Understanding Lvalues and Rvalues

(corrected)

Dan Saks

Core C++ Conference
May, 2019

1

Lvalues and Rvalues

- *Lvalues* and *rvalues* aren't really language features.
- Rather, they're semantic properties of expressions.
- Understanding them provides valuable insights into:
  - the behavior of built-in operators
  - the code generated to execute those operators
  - the meaning of some otherwise cryptic compiler error messages
  - reference types
  - overloaded operators

5

Lvalues and Rvalues Have Evolved

- In early C, the concepts of lvalue and rvalue were fairly simple.

- Early C++ added classes, `const`, and references.
- The concepts got more complicated.

- Modern C++ added rvalue references.
- The concepts got even more complicated.

- This talk explains the origins of the concepts of lvalue and rvalue, from this historical perspective.

6

Lvalues

- In *The C Programming Language*, Kernighan and Ritchie wrote:
  - The name “lvalue” comes from the assignment expression

    `E1 = E2`

    in which the ***left*** operand `E1` must be an lvalue expression.
  - An ***lvalue*** is an expression referring to an object.
  - An ***object*** is a region of storage.

7

Lvalues and Rvalues

```
int n;  // a definition for an integer object named n
~~~
n = 1;  // an assignment expression
```

- `n` is a sub-expression referring to an `int` object.
  - It's an *lvalue*.
- `1` is a sub-expression not referring to an object.
  - It's an *rvalue*.
- An ***rvalue*** is simply an expression that's not an lvalue.

Lvalues and Rvalues

- Here's a more complicated assignment:

```
x[i + 1] = abs(p->value);
```

- `x[i + 1]` is an expression. So is `abs(p->value)`.
- For the assignment to be valid:
  - The left operand must be an lvalue.
    - It must refer to an object.
  - The right operand can be either an lvalue or rvalue.
    - It can be any expression.

A Look Under the Hood

- Why make this distinction between lvalues and rvalues?
- One answer:
  - So that compilers can assume that rvalues don't necessarily occupy storage.
  - This offers considerable freedom in generating code for rvalue expressions.
- Again, let's consider the assignment in:

```
int n;  // a declaration for an integer object named n
~~~
n = 1;  // an assignment expression
```

10

Data Storage for Rvalues

- A compiler might represent 1 as named data storage initialized with the value 1, as if 1 were an lvalue.
- In assembly language, this might look something like:

```
one:            ; a label for the following location
    .word 1     ; allocate storage holding the value 1
```

- The compiler would generate code to copy from that initialized storage to the storage allocated for n:

```
    mov n, one  ; copy the value at one to location n
```

11

Data Storage for Rvalues

- Some machines provide instructions with an ***immediate operand***:
  - A source operand value can be part of an instruction.
- In assembly, this might look like:

```
mov n, #1        ; copy the value 1 to location n
```

- In this case:
  - The rvalue `1` never appears as an object in the data space.
  - Rather, it appears as part of an instruction in the code space.

12

Data Storage for Rvalues

- On some machines, the preferred way to put the value 1 into an object might be to:
  - clear the object,
  - then increment it.
- In assembly, this might look like:

```
clr n        ; set n to zero
inc n        ; increment n, effectively setting it to 1
```

- Data representing the values 0 and 1 appear nowhere in either the source or object code.

13

Must be an Lvalue == Can't be an Rvalue

- Now, suppose you write:

```
1 = n;       // obviously silly
```

- This is trying to change the value of the integer literal, `1`.
- Of course, C (and C++) reject it as an error.

- But why, exactly?
  - An assignment assigns a value to an object.
  - Its left operand must be an lvalue.
  - But `1` is not an lvalue; it's an rvalue.

14

Recap

- Every expression in C is either an lvalue or an rvalue.

- In general:
  - An *lvalue* is an expression that refers to an object.
  - An *rvalue* is simply any expression that isn't an lvalue.

- Caveat: Although this is also true for non-class types in C++, it's not true for class types.

15

Literals

- Most literals are rvalues, including:
  - numeric literals, such as 3 and `3.14159`
  - character literals, such as `'a'`
- They don't necessarily occupy data storage.

- However, character string literals, such as `"xyzzy"`, are lvalues.
- They occupy data storage.

16

Enumeration Constants

- When used in expressions, enumeration constants are also rvalues:

```
enum color { red, green, blue };
color c;
~~~
c = green;      // OK: c is an lvalue
blue = green;   // error: blue is an rvalue
```

17

Lvalues Used as Rvalues

- An lvalue can appear on either side of an assignment, as in:

```
int m, n;
~~~
m = n;      // OK: m and n are both lvalues
```

- Obviously, you can assign the value in `n` to the object designated by `m`.
- This assignment uses the lvalue expression `n` as an rvalue.
- Officially, C++ performs an ***lvalue-to-rvalue conversion***.

18

Operands of Other Operators

- The concepts of lvalue and rvalue apply in all expressions.
  - Not just assignment.
- For example, both operands of the binary `+` operator must be expressions.
  - Obviously, those expressions must have suitable types.
- But each operand can be either an lvalue or rvalue.

```
int x;
~~~
~~~ x + 2 ~~~   // OK: lvalue + rvalue
~~~ 2 + x ~~~   // OK: rvalue + lvalue
```

19

What About the Result?

- For built-in binary (non-assignment) operators such as `+`:
  - The operands may be lvalues or rvalues.
  - But what about the result?
- An expression such as `m + n` places its result:
  - not in `m`,
  - not in `n`,
  - but rather in a compiler-generated temporary object, often a CPU register.
- Such temporary objects are rvalues.

20

What About the Result?

- For example, this is (obviously?) an error:

```
m + 1 = n;        // error... but why?
```

- The `+` operator has higher precedence than `=`.
- Thus, the assignment expression is equivalent to:

```
(m + 1) = n;      // error... but why?
```

- It's an error because `m + 1` yields an rvalue.

21

Unary &

- &e is a valid expression only if e is an lvalue.

- Thus, &3 is an error.
- Again, 3 does not refer to an object, so it's not addressable.

- Although the operand must be an lvalue, the result is an rvalue.
- For example,

```
int n, *p;
~~~
p = &n;         // OK: n is an lvalue
&n = p;         // error: &n is an rvalue
```

22

Unary *

- In contrast to unary &, unary * yields an lvalue.
- A pointer p can point to an object, so *p is an lvalue.

```
int a[N];
int *p = a;
char *s = NULL;     // = nullptr in Modern C++
~~~
*p = 3;             // OK
*s = '\0';          // undefined behavior
```

- Note: Lvalue-ness is a compile-time property.
  - *s is an lvalue even if s is null.
  - If s is null, evaluating *s causes undefined behavior.

23

Unary *

- Again, the result of the * operator is an lvalue.
- However, its operand can be an rvalue.

- For example,

```
*(p + 1) = 4;    // OK
```

- Here, `p + 1` is an rvalue, but `*(p + 1)` is an lvalue.

- The assignment stores the value 4 into the object referenced by `*(p + 1)`.

24

Data Storage for Expressions

- Conceptually, rvalues (of non-class type) don't occupy data storage in the object program.
  - In truth, some might.

- C and C++ insist that you program as if non-class rvalues don't occupy storage.

- Conceptually, lvalues (of any type) occupy data storage.
  - In truth, the optimizer might eliminate some of them.
  - (But only when you won't notice.)

- C and C++ let you assume that lvalues always do occupy storage.

25

Non-Modifiable Lvalues

- In fact, not all lvalues can appear on the left of an assignment.
- An lvalue is ***non-modifiable*** if it has a const-qualified type.
- For example,

```
char const name[] = "dan";
~~~
name[0] = 'D';      // error: name[0] is const
```

- `name[0]` is an lvalue, but it's non-modifiable.
  - Each element of a const array is itself const.

26

Non-Modifiable Lvalues

- Lvalues and rvalues provide a vocabulary for describing subtle behavioral differences…
- …such as between enumeration constants and const objects.
- For example, this `MAX` is a constant of an unnamed enumeration type:

```
enum { MAX = 100 };
```

- Unscoped enumeration values implicitly convert to integer.

27

Non-Modifiable Lvalues

- When MAX appears in an expression, it yields an integer rvalue.
- Thus, you can't assign to it:

```
MAX += 3;            // error: MAX is an rvalue
```

- You can't take its address, either:

```
int *p = &MAX;       // error: again, MAX is an rvalue
```

28

Non-Modifiable Lvalues

- On the other hand, this MAX is a const-qualified object:

```
int const MAX = 100;
```

- When it appears in an expression, it's a non-modifiable lvalue.
- Thus, you still can't assign to it.

```
MAX += 3;                // error: MAX is non-modifiable
```

- However, you can take its address:

```
int const *p = &MAX;     // OK: MAX is an lvalue
```

29

Recap

- This table summarizes the behavior of lvalues and rvalues (of non-class type):

	can take the address of	can assign to
lvalue	yes	yes
non-modifiable lvalue	yes	no
(non-class) rvalue	no	no

30

Const Objects

- A const object is addressable.
  - The compiler may generate storage to hold the const object's value.
- The compiler might find that the program never needs storage for a particular const object.
  - It often does.
- In that case, the compiler need not allocate storage for that object.
- This behavior for const objects is analogous to the behavior for inline functions.

31

Reference Types

- The concepts of lvalues and rvalues help explain C++ ***reference types***.
- References provide an alternative to pointers as a way of associating names with objects.
- C++ libraries often use references instead of pointers as function parameters and return types.

32

Reference Types

- Consider the following code:

```
int i;          // define i as an integer object
~~~
int &ri = i;    // define ri as a "reference to int"
```

- The last line above:
  - defines `ri` with type "reference to `int`", and
  - initializes `ri` to refer to `i`.
- Hence, reference `ri` is an ***alias*** for `i`.

33

Reference Types

- A reference is essentially a pointer that's automatically dereferenced each time it's used.
- You can rewrite most, if not all, code that uses a reference as code that uses a const pointer, as in:

reference notation	equivalent pointer notation
`int &ri = i;`	`int *const cpi = &i;`
`ri = 4;`	`*cpi = 4;`
`int j = ri + 2;`	`int j = *cpi + 2;`

- A reference acts like a const pointer that's dereferenced (has a * in front of it) whenever you touch it.
- A reference yields an *lvalue*.

34

Initializing vs. Assigning

- ***Initializing*** a reference associates the reference with an object.
  - Initializing a reference is also known as ***binding***.
- ***Assigning*** to a reference stores through the reference and into the referenced object.
- For instance,

```
int &ri = i;     // binds reference to object
ri = 3;          // assigns to referenced object
```

35

References and Overloaded Operators

- What good are references?
- Why not just use pointers?

- References can provide friendlier function interfaces.

- More specifically, C++ has references so that overloaded operators can look just like built-in operators…

36

References and Overloaded Operators

```
enum month {
    Jan, Feb, Mar, ~~~, Dec, month_end
};
typedef enum month month;
~~~

for (month m = Jan; m <= Dec; ++m) {
    ~~~
}
```

- This code compiles and executes as expected in C.

- However, it doesn't compile in C++…

37

References and Overloaded Operators

- In C++, the built-in ++ won't accept an operand of enumeration type.
- You need to overload ++ for `month`.
- Let's try it without references…

38

References and Overloaded Operators

```
void operator++(month x) {          // pass by value
    x = static_cast<month>(x + 1);
}
```

- Using this definition, `++m` compiles, but doesn't increment `m`.
  - It increments a copy of `m` in parameter `x`.
- Also, this implementation lets you apply ++ to an rvalue, as in:

```
++Apr;       // compiles, but shouldn't
```

- A proper overloaded ++ should behave like the built-in ++, as in:

```
++42;        // compile error: can't increment an rvalue
```

39

References and Overloaded Operators

- We need a ++ that passes in a month it can modify:

```
void operator++(month *x) {         // pass by address?
    *x = static_cast<month>(*x + 1);
}
```

- In fact, this function definition won't compile.
  - You can't overload an operator with a parameter of pointer type.
- Even if the definition compiled, it wouldn't work like a built-in ++:

```
++m;        // looks right but doesn't compile
++&m;       // looks wrong and doesn't compile
```

40

References and Overloaded Operators

- We really need a ++ that can modify a month object…
- … but without passing explicitly by address:

```
void operator++(month &x) {         // pass by reference
    x = static_cast<month>(x + 1);
}
```

- Using this definition:

```
++m;       // compiles, increments m, and looks right
```

- As a bonus, this ++ operator won't accept an rvalue:

```
++Apr;     // compile error
```

41

References and Overloaded Operators

- Actually, a proper prefix ++ doesn't return void.
- It returns the incremented object by reference:

```
month &operator++(month &x) {        // non-void return
    return x = static_cast<month>(x + 1);
}
```

- This enables overloaded ++ to act even more like a built-in operator:

```
int j = ++i;    // OK
month n = ++m;  // OK
```

42

"Reference to Const" Parameters

- Just as you can have "pointer to const" parameters...
- You can also have "reference to const" parameters:

```
R f(T const &t);
```

- A "reference to const" parameter will accept an argument that's either const or non-const.
- In contrast, a reference (to non-const) parameter will accept only a non-const argument.
- When it appears in an expression, a "reference to const" yields a *non-modifiable lvalue*.

43

"Reference to Const" Parameters

- For the most part, a function declared as:

```
R f(T const &t);    // by "reference to const"
```

  has the same outward behavior as a function declared as:

```
R f(T t);           // by value
```

- That is, the calls look and act very much the same…

44

"Reference to Const" Parameters

- Either way you declare `f`, you write the argument expression the same way:

```
T x;
~~~
f(x);   // by value, or by "reference to const"?
```

- Either way, calling `f` can't alter the actual argument, `x`:
  - By value: `f` has access only to a copy of `x`, not `x` itself.
  - By "reference to const": `f`'s parameter is declared to be non-modifiable.

45

Why Use "Reference to Const"?

- Why pass by "reference to const" instead of by value?
- Passing by "reference to const" might be much more efficient than passing by value.
- It depends on the cost to make a copy.

46

References and Temporaries

- A "pointer to T" can point only to an lvalue of type T.
- Similarly, a "reference to T" binds only to an lvalue of type T.
- For example, these are both compile errors:

```
int *pi = &3;        // can't apply & to 3
int &ri = 3;         // can't bind this, either
```

- These are also compile errors:

```
int i;
~~~
double *pd = &i;     // can't convert pointers
double &rd = i;      // can't bind this, either
```

47

References and Temporaries

- There's an exception to the rule that a reference must bind to an lvalue of the referenced type:
  - A "reference to const T" can bind to an expression x that's not an lvalue of type T …
  - … if there's a conversion from x's type to T.
- In this case, the compiler creates a temporary object to hold a copy of x converted to T.
  - This is so the reference has something to bind to.

48

References and Temporaries

- For example, consider:

```
int const &ri = 3;
```

- When program execution reaches this declaration, the program:
  1. creates a temporary `int` to hold the 3, and
  2. binds `ri` to the temporary.
- When execution leaves the scope containing `ri`, the program:
  3. destroys the temporary.

49

References and Temporaries

- Given:

```
double const &rd = ri;  // ri from the previous slide
```

- When program execution reaches this declaration, the program:
  1. converts the value of `ri` from `int` to `double`,
  2. creates a temporary `double` to hold the converted result, and
  3. binds `rd` to the temporary.

- Again, when execution leaves the scope containing `rd`, the program:
  4. destroys the temporary.

50

References and Temporaries

- This special behavior enables passing by "reference to const" to consistently have the same outward behavior as passing by value.
- For example, compare this with the code on the next slide:

```
long double x;
void f(long double       ld);  // by value
~~~

f(x);   // passes a copy of x
f(1);   // passes a copy of...
        // ...1 converted to long double
```

51

References and Temporaries

- This is the same example, except it uses a “reference to const” parameter in place of a value parameter:

```
long double x;
void f(long double const &ld);  // by reference to const
~~~

f(x);   // passes a reference to x
f(1);   // passes a reference to a temporary...
        // ...containing 1 converted to long double
```

- Either way, the function calls behave the same.

52

Mimicking Built-In Operators

- Recall the behavior of the built-in + operator:
  - The operands may be lvalues or rvalues.
  - The result is always an rvalue.
- How do you declare an overloaded operator with the same behavior?
- Consider a rudimentary (character) string class with + as a concatenation operator…

53

Mimicking Built-In Operators

- You can declare operator + as a non-member, as in:

```
class string {
public:
    string(string const &);
    string(char const *);   // converting constructor
    string &operator=(string const &);
    ~~~
};

string operator+(string const &lo, string const &ro);
```

54

Mimicking Built-In Operators

```
string operator+(string const &lo, string const &ro);
```

- Parameters lo and ro accept arguments that are either lvalues or rvalues:

```
string s = "hello";
string t = "world";
~~~
s = s + ", " + t;
```

- The compiler applies the converting constructor implicitly:

```
s = s + string(", ") + t;   // lvalue + rvalue + lvalue
```

55

Mimicking Built-In Operators

```
string operator+(string const &lo, string const &ro);
```

- The function returns its result by value.
- Calling this operator + yields an rvalue:

```
string *p = &(s + t);    // error: can't take the address
```

56

References

- C++11 introduced another kind of reference.
- What C++03 calls ***"references"***, C++11 calls ***"lvalue references"***.
- This distinguishes them from C++11's new ***"rvalue references"***.
- Except for the name change, lvalue references in C++11 behave just like references in C++03.

57

Rvalue References

- Whereas an ***lvalue reference*** declaration uses the ***&*** operator, an ***rvalue reference*** uses the ***&&*** operator.
- For example, this declares `ri` to be an "rvalue reference to `int`":

```
int &&ri = 10;
```

- You can use rvalue references as function parameters and return types, as in:

```
double &&f(int &&ri);
```

- You can also have an "rvalue reference to `const`", as in:

```
int const &&rci = 20;
```

58

Rvalue References

- Rvalue references bind only to rvalues.
- This is true even for "rvalue reference to `const`".
- For example,

```
int n = 10;
int &&ri = n;       // error: n is an lvalue
int const &&rj = n; // error: n is an lvalue
```

59

Move Operations

- Modern C++ uses rvalue references to implement move operations that can avoid unnecessary copying:

```
class string {
public:
    // copy operations
    string(string const &);                // constructor
    string &operator=(string const &);     // assignment

    // move operations
    string(string &&) noexcept;            // constructor
    string &operator=(string &&) noexcept; // assignment
};
```

60

Value Categories

- Modern C++ introduces a more complex categorization of expressions:

```
              expression
             ↙          ↘
        glvalue          rvalue
       ↙       ↘        ↙      ↘
  lvalue        xvalue          prvalue
```

- The newer categories are:
  - ***glvalue***: a "generalized" lvalue
  - ***prvalue***: a "pure" rvalue
  - ***xvalue***: an "expiring" lvalue

61

Thanks for Listening