Proposed C++14 Value Classification

Document #: WG21 N3550
Date: 2013-03-12
Revises: None
Project: JTC1.22.32 Programming Language C++
Reply to: Walter E. Brown <webrown.cpp@gmail.com>

Contents

1 Introduction

CWG issue 1585 (reproduced here in its entirety from Revision 82 of the C++ Standard Core Language Issues List) proposes to review the value classification of some member access expressions:

> (From messages 22742 through 22750 and 22765 through 22769.)
> According to 5.2.5 [expr.ref] paragraph 4,
>
> > If **`E2`** is declared to have type "reference to **`T`**," then **`E1.E2`** is an lvalue. . .
>
> This applies to rvalue reference types as well as to lvalue reference types, based on the rationale from 5 [expr] paragraph 7 that
>
> > In general. . . named rvalue references are treated as lvalues and unnamed rvalue references to objects are treated as xvalues. . .
>
> Since a non-static data member has a name, it would appear most naturally to fall into the lvalue category. This makes sense as well from the perspective that the target of such a reference does not bear any necessary correlation with the value category of the object expression; in particular, an xvalue object might have an rvalue reference member referring to a different object from which it would be an error to move.
>
> On the other hand, rvalue reference members have limited utility and are likely only to occur as the result of template argument deduction in the context of perfect forwarding, such as using a **`std::pair`** to forward values. In such cases, a **`first`** or **`second`** member of rvalue reference type would be most naturally treated as having the same value category as that of the object expression. The utility of this usage may outweigh the safety considerations that shaped the current policy.

Because the C++11 value classification scheme is widely distributed throughout the text of the standard, it seems unnecessarily difficult to assess the global impact of the direction suggested in the last paragraph of the issue. We therefore propose a centralized specification of the scheme.

2 Proposal

Independent of the outcome of CWG 1585, we propose to **centralize the specification of value classification**, and correspondingly to **excise the current widely distributed specifications**. If the following wording is incorrect or incomplete in any detail, a revision of this proposal will address any defects that CWG identifies. All citations are from WG21 draft [DuT12]. Green text is proposed for addition; text in ~~red~~ is to be deleted. Footnotes are for clarification, and are not part of the proposed wording.

2.1 Proposed value classification rules

The following comprehensive value classification scheme (inspired by and subsuming the Note that is [expr]/7) is proposed for inclusion as a new paragraph following [basic.lval]/1. The wording is drafted[1] under the assumption that CWG decides in favor of the direction suggested in the last paragraph of the above-cited issue. If CWG decides against making a change in response to issue 1585, item (5) should be redrafted so as to correspond to the status quo.

Unless specified otherwise, an expression's value category is determined according to the following ordered list of rules:

1. The value classification of an expression enclosed in parentheses is the same as the expression's value classification without the enclosing parentheses.
2. An unparenthesized expression ***E*** is classified as a **prvalue** if it is a constant expression [expr.const] whose result is of non-arrary type.
3. Otherwise, ***E*** is classified as an **lvalue** if:
   a) it is an *id-expression* [expr.prim.general], or
   b) the type of its result is (i) an lvalue reference type,[2] (ii) an array type,[3] or (iii) a function type, a non-static member function type, or an rvalue reference to either.
4. Otherwise, if its result has an rvalue reference type, ***E*** is classified as an **xvalue** if:
   a) it is a cast expression, or
   b) it is an explicit or implicit function call expression.
5. Otherwise, if ***E*** is a class member access [expr.ref], then ***E***'s value classification is the same as the value classification of its object expression.
6. Otherwise, ***E*** is classified as a **prvalue**.

[*Example:* from [expr]/7 *— end example*]

2.2 Proposed excisions

The following list of proposed excisions is in addition to [expr]/7, which was handled in the previous subsection. Items are presented in approximate order of occurrence:

[lex.bool]/1

The Boolean literals are the keywords **`false`** and **`true`**. Such literals ~~are prvalues and~~ have type **`bool`**.

[2] The return types in the lists of operator functions in [over.built] identify those native operators that return an lvalue reference result: (i) prefix `++` and `--`, (ii) unary `*`, (iii) `[]`, and (iv) `=` and the compound assignment operators.

[3] This includes string literals.

[lex.nullptr]/1

The pointer literal is the keyword `nullptr`. It ~~is a prvalue of~~ has type `std::nullptr_t`. [*Note:* ... — *end note*]

[conv]/6

... ~~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 a prvalue otherwise.~~ ...

[expr]/5

... The expression designates the object or function denoted by the reference~~, and the expression is an lvalue or an xvalue, depending on the expression~~.

[expr.prim.general]/1

... ~~A string literal is an lvalue; all other literals are prvalues.~~

[expr.prim.general]/6

... ~~The presence of parentheses does not affect whether the expression is an lvalue.~~ ...

[expr.prim.general]/7

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

[expr.prim.general]/9

... ~~The result is an lvalue if the member is a static member function or a data member and a prvalue otherwise.~~ ...

[expr.prim.general]/10

... ~~The result is an lvalue if the member is a static member function or a data member and a prvalue otherwise.~~

[expr.prim.general]/11

... ~~The result is a prvalue.~~

[expr.sub]/1

... The result ~~is an lvalue of~~ has type "`T`." ...

[expr.call]/10

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

[expr.type.conv]/1

... If the expression list specifies more than a single value, the type shall be a class with a suitably declared constructor (8.5, 12.1), and the expression `T(x1, x2, ...)` is equivalent in effect to the declaration `T t(x1, x2, ...)` for some invented temporary variable `t` with the result being the value of `t` ~~as a prvalue~~.

[expr.type.conv]/2

The expression `T()`, where `T` is a *simple-type-specifier* or *typename-specifier* for a non-array complete object type or the (possibly cv-qualified) void type, creates a ~~prvalue~~ value of the specified type, ~~whose value is that~~ produced by value-initializing (8.5) an object of type `T`; no initialization is done for the `void()` case. [*Note:* ... — *end note*]

[expr.type.conv]/3

Similarly, a *simple-type-specifier* or *typename-specifier* followed by a *braced-init-list* creates a temporary object of the specified type direct-list-initialized (8.5.4) with the specified *braced-init-list*, and its value is that temporary object ~~as a prvalue~~.

[expr.ref]/4

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. . . .

— . . . ~~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 a prvalue.~~ . . .

— If `E2` is a (possibly overloaded) member function, function overload resolution (13.3) is used to determine whether `E1.E2` refers to a static or a non-static 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 lvalue; the expression~~ designates the static member function. . . .

— Otherwise, if `E1.E2` refers to a non-static member function and the type of `E2` is "function of parameter-type-list *cv ref-qualifier*$_{opt}$ returning `T`", then `E1.E2` ~~is a prvalue. The expression~~ designates a non-static member function. . . .

— . . .

— If `E2` is a member enumerator and the type of `E2` is `T`, the ~~expression `E1.E2` is a prvalue. The~~ type of `E1.E2` is `T`.

[expr.post.incr]/1

. . . ~~The result is a prvalue.~~ . . .

[expr.dynamic.cast]/2

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

[expr.dynamic.cast]/5

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

[expr.typeid]/1

The result of a typeid expression is ~~an lvalue~~ of static type `const std::type_info` (18.7.1) and . . . .

[expr.static.cast]/1

. . . ~~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 a prvalue.~~ . . .

[expr.static.cast]/1

. . . 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 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 expression `v`. . . .

[expr.reinterpret.cast]/1

. . . 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 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 expression `v`. . . .

[expr.const.cast]/1

. . . If ~~`T` is an lvalue reference to object type, the result is an lvalue; if `T` is an rvalue reference to object type, the result is an xvalue; otherwise,~~ the result is 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 expression `v`. . . .

[expr.unary.op]/1

The unary * operator performs indirection: the expression to which it is applied shall be a pointer to an object type, or a pointer to a function type and the result ~~is an lvalue referring~~ refers to the object or function to which the expression points. . . .

[expr.unary.op]/2

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

[expr.unary.op]/3

. . . If the operand is a *qualified-id* naming a non-static member `m` of some class `C` with type `T`, the result has type "pointer to member of class `C` of type `T`" and ~~is a prvalue designating~~ designates `C::m`. Otherwise, if the type of the expression is `T`, the result has type "pointer to `T`" and ~~is a prvalue~~ yields a value that is the address of the designated object (1.7) or a pointer to the designated function. [*Note:* . . . — *end note*]

[expr.pre.incr]/1

. . . The result is the updated operand; ~~it is an lvalue, and~~ it is a bit-field if the operand is a bit-field. . . .

[expr.cast]/1

. . . ~~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 a prvalue.~~ . . .

[expr.mptr.oper]/6

. . . ~~The result of a `.*` expression whose second operand is a pointer to a data member is 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.~~ . . .

[expr.cond]/2

. . . and one of the following shall hold:

— 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 a prvalue~~.

— Both the second and the third operands have type `void`; the result is of type `void` ~~and is a prvalue~~. [*Note:* . . . — *end note*]

[expr.cond]/4

If the second and third operands are glvalues of the same value category and have the same type, the result is of that type ~~and 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.

[expr.cond]/5

Otherwise, ~~the result is a prvalue. If~~ if the second and third operands do not have the same type, and either has (possibly cv-qualified) class type, overload resolution is used to determine the conversions (if any) to be applied to the operands (13.3.1.2, 13.6). . . .

[expr.cond]/6

Lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are performed on the second and third operands. After those conversions, one of the following shall hold:

— . . . If the operands have class type, the result is a ~~prvalue~~ temporary of the result type, which . . . .

— . . .

[expr.comma]/1

. . . The type and value of the result are the type and value of the right operand; the result ~~is of the same value category as its right operand, and~~ is a bit-field if its right operand is a glvalue and a bit-field. . . .

[over.match.conv]/1

. . .

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

[temp.param]/6

A non-type non-reference template-parameter ~~is a prvalue. It~~ shall not be assigned to or in any other way have its value changed. . . .

3 Acknowledgments

Many thanks to the readers of early drafts of this paper for their helpful feedback.

4 Bibliography

[DuT12] Stefanus Du Toit: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N3485 (post-Portland mailing), 2012-11-02. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3485.pdf.

5 Revision history

Revision	Date	Changes
1.0	2013-03-12	• Published as N3550.