Document: PL22.16/09-0200 = WG21 N3010
Author: William M. Miller
Edison Design Group
Date: 2009-11-09
Revision: 1

Rvalue References as "Funny" Lvalues

I. Background

Rvalue references were introduced into C++0x to provide a mechanism for capturing an rvalue temporary (which could previously be done in C++ using traditional lvalue references to const) and allowing modification of its value (which could not). When used in the contexts of reference binding, overload resolution, and template argument deduction, it was desired that a function returning an rvalue reference should behave like a traditional function returning an rvalue. The most straightforward way of achieving that goal was to classify such rvalue reference return values as rvalues, and that approach is embodied in the current draft.

Unfortunately, however, rvalues have certain characteristics that are incompatible with the intended uses for rvalue references. In particular:

- Rvalues are anonymous and can be copied at will, with the copy assumed to be equivalent to the original. Rvalue references, however, designate a specific object in memory (even if it is a temporary), and that identity must be maintained.
- The type of an rvalue is fully known – that is, its type must be complete, and its static type is the same as its dynamic type. By contrast, an rvalue reference must support polymorphic behavior and should be able to have an incomplete type.
- The type of a non-class rvalue is never cv-qualified. An rvalue reference, however, can be bound to a const or volatile object, and that qualification must be preserved.

In addition, rvalue references (like traditional lvalue references) can be bound to functions. Treating an rvalue reference return value as an rvalue, however, introduces the novel concept of a function rvalue into the language. There was previously no such idea – a function lvalue used in an rvalue context becomes a pointer-to-function rvalue, not a function rvalue – so the current draft Standard does not describe how such rvalues are to be treated. In particular, function calls and conversions to function pointers are specified in terms of function lvalues, so most plausible uses of rvalue references to functions are undefined in the current wording.

One possible approach to resolving these problems would be to maintain the current approach of treating an rvalue reference return value as an rvalue but to add various caveats to the specification of rvalues so that those coming from rvalue references would have special characteristics. This could be called the "funny rvalue" approach. However, further examination of the current wording of the draft Standard indicates that the problems listed above are probably only the tip of the iceberg: many of the specifications that should apply to the objects to which rvalue references refer, such as object lifetime, aliasing rules, etc., are phrased in terms of lvalues, so the list of rvalue caveats could get quite long.

This suggests an alternative approach: that rvalue reference return values should actually be seen as lvalues, with a few exceptions to allow them to be treated as rvalues in the cases where that is intended,

i.e., in reference binding, overload resolution, and template argument deduction. This idea, dubbed the “funny lvalue” approach, is the subject of this paper.

(The problems described above are discussed in more detail in the following issues in the Core Language Issues List: 664, 690, 846, and 863.)

II. Overview of Changes

The detailed wording changes given in the next section implement the following conceptual modifications to the existing specification:

- There are two kinds of lvalues: traditional lvalues as described in the current WP and lvalues that reflect the binding of rvalue references to objects. The latter are called “rref lvalues” in the revised wording, while traditional lvalues become “non-rref lvalues.” Many references to lvalues in the current wording are unchanged and intentionally apply to both kinds.
- Rref lvalues are created by three kinds of expressions:
  - Function calls (explicit or implicit) that return rvalue references to objects
  - Casts to rvalue references to objects
  - Member selection and pointer-to-data-member dereference expressions where the object expression is an rref lvalue

  As a result, function lvalues are always non-rref lvalues and there are no function rvalues.
- In general, operators whose operands or results are currently lvalues are changed to require and/or produce non-rref lvalues.
- Reference binding and overload resolution are changed to treat rref lvalues the same as rvalues.

III. Detailed wording changes

The principal sections in which the concepts are introduced (3.10, 5) are presented first, followed by the various ancillary changes. The changes are shown relative to the current working paper, N3000. Additional edits will be required to some pending resolutions to various core language issues, but these are not yet incorporated into this document. The terms “rref lvalue” and “non-rref lvalue” are subject to change; various suggestions for alternatives have been made, but none so far have commended themselves as clearly superior.

3.10 Lvalues and rvalues (paragraphs 1-6)

Every expression is either an *lvalue* or an *rvalue*.

An lvalue refers to an object or function. Some rvalue expressions — those of (possibly cv-qualified) class or array type — also refer to objects.[51]

[*Note:* some built-in operators and function calls yield lvalues. [*Example:* if `E` is an expression of

pointer type, then `*E` is an lvalue expression referring to the object or function to which `E` points. As another example, the function

```
int& f();
```

yields an lvalue, so the call `f()` is an lvalue expression. —*end example*] —*end note*]

There are two kinds of lvalues. Some expressions, such as calls to functions that return an rvalue reference to an object type, yield *rref lvalues*. All other lvalues are *non-rref lvalues*. Unless otherwise specified, statements in this International Standard that mention lvalues apply to both kinds.

[*Note:* some built-in operators expect lvalue operands. [*Example:* built-in assignment operators all expect their left-hand operands to be **non-rref** lvalues. —*end example*] Other built-in operators yield rvalues, and some expect them. [*Example:* the unary and binary + operators expect rvalue arguments and yield rvalue results. —*end example*] The discussion of each built-in operator in Clause 5 indicates whether it expects lvalue operands and**/or** ~~whether it~~ yields an lvalue**, and, if so, which kind of lvalue**. —*end note*]

The result of calling a function that does not return ~~an lvalue~~ **a** reference is an rvalue. User defined operators are functions, and whether such operators expect or yield lvalues is determined by their parameter and return types.

An expression ~~which~~ **that** holds a temporary object resulting from a cast to a type other than ~~an lvalue~~ **a** reference type is an rvalue (this includes the explicit creation of an object using functional notation (5.2.3)).

...

5 Expressions (paragraphs 5-6)

If an expression initially has the type “~~lvalue~~ reference to T” (8.3.2, 8.5.3), the type is adjusted to T prior to any further analysis, the expression designates the object or function denoted by the ~~lvalue~~ reference, and the expression is an lvalue.

~~If an expression initially has the type “rvalue reference to T” (8.3.2, 8.5.3), the type is adjusted to “T” prior to any further analysis, and the expression designates the object or function denoted by the rvalue reference. If the expression is the result of calling a function, whether implicitly or explicitly, it is an rvalue;~~ **An lvalue expression is an rref lvalue if it is:**

- **the result of calling a function, whether implicitly or explicitly, whose return type is an rvalue reference to an object type,**
- **a cast to an rvalue reference to an object type,**
- **a member selection expression designating a non-static data member in which the object expression is an rref lvalue, or**
- **a `.*` pointer-to-member expression in which the first operand is an rref lvalue and the**

second operand is a pointer to data member;

otherwise, it is ~~an~~ **a non-rref** lvalue. [*Note:* In general, the effect of this rule is that named rvalue references are treated as **non-rref** lvalues and unnamed rvalue references are treated as ~~rvalues~~ **rref lvalues**. —*end note*]

[*Example:*

```
struct A {
  int m;
};
A&& operator+(A, A);
A&& f();

A a;
A&& ar = a static_cast<A&&>(a);
```

The expressions `f()`, **`f().m`**, **`static_cast<A&&>(a)`**, and `a + a` are ~~rvalues~~ **rref lvalues** of type `A`. The expression `ar` is ~~an~~ **a non-rref** lvalue of type `A`. —*end example*]

1.3.4 dynamic type: no change (applies to both kinds of lvalues)

1.9 Program execution ¶12: no change

> Accessing an object designated by a `volatile` lvalue (3.10), ...are all *side effects*...

3.2 One definition rule ¶4: no change

- an lvalue-to-rvalue conversion is applied to an lvalue referring to an object of type T...
- ...
- an lvalue of type T is assigned to...

3.8 Object lifetime: no change (rules are the same for both kinds of lvalues)

4 Standard conversions ¶3:

> An expression `e` can be *implicitly converted* to a type `T`... The result is an lvalue if `T` is ~~an lvalue~~ **a** reference type (8.3.2) and an rvalue otherwise...

4.1 Lvalue-to-rvalue conversion: no change (applies to both kinds of lvalues)

5.1.1 General ¶1, 3, 6, 7:

> ...A string literal is ~~an~~ **a non-rref** lvalue; ...
>
> ...The result is the entity denoted by the identifier, *qualified-id*, *operator-function-id* or *literal-operator-id*. The result is ~~an~~ **a non-rref** lvalue if the entity is a function or variable **and an rvalue otherwise**...

...The result is ~~an~~ **a non-rref** lvalue if the entity is a function, variable, or data member **and an rvalue otherwise**... The result is ~~an~~ **a non-rref** lvalue if the member is a static member function or a data member **and an rvalue otherwise**.

...The result is ~~an~~ **a non-rref** lvalue if the member is a function or a variable **and an rvalue otherwise**.

5.1.2 Lambda expressions ¶18:

```
decltype((x)) y2 = y1;   // y2 has type float const& because this lambda
                         // is not mutable and x is an a non-rref lvalue
```

5.2.1 Subscripting ¶1:

...The result is ~~an~~ **a non-rref** lvalue of type "T." ...

5.2.2 Function call ¶10:

A function call is an ~~lvalue if and only if the result type is an lvalue reference~~ **rref lvalue if the result type is an rvalue reference, a non-rref lvalue if the result type is an lvalue reference, and an rvalue otherwise**.

5.2.5 Class member access ¶3-4:

Footnote 64: Note that if `E1` has the type "pointer to class `X`," then `(*(E1))` is ~~an~~ **a non-rref** lvalue.

If `E2` is declared to have type "reference to `T`," then `E1.E2` is ~~an~~ **a non-rref** lvalue; the type of `E1.E2` is `T`. Otherwise, one of the following rules applies.

- If `E2` is a static data member and the type of `E2` is `T`, then `E1.E2` is an **a non-rref** lvalue; the expression designates the named member of the class. The type of `E1.E2` is `T`.
- ...If `E1` is an lvalue, then `E1.E2` is an lvalue **of the same kind (rref or non-rref);** otherwise, it is an rvalue...
- If `E2` is a (possibly overloaded) member function...
  - If it refers to a static member function and the type of `E2` is "function of parameter-type-list returning `T`", then `E1.E2` is ~~an~~ **a non-rref** lvalue

5.2.6 Increment and decrement ¶1:

...The operand shall be a modifiable **non-rref** lvalue...

5.2.7 Dynamic cast ¶2, 5:

...If `T` is an lvalue reference type, `v` shall be ~~an~~ **a non-rref** lvalue of a complete class type, and the result is ~~an~~ **a non-rref** lvalue of the type referred to by `T`. If `T` is an rvalue reference type, `v`

shall be an expression having a complete class type, and the result is an ~~rvalue~~ **rref lvalue** of the type referred to by `T`.

...The result is ~~an~~ **a non-rref** lvalue if `T` is an lvalue reference, or an ~~rvalue~~ **rref lvalue** if `T` is an rvalue reference.

5.2.8 Type identification ¶1-2:

The result of a `typeid` expression is ~~an~~ **a non-rref** lvalue...

When `typeid` is applied to an lvalue expression... *[No change: applies to both kinds of lvalue.]*

5.2.9 Static cast ¶1-2:

...If `T` is an lvalue reference type, the result is ~~an~~ **a non-rref** lvalue; **if `T` is an rvalue reference type, the result is an rref lvalue;** otherwise, the result is an rvalue...

~~An~~ **A non-rref** lvalue of type “*cv1* `B`,” where `B` is a class type, can be cast to type “reference to *cv2* `D`,” where `D` is a class derived (Clause 10) from `B`, if a valid standard conversion from “pointer to `D`” to “pointer to `B`” exists (4.10), *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*, and `B` is neither a virtual base class of `D` nor a base class of a virtual base class of `D`. The result has type “*cv2* `D`.” An rvalue **or rref lvalue** of type “*cv1* `B`” may be cast to type “rvalue reference to *cv2* `D`” with the same constraints as for ~~an~~ **a non-rref** lvalue of type “*cv1* `B`.” ...

5.2.10 Reinterpret cast ¶1, 11:

...If `T` is an lvalue reference type, the result is ~~an~~ **a non-rref** lvalue; if `T` is an rvalue reference type, the result is an ~~rvalue~~ **rref lvalue**; otherwise, the result is an rvalue and the lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are performed on the the expression `v`.

...The result is ~~an~~ **a non-rref** lvalue for lvalue references or an ~~rvalue~~ **rref lvalue** for rvalue references.

5.2.11 Const cast ¶1, 4:

...If `T` is an lvalue reference type, the result is ~~an~~ **a non-rref** lvalue; if `T` is an rvalue reference type, the result is an ~~rvalue~~ **rref lvalue**; otherwise, the result is an rvalue and the lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are performed on the the expression `v`.

~~An~~ **A non-rref** lvalue of type `T1` can be explicitly converted to ~~an~~ **a non-rref** lvalue of type `T2` using the cast `const_cast<T2&>`... Similarly, for two object types `T1` and `T2`, an expression of type `T1` can be explicitly converted to an ~~rvalue~~ **rref lvalue** of type `T2` using the cast `const_cast<T2&&>`...

5.3.1 Unary operators ¶1, 3:

> The unary `*` operator performs *indirection*... the result is ~~an~~ **a non-rref** lvalue referring to...
>
> The result of the unary `&` operator is a pointer to its operand. The operand shall be ~~an~~ **a non-rref** lvalue or a *qualified-id*...

5.3.2 Increment and decrement ¶1:

> The operand of prefix ++... shall be a modifiable **non-rref** lvalue... The result... is ~~an~~ **a non-rref** lvalue...

5.4 Explicit type conversion (cast notation) ¶1:

> ...The result is ~~an~~ **a non-rref** lvalue if `T` is an lvalue reference type **and an rref lvalue if `T` is an rvalue reference type;** ~~,~~ otherwise the result is an rvalue.

5.5 Pointer-to-member operators ¶6:

> ...The result of a `.*` expression is an lvalue only if its first operand is an lvalue and its second operand is a pointer to data member**; the resulting lvalue is of the same kind (rref or non-rref) as the first operand**. The result of an `->*` expression is ~~an~~ **a non-rref** lvalue only if its second operand is a pointer to data member**; otherwise, the result is an rvalue**...

5.16 Conditional operator ¶3-4:

> Otherwise, if the second and third operand have different types, **or if one is an rref lvalue and the other is a non-rref lvalue,** and either has (possibly cv-qualified) class type or if both are lvalues of the same type except for cv-qualification, an attempt is made to convert each of those operands to the type of the other.
>
> - If `E2` is ~~an~~ **a non-rref** lvalue: `E1` can be converted to match `E2` if `E1` can be implicitly converted (Clause 4) to the type “lvalue reference to `T2`”, subject to the constraint that in the conversion the reference must bind directly (8.5.3) to `E1`.
>
> - If `E2` is an rvalue **or rref lvalue**, or if the conversion above cannot be done...
>
> If the second and third operands are lvalues and have the same type, the result is of that type and is an lvalue **of the same kind (rref or non-rref)** and it is a bit-field if the second or the third operand is a bit-field, or if both are bit-fields.

5.17 Assignment and compound assignment operators ¶1:

> ...All require a modifiable **non-rref** lvalue as their left operand and return ~~an~~ **a non-rref** lvalue referring to the left operand...

5.18 Comma operator ¶1:

> ...the result is an lvalue **of the same kind (rref or non-rref)** if its right operand is an lvalue **and an rvalue otherwise**, and is a bit-field if its right operand is an lvalue and a bit-field.

6.6.3 The `return` statement ¶2:

> ...[*Note:* A copy operation associated with a return statement may be elided or considered as an ~~rvalue~~ **rref lvalue** for the purpose of overload resolution in selecting a constructor (12.8). — *end note*]...

7.1.6.2 Simple type specifiers ¶4 bullet 3:

- otherwise, if `e` is ~~an~~ **a non-rref** lvalue, `decltype(e)` is `T&`, where `T` is the type of `e`;

8.5.3 References ¶5:

- If the reference is an lvalue reference and the initializer expression
  - is ~~an~~ **a non-rref** lvalue (but is not a bit-field), and “*cv1* `T1`” is reference-compatible with “*cv2* `T2`,” or
  - has a class type (i.e., `T2` is a class type), where `T1` is not reference-related to `T2`, and can be implicitly converted to ~~an~~ **a non-rref** lvalue of type “*cv3* `T3`,” ...
- Otherwise, the reference shall be an lvalue reference to a non-volatile const type (i.e., *cv1* shall be `const`), or the reference shall be an rvalue reference and the initializer expression shall be an rvalue **or an rref lvalue**...
  - if `T1` and `T2` are class types and
    - the initializer expression is an rvalue **or an rref lvalue** and “*cv1* `T1`” is reference-compatible with “*cv2* `T2`”, or
    - `T1` is not reference-related to `T2` and the initializer expression can be implicitly converted to an rvalue **or an rref lvalue** of type “*cv3* `T3`” (this conversion is selected by enumerating the applicable conversion functions (13.3.1.6) and choosing the best one through overload resolution (13.3)),

    then the reference is bound to the initializer expression ~~rvalue~~ **object** in the first case and to the object that is the result of the conversion in the second case...
  - If the initializer expression is an rvalue **or an rref lvalue**, with `T2` an array type, and “*cv1* `T1`” is reference-compatible with “*cv2* `T2`,” the reference is bound to the object represented by the rvalue (see 3.10) **or designated by the lvalue**.

12.2 Temporary objects ¶1:

> Temporaries of class type are created in various contexts: binding an rvalue **or rref lvalue** to a reference (8.5.3)...

12.8 Copying class objects ¶18:

> When the criteria for elision of a copy operation are met and the object to be copied is

designated by ~~an~~ **a non-rref** lvalue, overload resolution to select the constructor for the copy is first performed as if the object were designated by an ~~rvalue~~ **rref lvalue**. If overload resolution fails, or if the type of the first parameter of the selected constructor is not an rvalue reference to the object's type (possibly cv-qualified), overload resolution is performed again, considering the object as ~~an~~ **a non-rref** lvalue...

13.3 Overload resolution ¶2 bullet 7:

- invocation of a conversion function for conversion to ~~an~~ **a non-rref** lvalue or class rvalue to which a reference (8.5.3) will be directly bound (13.3.1.6).

13.3.1 Candidate functions and argument lists ¶5:

...For non-static member functions declared without a *ref-qualifier*, an additional rule applies:

- even if the implicit object parameter is not const-qualified, an rvalue temporary **or rref lvalue** can be bound to the parameter as long as in all other respects the ~~temporary~~ **argument** can be converted to the type of the implicit object parameter. [*Note:* The fact that such ~~a temporary~~ **an argument** is an rvalue **or rref lvalue** does not affect the ranking of implicit conversion sequences (13.3.3.2). —*end note*]

13.3.1.4 Copy-initialization of class by user-defined conversion ¶1 bullet 2:

- ...Conversion functions that return "reference to `X`" return lvalues ~~or rvalues, depending on the type of reference,~~ of type `X` and are therefore considered to yield `X` for this process of selecting candidate functions.

13.3.1.5 Initialization by conversion function ¶1 bullet 1:

- ...Conversion functions that return "reference to *cv2* `X`" return lvalues ~~or rvalues, depending on the type of reference,~~ of type "*cv2* `X`" and are therefore considered to yield `X` for this process of selecting candidate functions.

13.3.1.6 Initialization by conversion function for direct reference binding ¶1:

Under the conditions specified in 8.5.3, a reference can be bound directly to ~~an~~ **a non-rref** lvalue or class rvalue that is the result of applying a conversion function...

13.3.2 Viable functions ¶3:

...the implicit conversion sequence includes the operation of binding the reference, and the fact that an lvalue reference to non-const cannot be bound to an rvalue **or rref lvalue** can affect the viability of the function (see 13.3.3.1.4).

13.3.3.1.4 Reference binding ¶3:

A standard conversion sequence cannot be formed if it requires binding an lvalue reference to non-`const` to an rvalue **or an rref lvalue**...

13.3.3.2 Ranking implicit conversion sequences ¶3 bullet 1 sub-bullet 4:

- `S1` and `S2` are reference bindings (8.5.3) and neither refers to an implicit object parameter of a non-static member function declared without a ref-qualifier, and `S1` binds an rvalue reference to an rvalue **or rref lvalue** and `S2` binds an lvalue reference.

14.4.2 Template non-type arguments ¶5 bullet 3:

- ...The *template-parameter* is bound directly to the template-argument, which shall be ~~an~~ **a non-rref** lvalue.

14.9.2.1 Deducing template arguments from a function call ¶3:

...If `P` is an rvalue reference to a cv-unqualified template parameter and the argument is ~~an~~ **a non-rref** lvalue, the type “lvalue reference to `A`” is used in place of `A` for type deduction...

15.1 Throwing an exception ¶3:

...The temporary is ~~an~~ **a non-rref** lvalue and is used to initialize the variable named in the matching handler (15.3)...