22.4 — std::move

ALEX SEPTEMBER 20, 2023

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:

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

template<class T>
void myswapCopy(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';

    myswapCopy(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 casts (using static_cast) its argument into an r-value reference, so that move semantics can be invoked. Thus, we can use std::move to cast an l-value into a type that will prefer being moved over being copied. std::move is defined in the utility header.

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

```
#include <iostream>
#include <string>
#include <utility> // for std::move

template<class T>
void myswapMove(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';

    myswapMove(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.

```
#include <iostream>
#include <string>
#include <utility> // for std::move
#include <vector>

int main()
{
    std::vector<std::string> v;

    // We use std::string because it is movable (std::string_view is not)
    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'; // The result of this is indeterminate
    std::cout << "vector:" << v[0] << ' ' << v[1] << '\n';

    return 0;
}
```

On the author's machine, 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.

Moved from objects will be in a valid, but possibly indeterminate state

When we move the value from a temporary object, it doesn't matter what value the moved-from object is left with, because the temporary object will be destroyed immediately anyway. But what about lvalue objects that we've used std::move() on? Because we can continue to access these objects after their values have been moved (e.g. in the example above, we print the value of `str` after it has been moved), it is useful to know what value they are left with.

There are two schools of thought here. One school believes that objects that have been moved from should be reset back to some default / zero state, where the object does not own a resource any more. We see an example of this above, where `str` has been cleared to the empty string.

The other school believes that we should do whatever is most convenient, and not constrain ourselves to having to clear the moved-from object if its not convenient to do so.

So what does the standard library do in this case? About this, the C++ standard says, "Unless otherwise specified, moved-from objects [of types defined in the C++ standard library] shall be placed in a valid but unspecified state."

In our example above, when the author printed the value of `str` after calling `std::move` on it, it printed an empty string. However, this is not required, and it could have printed any valid string, including an empty string, the original string, or any other valid string. Therefore, we should avoid using the value of a moved-from object, as the results will be implementation-specific.

In some cases, we want to reuse an object whose value has been moved (rather than allocating a new object). For example, in the implementation of myswapMove() above, we first move the resource out of `a`, and then we move another resource into `a`. This is fine because we never use the value of `a` between the time where we move it out and the time where we give `a` a new determinate value.

With a moved-from object, it is safe to call any function that does not depend on the current value of the object. This means we can set or reset the value of the moved-from object (using `operator=`, or any kind of `clear()` or `reset()` member function). We can also test the state of the moved-from object (e.g. using `empty()` to see if the object has a value). However, we should avoid functions like `operator[]` or `front()` (which returns the first element in a container), because these functions depend on the container having elements, and a moved-from container may or may not have elements.

Key insight

`std::move()` gives a hint to the compiler that the programmer doesn't need the value of an object any more. Only use `std::move()` on persistent objects whose value you want to move, and do not make any assumptions about the value of the object beyond that point. It is okay to give a moved-from object a new value (e.g. using `operator=`) after the current value has been moved.

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.

Related content

There is a useful variant of `std::move()` called `std::move_if_noexcept()` that returns a movable r-value if the object has a `noexcept` move constructor, otherwise it returns a copyable l-value. We cover this in lesson 27.10 -- std::move_if_noexcept (https://www.learncpp.com/cpp-tutorial/stdmove_if_noexcept/).

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.

Next lesson
22.5 std::unique_ptr

Back to table of contents

Previous lesson
22.3 Move constructors and move assignment