Move Semantics in C++

•••

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

Attendance

bit.ly/3IByFQJ







Announcements

Announcements

- Only 3 lectures left (including today)!

8	Can start 13. Move Semantics assignment 2	MAY 25 14. std::optional and Type Safety
9	after this lecture 15. RAII, Smart Pointers, and Building C++ Projects	Optional: No Class, Extra Office Hours
10	Optional: No Class, Extra Office Hours	No class, No office hours

1

Important Announcements

- Wednesday, June 7th at 11:59pm PT is the last day we
 can accept any assignments (1 or 2)
- **We want everyone to pass!** Please turn in your assignments! Come to office hours, post on Ed, email us to get help!
- Reminder you need to complete assignment 1 and 2
 without build errors to pass the class

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

- l-values can appear on the left orright of an =

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

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

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

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

l-values have names

l-values are **not temporary**

- l-values can appear on the left orright 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 theright of an =

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

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

l-values have names

l-values are <u>not temporary</u>

- r-values can ONLY appear on theright of an =
- 3 is an **r-value**

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

- l-values can appear on the left orright 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 theright of an =
- 3 is an **r-value**

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

r-values don't have names

r-values are **temporary**

I-values live until the end of the scope r-values live until the end of the line

```
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;
```

```
//3 is an r-value
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;
```

```
//3 is an r-value
int x = 3;
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;
```

```
//3 is an r-value
int x = 3;
                           //0x02248837 is an r-value
int *ptr = 0x02248837;
vector<int> v1\{1, 2, 3\}; \frac{1}{1} is an r-value, \frac{1}{1} 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;
```

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

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

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

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

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

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

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

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 object to be a **copy** of an existing object Type::operator=(const Type& other)
 - Destructor: deallocate the memory of an existing object
 Type::~Type()

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 Interlude: 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}
```

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

   vector<int> nums2; // (2)

   nums2 = make_me_a_vec(23456); // (2)
```

Options:

- 1. Copy Constructor
- 2. Copy Assignment Operator
- 3. Destructor

```
copy constructor
int main() {
   vector<int> nums1 = make_me_a_vec(12345); // (1)
                                    destructor
                                                 // (2)
   vector<int> nums2;
   nums2 = make_me_a_vec(23456);
                                                 // (2)
```

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

Default constructor
    vector<int> nums2; // (2)

nums2 = make_me_a_vec(23456); // (2)
}
```

```
copy constructor
int main() {
   vector<int> nums1 \stackrel{\triangleright}{=} make_me_a_vec(12345); // (1)
 Default constructor \
                                           destructor
                                                           // (2)
   vector<int> nums2;
copy assignment
    nums2 = make_me_a_vec(23456);
                                                           // (2)
                        destructor
```

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?

Time to Ponder

Only I-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 Operator

```
template <typename T>
vector<T>& vector<T>::operator=(const vector<T>& other) {
    if (&other == this) return *this;
    _size = other._size;
                                            std::copy is a generic copy function
                                            used to copy a range of elements
    _capacity = other._capacity;
                                            from one container to another.
    delete[] _elems;
    _elems = new T[other._capacity];
    std::copy(other._elems, other._elems + other._size, _elems);
    return *this;
```

Recall: Vector Copy Assignment Operator

```
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);
```

Only l-values can be referenced using &!

Recall: Vector Copy Assignment Operator

```
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
}
```

why is this possible?

Recall: Vector Copy Assignment Operator

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 Assignment Operator

```
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?

```
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;
}
```

```
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

```
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

```
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

```
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**

```
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

Can we do better?

- make_me_a_vec creates a vector using the default constructo
- 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?

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

```
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;
    vec2 = vec1;
    vec1.push_back("Sure hope vec2 doesn't see this!");
}
```

```
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;
    vec2 = vec1;
    vec1.push_back("Sure hope vec2 doesn't see this!");
} //BAD!
```

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

But what about this?

Okay so we need both a copy assignment AND a move assignment

```
int main() {
    vector<string> vec1 = {"hello", "world"};
    vector<string> vec2;
    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

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

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

Using copy assignment

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

Questions?

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

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 1-values
//num is a reference to int
```

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)
```

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)
{
    if (&other == this) return *this;
    _size = other._size;
    _capacity = other._capacity;
    //we can steal the array
    delete[] _elems;
    _elems = other._elems
    return *this;
```

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

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

```
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!

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

What if we wanted to declare and initialize a vec on the same line?

```
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!")
}
```

What if we wanted to declare and initialize a vec on the same line?

```
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!")
}
```

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 I-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 will use move constructor
    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= or constructor to use based on whether the RHS is an I-value or an r-value

Let's do it with our copy constructor!

copy constructor

move 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;
```

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: Wherever 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!")
}
```

Where else should we use std::move?

Rule of Thumb: Wherever 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!

The 6 Special Member Functions

- 1. **Default constructor:** Initializes an object to a default state
- **2. Copy constructor:** Creates a new object by copying an existing object
- **3. Move constructor:** Creates a new object by moving the resources of an existing object
- **4. Copy Assignment Operator:** Assigns the contents of one object to another object
- **5. Move Assignment Operator:** Moves the resources of one object to another object
- **6. Destructor**: Frees any dynamically allocated resources owned by an object when it is destroyed

Some Philosophy about SMFs

Three Guiding Rules

- Rule of Zero
- Rule of Three
- Rule of Five

Rules of Zero

- If you can avoid defining default operations, do
- Why? It's the simplest and gives the cleanest semantic
- **Example:** Since std::map and std::string have all the special functions, no further work is needed.

```
Class Named_map {
public:
    // ... no default operations declared ...
private:
    std::string name;
    std::map<int, int> rep;
};

Named_map nm;    // default construct
Named_map nm2 {nm};    // copy construct
```

Rules of Three

- If you need to implement a custom destructor, you almost certainly need to define a copy constructor and copy assignment operator
- **Why?** You are probably managing your own memory somehow, so the shallow copies provided by the default operations won't work correctly

Rules of Five

- If you define custom copy constructor/assignment operator, you should define move constructor/assignment operator as well
- Why? This is about efficiency rather than correctness. It's inefficient to make extra copies (although it is "correct")

The 6 Special Member Functions

- Default constructor: A constructor that takes no arguments and initializes an object to a default state. If a class has dynamically allocated resources, the default constructor should initialize them to a valid state.
- Copy constructor: A constructor that creates a new object by copying an existing object of the same type. This function is called when an object is passed by value or returned by value.
- Move constructor: A constructor that creates a new object by moving the resources of an existing object of the same type. This function is called when an object is moved, typically as an rvalue reference.

The 6 Special Member Functions

- Copy assignment operator: An overloaded assignment operator that assigns the contents of one object to another object of the same type. This function is called when an object is assigned to another object of the same type.
- Move assignment operator: An overloaded assignment operator that moves the resources of one object to another object of the same type. This function is called when an object is moved, typically as an rvalue reference.
- Destructor: A special member function that is called when an object is destroyed, typically when it goes out of scope or is deleted. This function is responsible for freeing any dynamically allocated resources that the object owns.

Thanks for coming!

•••

Next time: std::optional