22.4 — std::move

ALEX[1] MAY 8, 2024

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 <typename 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 <typename 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/)[2].

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

3

[Back to table of contents](#)

4

[Previous lesson](#)
22.3 Move constructors and move assignment

5

6

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

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[8] are connected to your provided email address.

191 COMMENTS Newest

NordicCat
January 15, 2025 10:04 am PST

```
int length{};
cout << "Enter the number of names: ";
cin >> length;

assert(length > 0);
vector<string> myVector(length);

for (int i = 0; i < length; i++) {

    string temp{};
    cout << "Enter name " << i + 1 << ": ";
    cin >> temp;
    myVector.push_back(std::move(temp));
}

for (const auto& names : myVector) {
    cout << names << '\n';
}
```

for this case we don't need to use any reset() func but should we use any reset function in general when we want to transfer resources from ob1 -> obj2 ?

0 Reply

RimShaun
September 16, 2024 1:04 am PDT

Hello, I have the following code

```
#include <iostream>

void foo(std::string&& bar){

    std::string bar2{"asf"};
    bar2 = bar;
    std::cout << bar2 << " " << bar << std::endl;
    bar2 = std::move(bar);
    std::cout << bar2 << " " << bar << std::endl;

}

int main() {
    foo(std::string("hoo"));

    return 0;
}
```

I got the output:

```
hoo hoo
hoo
```

My question is the following: if std::move() just casts an l-value to an r-value reference, why are move semantics not used on the statement "bar2 = bar;"? Since bar is already an r-value reference, shouldn't it be considered by the compiler as the same thing as std::move(bar)?

Last edited 9 months ago by RimShaun

1 Reply

RimShaun

Reply to RimShaun [9] September 18, 2024 1:14 am PDT

I reread the course and it's because bar is an lvalue of type std::string&& so bar2 = bar is going to be a copy assignment. Then does that mean that std::move actually casts to an rvalue reference as a type, but since it's a function, std::move(bar) itself has a value category of rvalue (a function return)?
But then why would std::move need to change/cast the type of bar for bar2 to perform a move assignment? It just needs to have a value category of rvalue (and being a string), no ?

Last edited 9 months ago by RimShaun

1 Reply

Alex Author

Reply to RimShaun [10] September 22, 2024 8:20 pm PDT

Move semantics are used. This version of your program makes it clearer:

```
#include <iostream>

void fooCopy(std::string&& from)
{
    std::string to{"asf"};
    std::cout << to << "/" << from << '\n';
    to = from; // copy
    std::cout << to<< "/" << from << '\n'; // prints hoo/hoo
}

void fooMove(std::string&& from)
{
    std::string to{"asf"};
    std::cout << to << "/" << from << '\n';
    to = std::move(from); // move
    std::cout << to << "/" << from << '\n'; //prints hoo/ (on GCC)
}

int main()
{
    fooCopy(std::string("hoo"));
    fooMove(std::string("hoo"));

    return 0;
}
```

That said, moved-from objects are placed in a valid but unspecified state, so printing `bar` after moving from it may yield a different result depending on the platform.

1 Reply

Selviniah

August 4, 2024 12:28 pm PDT

```
Object1 = std::move(Object2);
Object1 = static_cast<Test&&>(Object2);
```

Is this both completely same?

0 Reply

Alex Author

Reply to Selviniah [11] August 5, 2024 3:38 pm PDT

Yes, if Object2 isn't a reference.

For this reason, `std::move` is typically defined as `static_cast<typename std::remove_reference<T>::type&&>(t)` so it can handle reference types too.

1 Reply

Buckley

May 6, 2024 7:51 pm PDT

Why is the template a `class` instead of `typename`?

0 Reply

Alex Author

Reply to Buckley [12] May 8, 2024 2:57 pm PDT

Old code. I've updated it to use typename.

2 Reply

Suiren

March 21, 2024 5:15 am PDT

So cool modern cpp tutorial. Would you plan extending this chapter in remaster version Alex?

0 Reply

Alex Author

Reply to Suiren [13] March 21, 2024 2:10 pm PDT

I don't think I have any updates on my todo for this chapter. Is there something you think is missing?

0 Reply

Suiren

Reply to Alex [14] March 21, 2024 10:58 pm PDT

Not actually, just gratitude for your effort :)

0 Reply

Great Ape
February 21, 2024 12:21 pm PST

I don't quite understand why the moved-from objects would be left in an indeterminate state. There are 2 possibilities:

(1) The moved-from object held dynamically allocated memory. In this case, the move constructor / assignment operator would have to set the object's pointer to null, in order to implement move semantics correctly. Thus its final state is determinate.
(2) The moved-from object didn't hold dynamically allocated memory. Then there is no reason to use move semantics, as there is no "ownership" to transfer, and copy semantics are all that's needed.

Did I miss something?

0 Reply

> 
>
> **Alex** Author
> Reply to Great Ape[15] February 23, 2024 1:48 pm PST
>
> The term "indeterminate" here really means "unspecified". A moved-from object can do whatever it wants, so long as it remains in some valid state. So it's better to just not assume anything about what state it's left in (and this is fine, because we typically don't need to access the state of moved-from objects).
>
> 0 Reply

Lilac
December 15, 2023 10:13 pm PST

So basically `std::vector::emplace_back` uses move semantics?

0 Reply

> 
>
> **Alex** Author
> Reply to Lilac[16] December 18, 2023 2:49 pm PST
>
> Both push_back and emplace_back can use move semantics. The difference is that emplace_back employs another feature called perfect forwarding (which isn't yet covered in this series) to avoid a move in some cases.
>
> Remember that most often, a move is implemented as a copy -- so avoiding a move on a non-movable type is valuable.
>
> 0 Reply

VexedBannister
September 30, 2023 12:18 pm PDT

> "std::move can also be useful when sorting an array of elements"

Does std::swap use move semantics?

0 Reply

Alex Author

Reply to VexedBannister 17 October 2, 2023 12:21 pm PDT

Yes, if they are defined.

1 Reply

Raid

September 25, 2023 12:45 am PDT

So according to examples in 22.3, should move semantics only work with pointer types? `m_ptr=a.m_ptr` is easy to understand.
Can it work with any types like `myswapCopy` shows?

0 Reply

Alex Author

Reply to Raid 18 September 26, 2023 8:03 pm PDT

Ideally we want move semantics to actually do a move. This is typically done by "transferring" resources via reassigning a pointer. Stack-allocated data can't be moved -- it must be copied.

Last edited 1 year ago by Alex

0 Reply

noctis

July 21, 2023 6:25 am PDT

```
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.
```

How does this even work `static_cast<T&&>(Obj)` ? - (1)

My View - Here we are creating a `r-value reference` but `Obj` is `lvalue`/`lvalueReference`.
Static_cast creates a temporary object of a given object to a object of requested type in general (e.g - `static_cast<double>(int{5})`).

So in the above (1), are we creating a `temporary Obj` and binding a reference to that r-value ?

0 Reply

Alex Author

Reply to noctis [19] July 23, 2023 12:29 pm PDT

When we cast to a reference type, a temporary reference is created (not a temporary value).

So `static_cast<T&&>(Obj)` creates a temporary rvalue reference bound to `Obj`.

2 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/stdmove_if_noexcept/
3. https://www.learncpp.com/cpp-tutorial/stdunique_ptr/
4. https://www.learncpp.com/
5. https://www.learncpp.com/cpp-tutorial/move-constructors-and-move-assignment/
6. https://www.learncpp.com/stdmove/
7. https://www.learncpp.com/cpp-tutorial/stdinitializer_list/
8. https://gravatar.com/
9. https://www.learncpp.com/cpp-tutorial/stdmove/#comment-602017
10. https://www.learncpp.com/cpp-tutorial/stdmove/#comment-602079
11. https://www.learncpp.com/cpp-tutorial/stdmove/#comment-600483
12. https://www.learncpp.com/cpp-tutorial/stdmove/#comment-596766
13. https://www.learncpp.com/cpp-tutorial/stdmove/#comment-594913
14. https://www.learncpp.com/cpp-tutorial/stdmove/#comment-594945
15. https://www.learncpp.com/cpp-tutorial/stdmove/#comment-593895
16. https://www.learncpp.com/cpp-tutorial/stdmove/#comment-591080
17. https://www.learncpp.com/cpp-tutorial/stdmove/#comment-588004
18. https://www.learncpp.com/cpp-tutorial/stdmove/#comment-587725
19. https://www.learncpp.com/cpp-tutorial/stdmove/#comment-584180
20. https://g.ezoic.net/privacy/learncpp.com