Document: PL22.16/10-0045 = WG21 N3055
Author: William M. Miller
Edison Design Group
Date: 2010-03-12
Revision: 6

A Taxonomy of Expression Value Categories

Revision History:

Revision 6 (PL22.16/10-0045 = WG21 N3055):

- Addressed comments from CWG review during morning session of 2010-03-12.

Revision 5 (PL22.16/10-0045 = WG21 D3055):

- Addressed comments from CWG review during morning session of 2010-03-11.
- Changed the title to reflect the new approach taken by the paper.
- Defined the term *value category* and used it at various places.

Revision 4 (PL22.16/10-0045 = WG21 D3055):

- Renamed categories per discussion in CWG on 2010-03-10, adding and removing citations to be modified.

Revision 3 (PL22.16/10-0045 = WG21 D3055):

- Changed the terminology from “rref lvalue” and “non-rref lvalue” to *x*value and added the inheritance diagram illustrating their relationships.
- Changed the normative text in 5¶6 into a note.
- Extracted the treatment of initializing rvalue references to function type into a separate sub-bullet in 8.5.3¶5.
- Changed 8.5.3¶5 to handle the case of binding a non-class rvalue reference to the result of a conversion function for a class object initializer.
- Removed changes to 13.3¶2 bullet 7 and 13.3.1.6¶1.

Revision 2 (PL22.16/10-0020 = WG21 N3030):

- Clarified that rvalue references to function types produce non-rref lvalues (previously stated in the general wording in 5¶6 but inadvertently contradicted in the detailed wording of 5.2.2, 5.2.9, etc.).
- Changed 8.5.3¶5 to allow rvalue references to function types to bind to function lvalues.

Revision 1 (PL22.16/09-0200 = WG21 N3010):

- Added section I (Background).
- First numbered document.

Revision 0:

- Unnumbered document reviewed by CWG during the Santa Cruz (October, 2009) meeting.

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, the term "rvalue" implies 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` non-class 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 resolution for 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 embodied in earlier versions of this paper.

After extensive discussions in the Core Working Group at the Pittsburgh (March 8-13, 2010) meeting, a third approach was suggested and is adopted in this version of the paper, namely, to establish

terminology for all the categories into which expressions can be divided and to change the various mentions of "lvalue" and "rvalue" in the Working Paper to use the more specific terms where needed. The terms "lvalue" and "rvalue" are retained and are intended to reflect their meanings in the Library clauses, and additional terms are defined for the remaining categories.

(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:

- The traditional dichotomy between lvalues and rvalues is subdivided into five overlapping categories. The result of binding an rvalue reference to an object is called an "xvalue" (inspired by the fact that the canonical use of an rvalue reference is to move the resources of a value that is "eXpiring," i.e., at the end of its lifetime). Traditional lvalues retain that name. Together, xvalues and lvalues are called "glvalues" ("generalized" lvalues). Rvalues, as currently known in the core language clauses, are renamed to "prvalues" ("pure" rvalues). Together, prvalues and xvalues are called "rvalues," reflecting the use of that term in the library clauses and the intended uses in reference binding, overload resolution, and template argument deduction. This taxonomy provides the means to refer collectively to expressions that designate objects in memory or functions ("glvalues") and to expressions that can be used to initialize rvalue references (the new "rvalues"), as well as to designate the specific kinds of values when that is necessary.
- Xvalues are created by three kinds of expressions:
  - function calls (explicit or implicit) that return rvalue references to objects;
  - casts to rvalue references to objects;
  - class member access and pointer-to-data-member dereference expressions where the object expression is an xvalue.

  As a result, function values are always lvalues and there are no function 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 N3035. Additional edits will be required to some pending resolutions to various core language issues, but these are not part of this document.

3.10 Lvalues and rvalues (paragraphs 1-10, 15)

~~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();
```

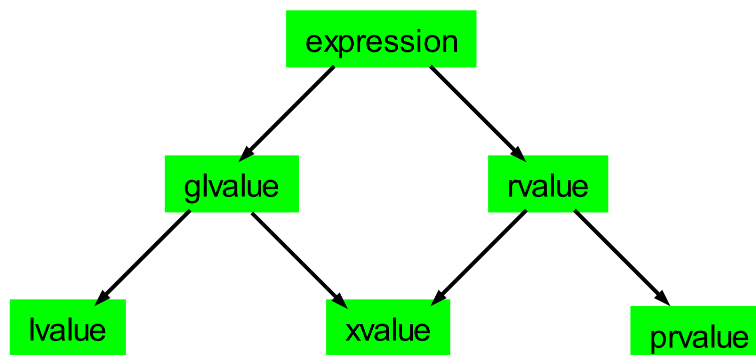
~~yields an lvalue, so the call `f()` is an lvalue expression. —*end example*] —*end note*]~~

~~*Note:* some built-in operators expect lvalue operands. [*Example:* built-in assignment operators all expect their left-hand operands to be 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 whether it yields an lvalue. —*end note*]~~

~~The result of calling a function that does not return an lvalue 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 holds a temporary object resulting from a cast to a type other than an lvalue reference type is an rvalue (this includes the explicit creation of an object using functional notation (5.2.3)).~~

Expressions are categorized according to the following taxonomy:



- **An *lvalue* (so-called, historically, because lvalues could appear on the left-hand side of an assignment expression) designates a function or an object. [*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 result of calling a function whose return type is an lvalue reference is an lvalue. —*end example*]**

- **An *xvalue* (an "eXpiring" value) also refers to an object, usually near the end of its lifetime (so that its resources may be moved, for example). An xvalue is the result of certain kinds of expressions involving rvalue references (8.3.2). [*Example:* The result of calling a function whose return type is an rvalue reference is an xvalue. —*end example*]**

- **A *glvalue* ("generalized" lvalue) is an lvalue or an xvalue.**

- **An *rvalue* (so-called, historically, because rvalues could appear on the right-hand side of an assignment expression) is an xvalue, a temporary object (12.2) or sub-object thereof, or a value that is not associated with an object.**

- **A *prvalue* ("pure" rvalue) is an rvalue that is not an xvalue. [*Example:* The result of calling a function whose return type is not a reference is a prvalue. The value of a literal such as `12`, `7.3e5`, or `true`, is also a prvalue. —*end example*]**

Every expression belongs to exactly one of the fundamental classifications in this taxonomy: lvalue, xvalue, or prvalue. This property of an expression is called its *value category*. [*Note:* The discussion of each built-in operator in Clause 5 indicates the category of the value it yields and the value categories of the operands it expects. For example, the built-in assignment operators expect that the left operand is an lvalue and that the right operand is a prvalue and yield an lvalue as the result. User-defined operators are functions, and the categories of values they expect and yield are determined by their parameter and return types. —*end note*]

Whenever ~~an lvalue~~ **a glvalue** appears in a context where ~~an rvalue~~ **a prvalue** is expected, the ~~lvalue~~ **glvalue** is converted to ~~an rvalue~~ **a prvalue**; see 4.1, 4.2, and 4.3.

The discussion of reference initialization in 8.5.3 and of temporaries in 12.2 indicates the behavior of lvalues and rvalues in other significant contexts.

Class ~~rvalues~~ **prvalues** can have cv-qualified types; non-class ~~rvalues~~ **prvalues** always have cv-unqualified types. ~~Rvalues~~ **Prvalues** shall always have complete types or the `void` type; in addition to these types, ~~lvalues~~ **glvalues** can also have incomplete types.

An lvalue for an object is necessary in order to modify the object except that an rvalue of class type can also be used to modify its referent under certain circumstances. [*Example:* a member function called for an object (9.3) can modify the object. —*end example*]

...

If a program attempts to access the stored value of an object through ~~an lvalue~~ **a glvalue** of other than one of the following types the behavior is undefined[52]

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~~;~~ **and** the expression designates the object or function denoted

by the ~~lvalue~~ reference, and the expression is an lvalue **or an xvalue, depending on the expression**.

~~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; otherwise, it is an lvalue.~~ [Note: **An expression is an xvalue if it is:**

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

In general, the effect of this rule is that named rvalue references are treated as lvalues and unnamed rvalue references **to objects** are treated as ~~rvalues~~ **xvalues; rvalue references to functions are treated as lvalues whether named or not**. —*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~~ **xvalues** ~~of type A~~. The expression `ar` is an lvalue **of type A**. —*end example*]

1.3.4:

dynamic type
the type of the most derived object (1.8) to which the ~~lvalue~~ **glvalue** denoted by ~~an lvalue~~ **a glvalue** expression refers. [*Example:* if a pointer (8.3.1) `p` whose static type is "pointer to class `B`" is pointing to an object of class `D`, derived from `B` (Clause 10), the dynamic type of the expression `*p` is "`D`." References (8.3.2) are treated similarly. —*end example*] The dynamic type of ~~an rvalue~~ **a prvalue** expression is its static type.

1.9 Program execution ¶12:

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

Evaluation of an expression (or a sub-expression) in general includes both value computations (including determining the identity of an object for ~~lvalue~~ **glvalue** evaluation and fetching a value previously assigned to an object for ~~rvalue~~ **prvalue** evaluation) and initiation of side effects...

2.14.6 Boolean literals ¶1:

...Such literals are ~~rvalues~~ **prvalues** and have type `bool`.

2.14.7 Pointer literals ¶1:

...It is ~~an rvalue~~ **a prvalue** of type `std::nullptr_t`.

3 Basic concepts ¶1:

[*Note:* this Clause presents the basic concepts of the C++ language. It explains the difference between an object and a name and how they relate to the ~~notion of an lvalue~~ **value categories for expressions**. It introduces the concepts...

3.2 One definition rule ¶4:

- ...
- an lvalue-to-rvalue conversion is applied to ~~an lvalue~~ a glvalue referring to an object of type `T`...
- ...

3.8 Object lifetime ¶6:

Similarly, before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any ~~lvalue which~~ **glvalue that** refers to the original object may be used but only in limited ways. Such ~~an lvalue~~ **a glvalue** refers to allocated storage (3.7.4.2), and using the properties of the ~~lvalue which~~ **glvalue that** do not depend on its value is well-defined. The program has undefined behavior if:

- an lvalue-to-rvalue conversion (4.1) is applied to such ~~an lvalue~~ **a glvalue**,
- the ~~lvalue~~ **glvalue** is used to access a non-static data member or call a non-static member function of the object, or
- the ~~lvalue~~ **glvalue** is implicitly converted (4.10) to a reference to a base class type, or
- the ~~lvalue~~ **glvalue** is used as the operand of a `static_cast` (5.2.9) except when the conversion is ultimately to *cv* `char&` or *cv* `unsigned char&`, or
- the ~~lvalue~~ **glvalue** is used as the operand of a `dynamic_cast` (5.2.7) or as the operand of `typeid`.

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 reference type **or an rvalue reference to function type** (8.3.2)**, an xvalue if `T` is an rvalue reference to object type,** and ~~an rvalue~~ **a prvalue** otherwise. The expression `e` is used as ~~an lvalue~~ **a glvalue** if and only if the initialization uses it as ~~an lvalue~~ **a glvalue**.

4.1 Lvalue-to-rvalue conversion ¶1-2:

~~An lvalue~~ **A glvalue** (3.10) of a non-function, non-array type `T` can be converted to ~~an rvalue~~ **a prvalue**. **[*Footnote:* For historical reasons, this conversion is called the "lvalue-to-rvalue" conversion, even though that name does not accurately reflect the taxonomy of expressions described in 3.10. *—end footnote]*** If `T` is an incomplete type, a program that necessitates this conversion is ill-formed. If the object to which the ~~lvalue~~ **glvalue** refers is not an object of type `T` and is not an object of a type derived from `T`, or if the object is uninitialized, a program that necessitates this conversion has undefined behavior. If `T` is a non-class type, the type of the ~~rvalue~~ **prvalue** is the cv-unqualified version of `T`. Otherwise, the type of the ~~rvalue~~ **prvalue** is `T`.[54] [*Footnote:* In C++ class ~~rvalues~~ **prvalues** can have cv-qualified types (because they are objects). This differs from ISO C, in which non-lvalues never have cv-qualified types. *—end footnote*]

When an lvalue-to-rvalue conversion occurs in an unevaluated operand or a subexpression thereof (Clause 5) the value contained in the referenced object is not accessed. Otherwise, if the ~~lvalue~~ **glvalue** has a class type, the conversion copy-initializes a temporary of type `T` from the ~~lvalue~~ **glvalue** and the result of the conversion is ~~an rvalue~~ **a prvalue** for the temporary. Otherwise, if the ~~lvalue~~ **glvalue** has (possibly cv-qualified) type `std::nullptr_t`, the ~~rvalue~~ **prvalue** result is a null pointer constant (4.10). Otherwise, the value contained in the object indicated by the ~~lvalue~~ **glvalue** is the ~~rvalue~~ **prvalue** result.

4.2 Array-to-pointer conversion ¶1:

An lvalue or rvalue of type "array of `N` `T`" or "array of unknown bound of `T`" can be converted to ~~an rvalue~~ **a prvalue** of type "pointer to T". The result is a pointer to the first element of the array.

4.3 Function-to-pointer conversion ¶1:

An lvalue of function type T can be converted to ~~an rvalue~~ **a prvalue** of type "pointer to T ." The result is a pointer to the function.[55]

4.4 Qualification conversions ¶1-2:

~~An rvalue~~ **A prvalue** of type "pointer to *cv1* `T`" can be converted to ~~an rvalue~~ **a prvalue** of type "pointer to *cv2* `T`" if "*cv2* `T`" is more cv-qualified than "*cv1* `T`".

~~An rvalue~~ **A prvalue** of type "pointer to member of `X` of type *cv1* `T`" can be converted to ~~an-rvalue~~ **a prvalue** of type "pointer to member of `X` of type *cv2* `T`" if "*cv2* `T`" is more cv-qualified

than “*cv1* `T`”.

4.5 Integral promotions ¶1-6:

> ~~An rvalue~~ **A prvalue** of an integer type... can be converted to ~~an rvalue~~ **a prvalue** of type `int` if...; otherwise, the source ~~rvalue~~ **prvalue** can be converted to ~~an rvalue~~ **a prvalue** of type `unsigned int`.
>
> ~~An rvalue~~ **A prvalue** of type `char16_t`, `char32_t`, or `wchar_t` (3.9.1) can be converted to ~~an rvalue~~ **a prvalue** of the first of the following types... ~~an rvalue~~ **a prvalue** of type `char16_t`, `char32_t`, or `wchar_t` can be converted to ~~an rvalue~~ **a prvalue** of its underlying type.
>
> ~~An rvalue~~ **A prvalue** of an unscoped enumeration type whose underlying type is not fixed (7.2) can be converted to ~~an rvalue~~ **a prvalue** of the first of the following types... ~~an rvalue~~ **a prvalue** of an unscoped enumeration type can be converted to ~~an rvalue~~ **a prvalue** of the extended integer type...
>
> ~~An rvalue~~ **A prvalue** of an unscoped enumeration type whose underlying type is fixed (7.2) can be converted to ~~an rvalue~~ **a prvalue** of its underlying type. Moreover, if integral promotion can be applied to its underlying type, ~~an rvalue~~ **a prvalue** of an unscoped enumeration type whose underlying type is fixed can also be converted to ~~an rvalue~~ **a prvalue** of the promoted underlying type.
>
> ~~An rvalue~~ **A prvalue** for an integral bit-field (9.6) can be converted to ~~an rvalue~~ **a prvalue** of type `int`...
>
> ~~An rvalue~~ **A prvalue** of type `bool` can be converted to ~~an rvalue~~ **a prvalue** of type `int`...

4.6 Floating point promotion ¶1:

> ~~An rvalue~~ **A prvalue** of type `float` can be converted to ~~an rvalue~~ **a prvalue** of type `double`...

4.7 Integral conversions ¶1:

> ~~An rvalue~~ **A prvalue** of an integer type can be converted to ~~an rvalue~~ **a prvalue** of another integer type. ~~An rvalue~~ **A prvalue** of an unscoped enumeration type can be converted to ~~an rvalue~~ **a prvalue** of an integer type.

4.8 Floating point conversions ¶1:

> ~~An rvalue~~ **A prvalue** of floating point type can be converted to ~~an rvalue~~ **a prvalue** of another floating point type...

4.9 Floating-integral conversions ¶1-2:

> ~~An rvalue~~ **A prvalue** of a floating point type can be converted to ~~an rvalue~~ **a prvalue** of an integer type...
>
> ~~An rvalue~~ **A prvalue** of an integer type or of an unscoped enumeration type can be converted to

> ~~an rvalue~~ **a prvalue** of a floating point type...

4.10 Pointer conversions ¶1-3:

> A null pointer constant is an integral constant expression (5.19) ~~rvalue~~ **prvalue** of integer type that evaluates to zero or ~~an rvalue~~ **a prvalue** of type `std::nullptr_t`... A null pointer constant of integral type can be converted to ~~an rvalue~~ **a prvalue** of type `std::nullptr_t`. [*Note:* The resulting ~~rvalue~~ **prvalue** is not a null pointer value. —*end note*]
>
> ~~An rvalue~~ **A prvalue** of type "pointer to *cv* `T`," where `T` is an object type, can be converted to ~~an rvalue~~ **a prvalue** of type "pointer to *cv* `void`"...
>
> ~~An rvalue~~ **A prvalue** of type "pointer to *cv* `D`", where `D` is a class type, can be converted to ~~an rvalue~~ **a prvalue** of type "pointer to *cv* `B`"...

4.11 Pointer to member conversions ¶2:

> ~~An rvalue~~ **A prvalue** of type "pointer to member of `B` of type *cv* `T`", where `B` is a class type, can be converted to ~~an rvalue~~ **a prvalue** of type "pointer to member of `D` of type *cv* `T`"...

4.12 Boolean conversions ¶1:

> ~~An rvalue~~ **A prvalue** of arithmetic, unscoped enumeration, pointer, or pointer to member type can be converted to ~~an rvalue~~ **a prvalue** of type `bool`. A zero value, null pointer value, or null member pointer value is converted to `false`; any other value is converted to `true`. ~~An rvalue~~ **A prvalue** of type `std::nullptr_t` can be converted to ~~an rvalue~~ **a prvalue** of type `bool`; the resulting value is `false`.

5. Expressions ¶2, 9:

> [*Note:* ...Overloaded operators obey the rules for syntax specified in Clause 5, but the requirements of operand type, ~~lvalue~~ **value category**, and evaluation order are replaced by the rules for function call...
>
> Whenever ~~an lvalue~~ **a glvalue** expression appears as an operand of an operator that expects ~~an rvalue~~ **a prvalue** for that operand, the lvalue-to-rvalue (4.1), array-to-pointer (4.2), or function-to-pointer (4.3) standard conversions are applied to convert the expression to ~~an rvalue~~ **a prvalue**. [*Note:* because cv-qualifiers are removed from the type of an expression of non-class type when the expression is converted to ~~an rvalue~~ **a prvalue**, an lvalue expression of type `const int` can, for example, be used where ~~an rvalue~~ **a prvalue** expression of type `int` is required. —*end note*]

5.1.1 General ¶1-3, 6, 7-8:

> ...A string literal is an lvalue; all other literals are ~~rvalues~~ **prvalues**.
>
> The keyword `this` names a pointer... The expression is ~~an rvalue~~ **a prvalue**.
>
> ...The result is the entity denoted by the identifier, *qualified-id*, *operator-function-id* or *literal-*

operator-id. The result is an lvalue if the entity is a function or variable **and a prvalue otherwise**...

...The result is an lvalue if the entity is a function, variable, or data member **and a prvalue otherwise**... The result is an lvalue if the member is a static member function or a data member **and a prvalue otherwise**.

...The result is an lvalue if the member is a function or a variable **and a prvalue otherwise**.

A *nested-name-specifier* that names an enumeration... The result is ~~an rvalue~~ **a prvalue**.

5.1.2 Lambda expressions ¶2, 17:

The evaluation of a *lambda-expression* results in ~~an rvalue~~ **a prvalue** temporary (12.2)...

...[*Note:* the cast ensures that the transformed expression is ~~an rvalue~~ **a prvalue**. —*end note*]

5.2.2 Function call ¶10:

A function call is an lvalue ~~if and only if the result type is an lvalue reference~~ **if the result type is an lvalue reference or an rvalue reference to function type, an xvalue if the result type is an rvalue reference to object type, and a prvalue otherwise**.

5.2.3 Explicit type conversion (functional notation) ¶1-3:

...with the result being the value of `t` as ~~an rvalue~~ **a prvalue**.

The expression `T()`... creates ~~an rvalue~~ **a prvalue** of the specified type... [*Note:* if `T` is a non-class type that is cv-qualified, the *cv-qualifier*s are ignored when determining the type of the resulting ~~rvalue~~ **prvalue** (3.10). —*end note*]

...its value is that temporary object as ~~an rvalue~~ **a prvalue**.

5.2.5 Class member access ¶3-4:

...Abbreviating *object-expression.id-expression* as `E1.E2`, then the type and ~~lvalue properties~~ **value category** of this expression are determined as follows...

If `E2` is declared to have type "reference to `T`," then `E1.E2` is an 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 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; **if `E1` is an xvalue, then `E1.E2` is an xvalue;** otherwise, it is ~~an rvalue~~ **a prvalue**...
- If `E2` is a (possibly overloaded) member function...
    - If it refers to a static member function...

- Otherwise, if `E1.E2` refers to a non-static member function... then `E1.E2` is ~~an rvalue~~ **a prvalue**...

- If `E2` is a nested type...
- If `E2` is a member enumerator and the type of `E2` is `T`, the expression `E1.E2` is ~~an rvalue~~ **a prvalue**...

5.2.6 Increment and decrement ¶1:

...The result is ~~an rvalue~~ **a prvalue**...

5.2.7 Dynamic cast ¶2, 5:

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

...The result is an lvalue if `T` is an lvalue reference, or an ~~rvalue~~ **xvalue** if `T` is an rvalue reference...

5.2.8 Type identification ¶2-3, 5:

When `typeid` is applied to ~~an lvalue~~ **a glvalue** expression whose type is a polymorphic class type (10.3), the result refers to a `std::type_info` object representing the type of the most derived object (1.8) (that is, the dynamic type) to which the ~~lvalue~~ **glvalue** refers. If the ~~lvalue~~ **glvalue** expression is obtained by applying the unary `*` operator to a pointer[67] and the pointer is a null pointer value (4.10), the `typeid` expression throws the `std::bad_typeid` exception (18.7.3).

When `typeid` is applied to an expression other than ~~an lvalue~~ **a glvalue** of a polymorphic class type, the result refers to a `std::type_info` object representing the static type of the expression.

...

The top-level cv-qualifiers of the ~~lvalue~~ **glvalue** expression or the type-id that is the operand of typeid are always ignored...

5.2.9 Static cast ¶1-3, 11-13:

...If `T` is an lvalue reference type **or an rvalue reference to function type**, the result is an lvalue; **if `T` is an rvalue reference to object type, the result is an xvalue;** otherwise, the result is ~~an rvalue~~ **a prvalue**...

An 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~~ **xvalue** of type “*cv1* `B`” may be cast to type “rvalue reference to *cv2* `D`” with the same constraints as for an lvalue of type “*cv1* `B`.” ...

~~An lvalue~~ **A glvalue** of type “*cv1* `T1`” can be cast to type “rvalue reference to *cv2* `T2`” if “*cv2* `T2`” is reference-compatible with “*cv1* `T1`” (8.5.3)...

Otherwise, an expression `e` can be explicitly converted to a type `T`... The expression `e` is used as ~~an lvalue~~ **a glvalue** if and only if the initialization uses it as ~~an lvalue~~ **a glvalue**.

~~An rvalue~~ **A prvalue** of type “pointer to *cv1* `B`,” where `B` is a class type, can be converted to ~~an rvalue~~ **a prvalue** of type “pointer to *cv2* `D`,” where...If the ~~rvalue~~ **prvalue** of type “pointer to *cv1* `B`” points to a `B`...

~~An rvalue~~ **A prvalue** of type “pointer to member of `D` of type *cv1* `T`” can be converted to ~~an rvalue~~ **a prvalue** of type “pointer to member of `B`” of type...

~~An rvalue~~ **A prvalue** of type “pointer to *cv1* `void`” can be converted to ~~an rvalue~~ **a prvalue** of type...

5.2.10 Reinterpret cast ¶1, 6-8, 10-11:

...If `T` is an lvalue reference type **or an rvalue reference to function type**, the result is an lvalue; if `T` is an rvalue reference **to object** type, the result is an ~~rvalue~~ **xvalue**; otherwise, the result is ~~an rvalue~~ **a prvalue** 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`...

...Except that converting ~~an rvalue~~ **a prvalue** of type “pointer to `T1`” to the type...

... When ~~an rvalue~~ **a prvalue** `v` of type “pointer to `T1`” is converted... Converting ~~an rvalue~~ **a prvalue** of type “pointer to `T1`” to the type...

...converting ~~an rvalue~~ **a prvalue** of one type to the other type...

~~An rvalue~~ **A prvalue** of type “pointer to member of `X` of type `T1`” can be explicitly converted to ~~an rvalue~~ **a prvalue** of type “pointer to member of `Y` of type `T2`” if `T1` and `T2` are both function types or both object types.[70] The null member pointer value (4.11) is converted to the null member pointer value of the destination type. The result of this conversion is unspecified, except in the following cases:

- converting ~~an rvalue~~ **a prvalue** of type “pointer to member function” to a different pointer to member function type and back to its original type yields the original pointer to member value.
- converting ~~an rvalue~~ **a prvalue** of type “pointer to data member of `X` of type `T1`” to the type “pointer to data member of `Y` of type `T2`” (where the alignment requirements of `T2` are no stricter than those of `T1`) and back to its original type yields the original pointer

to member value.

...The result is an lvalue for **an** lvalue reference~~s~~ **type or an rvalue reference to function type** ~~or~~ **and** an ~~rvalue~~ **xvalue** for **an** rvalue reference~~s~~ **to object type**...

5.2.11 Const cast ¶1, 3-4, 9-10:

...If `T` is an lvalue reference type **or an rvalue reference to function type**, the result is an lvalue; if `T` is an rvalue reference **to object** type, the result is an ~~rvalue~~ **xvalue**; otherwise, the result is ~~an rvalue~~ **a prvalue** 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 rvalue~~ **a prvalue** of type `T1` may be explicitly converted...

...Similarly, for two object types `T1` and `T2`, an expression of type `T1` can be explicitly converted to an ~~rvalue~~ **xvalue** of type `T2` using the cast `const_cast<T2&&>`...

Casting from an lvalue of type `T1` to an lvalue of type `T2` using an lvalue reference cast or casting from an expression of type `T1` to an ~~rvalue~~ **xvalue** of type `T2` using an rvalue reference cast casts away constness if a cast from ~~an rvalue~~ **a prvalue** of type “pointer to `T1`” to the type “pointer to `T2`” casts away constness.

Casting from ~~an rvalue~~ **a prvalue** of type “pointer to data member of `X` of type `T1`” to the type “pointer to data member of `Y` of type `T2`” casts away constness if a cast from ~~an rvalue~~ **a prvalue** of type “pointer to `T1`” to the type “pointer to `T2`” casts away constness.

5.3.1 Unary operators ¶1-2:

...[*Note:*...this lvalue must not be converted to ~~an rvalue~~ **a prvalue**, see 4.1. —*end note*]

The result of each of the following unary operators is ~~an rvalue~~ **a prvalue**.

5.4 Explicit type conversion (cast notation) ¶1, 4-5:

...The result is an lvalue if `T` is an lvalue reference type **or an rvalue reference to function type and an xvalue if `T` is an rvalue reference to object type;**~~,~~ otherwise the result is ~~an rvalue~~ **a prvalue**. [*Note:* if `T` is a non-class type that is *cv-qualified*, the *cv-qualifier*s are ignored when determining the type of the resulting ~~rvalue~~ **prvalue**; see 3.10. —*end note*]

...the conversion is valid even if the base class is inaccessible:

- ...
- a pointer to an object of an unambiguous non-virtual base class type, ~~an lvalue or rvalue~~ **a glvalue** of an unambiguous non-virtual base class type, or a pointer to member of an unambiguous non-virtual base class type may be explicitly converted to a pointer, a reference, or a pointer to member of a derived class type, respectively.

The operand of a cast using the cast notation can be ~~an rvalue~~ **a prvalue** of type “pointer to

incomplete class type”...

5.5 Pointer-to-member operators ¶6:

...The result of a `.*` expression **whose second operand is a pointer to a data member** is ~~an lvalue only if its first operand is an lvalue and its second operand is a pointer to data member~~ **of the same value category (3.10) as its first operand. The result of a `.*` expression whose second operand is a pointer to a member function is a prvalue**. The result of an `->*` expression is an lvalue ~~only~~ if its second operand is a pointer to data member **and a prvalue otherwise**...

5.16 Conditional operator ¶2-6:

If either the second or the third operand has type (possibly cv-qualified) `void`...

- The second or the third operand (but not both) is a throw-expression (15.1); the result is of the type of the other and is ~~an rvalue~~ **a prvalue**.
- Both the second and the third operands have type `void`; the result is of type `void` and is ~~an rvalue~~ **a prvalue**...

Otherwise, if the second and third operand have different types and either has (possibly cv-qualified) class type**,** or if both are ~~lvalues~~ **glvalues** of **the same value category and** the same type except for cv-qualification, an attempt is made to convert each of those operands to the type of the other. The process for determining whether an operand expression `E1` of type `T1` can be converted to match an operand expression `E2` of type `T2` is defined as follows:

- If `E2` is an 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 an lvalue.
- **If `E2` is an xvalue: `E1` can be converted to match `E2` if `E1` can be implicitly converted to the type “rvalue reference to `T2`,” subject to the constraint that the reference must bind directly.**
- If `E2` is an rvalue or if **neither of** the conversion**s** above ~~cannot~~ **can** be done and at least one of the operands has (possibly cv-qualified) class type:
  - if `E1` and `E2` have class type, and the underlying class types are the same or one is a base class of the other: `E1` can be converted to match `E2` if the class of `T2` is the same type as, or a base class of, the class of `T1`, and the cv-qualification of `T2` is the same cv-qualification as, or a greater cv-qualification than, the cv-qualification of `T1`. If the conversion is applied, `E1` is changed to ~~an rvalue~~ **a prvalue** of type `T2` by copy-initializing a temporary of type `T2` from `E1` and using that temporary as the converted operand.
  - Otherwise (i.e., if E1 or E2 has a nonclass type, or if they both have class types

> > but the underlying classes are not either the same or one a base class of the other): E1 can be converted to match E2 if E1 can be implicitly converted to the type that expression E2 would have if E2 were converted to ~~an rvalue~~ **a prvalue** (or the type it has, if E2 is ~~an rvalue~~ **a prvalue**).
>
> Using this process, it is determined whether the second operand can be converted to match the third operand, and whether the third operand can be converted to match the second operand. If both can be converted, or one can be converted but the conversion is ambiguous, the program is ill-formed. If neither can be converted, the operands are left unchanged and further checking is performed as described below. If exactly one conversion is possible, that conversion is applied to the chosen operand and the converted operand is used in place of the original operand for the remainder of this section.

If the second and third operands are ~~lvalues~~ **glvalues of the same value category** and have the same type, the result is of that type and ~~is an lvalue~~ **value category** and it is a bit-field if the second or the third operand is a bit-field, or if both are bit-fields.

Otherwise, the result is ~~an rvalue~~ **a prvalue**. If the second and third operands...

...After those conversions, one of the following shall hold:

- The second and third operands have the same type; the result is of that type. If the operands have class type, the result is ~~an rvalue~~ **a prvalue** temporary of the result type...

5.18 Comma operator ¶1:

...the result is ~~an lvalue if~~ **of the same value category as** its right operand~~ is an lvalue~~, and is a bit-field if its right operand is ~~an lvalue~~ **a glvalue** and a bit-field.

7.1.6.1 Simple type specifiers ¶6:

If an attempt is made to refer to an object defined with a volatile-qualified type through the use of ~~an lvalue~~ **a glvalue** with a non-volatile-qualified type, the program behavior is undefined.

8.3.2 References ¶1:

...[*Example:*

> `typedef int& A;`
> `const A aref = 3;` *// ill-formed;* ***lvalue reference to*** *non-*`const` ~~*reference*~~
> *// initialized with rvalue*

The type of `aref` is "**lvalue** reference to `int`", not "~~`const`~~ **lvalue** reference to **`const`** `int`". —*end example*]

8.3.4 Arrays ¶5:

[*Note:* conversions affecting ~~lvalues~~ **expressions** of array type are described in 4.2. Objects of array types cannot be modified, see 3.10. —*end note*]

8.5 Initializers ¶16:

The semantics of initializers are as follows...

- ...
- If the destination type is a (possibly cv-qualified) class type:
  - If the initialization is direct-initialization...
  - Otherwise... The temporary is ~~an rvalue~~ **a prvalue**...

8.5.3 References ¶5:

- If the reference is an lvalue reference and the initializer expression...
- 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 have a function type**...
  - **If `T1` is a function type, then**
    - **if `T2` is the same type as `T1`, the reference is bound to the initializer expression lvalue;**
    - **if `T2` is a class type and the initializer expression can be implicitly converted to an lvalue of type `T1` (this conversion is selected by enumerating the applicable conversion functions (13.3.1.6) and choosing the best one through overload resolution (13.3)), the reference is bound to the function lvalue that is the result of the conversion;**
    - **otherwise, the program is ill-formed.**
  - ~~If~~ **Otherwise, if** ~~`T1` and~~ `T2` ~~are~~ **is a** class type~~s~~ and...

8.5.4 List-initialization ¶3:

List-initialization of an object or reference of type `T` is defined as follows:

- ...
- Otherwise, if `T` is a reference to class type or if `T` is any reference type and the initializer list has no elements, ~~an rvalue~~ **a prvalue** temporary of the type referenced by `T` is list-initialized, and the reference is bound to that temporary...

9.3.2 The `this` pointer ¶1:

In the body of a non-static (9.3) member function, the keyword `this` is ~~an rvalue~~ **a prvalue** expression...

10.2 Member name lookup ¶12:

An explicit or implicit conversion from a pointer to or ~~an lvalue~~ **an expression designating an object** of a derived class to a pointer or reference to one of its base classes shall unambiguously refer to a unique object representing the base class...

12.1 Constructors ¶14:

During the construction of a `const` object, if the value of the object or any of its subobjects is accessed through ~~an lvalue~~ **a glvalue** that is not obtained, directly or indirectly, from the constructor's `this` pointer, the value of the object or subobject thus obtained is unspecified. [*Example:*...

12.2 Temporary objects ¶1:

Temporaries of class type are created in various contexts: binding ~~an rvalue to~~ a reference **to a prvalue** (8.5.3), returning ~~an rvalue~~ **a prvalue** (6.6.3), a conversion that creates ~~an rvalue~~ **a prvalue** (4.1, 5.2.9, 5.2.11, 5.4), throwing an exception (15.1), entering a handler (15.3), and in some initializations (8.5)...

12.7 Construction and destruction ¶3:

To explicitly or implicitly convert a pointer (~~an lvalue~~ **a glvalue**) referring to an object of class `X` to a pointer (reference) to a direct or indirect base class `B` of `X`...

13.3 Overload resolution ¶2:

Overload resolution selects the function to call in seven distinct contexts within the language:

- ...
- invocation of a conversion function for conversion to ~~an lvalue~~ **a glvalue** or class ~~rvalue~~ **prvalue** 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~~ 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 does not affect the ranking of implicit conversion sequences (13.3.3.2). *—end note*]

13.3.1.1.1 Call to named function ¶2:

[*Footnote:*Note that cv-qualifiers on the type of objects are significant in overload resolution for both ~~lvalue~~ **glvalue** and class ~~rvalue~~ **prvalue** objects. *—end footnote*]

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

...the candidate functions are selected as follows:

- ...
- ...Conversion functions that return "reference to `X`" return lvalues or ~~rvalues~~ **xvalues**, depending on the type of reference...

13.3.1.5 Initialization by conversion function ¶1:

...the candidate functions are selected as follows:

- ...
- ...Conversion functions that return "reference to *cv2* `X`" return lvalues or ~~rvalues~~ **xvalues**, depending on the type of reference...

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 lvalue~~ **a glvalue** or class ~~rvalue~~ **prvalue** that is the result of applying a conversion function to an initializer expression...

13.3.2 Viable functions ¶3:

If the parameter has reference type, 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 **and that an rvalue reference cannot be bound to an lvalue** can affect the viability of the function (see 13.3.3.1.4).

13.3.3.1 Implicit conversion sequences ¶2, 6:

Implicit conversion sequences are concerned only with the type, cv-qualification, and ~~lvalue-ness~~ **value category** of the argument...

When the parameter type is not a reference, the implicit conversion sequence models a copy-initialization of the parameter from the argument expression. The implicit conversion sequence is the one required to convert the argument expression to ~~an rvalue~~ **a prvalue** of the type of the parameter...

13.3.3.1.1 Standard conversion sequences ¶1:

...[*Note:* these categories are orthogonal with respect to ~~lvalue-ness~~ **value category**, cv-qualification, and data representation: the Lvalue Transformations do not change the cv-qualification or data representation of the type; the Qualification Adjustments do not change the ~~lvalue-ness~~ **value category** or data representation of the type; and the Promotions and Conversions do not change the ~~lvalue-ness~~ **value category** or cv-qualification of the type. —*end note*]

13.3.3.2 Ranking implicit conversion sequences ¶3:

Two implicit conversion sequences of the same form are indistinguishable conversion sequences unless one of the following rules applies:

- Standard conversion sequence `S1` is a better conversion sequence than standard conversion sequence `S2` if
    - ...
    - 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 and S2 binds an lvalue reference.

      [*Example:*

      ```
      int i;
      int f1();
      int&& f2();
      int g(const int&);
      int g(const int&&);
      int j = g(i);        // calls g(const int&)
      int k = g(f1());     // calls g(const int&&)
      int l = g(f2());     // calls g(const int&&)
      ...
      ```

  —*end example*]

14.2 Template parameters ¶6:

A non-type non-reference *template-parameter* is ~~an rvalue~~ a prvalue...