

Move Semantics in C++

...

A fancy way to say “how can we avoid making unnecessary copies of resources?”



masks required

Attendance

bit.ly/3UEprqT



Announcements

- Assignment 1 grades are out! Go to paperless.stanford.edu to view your grade and feedback!
- Assignment 2 due date moved to Friday, December 2nd @ 11:59pm PT
 - Partners allowed!

Today



- L values vs r values
- SMF Recap
- What the heck is &&??
 - Aka move assignment operator and move constructor the last two special member functions

Definition: **l-values** vs **r-values**

- **l-values** can appear on the **left** or **right** of an =

Definition: l-values vs r-values

- l-values can appear on the left or right of an =
- x is an l-value

```
int x = 3;  
int y = x;
```

Definition: l-values vs r-values

- l-values can appear on the left or right of an =
- x is an l-value

```
int x = 3;  
int y = x;
```

l-values have names

l-values are not temporary

left or right

name

non-temporary

Definition: l-values vs r-values

- **l-values** can appear on the **left** or **right** of an =
- **x** is an **l-value**
- **r-values** can ONLY appear on the **right** of an =

```
int x = 3;  
int y = x;
```

l-values have names

l-values are not temporary

Definition: l-values vs r-values

- **l-values** can appear on the **left** or **right** of an =
- `x` is an **l-value**

```
int x = 3;  
int y = x;
```

- **r-values** can ONLY appear on the **right** of an =
- `3` is an **r-value**

```
int x = 3;  
int y = x;
```

l-values have names

l-values are not temporary

Definition: **l-values** vs **r-values**

a value, temporary

- **l-values** can appear on the **left** or **right** of an =
- `x` is an **l-value**

```
int x = 3;  
int y = x;
```

l-values have names

l-values are not temporary

- **r-values** can ONLY appear on the **right** of an =
- `3` is an **r-value**

```
int x = 3;  
int y = x;
```

r-values don't have names

r-values are temporary

l-values live until the end of the scope

r-values live until the end of the line

Find the **r-values**! (Only consider the items on the *right* of **=** signs)

```
int x = 3;
int *ptr = 0x02248837;
vector<int> v1{1, 2, 3};
auto v4 = v1 + v2;
size_t size = v.size();
v1[1] = 4*i;
ptr = &x;
v1[2] = *ptr;
MyClass obj;
x = obj.public_member_variable;
```

Find the **r-values**! (Only consider the items on the *right* of `=` signs)

```
int x = 3;           //3 is an r-value
int *ptr = 0x02248837;
vector<int> v1{1, 2, 3};
auto v4 = v1 + v2;
size_t size = v.size();
v1[1] = 4*i;
ptr = &x;
v1[2] = *ptr;
MyClass obj;
x = obj.public_member_variable;
```

Find the **r-values**! (Only consider the items on the *right* of **=** signs)

```
int x = 3;           //3 is an r-value
int *ptr = 0x02248837; //0x02248837 is an r-value
vector<int> v1{1, 2, 3};
auto v4 = v1 + v2;
size_t size = v.size();
v1[1] = 4*i;
ptr = &x;
v1[2] = *ptr;
MyClass obj;
x = obj.public_member_variable;
```

Find the **r-values**! (Only consider the items on the *right* of `=` signs)

```
int x = 3;           //3 is an r-value
int *ptr = 0x02248837; //0x02248837 is an r-value
vector<int> v1{1, 2, 3}; //{1, 2, 3} is an r-value, v1 is an l-value
auto v4 = v1 + v2;
size_t size = v.size();
v1[1] = 4*i;
ptr = &x;
v1[2] = *ptr;
MyClass obj;
x = obj.public_member_variable;
```

Find the **r-values**! (Only consider the items on the *right* of `=` signs)

<code>int x = 3;</code>	<code>//3 is an r-value</code>
<code>int *ptr = 0x02248837;</code>	<code>//0x02248837 is an r-value</code>
<code>vector<int> v1{1, 2, 3};</code>	<code>//{1, 2, 3} is an r-value, v1 is an l-value</code>
<code>auto v4 = v1 + v2;</code>	<code>//v1 + v2 is an r-value</code>
<code>size_t size = v.size();</code>	
<code>v1[1] = 4*i;</code>	
<code>ptr = &x;</code>	
<code>v1[2] = *ptr;</code>	
<code>MyClass obj;</code>	
<code>x = obj.public_member_variable;</code>	

Find the **r-values**! (Only consider the items on the *right* of `=` signs)

<code>int x = 3;</code>	<code>//3 is an r-value</code>
<code>int *ptr = 0x02248837;</code>	<code>//0x02248837 is an r-value</code>
<code>vector<int> v1{1, 2, 3};</code>	<code>//{1, 2, 3} is an r-value, v1 is an l-value</code>
<code>auto v4 = v1 + v2;</code>	<code>//v1 + v2 is an r-value</code>
<code>size_t size = v.size();</code>	<code>//v.size() is an r-value</code>
<code>v1[1] = 4*i;</code>	
<code>ptr = &x;</code>	
<code>v1[2] = *ptr;</code>	
<code>MyClass obj;</code>	
<code>x = obj.public_member_variable;</code>	

Find the **r-values**! (Only consider the items on the *right* of `=` signs)

<code>int x = 3;</code>	<code>//3 is an r-value</code>
<code>int *ptr = 0x02248837;</code>	<code>//0x02248837 is an r-value</code>
<code>vector<int> v1{1, 2, 3};</code>	<code>//{1, 2, 3} is an r-value, v1 is an l-value</code>
<code>auto v4 = v1 + v2;</code>	<code>//v1 + v2 is an r-value</code>
<code>size_t size = v.size();</code>	<code>//v.size() is an r-value</code>
<code>v1[1] = 4*i;</code>	<code>//4*i is an r-value, v1[1] is an l-value</code>
<code>ptr = &x;</code>	
<code>v1[2] = *ptr;</code>	
<code>MyClass obj;</code>	
<code>x = obj.public_member_variable;</code>	

Find the **r-values**! (Only consider the items on the *right* of `=` signs)

<code>int x = 3;</code>	<code>//3 is an r-value</code>
<code>int *ptr = 0x02248837;</code>	<code>//0x02248837 is an r-value</code>
<code>vector<int> v1{1, 2, 3};</code>	<code>//{1, 2, 3} is an r-value, v1 is an l-value</code>
<code>auto v4 = v1 + v2;</code>	<code>//v1 + v2 is an r-value</code>
<code>size_t size = v.size();</code>	<code>//v.size() is an r-value</code>
<code>v1[1] = 4*i;</code>	<code>//4*i is an r-value, v1[1] is an l-value</code>
<code>ptr = &x;</code>	<code>//&x is an r-value</code>
<code>v1[2] = *ptr;</code>	
<code>MyClass obj;</code>	
<code>x = obj.public_member_variable;</code>	

Find the **r-values**! (Only consider the items on the *right* of `=` signs)

<code>int x = 3;</code>	<code>//3 is an r-value</code>
<code>int *ptr = 0x02248837;</code>	<code>//0x02248837 is an r-value</code>
<code>vector<int> v1{1, 2, 3};</code>	<code>//{1, 2, 3} is an r-value, v1 is an l-value</code>
<code>auto v4 = v1 + v2;</code>	<code>//v1 + v2 is an r-value</code>
<code>size_t size = v.size();</code>	<code>//v.size() is an r-value</code>
<code>v1[1] = 4*i;</code>	<code>//4*i is an r-value, v1[1] is an l-value</code>
<code>ptr = &x;</code>	<code>//&x is an r-value</code>
<code>v1[2] = *ptr;</code>	<code>//*ptr is an l-value</code>
<code>MyClass obj;</code>	
<code>x = obj.public_member_variable;</code>	

Find the **r-values**! (Only consider the items on the *right* of `=` signs)

<code>int x = 3;</code>	<code>//3 is an r-value</code>
<code>int *ptr = 0x02248837;</code>	<code>//0x02248837 is an r-value</code>
<code>vector<int> v1{1, 2, 3};</code>	<code>//{1, 2, 3} is an r-value, v1 is an l-value</code>
<code>auto v4 = v1 + v2;</code>	<code>//v1 + v2 is an r-value</code>
<code>size_t size = v.size();</code>	<code>//v.size() is an r-value</code>
<code>v1[1] = 4*i;</code>	<code>//4*i is an r-value, v1[1] is an l-value</code>
<code>ptr = &x;</code>	<code>//&x is an r-value</code>
<code>v1[2] = *ptr;</code>	<code>//*ptr is an l-value</code>
<code>MyClass obj;</code>	<code>//obj is an l-value</code>
<code>x = obj.public_member_variable;</code>	

Find the **r-values**! (Only consider the items on the *right* of `=` signs)

<code>int x = 3;</code>	<code>//3 is an r-value</code>
<code>int *ptr = 0x02248837;</code>	<code>//0x02248837 is an r-value</code>
<code>vector<int> v1{1, 2, 3};</code>	<code>//{1, 2, 3} is an r-value, v1 is an l-value</code>
<code>auto v4 = v1 + v2;</code>	<code>//v1 + v2 is an r-value</code>
<code>size_t size = v.size();</code>	<code>//v.size() is an r-value</code>
<code>v1[1] = 4*i;</code>	<code>//4*i is an r-value, v1[1] is an l-value</code>
<code>ptr = &x;</code>	<code>//&x is an r-value</code>
<code>v1[2] = *ptr;</code>	<code>//*ptr is an l-value</code>
<code>MyClass obj;</code>	<code>//obj is an l-value</code>
<code>x = obj.public_member_variable;</code>	<code>//obj.public_member_variable is l-value</code>

Last time...

特殊成员函数

- Special Member Functions (SMFs) get called for specific tasks

- **Copy constructor**: create a new object as a **copy** of an existing object

`Type::Type(const Type& other)`

copy constructor

- **Copy assignment**: reassign a new object to be a **copy** of an existing object

`Type::operator=(const Type& other)`

copy assignment

- **Destructor**: deallocate the memory of an existing object

`Type::~~Type()`

destructor

Last time...

- Special Member Functions (SMFs) get called for specific tasks
 - **Copy constructor**: create a new object as a **copy** of an existing object
`Type::Type(const Type& other)`
 - **Copy assignment**: reassign a new object to be a **copy** of an existing object
`Type::operator=(const Type& other)`
 - **Destructor**: deallocate the memory of an existing object
`Type::~~Type()`
- SMFs are automatically generated for you
 - But if you're managing pointers to allocated to memory, do it yourself

Quick Intertude: make_me_a_vec

```
vector<int> make_me_a_vec(int num) {  
    vector<int> res;  
    while (num != 0) {  
        res.push_back(num%10);  
        num /= 10;  
    }  
    return res;  
}
```

Example:

```
vector<int> myvec = make_me_a_vec(123);  
// myvec = {3, 2, 1}
```

push_back

What Special Member Function gets called at each point?

```
int main() {  
    vector<int> nums1 = make_me_a_vec(12345);    // (1)  
  
    vector<int> nums2;                            // (2)  
  
    nums2 = make_me_a_vec(23456);                // (3)  
}
```

What Special Member Function gets called at each point?

```
int main() {  
    vector<int> nums1 = make_me_a_vec(12345); // (1)  
  
    vector<int> nums2;  
    nums2 = make_me_a_vec(23456); // (3)  
}
```

copy constructor (points to the assignment operator in line 1)

destructor (points to the `make_me_a_vec` function in line 1)

copy assignment (points to the assignment operator in line 3)

destructor (points to the `make_me_a_vec` function in line 3)

The Central Problem

```
nums2 = make_me_a_vec(23456);
```

We need to find a way to **move** the result of **make_me_a_vec** to `nums2`, so that we don't create two objects (and immediately destroy one)

Question: Why don't we just return `vector&` instead of `vector` in `make_me_a_vec`?

Questions?

Only l-values can be referenced using &

```
int main() {  
    vector<int> vec;  
    change(vec);  
}  
  
void change(vector<int>& v){...}  
//v is a reference to vec
```

```
int main() {  
    change(7);  
    //this will compile error  
}  
  
//we cannot take a reference to  
//a literal!  
void change(int& v){...}
```

Vector Copy Assignment

*from one container
to another*

```
template <typename T>
vector<T>& vector<T>::operator=(const vector<T>& other) {
    if (&other == this) return *this;
    _size = other._size;
    _capacity = other._capacity;
    delete[] _elems;
    _elems = new T[other._capacity];
    std::copy(other._elems, other._elems + other._size, _elems);
    return *this;
}
```

`std::copy` is a generic copy function
used to copy a range of elements
from one container to another.

Recall: Vector Copy Assignment

```
template <typename T>
vector<T>& vector<T>::operator=(const vector<T>& other) {
```

but wait ...


```
int main() {
    vector<int> vec;
    vec.operator=(make_me_a_vec(123));
}
```

```
vector<int> make_me_a_vec(int num);
```


Recall: Vector Copy Constructor

Only l-values can be referenced using &!

```
template <typename T>
vector<T>& vector<T>::operator=(const vector<T>& other) {
```



left value

but wait ...

```
int main() {
    vector<int> vec;
    vec = make_me_a_vec(123); //make_me_a_vec(123) is an r-value
}
```


why is this possible?

right value

Recall: Vector Copy Constructor

Only l-values can be referenced using &!

```
template <typename T>
vector<T>& vector<T>::operator=(const vector<T>& other) {
```



but wait ...

```
int main() {
    vector<int> vec;
    vec = make_me_a_vec(123); //make_me_a_vec(123) is an r-value
}
```

rvalues can be bound to `const` & (we promise not to change them)

Recall: Vector Copy Constructor

```
template <typename T>
vector<T>& vector<T>::operator=(const vector<T>& other) {
```

rvalues can be bound to `const` & (we promise not to change them)

```
int main() {
    vector<int> vec;
    vec = make_me_a_vec(123); //make_me_a_vec(123) is an r-value
}
```

passing by & avoids making unnecessary copies... but does it?

How many arrays will be allocated, copied and destroyed here?

```
int main() {  
    vector<int> vec;  
    vec = make_me_a_vec(123);  
}
```

```
vector<int> make_me_a_vec(int num) {  
    vector<int> res;  
    while (num != 0) {  
        res.push_back(num%10);  
        num /= 10;  
    }  
    return res;  
}
```

How many arrays will be allocated, copied and destroyed here?

```
int main() {  
    vector<int> vec;  
    vec = make_me_a_vec(123); //make_me_a_vec(123) is an r-value  
}
```

- vec is created using the default constructor
- make_me_a_vec creates a vector using the default constructor and returns it
- vec is reassigned to a **copy** of that return value using **copy assignment**
- **copy assignment** creates a new array and **copies** the contents of the old one
- The original return value's lifetime ends and it calls its **destructor**
- vec's lifetime ends and it calls its **destructor**

How many arrays will be allocated, copied and destroyed here?

```
int main() {  
    vector<int> vec;  
    vec = make_me_a_vec(123); //make_me_a_vec(123) is an r-value  
}
```

- vec is created using the **default constructor**
- make_me_a_vec creates a vector using the **default constructor** and returns it
- vec is reassigned to a **copy** of that return value using **copy assignment**
- **copy assignment** creates a new array and **copies** the contents of the old one
- The original return value's lifetime ends and it calls its **destructor**
- vec's lifetime ends and it calls its **destructor**

Recall: **copy assignment** creates a new array and copies the contents of the old one...

```
template <typename T>
vector<T>& vector<T>::operator=(const vector<T>& other) {
    if (&other == this) return *this;
    _size = other._size;
    _capacity = other._capacity;
    delete[] _elems;
    _elems = new T[other._capacity];
    std::copy(other._elems, other._elems + other._size, _elems);
    return *this;
}
```

copy assignment creates a new array and copies the contents of the old one... what if it didn't?

```
template <typename T>
vector<T>& vector<T>::operator=(const vector<T>& other) {
    if (&other == this) return *this;
    _size = other._size;
    _capacity = other._capacity;
    _elems = other._elems;
    return *this;
}
```

Let's call this **move assignment**

Is this allowed?

This works!

```
int main() {  
    vector<int> vec;  
    vec = make_me_a_vec(123);  
}
```

This works!

```
int main() {  
    vector<int> vec;  
    vec = make_me_a_vec(123);  
}
```

But what about this?

```
int main() {  
    vector<string> vec1 = {"hello", "world"}  
    vector<string> vec2 = vec1;  
    vec1.push_back("Sure hope vec2 doesn't see this!")  
}
```

This works!

```
int main() {  
    vector<int> vec;  
    vec = make_me_a_vec(123);  
}
```

But what about this?

```
int main() {  
    vector<string> vec1 = {"hello", "world"};  
    vector<string> vec2 = vec1;  
    vec1.push_back("Sure hope vec2 doesn't see this!");  
} //BAD!
```

How do we know when to use **move assignment** and
when to use **copy assignment**?

How do we know when to use **move assignment** and when to use **copy assignment**?

When the item on the right of the = is an **r-value** we should use **move assignment**

r-value ~ move assignment

How do we know when to use **move assignment** and when to use **copy assignment**?

When the item on the right of the = is an **r-value** we should use **move assignment**

Why? **r-values** are always about to die, so we can steal their resources

Using move assignment

move

```
int main() {  
    vector<int> vec;  
    vec = make_me_a_vec(123);  
}
```

Using copy assignment

copy

```
int main() {  
    vector<string> vec1 = {"hello", "world"}  
    vector<string> vec2 = vec1;  
    vec1.push_back("Sure hope vec2 doesn't see this!")  
} //and vec2 never saw a thing
```


Questions?

How to make two different assignment operators?

Overload `vector::operator=`!

How to make two different assignment operators?

Overload `vector::operator=`!

How? Introducing... the **r-value reference**

`&&`

(This is different from the l-value reference `&` you have seen before)

(it has one more ampersand)

Overloading with &&

```
int main() {  
    int x = 1;  
    change(x); //this will call version 2  
    change(7); //this will call version 1  
}  
  
void change(int&& num){...} //version 1 takes r-values  
void change(int& num){...}  //version 2 takes l-values  
//num is a reference to vec
```

Copy assignment

```
vector<T>& operator=(const vector<T>& other)
{

    if (&other == this) return *this;
    _size = other._size;
    _capacity = other._capacity;

    //must copy entire array
    delete[] _elems;
    _elems = new T[other._capacity];
    std::copy(other._elems,
other._elems + other._size,
_elems);
    return *this;
}
```

Move assignment

```
vector<T>& operator=(vector<T>&& other)
```

释放内存
重新分配内存

Copy assignment

```
vector<T>& operator=(const vector<T>& other)
{

    if (&other == this) return *this;
    _size = other._size;
    _capacity = other._capacity;

    //must copy entire array
    delete[] _elems;
    _elems = new T[other._capacity];
    std::copy(other._elems,
other._elems + other._size,
_elems);
    return *this;
}
```

释放内存, 改变指针指向

Move assignment

```
vector<T>& operator=(vector<T>&& other)
{

    if (&other == this) return *this;
    _size = other._size;
    _capacity = other._capacity;

    //we can steal the array
    delete[] _elems;
    _elems = other._elems
    return *this;
}
```

This works!

```
int main() {  
    vector<int> vec;  
    vec = make_me_a_vec(123); //this will use move assignment  
    vector<string> vec1 = {"hello", "world"}  
    vector<string> vec2 = vec1; //this will use copy assignment  
    vec1.push_back("Sure hope vec2 doesn't see this!")  
}
```

The compiler will pick which `vector::operator=` to use based on whether the RHS is an **l-value** or an **r-value**

Can we make it even better?

Move assignment

```
vector<T>& operator=(vector<T>&& other)
{
    if (&other == this) return *this;
    _size = other._size;
    _capacity = other._capacity;

    //we can steal the array
    delete[] _elems;
    _elems = other._elems
    return *this;
}
```

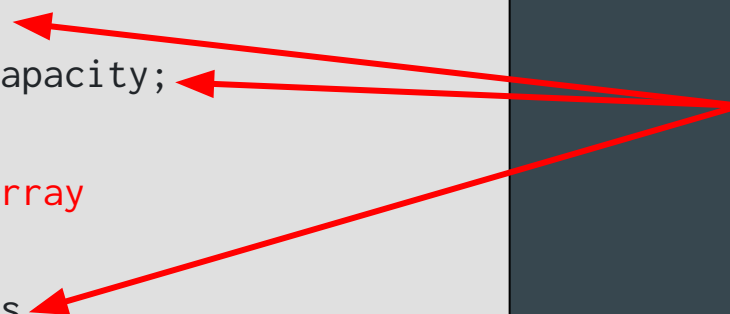

Can we make it even better?

Move assignment

```
vector<T>& operator=(vector<T>&& other)
{
    if (&other == this) return *this;
    _size = other._size;
    _capacity = other._capacity;

    //we can steal the array
    delete[] _elems;
    _elems = other._elems
    return *this;
}
```

Technically, these
are also making
copies (using
int/ptr copy
assignment)



Introducing... `std::move`

- `std::move(x)` doesn't do anything except cast x as an r-value
- It is a way to force C++ to choose the && version of a function

```
int main() {  
    int x = 1;  
    change(x); //this will call version 2  
    change(std::move(x)); //this will call version 1  
}  
  
void change(int&& num){...} //version 1 takes r-values  
void change(int& num){...}  //version 2 takes l-values
```

Can we make it even better?

Move assignment

```
vector<T>& operator=(vector<T>&& other)
{
    if (&other == this) return *this;
    _size = other._size;
    _capacity = other._capacity;

    //we can steal the array
    delete[] _elems;
    _elems = other._elems
    return *this;
}
```

We can force
move assignment
rather than copy
assignment of
these ints by
using `std::move`!

Can we make it even better?

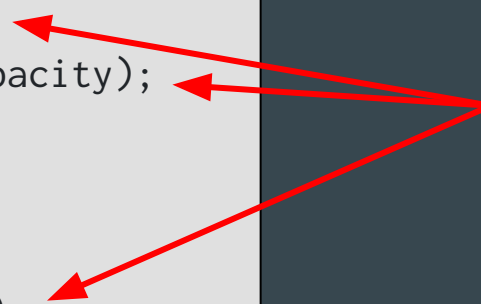
Move assignment

std::move(...);

```
vector<T>& operator=(vector<T>&& other)
{
    if (&other == this) return *this;
    _size = std::move(other._size);
    _capacity = std::move(other._capacity);

    //we can steal the array
    delete[] _elems;
    _elems = std::move(other._elems);
    return *this;
}
```

We can force
move assignment
rather than copy
assignment of
these ints by
using std::move!



This works!

```
int main() {  
    vector<int> vec;  
    vec = make_me_a_vec(123); //this will use move assignment  
    vector<string> vec1 = {"hello", "world"}  
    vector<string> vec2 = vec1; //this will use copy assignment  
    vec1.push_back("Sure hope vec2 doesn't see this!")  
}
```

The compiler will pick which `vector::operator=` to use based on whether the RHS is an **l-value** or an **r-value**

This works!

```
int main() {  
    vector<int> vec;  
    vec = make_me_a_vec(123); //this will use move assignment  
    vector<string> vec1 = {"hello", "world"}  
    vector<string> vec2 = vec1; //this will use copy assignment  
    vec1.push_back("Sure hope vec2 doesn't see this!")  
}
```

The compiler will pick which `vector::operator=` to use based on whether the RHS is an **l-value** or an **r-value**

This works!

```
int main() {  
    vector<int> vec;  
    vec = make_me_a_vec(123); //this will use move assignment  
    vector<string> vec1 = {"hello", "world"}  
    vector<string> vec2 = vec1; //this will use copy construction  
    vec1.push_back("Sure hope vec2 doesn't see this!")  
}
```

The compiler will pick which `vector::operator=` to use based on whether the RHS is an **l-value** or an **r-value**

This works!

```
int main() {  
    vector<int> vec;  
    vec = make_me_a_vec(123); //this will use move assignment  
    vector<string> vec1 = {"hello", "world"} //this should use move  
    vector<string> vec2 = vec1; //this will use copy construction  
    vec1.push_back("Sure hope vec2 doesn't see this!")  
}
```

The compiler will pick which `vector::operator=` to use based on whether the RHS is an **l-value** or an **r-value**

Let's do it with our copy constructor!

copy constructor

```
vector<T>(const vector<T>& other) {  
    if (&other == this) return *this;  
    _size = other._size;  
    _capacity = other._capacity;  
  
    //must copy entire array  
    delete[] _elems;  
    _elems = new T[other._capacity];  
    std::copy(other._elems,  
other._elems + other._size,  
_elems);  
    return *this;  
}
```

move constructor

Let's do it with our copy constructor!

copy constructor

```
vector<T>(const vector<T>& other) {  
    if (&other == this) return *this;  
    _size = other._size;  
    _capacity = other._capacity;  
  
    //must copy entire array  
    delete[] _elems;  
    _elems = new T[other._capacity];  
    std::copy(other._elems,  
other._elems + other._size,  
_elems);  
    return *this;  
}
```

move constructor

```
vector<T>(vector<T>&& other)
```

Let's do it with our copy constructor!

copy constructor

```
vector<T>(const vector<T>& other) {  
    if (&other == this) return *this;  
    _size = other._size;  
    _capacity = other._capacity;  
  
    //must copy entire array  
    delete[] _elems;  
    _elems = new T[other._capacity];  
    std::copy(other._elems,  
other._elems + other._size,  
_elems);  
    return *this;  
}
```

move constructor

```
vector<T>(vector<T>&& other) {  
    if (&other == this) return *this;  
  
    _size = std::move(other._size);  
    _capacity =  
        std::move(other._capacity);  
  
    //we can steal the array  
    delete[] _elems;  
    _elems = std::move(other._elems);  
    return *this;  
}
```

Where else should we use `std::move`?

Where else should we use `std::move`?

Rule of Thumb: Whenever we take in a `const &` parameter in a class member function and assign it to something else in our function

vector::push_back

Copy push_back

```
void push_back(const T& element) {  
    elems[_size++] = element;  
    //this is copy assignment  
}
```

Move push_back

```
void push_back(T&& element) {  
    elems[_size++] =  
        std::move(element);  
    //this forces T's move  
    //assignment  
}
```

Be careful with `std::move`

```
int main() {  
    vector<string> vec1 = {"hello", "world"}  
    vector<string> vec2 = std::move(vec1);  
    vec1.push_back("Sure hope vec2 doesn't see this!")  
}
```

Be careful with `std::move`

```
int main() {  
    vector<string> vec1 = {"hello", "world"}  
    vector<string> vec2 = std::move(vec1);  
vec1.push_back("Sure hope vec2 doesn't see this!");  
}
```

- After a variable is moved via `std::move`, it should never be used until it is reassigned to a new variable!
- The C++ compiler *might* warn you about this mistake, but the code above compiles!

Where else should we use `std::move`?

Rule of Thumb: Whenever we take in a `const` & parameter in a class member function and assign it to something else in our function

Don't use `std::move` outside of class definitions, never use it in application code!

TLDR: Move Semantics

- If your class has **copy constructor** and **copy assignment** defined, you should also define a **move constructor** and **move assignment**
- Define these by overloading your copy constructor and assignment to be defined for `Type&& other` as well as `Type& other`
- Use `std::move` to force the use of other types' move assignments and constructors
- All `std::move(x)` does is cast `x` as an rvalue
- Be wary of `std::move(x)` in main function code!