Value Categories in C

Barry Revzin · Follow
5 min read · Sep 4, 2017

Listen Share

The meaning of value categories has changed over time and can be quite a confusing topic, prone to misconception. This post will try to illuminate what, in C++17, these oddly named things actually are.

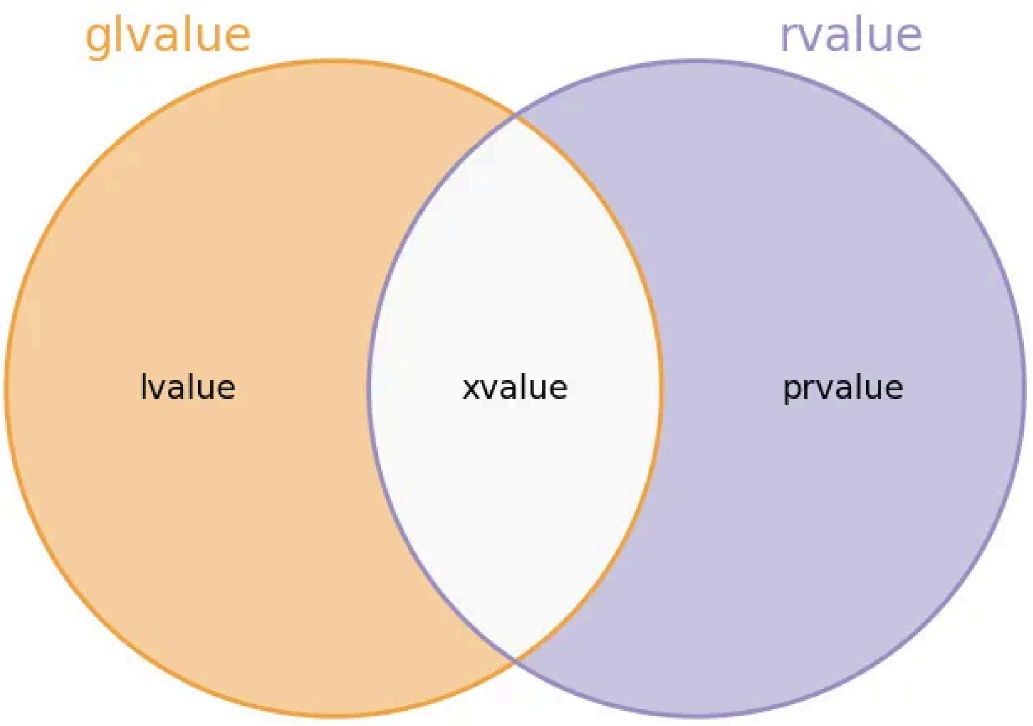
The most important thing to remember is that value categories are a taxonomy of *expressions*. They are not categories of objects or variables or types. Getting this wrong is an immediate source of problems. Consider:

```
1   void foo(int& );  // #1
2   void foo(int&& ); // #2
3
4   int&& r = 42;
5   foo(r);
```

ex1.h hosted with ❤ by GitHub view raw

The *variable* `r` is an *rvalue* reference. But the *expression* `r` on line 5 is an lvalue. As such, #1 is invoked. It doesn't matter that the type of `r` matches (exactly, even) #2. The value category mismatch prevents that candidate from being viable.

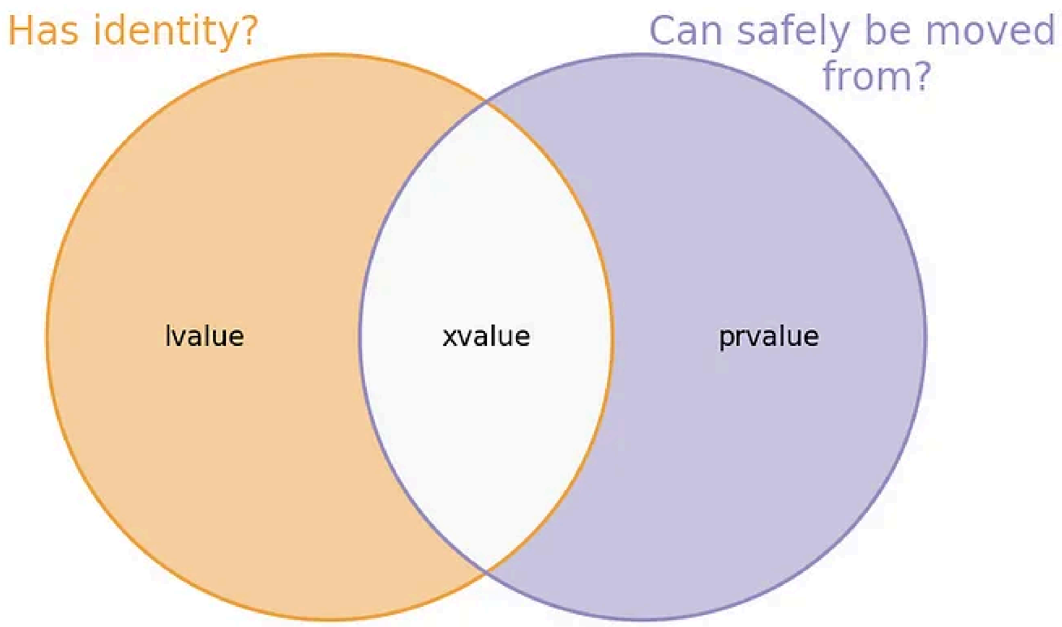
Each expression, from the arbitrary complex down to the simple identifiers or literals, has a category that it corresponds to. This taxonomy can, to me, best be viewed as a Venn diagram:



That is, every expression is exactly one of an *lvalue*, an *xvalue*, or a *prvalue*. Those three are, in turn, subsets of two broader value categories: *glvalues* (the union of lvalues and xvalues) and *rvalues* (the union of xvalues and prvalues). Besides alphabet soup, what do these terms actually mean? In the new standard, the definitions are quite helpful. The three key ones are, from [basic.lval] (the other two are defined by their position in the Venn diagram):

- A *glvalue* is an expression whose evaluation determines the identity of an object, bit-field, or function.
- A *prvalue* is an expression whose evaluation initializes an object or a bit-field, or computes the value of the operand of an operator, as specified by the context in which it appears.
- An *xvalue* is a glvalue that denotes an object or bit-field whose resources can be reused (usually because it is near the end of its lifetime).

A common way of describing these categories used to involve describing the higher-level groupings by what properties those expressions have. That is:



I think this is still a helpful way of thinking about the categories, even if it's not strictly accurate anymore. The "has identity" part remains valid — indeed the standard definition of *glvalue* itself is basically this criteria.

But a big part of the change in P0135 ("Guaranteed copy elision through simplified value categories") is that prvalues themselves do not necessarily have to lead to the existence of objects (only if a "temporary materialization" is necessary) and as such, it doesn't quite make sense to talk about whether those resources can be safely reused. Indeed, C++17, prvalues are not moved from! Let's take a seemingly simple example:

```
T var = T();
```

For some type `T`. In C++03, the expression `T()` is an rvalue, but this is copy-construction of a new variable named `var`. In C++11, the expression `T()` is a prvalue, and this is move-construction. In both cases, the copy/move will likely get elided, even if there are side effects. However, in C++17, *there is no move*. It's important to repeat this for emphasis. The prvalue is *not moved from*. This is value-initializing `var` and is exactly equivalent to:

```
T var();
```

(Or, at least, would be if the above weren't a function declaration. What `T var = T()` means today is declaring a variable `var` of type `T` constructed using the initializer `()`. There is not an easy way to express that in other terms).

```
struct NonMoveable { /* ... */ };
NonMoveable make() { /* how to make this work without a copy? */ }

auto x = make(); // error, can't perform the move you didn't want,
                 // even though compiler would not actually call it
```

p0135.h hosted with ❤ by GitHub view raw

In C++14, this code is ill-formed, and the comments say it all. The move constructor *must be valid*, even if you don't need. In C++17, this is no longer true, and the code is fine.

What's most important is to not get lost in the history of the names. Originally, the two value categories (lvalues and rvalues) were so named because those expressions could appear on the left or right side of an assignment expression. But there are lvalues that cannot appear on the left hand side of assignment (for instance, an identifier referring to a `const` object — such an object is not assignable) and there are rvalues that can (for instance, given `struct S{};` the expression `S{} = S{}` is well-formed).

Value categories are mirrored in the type system. The standard gives a method to determine an expression's value category using `decltype`, as defined in [dcl.type.simple]/4. Given an expression, `expr`, we can observe the type of `decltype((expr))` (the extra parentheses are *not* a typo!). If that resulting type is an lvalue reference type, the expression is an lvalue. If that resulting type is an rvalue reference type, the expression is an xvalue. Otherwise (if the resulting type is not a reference type), the expression is a prvalue. For instance:

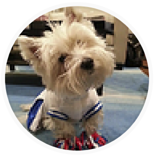
```cpp
#include <iostream>

namespace detail {
    template <class T> struct value_category      { static constexpr char const* value = "prval
    template <class T> struct value_category<T&>  { static constexpr char const* value = "lvalu
    template <class T> struct value_category<T&&> { static constexpr char const* value = "xvalu
}

#define PRINT_VALUE_CAT(expr) std::cout << #expr << " is a " << ::detail::value_category<declty

struct S { int i; };

int main ()
{
    int&& r = 42;
    PRINT_VALUE_CAT(4); // prvalue
    PRINT_VALUE_CAT(r); // lvalue
    PRINT_VALUE_CAT(std::move(r)); // xvalue

    PRINT_VALUE_CAT(S{0}); // prvalue
    PRINT_VALUE_CAT(S{0}.i); // xvalue (gcc erroneously calls this a prvalue)
}
```

ex2.cpp hosted with ❤ by GitHub view raw

Only rvalues can bind to rvalue references, only lvalues can bind to *non-const* lvalue references (the one exception here is that rvalues can indeed bind to const lvalue references, something that Herb Sutter calls the most important const). This allows us to differentiate in the type system between lvalues and rvalues (to make it clear when an object's resources can be cannibalized safely or when they needs to be copied), but there is no way to differentiate xvalues and prvalues in overload resolution. Indeed, such a differentiation probably wouldn't be helpful anyway.

To wrap up, expressions in C++ are divided into five overlapping value categories. glvalues are expressions that evaluate to locations — they have identity. prvalues are used for initialization, they do not denote objects, although they can "materialize" an object in those cases where necessary. xvalues are glvalues that denote objects that are at the end of their lifetime or are otherwise marked for being able to have their resources reused. lvalues and xvalues have identities. xvalues can be safely moved from. prvalues are, again, used for initialization. Value categories are categories of expressions, not of objects. Value categories are not about assignment.

Cpp Cplusplus

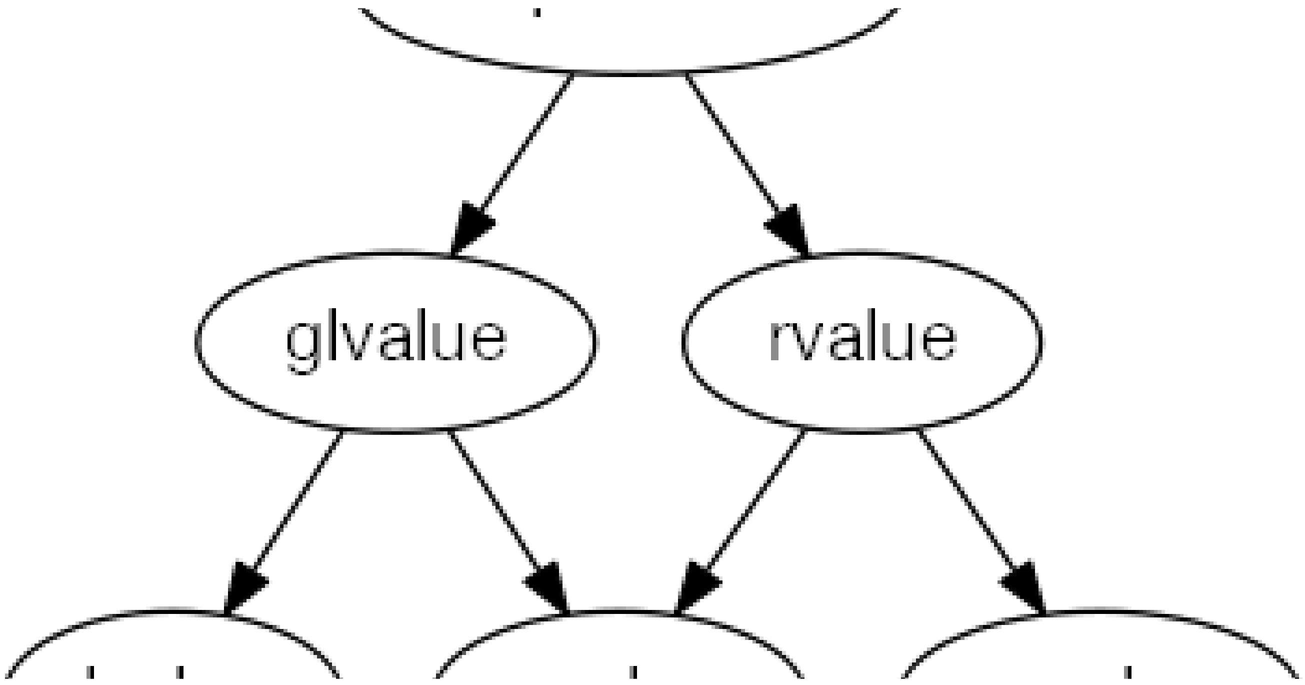


Follow

Written by Barry Revzin

288 Followers

More from Barry Revzin

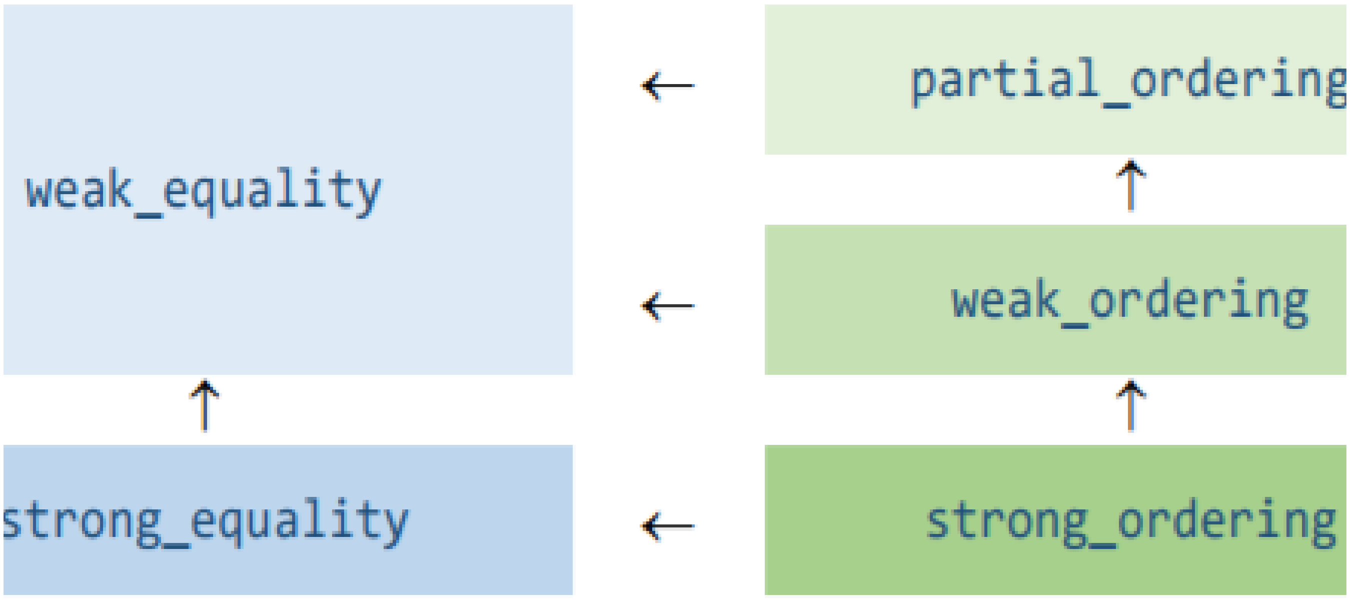


Barry Revzin

xvalues and prvalues: The Next Generation

C++11 introduced the ability for code to differentiate between lvalues and rvalues — a pretty powerful feature that is what move semantics...

Feb 11, 2017 24



Barry Revzin

Implementing the spaceship operator for optional

Last week, the C++ Standards Committee added operator<=>, known as the spaceship operator, to the working draft for what will eventually...

Nov 17, 2017 292 1