Move semantics

Matt Warner

1 lvalues and rvalues

The term "lvalue" stands for "locator value". In early C, an lvalue was considered to be something that could appear on the left-hand side of an assignment operator ('='). It was named as such because it represents a value that can be located in memory.

The term "rvalue" stands for "right-hand side value". In early C, an rvalue was something that could appear only on the right-hand side of an assignment operator ('='). Rvalues were typically temporary values that did not have a persistant memory location.

Heres a simplifed Definition for lvalues and rvalues:

- **lvalue** - An object that occupies some identifiable location in memory
- **rvalue** - Temporary values that are used in expressions but do not persist beyond the expression in which they appear.

lvalue examples

```
int i;    // i is a lvalue
int* p = &i; // i's address is identifiable
i = 2; // Memory content is modified

class dog;  // lvalue of user defined type (class)
dog dl;

```

Note:-

Most variables in C++ code are lvalues

rvalue examples

```
int x = 2; // 2 is an rvalue
int x = i+2;  // (i+2) is an rvalue
int* p = &(i+2); // ERROR
i+2 = 4; // ERROR
2 = i; // ERROR

dog d1;
d1 = dog(); // dog() is rvalue of user defined type (class)

int sum(int x, int y) { return x+y; }
int i = sum(3,4); // sum(3,4) is rvalue
```

l-value references

the term "l-value reference" just refers to references in C++. Here are some examples of those:

```
int i;
int &r = i; // r is a reference to i;

int &r = 5; // Error

// Exception: Constant lvalue reference can be assigned a rvalue
const int &r = 5;


```

```
int square(int& x) { return x * x; }

square(i); // OK
square(40); // Error!

// Workaround:
int square(const int& x) { return x*x; } // square(40) and square(i) both work with this

```

In summary, reference variables can only refer to lvalues, except for const reference variables. They can refer to rvalues.

1.1 r-value references

An r-value referene in C++ is a reference type that was introduced to support move semantics, which allows efficient transfer of resources.

> **Note:-**
>
> The transfer of ownership facilitated by move semantics primarily applies to resources managed on the heap.

To understand the basic concept of r-value refernces, lets take a look at this example:

```
void print_int(int &i) { std::cout << "l-value reference: " << i << std::endl} // takes a l-value reference to i as parameter
void print_int(int &&i) { std::cout << "r-value reference: " << i << std::endl} // takes a r-value reference  to i as parameter
```

The following code demonstrates how to call both functions.

```
int a = 5;
print_int(a) // calls l-reference function (called with an l-value)
print_int(10) // calls r-reference function (called with an r-value)
```

As you can see, in its simplest form, functions with r-value reference parameters expect an r-value, and l-value references expect an l-value. To understand why r-value references are used to allow efficient tranfer of resources, we must take a look at **std::move**.

std::move is a utility function provided by C++ to facilitate move semantics. Its purpose is to cast a given argument into an r-value reference, which allows the programmer to indicate that they are potentially willing to "move" resources from the object being passed in. Take a look at this example:

```
std::vector<int> vec1{1,2,3,4};
std::vector<int> vec2{5,6,7,8};

// Assigns copys vec2 into vec1
vec1 = vec2;
```

In this example, we want vec1 to have the same contents of vec2. Without using std::move, we need copy the contents of vec2 and give them to vec1. However, using std::move, we can tranfer the resources to vec1 instead.

```
vec1 = std::move(vec2);
```

Now, instead of of coping over the contents of the vector, we are tranfering ownership of vec2's resources from vec2 to vec1, which is less costly.

> **Note:-**
>
> After std::move, vec2 is in a valid but unspecified state and should not be used.