What is std::move(), and when should it be used?

Asked 11 years, 11 months ago Modified 6 months ago Viewed 425k times

931

320

1. What is it?
2. What does it do?
3. When should it be used?

Good links are appreciated.

c++ c++11 move-semantics c++-faq stdmove

Share Follow

edited Aug 21, 2019 at 14:57
Fabio says Reinstate Monica
4,563 7 39 57

asked Aug 5, 2010 at 9:42
Basilevs
20.7k 15 55 98

54 Bjarne Stroustrup explains move in A Brief Introduction to Rvalue References – DumbCoder Aug 5, 2010 at 9:49

2 Move semantics – Basilevs Sep 11, 2014 at 4:12

16 This question is referring to `std::move(T && t)` ; there also exists a `std::move(InputIt first, InputIt last, OutputIt d_first)` which is an algorithm related to `std::copy` . I point it out so others aren't as confused as I was when first confronted with a `std::move` taking three arguments. en.cppreference.com/w/cpp/algorithm/move – josaphatv Oct 17, 2016 at 22:48

1 Recommend reading this if you don't have much of an idea what lvalue and rvalue references mean internalpointers.com/post/... – Karan Singh Jun 25, 2020 at 15:22

To clarify, this question is about the `std::move` from `<utility>` , not the `std::move` from `<algorithm>` . – Adrian McCarthy Apr 18 at 13:43

8 Answers

Sorted by:
Trending sort available
Highest score (default)

446

1. "What is it?"

While `std::move()` is technically a function - I would say **it isn't *really* a function**. It's sort

> You can plunder me, **move** anything I'm holding and use it elsewhere (since I'm going to be destroyed soon anyway)".

in other words, when you use `std::move(x)`, you're allowing the compiler to cannibalize `x`. Thus if `x` has, say, its own buffer in memory - after `std::move()`ing the compiler can have another object own it instead.

You can also move from a prvalue (such as a temporary you're passing around), but this is rarely useful.

3. "When should it be used?"

Another way to ask this question is "What would I cannibalize an existing object's resources for?" well, if you're writing application code, you would probably not be messing around a lot with temporary objects created by the compiler. So mainly you would do this in places like constructors, operator methods, standard-library-algorithm-like functions etc. where objects get created and destroyed automagically a lot. Of course, that's just a rule of thumb.

A typical use is 'moving' resources from one object to another instead of copying. @Guillaume links to this page which has a straightforward short example: swapping two objects with less copying.

```
template <class T>
swap(T& a, T& b) {
    T tmp(a);   // we now have two copies of a
    a = b;      // we now have two copies of b (+ discarded a copy of a)
    b = tmp;    // we now have two copies of tmp (+ discarded a copy of b)
}
```

using move allows you to swap the resources instead of copying them around:

```
template <class T>
swap(T& a, T& b) {
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}
```

Think of what happens when `T` is, say, `vector<int>` of size n. In the first version you read and write 3*n elements, in the second version you basically read and write just the 3 pointers to the vectors' buffers, plus the 3 buffers' sizes. Of course, class `T` needs to know how to do the moving; your class should have a move-assignment operator and a move-

description you've given it just seems like it's a shallow copy instead of a deep copy. – Zebrafish Dec 30, 2016 at 10:52

25 @TitoneMaurice: Except that it's not a copy - as the original value is no longer usable. – einpoklum Dec 30, 2016 at 11:52

7 @Zebrafish you couldn't be more wrong. A shallow copy leaves the original in the exact same state, a move usually results in the original being empty or in an otherwise valid state. – rubenvb Jan 10, 2018 at 22:38

59 @rubenvb Zebra isn't entirely wrong. While it's true that the original cannabilised object is usually deliberately sabotaged to avoid confusing errors (e.g. set its pointers to nullptr to signal that it no longer owns the pointees), the fact that the whole move is implemented by simply copying a pointer from the source to the destination (and deliberately avoiding doing anything with the pointee) is indeed reminiscent of a shallow copy. In fact, I would go so far as to say that a move *is* a shallow copy, followed optionally by a partial self-destruct of the source. *(cont.)* – Lightness Races in Orbit Nov 1, 2018 at 18:54

16 *(cont.)* If we permit this definition (and I rather like it), then @Zebrafish's observation isn't wrong, just slightly incomplete. – Lightness Races in Orbit Nov 1, 2018 at 18:55

395

[Wikipedia Page on C++11 R-value references and move constructors](#)

1. In C++11, in addition to copy constructors, objects can have move constructors. (And in addition to copy assignment operators, they have move assignment operators.)
2. The move constructor is used instead of the copy constructor, if the object has type "rvalue-reference" (`Type &&`).
3. `std::move()` is a cast that produces an rvalue-reference to an object, to enable moving from it.

It's a new C++ way to avoid copies. For example, using a move constructor, a `std::vector` could just copy its internal pointer to data to the new object, leaving the moved object in an moved from state, therefore not copying all the data. This would be C++-valid.

Try googling for move semantics, rvalue, perfect forwarding.

Share Follow

edited Feb 23, 2020 at 15:41
Noch
7 3

answered Aug 5, 2010 at 9:52
Scharron
16.6k 6 42 63

46 Move-semantics require the moved object remain *valid*, which is not an incorrect state. (Rationale: It still has to destruct, make it work.) – GManNickG Aug 5, 2010 at 16:37

19 @GMan: well, it has to be in a state that is safe to destruct, but, AFAIK, it does not have to be

`move` its only a cast to pass a value from one point to another where the original lvalue will no longer be used. – Manu343726 Jul 3, 2014 at 19:35

29 I'd go further. `std::move` itself does "nothing" - it has zero side effects. It just signals to the compiler that the programmer doesn't care what happens to that object any more. i.e. it gives *permission* to other parts of the software to move from the object, but it doesn't require that it be moved. In fact, the recipient of an rvalue reference doesn't have to make any promises about what it will or will not do with the data. – Aaron McDaid Aug 20, 2015 at 14:17

162

You can use move when you need to "transfer" the content of an object somewhere else, without doing a copy (i.e. the content is not duplicated, that's why it could be used on some non-copyable objects, like a unique_ptr). It's also possible for an object to take the content of a temporary object without doing a copy (and save a lot of time), with std::move.

This link really helped me out :

http://thbecker.net/articles/rvalue_references/section_01.html

I'm sorry if my answer is coming too late, but I was also looking for a good link for the std::move, and I found the links above a little bit "austere".

This puts the emphasis on r-value reference, in which context you should use them, and I think it's more detailed, that's why I wanted to share this link here.

Share Follow

edited May 1, 2019 at 7:30

q-l-p
3,740 3 14 33

answered Jun 21, 2012 at 23:47

Guillaume
1,721 2 15 21

28 Nice link. I always found the wikipedia article, and other links that I stumbled across rather confusing since they just throw facts at you, leaving it to you to figure out what the actual meaning/rationale is. While "move semantics" in a constructor is rather obvious, all those details about passing &&-values around are not... so the tutorial-style description was very nice. – Christian Stieber Jul 15, 2012 at 12:12

107

Q: What is `std::move` ?

A: `std::move()` is a function from the C++ Standard Library for casting to a rvalue reference.

Simplisticly `std::move(t)` is equivalent to:

```
static_cast<T&&>(t);
```

An implementation for std::move() is given in N2027: "A Brief Introduction to Rvalue References" as follows:

```
template <class T>
typename remove_reference<T>::type&&
std::move(T&& a)
{
    return a;
}
```

As you can see, `std::move` returns `T&&` no matter if called with a value (`T`), reference type (`T&`), or rvalue reference (`T&&`).

Q: What does it do?

A: As a cast, it does not do anything during runtime. It is only relevant at compile time to tell the compiler that you would like to continue considering the reference as an rvalue.

```
foo(3 * 5); // obviously, you are calling foo with a temporary (rvalue)

int a = 3 * 5;
foo(a);     // how to tell the compiler to treat `a` as an rvalue?
foo(std::move(a)); // will call `foo(int&& a)` rather than `foo(int a)` or
`foo(int& a)`
```

What it does **not** do:

- Make a copy of the argument
- Call the copy constructor
- Change the argument object

Q: When should it be used?

A: You should use `std::move` if you want to call functions that support move semantics with an argument which is not an rvalue (temporary expression).

This begs the following follow-up questions for me:

- What is move semantics? Move semantics in contrast to copy semantics is a programming technique in which the members of an object are initialized by 'taking over' instead of copying another object's members. Such 'take over' makes only sense with pointers and resource handles, which can be cheaply transferred by copying the

- Why can't the compiler figure it out on its own? The compiler cannot just call another overload of a function unless you say so. You must help the compiler choose whether the regular or move version of the function should be called.
- In which situations would I want to tell the compiler that it should treat a variable as an rvalue? This will most likely happen in template or library functions, where you know that an intermediate result could be salvaged (rather than allocating a new instance).

Share Follow

edited Jan 10 at 15:56

answered Feb 20, 2017 at 9:26

Christopher Oezbek
21k 6 53 81

8 Big +1 for code examples with semantics in comments. The other top answers define std::move using "move" itself - doesn't really clarify anything! --- I believe it's worth mentioning that not making a copy of the argument means that the original value cannot be reliably used. – t.y Jun 7, 2018 at 18:48

44

std::move itself doesn't really do much. I thought that it called the moved constructor for an object, but it really just performs a type cast (casting an lvalue variable to an rvalue so that the said variable can be passed as an argument to a move constructor or assignment operator).

So std::move is just used as a precursor to using move semantics. Move semantics is essentially an efficient way for dealing with temporary objects.

Consider Object `A = B + (C + (D + (E + F)));`

This is nice looking code, but E + F produces a temporary object. Then D + temp produces another temporary object and so on. In each normal "+" operator of a class, deep copies occur.

For example

```
Object Object::operator+ (const Object& rhs) {
    Object temp (*this);
    // logic for adding
    return temp;
}
```

The creation of the temporary object in this function is useless - these temporary objects will be deleted at the end of the line anyway as they go out of scope.

This avoids needless deep copies being made. With reference to the example, the only part where deep copying occurs is now E + F. The rest uses move semantics. The move constructor or assignment operator also needs to be implemented to assign the result to A.

Share Follow

edited Jul 28, 2021 at 18:33
Dark Dududu
3 3

answered Jun 5, 2013 at 7:55
user929404
2,025 22 26

3 you spoke about move semantics . you should add to your answer as how std::move can be used because the question asks about that. – Koushik Shetty Jun 5, 2013 at 8:01

2 @Koushik std::move doesnt do much - but is used to implement move semantics. If you don't know about std::move, you probably dont know move semantics either – user929404 Jun 5, 2013 at 8:56

1 "doesnt do much"(yes just a static_cast to to a rvalue reference). what actually does it do and y it does is what the OP asked. you need not know how std::move works but you got to know what move semantics does. furthermore, "but is used to implement move semantics" its the otherway around. know move semantics and you'l understand std::move otherwise no. move just helps in movement and itself uses move semantics. std::move does nothing but convert its argument to rvalue reference, which is what move semantics require. – Koushik Shetty Jun 5, 2013 at 9:08

11 "but E + F produces a temporary object" - Operator + goes left to right, not right to left. Hence `B+C` would be first! – Ajay Oct 15, 2015 at 7:20

only your answer explained it to me – Ulterior Nov 10, 2020 at 11:53

10

"What is it?" and **"What does it do?"** has been explained above.

I will give a example of **"when it should be used".**

For example, we have a class with lots of resource like big array in it.

```
class ResHeavy{ //  ResHeavy means heavy resource
    public:
        ResHeavy(int len=10):_upInt(new int[len]),_len(len){
            cout<<"default ctor"<<endl;
        }

        ResHeavy(const ResHeavy& rhs):_upInt(new int[rhs._len]),_len(rhs._len){
            cout<<"copy ctor"<<endl;
        }

        ResHeavy& operator=(const ResHeavy& rhs){
            _upInt.reset(new int[rhs._len]);
            _len = rhs._len;
            cout<<"operator= ctor"<<endl;
        }
```



```
// check array valid
```

```
    bool is_up_valid(){
        return _upInt != nullptr;
    }

    private:
        std::unique_ptr<int[]> _upInt; // heavy array resource
        int _len; // length of int array
};
```

Test code:

```
void test_std_move2(){
    ResHeavy rh; // only one int[]
    // operator rh

    // after some operator of rh, it becomes no-use
    // transform it to other object
    ResHeavy rh2 = std::move(rh); // rh becomes invalid

    // show rh, rh2 it valid
    if(rh.is_up_valid())
        cout<<"rh valid"<<endl;
    else
        cout<<"rh invalid"<<endl;

    if(rh2.is_up_valid())
        cout<<"rh2 valid"<<endl;
    else
        cout<<"rh2 invalid"<<endl;

    // new ResHeavy object, created by copy ctor
    ResHeavy rh3(rh2);  // two copy of int[]

    if(rh3.is_up_valid())
        cout<<"rh3 valid"<<endl;
    else
        cout<<"rh3 invalid"<<endl;
}
```

output as below:

```
default ctor
move ctor
rh invalid
rh2 valid
copy ctor
rh3 valid
```

We can see that `std::move` with `move constructor` makes transform resource easily.

Where else is `std::move` useful?

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

Cited:

- https://www.learncpp.com/cpp-tutorial/15-4-stdmove/

Share Follow

edited Apr 26, 2019 at 10:28

Lightness Races in Orbit
370k 73 621 1022

answered Aug 3, 2018 at 5:37

Jayhello
5,145 3 45 54

3

`std::move` itself does nothing rather than a `static_cast` . According to cppreference.com

> It is exactly equivalent to a static_cast to an rvalue reference type.

Thus, it depends on the type of the variable you assign to after the `move` , if the type has `constructors` or `assign operators` that takes a rvalue parameter, it may or may not *steal* the content of the original variable, so, it may leave the original variable to be in an `unspecified state` :

> Unless otherwise specified, all standard library objects that have been moved from being placed in a valid but unspecified state.

Because there is no special `move constructor` or `move assign operator` for built-in literal types such as integers and raw pointers, so, it will be just a simple copy for these types.

Share Follow

edited Jul 8, 2020 at 3:34

Yoon5oo
490 5 11

answered Jul 1, 2020 at 12:24

Tiger Yu
706 3 5

0

Here is a full example, using std::move for a (simple) custom vector

Expected output:

```
c: [10][11]
copy ctor called
copy of c: [10][11]
move ctor called
moved c: [10][11]
```


Code:

```
#include <iostream>
#include <algorithm>

template<class T> class MyVector {
private:
    T *data;
    size_t maxlen;
    size_t currlen;
public:
    MyVector<T> () : data (nullptr), maxlen(0), currlen(0) { }
    MyVector<T> (int maxlen) : data (new T [maxlen]), maxlen(maxlen),
currlen(0) { }

    MyVector<T> (const MyVector& o) {
        std::cout << "copy ctor called" << std::endl;
        data = new T [o.maxlen];
        maxlen = o.maxlen;
        currlen = o.currlen;
        std::copy(o.data, o.data + o.maxlen, data);
    }

    MyVector<T> (const MyVector<T>&& o) {
        std::cout << "move ctor called" << std::endl;
        data = o.data;
        maxlen = o.maxlen;
        currlen = o.currlen;
    }

    void push_back (const T& i) {
        if (currlen >= maxlen) {
            maxlen *= 2;
            auto newdata = new T [maxlen];
            std::copy(data, data + currlen, newdata);
            if (data) {
                delete[] data;
            }
            data = newdata;
        }
        data[currlen++] = i;
    }

    friend std::ostream& operator<<(std::ostream &os, const MyVector<T>& o) {
        auto s = o.data;
        auto e = o.data + o.currlen;;
        while (s < e) {
            os << "[" << *s << "]";
            s++;
        }
        return os;
    }
};

int main() {
    auto c = new MyVector<int>(1);
    c->push_back(10);
    c->push_back(11);
```

Share Follow

answered May 13, 2020 at 19:18

Goblinhack
2,553 1 23 26