustrup, Gabriel Dos Reis

Document #: N4164
Date: 2014-10-06
Reply to: Herb Sutter
(hsutter@microsoft.com)

Contents

1. Motivation

1.1. Overview

To the C++ programmer, a parameter of type C&& is always an rvalue reference—except when C is a template parameter type or `auto`, in which case it behaves very differently even though language-technically it is still an rvalue reference.

We intentionally overloaded the && syntax with this special case, but we did not give this special case a name. It needs a distinct name so that we can talk about it and teach it. This has already been discovered in the community, thanks to Scott Meyers in particular. [1]

In the absence of our giving this construct a distinct name, the community has been trying to make one. The one that is becoming popular is "universal reference." [1] Unfortunately, as discussed in §3.1 below, this is not an ideal name, and we need to give better guidance to a suitable name.

The name that has the most support in informal discussions among committee members, including the authors, is "**forwarding reference**." Interestingly, Meyers himself initially introduced the term "forwarding reference" in his original "Universal References" talk, [2] but decided to go with "universal references" because at the time he did not think that "forwarding references" reflected the fact that `auto&&` was also included; however, in §3.3 below we argue why `auto&&` is also a forwarding case and so is rightly included.

1.2. The naming problem

Consider this code:

```
void foo( X&& x );

template<class Y>
void bar( Y&& y );
```

These parameters are fundamentally different. Consider these questions, and how to teach them:

- What are the types of the function parameters?
- What arguments to they accept or reject?
- What is each parameter for?

Despite the syntactic similarity of X&& and Y&&, the answers are fundamentally different for `foo` and `bar`.

- `foo` takes an **rvalue** reference to **non-const**.
  bar takes an **lvalue or rvalue** reference to **everything**: `const`, `volatile`, both, **and** neither. *
- `foo` accepts **only rvalue** X objects.
  `bar` accepts **all** Y objects.
- `foo`'s parameter is for **capturing temporaries** (and other rvalues).
  `bar`'s parameter is for **forwarding** its argument onward.

The last point gets to the heart of the matter: The current T&& design was specifically designed to support *argument forwarding* uses.

* Note that in this bullet the correct word really is "everything," not "anything." In the body of `bar`, the parameter `y` is sometimes `const`, **and** sometimes `volatile`, **and** sometimes both, **and** sometimes neither. If `bar` were actually going to directly use its parameter, this would be madness—it would care whether its parameter, which to the function is a local variable, was `const` or not, `volatile` or not, and so on. But it doesn't care, because its purpose is not to use the parameter directly, but to be "neutral passthrough" code that is agnostic to what it's being given and faithfully pass it along to other code that will use the parameter and may care about the argument's `const`-ness, `volatile`-less, and rvalue-ness.

1.3. A wording problem

Note that this means we have an (arguably) incorrect normative statement in 8.3.2/2, which says:

> 2 A reference type that is declared using & is called an lvalue reference, and a reference type that is declared using && is called an rvalue reference. Lvalue references and rvalue references are distinct types. Except where explicitly noted, they are semantically equivalent and commonly referred to as references.

This statement about && is either untrue or misleading, at least insofar as the feature is used; for example, this states normatively that `bar`'s parameter is an rvalue reference. We think this illustrates the confusion surrounding the && punning.

2. Proposal

We (the committee) should guide the community to use the term "forwarding reference" as a clearer term. The simplest way to do that is to mention the name in the standard, even in a non-normative note.

2.1. Add the term "forwarding reference"

Change 8.3.2/6 as follows:

> 6 If a *typedef-name* (7.1.3, 14.1) or a *decltype-specifier* (7.1.6.2) denotes a type TR that is a reference to a type T, an attempt to create the type "lvalue reference to *cv* TR" creates the type "lvalue reference to T", while an attempt to create the type "rvalue reference to *cv* TR" is called a *forwarding reference* and creates the type TR. [*Example:* …

2.2. (Optional) Fix a wording problem

Change 8.3.2/2 as follows:

> 2 A reference type that is declared using & is called an lvalue reference, and a reference type that is declared using && is called an rvalue reference or a forwarding reference. Lvalue references and rvalue references and forwarding references are distinct types. Except where explicitly noted, they are semantically equivalent and commonly referred to as references.

3. Q&A

3.1. Q: Why not "universal reference"? A: It implies the wrong things.

"Universal reference" is a reasonable name with an obvious meaning—one that happens to be wrong in at least the authors' opinion. A "universal reference" must (obviously to many of us in retrospect) mean:

- a reference that can be used everywhere; or
- a reference that can be used for everything; or
- something similar.

And this is not the case, and is not the appropriate use of this construct. The concern is that some people will interpret that something having such a name is meant to be used "universally." And that's a bad thing to encourage by a name that will imply that to many people, even if we constantly put up disclaimers when we use it.

So the rub is that it is a name that just rolls off the tongue and is misleading because "universal references" aren't universal in the sense of how pervasively they should be used. Furthermore, they aren't even really references *per se*, but rather a set of rules for using references in a particular way in a particular context with some language support for that use, and that use is forwarding.

3.2. Q: But this is just reference collapsing, why do we need a name? A: Because reference collapsing is just the mechanism, and forwarding is the abstraction.

We need a name that says what it is and connotes how it should be used, not just how it is implemented. Similarly, we teach and write programs in terms of virtual functions (abstraction), not vtable dispatch (implementation mechanism).

3.3. Q: What about `auto&&`, including for local variables and `for( /*auto&&*/ e : c)`? A: Those are also forwarding cases.

Recall that `auto&&` also follows these special rules. Is it legitimate to call those forwarding cases too?

First, consider generic lambda parameters, which are just disguised template parameters. A parameter is a local variable, and in code like `[](auto&& x){ … }`, this is just as much a forwarding reference.

Next, consider range-`for`. C++11 range-`for` is expressed in terms of `auto&&`, to get the starting iterator using `auto && __range = range-init;` (stmt.ranged, 6.5.4/1). Further, we have a C++17 proposal for a range-based `for(x:c)` that has the local variable `x` be implicitly `auto&&`. That makes perfect sense, because the range-`for` *is a neutral forwarder* between a collection and calling code, so of course it should maintain the `const`-ness, rvalue-ness, etc. of what the iterator dereference operation returns.

Finally, just as with the extended range-`for` local variable, it is true in general that `auto&&` local variables are for forwarding. This was already answered in the main text by noting that a parameter is like a local variable. The only time a local variable like `auto&& local = /*...*/;` makes sense is in forwarding code. As this paper already said about the parameter case, here too `local` "is sometimes `const`, **and** sometimes `volatile`, **and** sometimes both, **and** sometimes neither. If *[the function]* were actually going to directly use *[this local variable]*, this would be madness—it would care whether its *[…]* local variable, was `const` or not, `volatile` or not, and so on. But it doesn't care, because its purpose is not to use the *[local variable]* directly, but to be "neutral passthrough" code that is agnostic to what it's being given and faithfully pass it along to other code that will use the parameter and care about the argument's `const`-ness, `volatile`-less, and rvalue-ness."

So yes, `auto&&` used for local variables really are forwarding references too.

4. Acknowledgments

Thanks to Gabriel Dos Reis, Stephan T. Lavavej, Scott Meyers, Bjarne Stroustrup and others for their comments and feedback on this topic and/or on drafts of this paper. Special thanks to Scott Meyers for popularizing the need for a name, and for being open to changing the specific name he began to popularize.

5. References

[1] S. Meyers. "Universal References in C++11" (isocpp.org blog, November 1, 2012).

[2] S. Meyers. "Universal References in C++11" (*C++ and Beyond*, Asheville, NC, USA, August 6, 2012).