

# Rvalue References and Move Semantics



Valentina Gaggero



#### Problems in C++ '98

```
99 ▼ static Holder retrieveHolder() =
100
101
          Holder h1(20000);
          Holder h2(10000);
102
103
          if (rand() == 0)
104
105
              return h1;
          return h2;
107
109
   ▼ int main()
111
112
          Holder my_holder=retrieveHolder();
113
          //....
114
          getchar();
115
```

```
vgaggero@iiticublap127:~/corsoCPP$ ./moveSem example
Holder class constructor
Holder class constructor
Holder class copy constructor
Holder class destructor
Holder class destructor
Holder class destructor
```

```
13 ▼ class Holder
14 {
     private:
16
         int32_t *m_data; //heap allocated, expensive copy
17
         enum { DefaultDataSize = 100000 };
18
         size_t m_size;
19
20
     public:
         Holder(uint32_t size)
23
             std::cout << "Holder class constructor" << std::endl;</pre>
24
             uint32_t currSize=size;
26
             if(size==0) currSize=DefaultDataSize;
27
             m_data=new int32_t[currSize];
28
             m_size=size;
29
             std::generate(m_data, m_data + m_size, rand);
30
31
32 ▼
         ~Holder()
34
             std::cout << "Holder class destructor" << std::endl;
35
             if(m_data!= nullptr)
36
                 delete[] m_data;
             m_size=0;
38
39
40 -
         Holder(const Holder &other)
41
             std::cout << "Holder class copy constructor" << std::endl;</pre>
42
43
             m_data = new int[other.m_size];
44
             std::copy(other.m_data, other.m_data + other.m_size, m_data);
45
             m_size = other.m_size;
46
47
48 ▼
         Holder& operator=(const Holder &other)
49
             std::cout << "Holder class copy assignment operator" << std::endl;
50
             if(this == &other) return *this;
             delete[] m_data;
53
             m_data = new int[other.m_size];
54
             std::copy(other.m_data, other.m_data + other.m_size, m_data);
             m_size = other.m_size;
             return *this;
56
                                                                  2
```

#### Problems in C++ '98

```
99 ▼ static Holder retrieveHolder()
  100
  101
            Holder h1(20000);
  102
            Holder h2(10000);
  103
  104
            if (rand() == 0)
  105
                return h1;
            return h2;
  107
  109
     ▼ int main()
  111
  112
            Holder my_holder=retrieveHolder();
  113
            //....
  114
            getchar();
  115
vgaggero@iiticublap127:~/corsoCPP
                                                Why not reuse the
Holder class constructor
```

```
temporary?
Holder class constructor
Holder Charactery constructor
Holder class destructor
Holder class destructor
Holder class destructor
                                               Move semantics
```

```
13 ▼ class Holder
14 {
15
     private:
         int32_t *m_data; //heap allocated, expensive copy
16
17
         enum { DefaultDataSize = 100000 };
         size_t m_size;
19
20
    public:
21
22 -
         Holder(uint32_t size)
23
             std::cout << "Holder class constructor" << std::endl;
24
25
             uint32_t currSize=size;
             if(size==0) currSize=DefaultDataSize:
27
             m_data=new int32_t[currSize];
             m_size=size;
29
             std::generate(m_data, m_data + m_size, rand);
30
         }
32 ▼
         ~Holder()
34
             std::cout << "Holder class deconstructor" << std::endl;
             delete[] m_data;
             m_size=0;
39
         Holder(const Holder &other)
             std::cout << "Holder class copy constructor" << std::endl;
             m_data = new int[other.m_size];
             rtd::copy(other.m_data, other.m_data + other.m_size, m_data);
             m_size = other.m_size;
             w& operator=(const Holder &other)
             std::cout << "Holder class copy assignment operator" << std::endl;
             if(this == &other) return *this;
             delete[] m_data;
             m_data = new int[other.m_size];
             std::copy(other.m_data, other.m_data + other.m_size, m_data);
             m_size = other.m_size;
             return *this;
         }
```

#### Solution: move constructor

The Move Constructor is a overloaded constructor called for temporaries.

It can "steal the guts" of the other object

```
Holder(Holder &&other)

{

m_data = other.m_data;

m_size = other.m_size;

other.m_data = nullptr;

other.m_size = 0;

}
```

**T&&:** new operator to indicate Reference to temporaries, i.e. **rvalue references** 

#### Solution: move constructor

The Move Constructor is a **overloaded constructor** called for **temporaries**. *It can "steal the guts" of the other object* 

```
Holder(Holder &&other) noexcept

{

m_data = other.m_data;

m_size = other.m_size;

other.m_data = nullptr;

other.m_size = 0;

}
```

**T&&:** new operator to indicate Reference to temporaries, i.e. **rvalue references** 

except: the function doesn't throw exception

### Solution: move constructor

In our example the move constructor is called instead of copy constructor because

the argument is a temporary object.

```
99 ▼ static Holder retrieveHolder()
100
101
           Holder h1(20000);
102
          Holder h2(10000);
103
104
           if (rand() == 0)
105
               return h1;
106
           return h2;
107
108
109
   ▼ int main()
110
111
          Holder my_holder=retrieveHolder();
112
113
          11 . . . . .
          getchar();
114
115
```

```
If Holder is a copy-only class ...

vgaggero@iiticublap127:~/corsoCPP$ ./moveSem_example

Holder class constructor

Holder class copy constructor

Holder class destructor

Holder class destructor

Holder class destructor
```

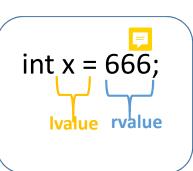
```
If Holder is movable class ...

vgaggero@iiticublap127:~/corsoCPP$ ./moveSem_example
Holder class constructor
Holder class constructor
Holder class move constructor
Holder class destructor
Holder class destructor
Holder class destructor
```

#### What we can move?

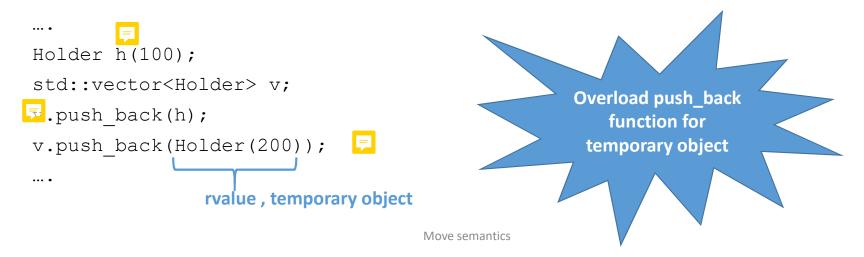
Everything that is a temporary





#### In C++:

- an rvalue is an unnamed object, like a temporary
- an Ivalue has either a name or address





### Overload on temporaries

Example: std::vector::push\_back(T&& element)

```
1 template <typename T>
                                              ✓ Use std::move to tell the compiler that an
   class vector {
 3
                                                object can be moved from
       void push_back(const T& element)
          if (!resize(m_size + 1))
                                              ✓ Turns Ivalues into rvalues, does not move
             return;
10
                                                itself.
          m elements[m size - 1] = element;
11
12
13
       void push back(T&& element)
          if (!resize(m size + 1))
16
17
             return;
18
                                                           The move assignment operator is called
          m elements[m size - 1] = std::move(element);
19
20
21 }
```

## Overload on temporaries

Example: std::vector::push\_back(T&& element)

```
1 template <typename T>
                                                        72
   class vector {
                                                        73
 3
                                                        74
 4
         . . .
                                                        75
                                                        76
        void push back(const T& element)
 6
                                                        77
                                                        78
 8
             if (!resize(m_size + 1))
 9
                 return;
10
                                                        81
11
             m elements[m size - 1] = element;
                                                        82
12
        }
13
                                                        84
14
        void push back(T&& element)
15
                                                        87
             if (!resize(m size + 1))
16
17
                 return;
18
            m elements[m size - 1] = std::move(element);
19
20
21 }
```

```
Holder(Holder &&other) noexcept
{
    std::cout << "Holder class move constructor operator" << std::endl;
    m_data = other.m_data;
    m_size = other.m_size;
    other.m_data = nullptr;
    other.m_size = 0;
}
Holder& operator=(Holder &&other) noexcept
{
    std::cout << "Holder class move assignment operator" << std::endl;
    if(m_data != nullptr)
        delete [] m_data;
    m_data = other.m_data;
    m_size = other.m_size;
    other.m_data = nullptr;
    other.m_size = 0;
}</pre>
```

#### Move assignment operator

```
118
                                                                                        vgaggero@iiticublap127: ~/corsoCPP
                                                                                       vgaggero@iiticublap127: ~/corsoCPP 80x24
                                                            Holder class deconstructor
122
                                                           vgaggero@iiticublap127:~/corsoCPP$ g++ -o moveSem_example move-semantics.cpp
123
                                                            /gaggero@iiticublap127:~/corsoCPP$ ./moveSem_example
                                                            Holder class constructor
125 ▼ int main()
                                                            Holder class constructor
126
                                                            Holder class move constructor
127
          Holder my_holder=retrieveHolder();
                                                           Holder class destructor
128
          Holder holder_first(100);
                                                            Holder class destructor
          Holder holder_second(200);
                                                           Holder class constructor
          //...
                                                           Holder class constructor
          cout << endl << "INIT:":
131
         cout << "my_holder size=" << my_holder.getSize()</pre>
132
                                                           INIT:my holder size=10000, holder first size=100, holder second size=200
133
          holder_first=my_holder;
                                                            Holder class copy assignment operator
          cout << "my_holder size=" << my_holder.getSize()</pre>
                                                            ny holder size=10000, holder first size=10000, holder second size=200
136
          holder_second=std::move(my_holder);
                                                            Holder class move assignment operator
          cout << "my_holder size=" << my_holder.getSize()</pre>
                                                            ny_holder size=0, holder_first size=10000, holder_second size=10000
139
          delete my holder;
          getchar();
141
                                                            Holder class destructor
142
                                                            Holder class destructor
143
                                                            Holder class destructor
144
                                                            gaggero@iiticublap127:~/corsoCPPS
```

my\_holder is empty.... I should not use it anymore!!

#### Move assignment operator

```
118
                                                                                       vgaggero@iiticublap127: ~/corsoCPP
120
                                                                                       vgaggero@iiticublap127: ~/corsoCPP 80x24
                                                            Holder class deconstructor
122
                                                           vgaggero@iiticublap127:~/corsoCPP$ g++ -o moveSem_example move-semantics.cpp
123
124
125 ▼ int main()
126
127
          Holder my_holder=retrieveHolder();
                                                                             ~Holder() =
                                                                  32 -
128
          Holder holder_first(100);
          Holder holder_second(200);
                                                                  33
          //...
                                                                                  std::cout << "Holder class destructor" << std::endl;
                                                                  34
131
          cout << endl << "INIT:":
                                                                  35
                                                                                  if(m_data!= nullptr)
          cout << "my_holder size=" << my_holder.getSize()</pre>
                                                                                       delete[] m_data;
                                                                  36
133
          holder_first=my_holder;
                                                                  37
                                                                                  m_size=0;
          cout << "my_holder size=" << my_holder.getSize()</pre>
136
137
          holder_second=std::move(my_holder);
          cout << "my_holder size=" << my_holder.getSize()</pre>
138
139
          delete my holder;
          getchar();
141
142
                                                            Holder class destructor
143
                                                            Holder class destructor
144
                                                            gaggero@iiticublap127:~/corsoCPP$
```

my holder is empty.... I should not use it anymore!!

## Implicity generated member functions

#### Rule of three

-

Rule of five

- ✓ Copy constructor
- ✓ Destructor
- ✓ Copy assignment operator



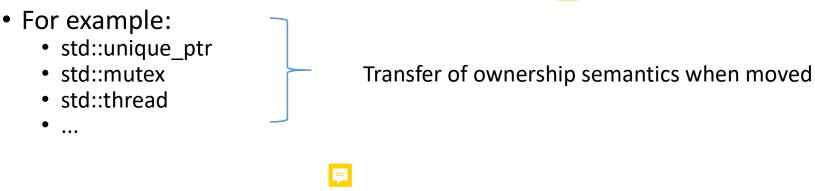
- ✓ Copy constructor
- ✓ Destructor
- ✓ Copy assignment operator
- ✓ Move constructor
- ✓ Move assignment operator

Remember: if you use =default, means that you are declaring the special member function anyway

**VG1** Valentina Gaggero, 5/21/2019

## Move-only classes

Some classes can only be *moved*, not copied.



How to: define move constructor only, no copy constructor. Eventually Move assignment operator could be defined.

## Move-only classes: example

```
class B {
    std::string objName;
public:
    B (B&&) = default;
    B& operator= (B&&) = default;
    B (const B&) = delete;
    B& operator= ( constB & ) = delete;
    B (const std::string & name) : objName(name) {}
    ~B() = default;
    std::string name() const { return objName;}
};
```



## Example: sort a vector of Holder objects

Changes done to preview example of Holder class:

- Added int m\_id member in the Holder class. Its value is initialized by std::rand function
- Added overload of operator <</li>
- (Only for debug purpose) Defined a static global variable for number of copies counting. It is incremented in copy constructor and in copy assignment operator

```
If Holder is a copy-only class ... 💆
150 ▼ int main()
151
                                                           vgaggero@iiticublap127:~/corsoCPP$ ./simpleExample
         vector<Holder> v;
                                                           Num of copies is 227
153
         for(auto i=0; i<100; i++)
                                                           After sort, num of copies is 897
            v.push_back(Holder(1000));
155
157
158
        cout<< "Num of copies is "<< numOfCopies <<endl;
                                                           If Holder is movable class ...
159
                                                           vgaggero@iiticublap127:~/corsoCPP$ ./simpleExample
        sort(v.begin(), v.end()); =
160
                                                          Num of copies is 0
162
        cout << "After sort, num of copies is " << numOfCopies</pre>
                                                         After sort, num of copies is 0
163
164
165
```

#### Question

```
▼ static Holder createHolder()
                                    If Horder class has both move-constructor and move-assignment,
167
168
         Holder bigH(60000) =
                                    how many "special member function" are invoked running this code?
        //....do something with bigH
169
170
         return bigH;
171
                                    Answer: only 1 constructor!!!
172
173
   ▼ int main()
                                    vgaggero@iiticublap127:~/corsoCPP$ ./move-ser
174
                                    Holder class constructor
175
      Holder obj=createHolder();
176
                                    Holder class destructor
177
         getchar();
178
               This is due to return
               value optimization
                                                                                                           16
                                                 Move semantics
```

### Return Value Optimization

"In the initialization of an object, when the source object is a nameless temporary and is of the same class type as the target object. When the nameless temporary is the operand of a return statement, this variant of copy elision is known as RVO, return value optimization".

https://en.cppreference.com/w/cpp/language/copy\_elision

- "... compilers may elide the copiyng (or moving) of a local object in a function that returns by value if:
- 1. the type of local object is the same as that returned by the function
- 2. the local object is what's being returned". [Effective Modern C++, Mayer]

The RVO is mandatory since c++ 17.

## Recap

- ✓ The move semantics is based on rvalue references (⊤ &&)
- ✓ An rvalue is a temporary object which will be destroyed at the end of expression
- ✓ When a caller passes an rvalue, the called function steals its data. (std::move(temp\_obj))
- ✓ The original data is a zombie and should never be accessed after move operation.
- ✓ Use noexcept in move constructor and move assignment operator
- ✓ Remember to set nullptr the moved object's data pointer

## Suggestion 1: safer move constructor

To avoid to forgot to clear the temporary object pointer (since c++ 14)

```
1 class BigObject
2 {
3    BigObject(BigObject &&other) noexcept
4    : lotsOfData{std::exchange(other.lotsOfData, nullptr)} {}
5    ...
6    int *lotsOfData;
7 };
```

# Suggestion 2: implement move assignment with copy-swap idiom

```
1 template <typename T>
 2 class vector {
        void push back(const T& element)
            if (!resize(m size + 1))
                 return;
10
11
             m elements[m size - 1] = element;
12
13
14
        void push back(T&& element)
15
16
             if (!resize(m size + 1))
17
                 return;
18
19
             m elements[m size - 1] = std::move(element);
20
21 }
```

```
BigObject &BigObject::operator=(BigObject &&other) noexcept

BigObject moved(std::move(other));

swap(moved);

return *this;

}

void BigObject::swap(BigObject &other) noexcept {
 using std::swap; // enable ADL
 swap(lotsOfData, other.lotsOfData);
}
```

```
Example of swap implementation without copying

1 void swap(BigObject &a, BigObject &b)
2 {
3     // temp will steal the guts of a, even though a is an lvalue!
4     BigObject temp = std::move(a);
5     a = std::move(b);
7     b = std::move(temp);
8 }
```

## THANKS !!!!