Rvalue References and Move Semantics

Valentina Gaggero

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;
106        return h2;
107    }
108
109
110 ▼ 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    {
15    private:
16        int32_t *m_data; //heap allocated, expensive copy
17        enum { DefaultDataSize = 100000 };
18        size_t m_size;
19
20    public:
21
22 ▼      Holder(uint32_t size)
23        {
24            std::cout << "Holder class constructor" << std::endl;
25            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()
33        {
34            std::cout << "Holder class destructor" << std::endl;
35            if(m_data!= nullptr)
36                delete[] m_data;
37            m_size=0;
38        }
39
40 ▼      Holder(const Holder &other)
41        {
42            std::cout << "Holder class copy constructor" << std::endl;
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        {
50            std::cout << "Holder class copy assignment operator" << std::endl;
51            if(this == &other) return *this;
52            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);
55            m_size = other.m_size;
56            return *this;
57        }
```

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;
106        return h2;
107    }
108
109
110 ▼ int main()
111    {
112        Holder my_holder=retrieveHolder();
113        //.....
114        getchar();
115    }
```

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

Holder class destructor
```

```
13 ▼ class Holder
14    {
15    private:
16        int32_t *m_data; //heap allocated, expensive copy
17        enum { DefaultDataSize = 100000 };
18        size_t m_size;
19
20    public:
21
22 ▼     Holder(uint32_t size)
23        {
24            std::cout << "Holder class constructor" << std::endl;
25            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()
33        {
34            std::cout << "Holder class deconstructor" << std::endl;
35            delete[] m_data;
36            m_size=0;
37        }
38
39 ▼     Holder(const Holder &other)
40        {
            std::cout << "Holder class copy constructor" << std::endl;
            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;

   r& 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;
56            return *this;
57    }
```

Why not reuse the temporary?

Solution: move constructor

The Move Constructor is a **overloaded constructor** called for **temporaries.**

It can "steal the guts" of the other object

```
73 ▼        Holder(Holder &&other)
74          {
75              m_data = other.m_data;
76              m_size = other.m_size;
77              other.m_data = nullptr;
78              other.m_size = 0;
79          }
80
```

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

```cpp
73 ▼    Holder(Holder &&other) noexcept
74       {
75           m_data = other.m_data;
76           m_size = other.m_size;
77           other.m_data = nullptr;
78           other.m_size = 0;
79       }
80
```

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

noexcept: 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
110 ▼  int main()
111    {
112        Holder my_holder=retrieveHolder();
113        //.....
114        getchar();
115    }
```

If Holder is a copy-only class …

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

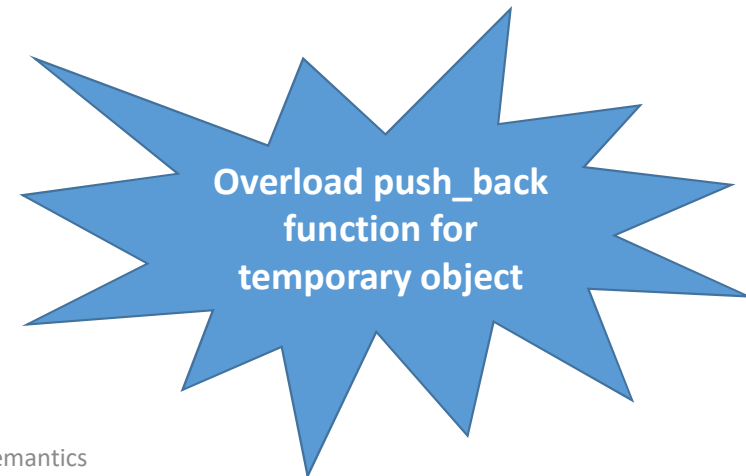Everything that is a temporary ➡️ **rvalue**

int x = 666;
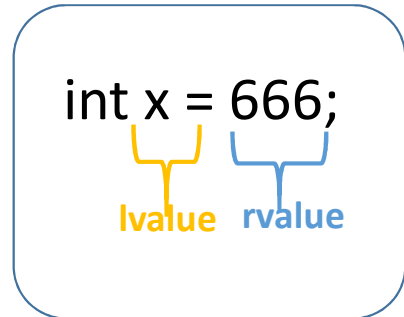
lvalue rvalue

In C++:

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

```
….
Holder h(100);
std::vector<Holder> v;
v.push_back(h);
v.push_back(Holder(200));
….
```

rvalue , temporary object

Overload push_back function for temporary object

Overload on temporaries

Example: std::vector::push_back(T&& element)

```cpp
 1  template <typename T>
 2  class vector {
 3
 4      ...
 5
 6      void push_back(const T& element)
 7      {
 8          if (!resize(m_size + 1))
 9              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  }
```

✓ Use **std::move** to tell the compiler that an object can be moved from

✓ Turns lvalues into rvalues, does not move itself.

The move assignment operator is called

Overload on temporaries

Example: std::vector::push_back(T&& element)

```
 1  template <typename T>
 2  class vector {
 3
 4      ...
 5
 6      void push_back(const T& element)
 7      {
 8          if (!resize(m_size + 1))
 9              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  }
```

```
70  Holder(Holder &&other) noexcept
71  {
72      std::cout << "Holder class move constructor operator" << std::endl;
73      m_data = other.m_data;
74      m_size = other.m_size;
75      other.m_data = nullptr;
76      other.m_size = 0;
77  }
78
79  Holder& operator=(Holder &&other) noexcept
80  {
81      std::cout << "Holder class move assignment operator" << std::endl;
82      if(m_data != nullptr)
83          delete [] m_data;
84      m_data = other.m_data;
85      m_size = other.m_size;
86      other.m_data = nullptr;
87      other.m_size = 0;
88  }
```

Move assignment operator

```cpp
118
119
120
121
122
123
124
125  ▼  int main()
126     {
127         Holder my_holder=retrieveHolder();
128         Holder holder_first(100);
129         Holder holder_second(200);
130         //...
131         cout << endl << "INIT:";
132         cout << "my_holder size=" << my_holder.getSize()
133
134         holder_first=my_holder;
135         cout << "my_holder size=" << my_holder.getSize()
136
137         holder_second=std::move(my_holder);
138         cout << "my_holder size=" << my_holder.getSize()
139         delete my_holder;
140         getchar();
141     }
142
143
144
```

```
vgaggero@iiticublap127: ~/corsoCPP
         vgaggero@iiticublap127: ~/corsoCPP 80x24
Holder class deconstructor
vgaggero@iiticublap127:~/corsoCPP$ g++  -o moveSem_example move-semantics.cpp
vgaggero@iiticublap127:~/corsoCPP$ ./moveSem_example
Holder class constructor
Holder class constructor
Holder class move constructor
Holder class destructor
Holder class destructor
Holder class constructor
Holder class constructor

INIT:my_holder size=10000, holder_first size=100, holder_second size=200

Holder class copy assignment operator
my_holder size=10000, holder_first size=10000, holder_second size=200

Holder class move assignment operator
my_holder size=0, holder_first size=10000, holder_second size=10000


Holder class destructor
Holder class destructor
Holder class destructor
vgaggero@iiticublap127:~/corsoCPP$ ▯
```

my_holder is empty…. I should not use it anymore!!

Move assignment operator


```cpp
118
119
120
121
122
123
124
125  ▼  int main()
126     {
127         Holder my_holder=retrieveHolder();
128         Holder holder_first(100);
129         Holder holder_second(200);
130         //...
131         cout << endl << "INIT:";
132         cout << "my_holder size=" << my_holder.getSize()
133
134         holder_first=my_holder;
135         cout << "my_holder size=" << my_holder.getSize()
136
137         holder_second=std::move(my_holder);
138         cout << "my_holder size=" << my_holder.getSize()
139         delete my_holder;
140         getchar();
141     }
142
143
144
```

```
vgaggero@iiticublap127: ~/corsoCPP
vgaggero@iiticublap127: ~/corsoCPP 80x24
Holder class deconstructor
vgaggero@iiticublap127:~/corsoCPP$ g++  -o moveSem_example move-semantics.cpp
```

```cpp
32  ▼      ~Holder()
33         {
34             std::cout << "Holder class destructor" << std::endl;
35  →         if(m_data!= nullptr)
36                 delete[] m_data;
37             m_size=0;
38         }
```

```
Holder class destructor
Holder class destructor
Holder class destructor
vgaggero@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.

- For example:
  - std::unique_ptr
  - std::mutex
  - std::thread
  - …

Transfer of ownership semantics when moved

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 <
- (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

```
150 ▼ int main()
151   {
152       vector<Holder> v;
153 ▼     for(auto i=0; i<100; i++)
154       {
155           v.push_back(Holder(1000));
156       }
157
158       cout<< "Num of copies is  "<< numOfCopies <<endl;
159
160       sort(v.begin(), v.end());
161
162       cout << "After sort, num of copies is " << numOfCopies <
163
164   }
165
```

If Holder is a copy-only class …

```
vgaggero@iiticublap127:~/corsoCPP$ ./simpleExample
Num of copies is   227
After sort, num of copies is 897
```

If Holder is movable class …

```
vgaggero@iiticublap127:~/corsoCPP$ ./simpleExample
Num of copies is   0
After sort, num of copies is 0
```

Question

```
166  ▼  static Holder createHolder()
167     {
168         Holder bigH(60000);
169         //....do something with bigH
170         return bigH;
171     }
172
173  ▼  int main()
174     {
175         Holder obj=createHolder();
176         ....
177         getchar();
178     }
```

If Holder class has both move-constructor and move-assignment, how many "special member function" are invoked running this code?

Answer: only 1 constructor!!!

```
vgaggero@iiticublap127:~/corsoCPP$ ./move-se
Holder class constructor

Holder class destructor
```

This is due to **return value optimization**

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 (T &&)

- ✓ 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)

```cpp
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 {
 3
 4      ...
 5
 6      void push_back(const T& element)
 7      {
 8          if (!resize(m_size + 1))
 9              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  }
```

```
 1  BigObject &BigObject::operator=(BigObject &&other) noexcept
 2  {
 3      BigObject moved(std::move(other));
 4      swap(moved);
 5      return *this;
 6  }
 7  void BigObject::swap(BigObject &other) noexcept {
 8      using std::swap; // enable ADL
 9      swap(lotsOfData, other.lotsOfData);
10  }
```

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
 6      a = std::move(b);
 7      b = std::move(temp);
 8  }
```

THANKS !!!!