M.4 — std::move

ALEX   APRIL 26, 2021

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 myswap(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';

    myswap(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 myswap() 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 myswap(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';

    myswap(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;
    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 uninitiatized 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.

Next lesson
M.5 std::move_if_noexcept

Back to table of contents

Previous lesson
M.3 Move constructors and move assignment

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies: POST COMMENT

Avatars from https://gravatar.com/ are connected to your provided email address.

132 COMMENTS Newest

Omer
February 5, 2022 6:49 am

Hi,
I have a quick question, move semantic is pretty obvious when dealing with dynamically allocated memory, but it's not obvious for me how it works with other non dynamically allocated data types, from what I understand in the case of dynamically allocated memory you just make the new pointer point to the preexisting data (so your essentially copying the address of the data instead of the data itself). Is that the same with fundamental data types? and if so How does that happen (because you're not dealing with pointers here).

I hope I was clear and understandable,
Thank you

0 Reply

> **Alex** Author
> Reply to Omer February 5, 2022 9:47 am
>
> Yup, move only works for things that can be moved. If move semantics isn't appropriate, copy semantics will be invoked instead.
>
> 1 Reply
>
> > **Omer**
> > Reply to Alex February 6, 2022 3:47 am
> >
> > Got it, Thanks a lot.
> >
> > 0 Reply

Rekt0707
December 12, 2021 9:36 pm

is moving fundamental types like int, char more efficient than copying them?

0 Reply

> **Alex** Author
> Reply to Rekt0707 December 15, 2021 9:12 am
>
> If they are dynamically allocated, yes. Otherwise no.
>
> 0 Reply

0xff
November 22, 2021 4:09 pm

In the comment of first snippet, you've said move constructor is invoked. However, there is no class with move constructor defined. In previous lesson it was told that there's no implicit move constructor unlike copy constructor. So, what move constructor did it invoke if move constructor isn't even present?

Last edited 3 months ago by 0xff

0 Reply

> **Alex** Author
> Reply to 0xff November 24, 2021 4:11 pm
>
> These examples are working with objects of type std::string, which has a move constructor.
>
> 2 Reply

Taylor7500
November 2, 2021 9:56 am

Based on this, is it good practice to use std::move() and move assignment wherever possible, and wherever it fits in the program?

Does that extend to fundamental data types or are they so small that the memory footprint isn't worth considering? For example, one object I've made is little more than a wrapper around an std::vector of doubles, with a few functions to manipulate them in particular ways. As such, it makes use of the push_back() function a fair amount - would it be better to add a std::move to every instance where I want to add an entry and don't care about the original identifier for that value after that line, or is doing that for doubles not worth the hassle?

0 Reply

> **Alex** Author
> Reply to Taylor7500 November 2, 2021 12:52 pm
>
> Copying fundamental types is fast so there is no need to move them. But yes, if you want std::vector::push_back() to be able to steal the resource from a named object that you're passing in, you can std::move it.
>
> 0 Reply

Tejero Joshua
October 12, 2021 5:19 pm

on the second panel of code you say that the code invokes move constructor/move assignment.

how is that happened? isn't std::move is just a templated function from standard library?

0 Reply

> **Alex** Author
> Reply to Tejero Joshua October 14, 2021 12:08 pm
>
> Move semantics only work with r-value references, so we use std::move() to convert our l-value references to r-value references so they can be moved instead of copied (if a move constructor exists).
>
> 2 Reply
>
> > **Tejero Joshua**
> > Reply to Alex October 17, 2021 6:23 pm
> >
> > Got it, thanks.
> >
> > 0 Reply

Asgar
September 15, 2021 8:07 am

About this line:
"In C++11, std::move is a standard library function that casts (using static_cast) its argument into an r-value reference."
Please remove the last word "reference". I think you meant r-value, not r-value reference.

0 Reply

> **Alex** Author
> Reply to Asgar September 19, 2021 10:50 am
>
> I did mean r-value reference. See https://en.cppreference.com/w/cpp/utility/move
> "It is exactly equivalent to a static_cast to an rvalue reference type."
>
> 0 Reply

New here
July 23, 2021 11:09 pm

Hello,

I have a question. So move semantics only work for pointers and references (because others cannot be moved without copying)?

and if yes, would be good if you could add a little more explanation for std::vector example, since I'm guessing (definitely not sure) in std::vector implementation they should have used reference version of ownership transfer! would be good if we could have an example of ownership transfer using references.

I know internally references are implemented using pointers, but, not sure what you do with the old reference (set to what) when the new reference owns the object. In case of pointers, we saw that you could simply set the old pointer to null.

0 Reply

Michael
April 24, 2021 12:19 pm

Hi Alex,
hi nascardriver,

I think it would be nice if you mention that std::move is just a type casting. I checked the header file and saw that std::move just uses a static_cast<std::string&&>(...). There is a different type in the angled bracket, I just wrote std::string for simplicity.

0 Reply

> **Jorge Diaz**
> Reply to Michael September 8, 2021 7:06 am
>
> So rewriting the "myswap" function like this
>
> ```
> template <class T>
> void myswap(T& a, T& b)
> {
>     T temp{ static_cast<T&&>(a) };
>     a = static_cast<T&&>(b);
>     b = static_cast<T&&>(temp);
> }
> ```
>
> yields the same result...
>
> That's a very interesting fact
>
> 0 Reply
>
> **Alex** Author
> Reply to Michael April 26, 2021 8:30 am
>
> Updated. Thanks for the suggestion!
>
> 0 Reply

Nishant
January 16, 2021 5:59 am

Hey,

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

template<class T>
void myswap(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';

    myswap(x, y);

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

    return 0;
}
```

I tried this and expected the compiler to throw an error saying you can't pass an l-value to an r value reference or something along those lines but that didn't happen.
Instead I got this:
x: abc
y: de
x: de
y: de

Which means the copy constructor did a shallow copy. Why is that?

Thanks.

0 Reply

> **Tomek**
> Reply to Nishant February 22, 2021 9:10 am
>
> i think that tmp is referencing the same variable as a
> so when you change a you also change temp
>
> 0 Reply

Lucky Abby
December 27, 2020 1:07 am

Hello nascardriver,

I don't know if this is bug or mistake I've made.
here is the code:

```
#include <iostream>
#include <algorithm>
#include <string>
#include <vector>

int main () {
    std::vector<std::string> v1{"spring","summer","winter","fall"};
    std::vector<std::string> v2(8);

    std::fill(v2.begin(), v2.end(), "=");
    std::move(v1.begin(), v1.end(), v2.begin()+2);

    std::cout << "v1: ";
    for(auto &e:v1) std::cout << e << ' '; std::cout << '\n';
    std::cout << "v2: ";
    for(auto &e:v2) std::cout << e << ' '; std::cout << '\n';

    return 0;
}
```

as you can see that vector v2 filled with "=" and v1 already initialized with values, but surprisingly after move operation, v1 supposed to be empty but filled with "=", the output is showing as follow:

v1: = = = =
v2: = = spring summer winter fall = =

could you please show me what is wrong here.

Thank you.

0 Reply

> **nascardriver** Sub-admin
> Reply to Lucky Abby December 27, 2020 2:58 am
>
> There's nothing wrong, this is valid behavior.
> When moving from a `std::string` (Or any type in general), the moved-from object is in a valid but unspecified state. Reading from it is safe, but there's no way to tell what the value is (Unless specified).
> By the looks of it, your standard library's `std::string` move simply swaps the `std::string`s.
>
> https://en.cppreference.com/w/cpp/string/basic_string/operator%3D
> Overload 2
>
> 0 Reply
>
> > **Lucky Abby**
> > Reply to nascardriver December 27, 2020 4:11 pm
> >
> > so, the meaning of unspecified is same as uninitialized, am I right?
> >
> > 0 Reply
> >
> > > **nascardriver** Sub-admin
> > > Reply to Lucky Abby December 28, 2020 3:10 am
> > >
> > > Accessing an uninitialized variable causes undefined behavior. That's not the case if the variable just has an unspecified value. You can safely read from the variable, but it doesn't make sense because you can't predict what the value will be.
> > >
> > > 0 Reply
> > >
> > > > **Lucky Abby**
> > > > Reply to nascardriver December 29, 2020 6:22 am
> > > >
> > > > Thanks
> > > >
> > > > 0 Reply