CORE C++ 2019
DAN SAKS

LVALUES AND RVALUES

- lvalues & rvalues aren't really language features.
- rather they are semantic properties of expressions.
- understanding them provides valuable insight into:
  - the behaviour of built-in operators.
  - the code generated to execute those operators.
  - the meaning of some otherwise cryptic compiler messages
  - reference types
  - overloaded operators

LVALUES AND RVALUES HAVE EVOLVED

- in early C, the concepts of lvalue & rvalue were fairly simple.
- early C++ added classes, const, & references.
- the concepts got more complicated.
- origins of lvalue & rvalue from hisorical persp.

LVALUES

- in "the C programming language", Kernighan & ritchie wrote:
  - the name "lvalue" comes from the assignment expression

    E1 = E2

    in which left operand E1 must be an lvalue expression.
  - an <u>lvalue</u> is an expression referring to an object
  - an <u>object</u> is a region of storage

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.
- its an lvalue.
- 1 is a sub-expression not referring to an object.
- its an rvalue.

- an rvalue is simply an expression thats not an lvalue.

LVALUES AND RVALUES

- `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 either be an lvalue or rvalue.

    it can be any expression.

A LOOK UNDER THE HOOD

- why make this distinction between lvalues & rvalues?

- so that compilers can assume that rvalues don't necessarily occupy storage.

- this offers considerable freedom in generating code for rvalue expressions.

```
int n;   // a declaration for an integer object
            named n
~~~

n = 1;   // an assignment expression
```

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 like:

```
one:                ; label for the following locat.
    .word 1         ; allocate storage for 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
```

DATA STORAGE FOR RVALUES

- most 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.

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.

MUST BE AN LVALUE == CANT BE AN RVALUE

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

- this is trying to change the value of integer literal 1.
- of course, C (and C++) reject it as 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, its an rvalue.

RECAP

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

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

- caveat:

  although this is true for non-class types in C++, its not true for class types.

LITERALS

- most literals are rvalues, including:
  - numeric literals, such as 3 & 3.14159
  - character literals, such as 'a'.

- they don't necessarily occupy data storage.

- however, string literals, such as "xyzzy", are lvalues.

- they occupy data storage.

ENUMERATION CONSTANTS

- when used in expressions, enumeration constants are also rvalues.

```
enum color {red, green, blue};
color c;
```

```
c = green;   ✓
blue = green; X
```

LVALUES USED AS RVALUES

- an lvalue can appear on either side of assignment :

```
int m, n;

~ ~ ~

m = n;        // OK: m & n are both lvalues
```

- this assignment uses the expression n as an rvalue.

- officially, C++ performs an lvalue-to-rvalue conversion.

OPERANDS OF OTHER OPERATORS

- the concepts of lvalue and rvalue apply in all expressions.
  - not just assignment.

- for example, both the operands of the binary operator "+" must be expressions.
  - obviously, those expressions must have suitable types.

- but each operand can either be an <u>lvalue</u> or <u>rvalue</u>.

```
int x;

~ ~ ~

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

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

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 <u>temporary objects</u> are <u>rvalues.</u>

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?
```

- its an error because "m+1" yields an rvalue.

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 is not addressable.
- although the operand must be an lvalue, the result is an rvalue.

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

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 behaviour
```

- 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 behaviour.

UNARY *

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

```
*(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).

DATA STORAGE FOR EXPRESSIONS

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

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

NON-MODIFIABLE LVALUES

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

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

- name [0] is an lvalue, but its non-modifiable.
  - each element of a const array is itself const.

NON MODIFIABLE LVALUES

- lvalues & rvalues provide a vocabulary for describing subtle behavioral differences.
  - such as between enumerations constants & const objects.

- for example, this MAX is a constant of an unnamed enumeration type.

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

- unscoped enumeration values implicitely convert to integer.

- when MAX appears in an expression, it yields an integer rvalue.

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

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

NON MODIFIABLE LVALUES

- on the other hand, this MAX is a const-qualified object.

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

- when it appears in an expression, its 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
```

---

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

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 behaviour for const objects is analogous to the behaviour for inline functions.

REFERENCE TYPES

- the concepts of lvalues & 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 insted of pointers as function parameters and return types.

```
int i;          // define i as an integer object

~ ~ ~

int &ri = i;    // define ri as a "ref. to int"
```

- "ri" is an <u>alias</u> for "i."

REFERENCE TYPES

- a reference is essentially a pointer thats automatically dereferenced each time its used.

- you can rewrite most, if not all, code that uses a reference as a 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 is dereferenced (has a * infront of it) whenever you touch it.

- a reference yields an <u>lvalue</u>.

INITIALIZING VS ASSIGNING

- initializing a reference associates the reference with an object.
  - initializing a reference is also known as binding.

- assigning a reference stores through the reference and into the referenced object.

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

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...

REFERENCES AND OVERLOADED OPERATORS

```
enum month {
   Jan, Feb, ~~~ , 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++...
- 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.

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.
```

REFERENCES AND OVERLOADED OPERATORS

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

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

- this works, but not like a builtin ++:

```
++m;    // doesn't compile.

++&m;   // compiles, increments m, but looks wrong
```

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
```

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 x = static_cast<month> (x+1);  return
}
```

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

```
int j = ++m;   // OK

month n = ++m;   // OK
```

"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 non-const argument.
- when it appears in an expression, a "reference to const" yields a non-modifiable lvalue.

"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 behaviour as a function declared as:

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

- that is, the calls look & act very much the same.

"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.

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.

REFERENCES AND TEMPORARIES

- a "pointer to T" can only point 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


int i;

double *pd = &i;   //Can't convert pointers

double &rd = i;    // can't bind this either
```

REFERENCES AND TEMPORARIES

- there's an exception to the rule that a reference must bind to an lvalue of referenced type:
  - a "reference to const T" can bind to an expression x that is 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.

REFERENCES AND TEMPORARIES

- `int const &ri = 3;`

- when the program execution reaches this declaration, the program:

  1. creates an int temporary to hold the 3, and

  2. binds ri to the temporary.

- when execution leaves the scope containing ri, the program:

  3. destroys the temporary

REFERENCES AND TEMPORARIES

```
double const &rd = ri;
```

- When the 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.

REFERENCES AND TEMPORARIES

- this special behaviour enables passing by "reference to const" to consistently have the same outward behaviour as passing by value.

-

```
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
```

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.

MIMICKING BUILT-IN OPERATORS

- recall the behaviour 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 behaviour?
- consider a rudimentary (character) string class with + as a concatenation operator.

MIMICKING BUILT-IN OPERATORS

- you can declare operator + as a non-member:

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

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

string s = "hello";
string t = "world";

~ ~ ~

s = s + ", " + t;

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

lvalue or rvalue

compiler applies converting constructor implicitly

MIMICKING BUILT-IN OPERATORS

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

- the function returns its result by value.
- calling this operator + yields an rvalue.

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

REFERENCES

- C++11 introduces 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.

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;
```

RVALUE REFERENCES

- rvalue references bind only to rvalues.
- this is true even for "rvalue reference to const".

```
int n = 10;

int && ri = n;        //error: n is an lvalue

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

MOVE OPERATIONS

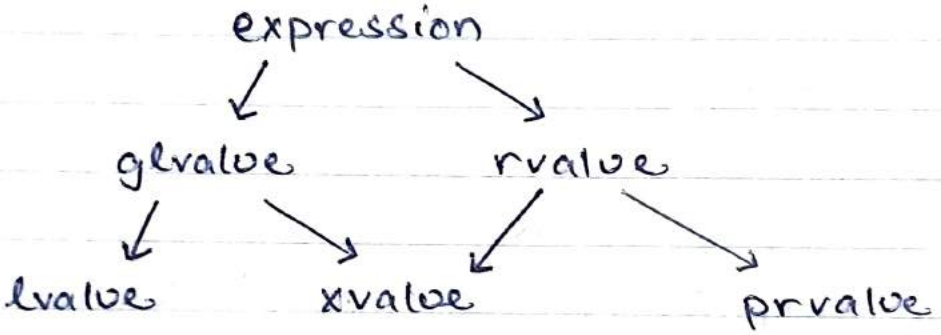
- modern C++ uses rvalue references to implement move operations that can avoid unnecessary copying:

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

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

VALUE CATEGORIES

- modern c++ introduces a more complex categorization of expressions:



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

—— x ——