15.4 — std::move

BY ALEX ON MARCH 4TH, 2017 | LAST MODIFIED BY ALEX ON MAY 20TH, 2018

Once you start using move semantics more regularly, you'll start to find cases where you want to invoke move semantics, but the objects you have to work with are l-values, not r-values. Consider the following swap function as an example:

```cpp
#include <iostream>
#include <string>

template<class T>
void swap(T& a, T& b)
{
  T tmp { a }; // invokes copy constructor
  a = b; // invokes copy assignment
  b = tmp; // invokes copy assignment
}

int main()
{
	std::string x{ "abc" };
	std::string y{ "de" };

	std::cout << "x: " << x << '\n';
	std::cout << "y: " << y << '\n';

	swap(x, y);

	std::cout << "x: " << x << '\n';
	std::cout << "y: " << y << '\n';

	return 0;
}
```

Passed in two objects of type T (in this case, std::string), this function swaps their values by making three copies. Consequently, this program prints:

```
x: abc
y: de
x: de
y: abc
```

As we showed last lesson, making copies can be inefficient. And this version of swap makes 3 copies. That leads to a lot of excessive string creation and destruction, which is slow.

However, doing copies isn't necessary here. All we're really trying to do is swap the values of a and b, which can be accomplished just as well using 3 moves instead! So if we switch from copy semantics to move semantics, we can make our code more performant.

But how? The problem here is that parameters a and b are l-value references, not r-value references, so we don't have a way to invoke the move constructor and move assignment operator instead of copy constructor and copy assignment. By default, we get the copy constructor and copy assignment behaviors. What are we to do?

std::move

In C++11, std::move is a standard library function that serves a single purpose -- to convert its argument into an r-value. We can pass an l-value to std::move, and it will return an r-value reference. std::move is defined in the utility header.

Here's the same program as above, but with a swap() function that uses std::move to convert our l-values into r-values so we can invoke move semantics:

```cpp
#include <iostream>
#include <string>
```

```cpp
#include <utility>

template<class T>
void swap(T& a, T& b)
{
  T tmp { std::move(a) }; // invokes move constructor
  a = std::move(b); // invokes move assignment
  b = std::move(tmp); // invokes move assignment
}

int main()
{
    std::string x{ "abc" };
    std::string y{ "de" };

    std::cout << "x: " << x << '\n';
    std::cout << "y: " << y << '\n';

    swap(x, y);

    std::cout << "x: " << x << '\n';
    std::cout << "y: " << y << '\n';

    return 0;
}
```

This prints the same result as above:

```
x: abc
y: de
x: de
y: abc
```

But it’s much more efficient about it. When tmp is initialized, instead of making a copy of x, we use std::move to convert l-value variable x into an r-value. Since the parameter is an r-value, move semantics are invoked, and x is moved into tmp.

With a couple of more swaps, the value of variable x has been moved to y, and the value of y has been moved to x.

Another example

We can also use std::move when filling elements of a container, such as std::vector, with l-values.

In the following program, we first add an element to a vector using copy semantics. Then we add an element to the vector using move semantics.

```cpp
#include <iostream>
#include <string>
#include <utility>
#include <vector>

int main()
{
    std::vector<std::string> v;
    std::string str = "Knock";

    std::cout << "Copying str\n";
    v.push_back(str); // calls l-value version of push_back, which copies str into the array element

    std::cout << "str: " << str << '\n';
    std::cout << "vector: " << v[0] << '\n';

    std::cout << "\nMoving str\n";

    v.push_back(std::move(str)); // calls r-value version of push_back, which moves str into the array
 element

```

```
    std::cout << "str: " << str << '\n';
    std::cout << "vector:" << v[0] << ' ' << v[1] << '\n';

    return 0;
}
```

This program prints:

```
Copying str
str: Knock
vector: Knock

Moving str
str:
vector: Knock Knock
```

In the first case, we passed push_back() an l-value, so it used copy semantics to add an element to the vector. For this reason, the value in str is left alone.

In the second case, we passed push_back() an r-value (actually an l-value converted via std::move), so it used move semantics to add an element to the vector. This is more efficient, as the vector element can steal the string's value rather than having to copy it. In this case, str is left empty.

At this point, it's worth reiterating that std::move() gives a hint to the compiler that the programmer doesn't need this object any more (at least, not in its current state). Consequently, you should not use std::move() on any persistent object you don't want to modify, and you should not expect the state of any objects that have had std::move() applied to be the same after they are moved!

Move functions should always leave your objects in a well-defined state

As we noted in the previous lesson, it's a good idea to always leave the objects being stolen from in some well-defined (deterministic) state. Ideally, this should be a "null state", where the object is set back to its uninitiazed or zero state. Now we can talk about why: with std::move, the object being stolen from may not be a temporary after all. The user may want to reuse this (now empty) object again, or test it in some way, and can plan accordingly.

In the above example, string str is set to the empty string after being moved (which is what std::string always does after a successful move). This allows us to reuse variable str if we wish (or we can ignore it, if we no longer have a use for it).

Where else is std::move useful?

std::move can also be useful when sorting an array of elements. Many sorting algorithms (such as selection sort and bubble sort) work by swapping pairs of elements. In previous lessons, we've had to resort to copy-semantics to do the swapping. Now we can use move semantics, which is more efficient.

It can also be useful if we want to move the contents managed by one smart pointer to another.

Conclusion

std::move can be used whenever we want to treat an l-value like an r-value for the purpose of invoking move semantics instead of copy semantics.

15.5 -- std::unique_ptr

Index

15.3 -- Move constructors and move assignment