Rvalue references and move semantics

Jacob Mossberg
gbgcpp – Gothenburg C++ Meetup
2017-09-06

Agenda

- What is the goal with move semantics?
- What is a rvalue?
- What is a rvalue reference?
- What is move semantics?
- Forcing move semantics
- Move semantics and compiler optimization
- Exercises

Agenda

- **What is the goal with move semantics?**
- What is a rvalue?
- What is a rvalue reference?
- What is move semantics?
- Forcing move semantics
- Move semantics and compiler optimization
- Exercises

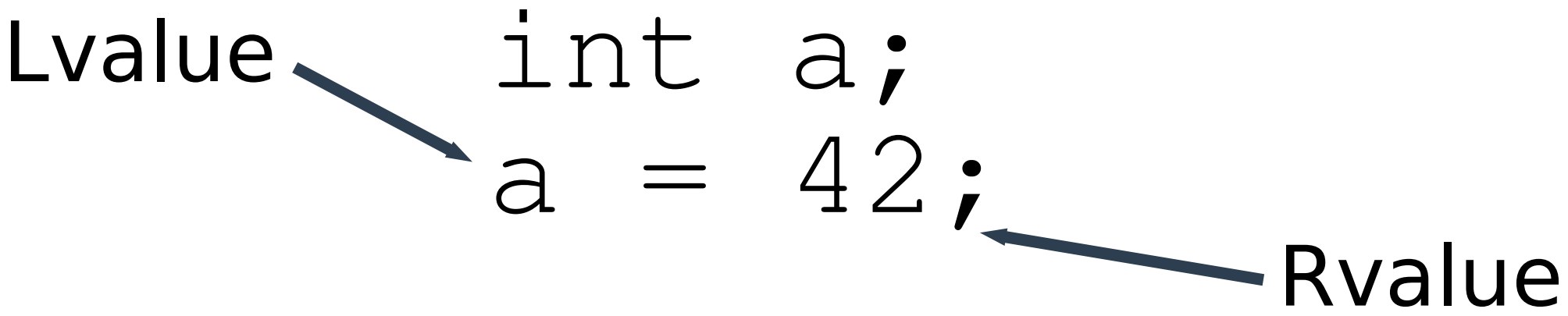
What is the goal with move semantics?

To steal resources from objects that are about to “die”, instead of making copies.

Agenda

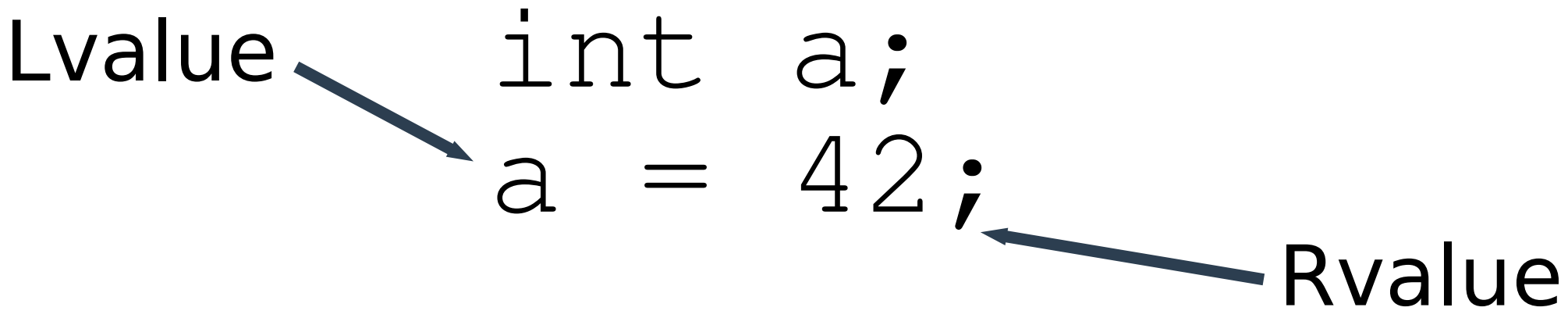
- What is the goal with move semantics?
- **What is a rvalue?**
- What is a rvalue reference?
- What is move semantics?
- Forcing move semantics
- Move semantics and compiler optimization
- Exercises

What is a rvalue?



What is a rvalue?

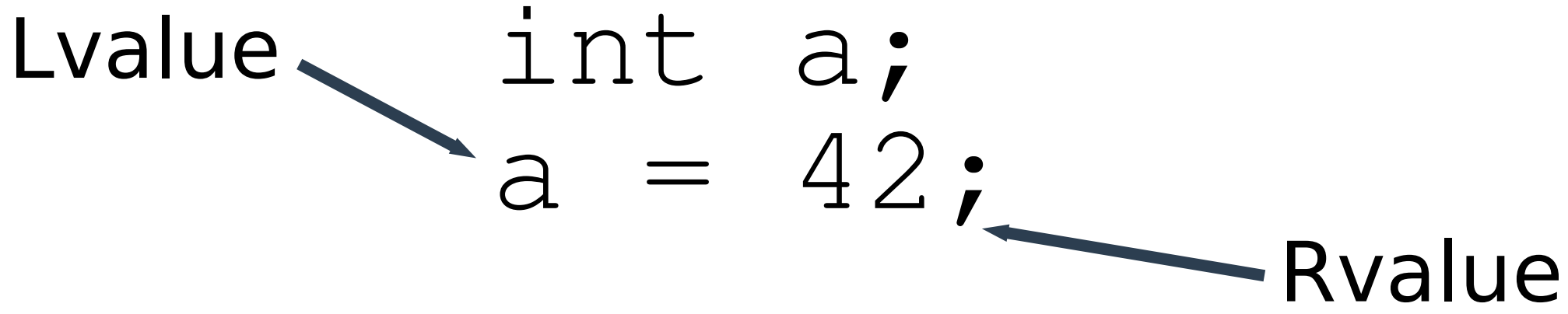
Lvalue may appear on the **left** hand side or right hand side of and assignment



What is a rvalue?

Lvalue may appear on the **left** hand side or right hand side of and assignment

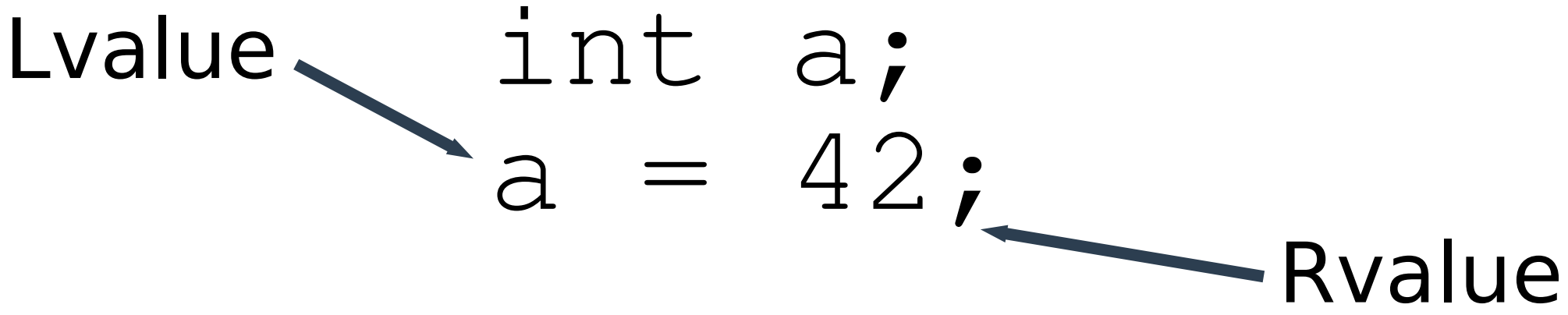
Lvalue refers to a location in memory. We can retrieve the address via the & operator.



What is a rvalue?

Lvalue may appear on the **left** hand side or right hand side of and assignment

Lvalue refers to a location in memory. We can retrieve the address via the & operator.

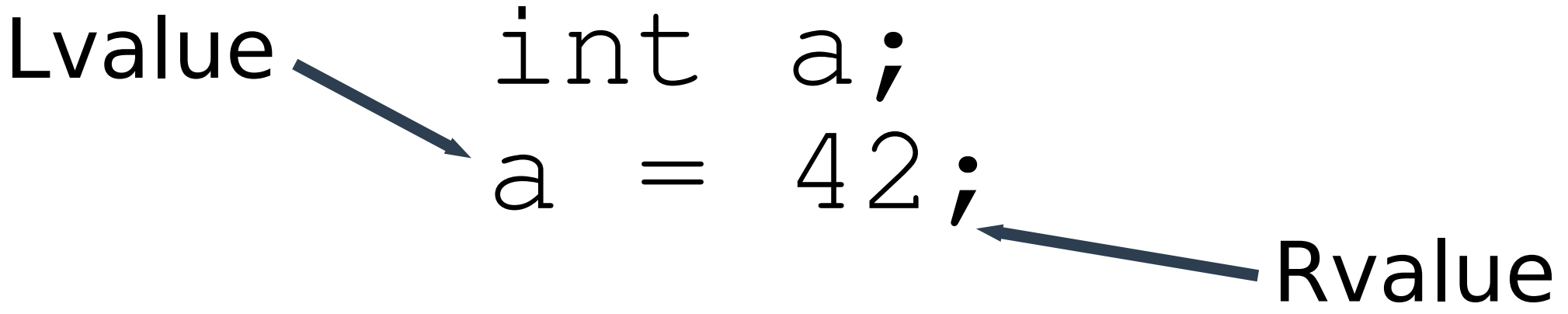


Rvalue may **only** appear on the **right** hand side of and assignment

What is a rvalue?

Lvalue may appear on the **left** hand side or right hand side of and assignment

Lvalue refers to a location in memory. We can retrieve the address via the & operator.



Rvalue may **only** appear on the **right** hand side of and assignment

Rvalue is often a temporary object

Lvalue and Rvalue examples

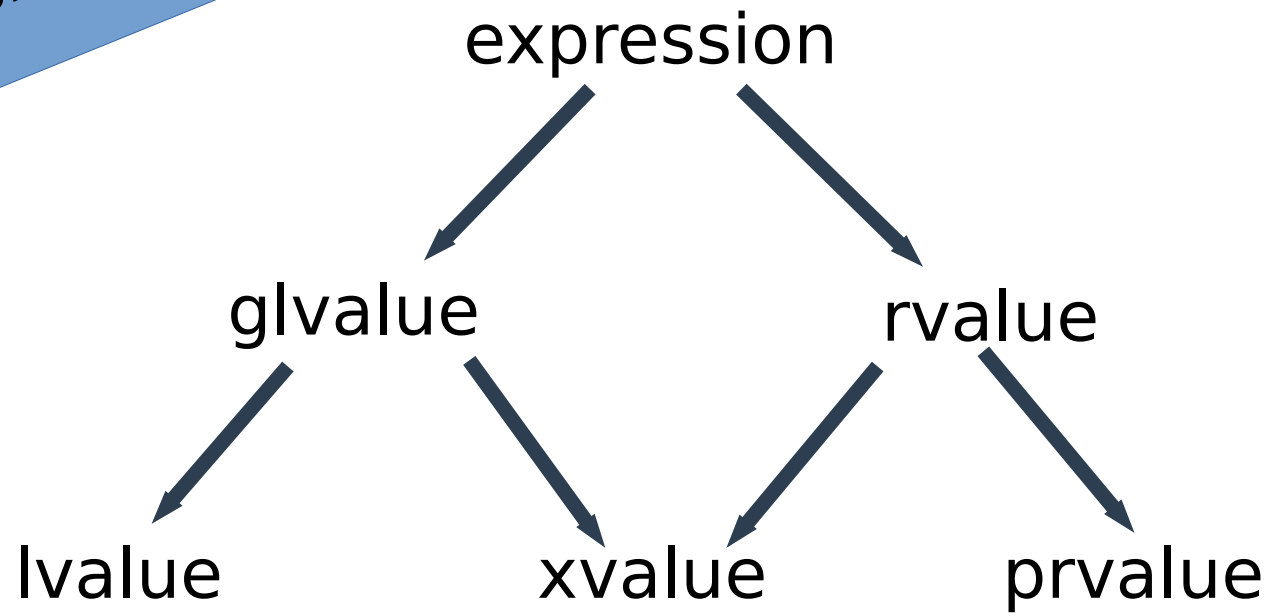
```
int a = 1; //a is lvalue, 1 is rvalue
a = 2;     //ok, a is lvalue

int& myint1();
myint1() = 3; //ok, myint1() is lvalue

int myint2();
myint2() = 4; //error, myint2() is
              //rvalue
```

Rvalues and Lvalues in C++14 standard

On page 78 of 1354...



Rvalues and Lvalues in C++14 standard

- An **lvalue** designates a function or an object.
- An **xvalue** (an “eXpiring” value) also refers to an object, usually near the end of its lifetime
- An **rvalue** is an *xvalue*, a temporary object or subobject thereof, or a value that is not associated with an object.
- A **glvalue** (“generalized” lvalue) is an *lvalue* or an *xvalue*.
- A **prvalue** (“pure” rvalue) is an *rvalue* that is not an *xvalue*.

Agenda

- What is the goal with move semantics?
- What is a rvalue?
- **What is a rvalue reference?**
- What is move semantics?
- Forcing move semantics
- Move semantics and compiler optimization
- Exercises

What is a Lvalue reference?

A “normal” reference, i.e. a **Lvalue** reference, is created by adding **&** to the type when declaring a variable

```
int a = 1;
int& lr{a}; //lr is a reference to a
lr = 2; //a becomes 2
int& lr2{3}; //ERROR, cannot bind to temporary
```

What is a Rvalue reference?

A **Rvalue** reference is created by adding **&&** to the type when declaring a variable

```
int&& rr{3}; //OK
```

rr is a rvalue reference and can bind to temporary holding 3

Function to show reference type

```
void print_reference_type(int& lref)
{
   cout << “I’m a Lvalue reference: ” << lref << endl;
}
void print_reference_type(int&& rref)
{
   cout << “I’m a Rvalue reference: ” << rref << endl;
}
```

Lvalue and Rvalue print outs

```
int a = 1;
print_reference_type(a);
print_reference_type(2);

I'm a Lvalue reference: 1
I'm a Rvalue reference: 2
```

Agenda

- What is the goal with move semantics?
- What is a rvalue?
- What is a rvalue reference?
- **What is move semantics?**
- Forcing move semantics
- Move semantics and compiler optimization
- Exercises

Class A constructor & destructor

```
class A {
public:
   A(const int& number); //constructor
   ~A();                 //destructor
private:
   int * _number;
};
```

Class A constructor & destructor

```
//constructor
A::A(const int& number): _number{new int{number}} {}

//destructor
A::~A()
{
    delete _number;
}
```

Class A constructor & destructor

```
A a1{3}; //use constructor
```

Class A copy constructor

```
A a1{3}; //use constructor
A a2{a1}; //we need a copy constructor
```

Class A copy constructor

```
class A {
public:
    ...
    A(const A& other); //copy constructor
    ...
};

A::A(const A& other) : number{new int{*other._number}} {}
```

Class A copy constructor

```
A a1{3}; //use constructor
A a2{a1}; //use copy constructor
```

Class A move constructor

```
A a1{3}; //use constructor
A a2{a1}; //use copy constructor
vector<A> v;
v.push_back(A{4}); //use copy constructor
```

Class A move constructor

```
A a1{3}; //use constructor
A a2{a1}; //use copy constructor
vector<A> v;
v.push_back(A{4}); //use copy constructor
                   //but A{4} looks like a rvalue!
```

Class A move constructor

```
A a1{3}; //use constructor
A a2{a1}; //use copy constructor
vector<A> v;
v.push_back(A{4}); //use copy constructor
                   //but A(4) looks like a rvalue!
```

Let us implement a move constructor!

Class A move constructor

```
class A {
public:
    ...
    A(A&& other); //move constructor
    ...
};
A::A(A&& other)
{
    number = other._number;
    other._number = nullptr;
}
```

Class A move constructor

```
class A {
public:
    ...
    A(A&& other); //move constructor
    ...
};
A::A(A&& other)
{
    number = other._number;
    other._number = nullptr;
}
```

We are “moving” the resources from **other** to the new instance of class A

Class A move constructor

```
A a1{3}; //use constructor
A a2{a1}; //use copy constructor
vector<A> v;
v.push_back(A{4}); //use move constructor
```

Class A copy assignment operator

```
A a1{3}; //use constructor
A a2{a1}; //use copy constructor
vector<A> v;
v.push_back(A{4}); //use move constructor
A a3{5};
a3 = a2; //need copy assignment operator
```

Class A copy assignment operator

```
class A {
public:
   ...
   A& operator=(const A& other); //copy assignment operator
   ...
};
A& A::operator=(const A& other)
{
   *_number = *other._number;
   return *this;
}
```

Class A copy assignment operator

```
A a1{3}; //use constructor
A a2{a1}; //use copy constructor
vector<A> v;
v.push_back(A{4}); //use move constructor
A a3{5};
a3 = a2; //use copy assignment operator
```

Class A move assignment operator

```
A a1{3}; //use constructor
A a2{a1}; //use copy constructor
vector<A> v;
v.push_back(A{4}); //use move constructor
A a3{5};
a3 = a2; //use copy assignment operator
a3 = A{6}; //use copy assignment operator
```

Class A move assignment operator

```
A a1{3}; //use constructor
A a2{a1}; //use copy constructor
vector<A> v;
v.push_back(A{4}); //use move constructor
A a3{5};
a3 = a2; //use copy assignment operator
a3 = A{6}; //use copy assignment operator,
           //but A{6} looks like a rvalue!
```

Class A move assignment operator

```
A a1{3}; //use constructor
A a2{a1}; //use copy constructor
vector<A> v;
v.push_back(A{4}); //use move constructor
A a3{5};
a3 = a2; //use copy assignment operator
a3 = A{6}; //use copy assignment operator,
           //but A{6} looks like a rvalue!
```

Let us implement a move assignment operator!

Class A move assignment operator

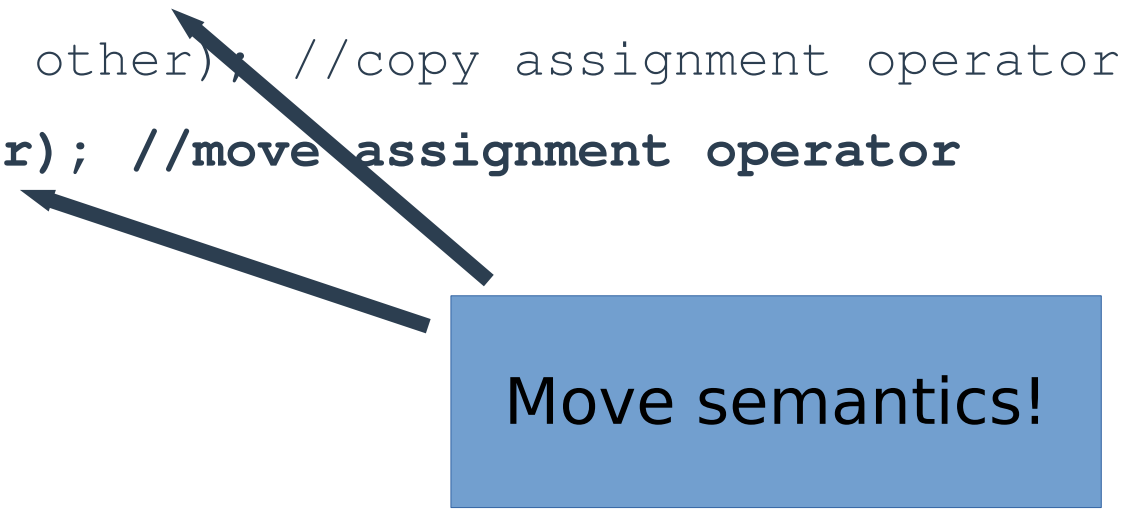
```
Class A {
public:
   ...
   A& operator=(A&& other); //move assignment operator
   ...
};
A& A::operator=(A&& other)
{
   delete _number; //delete old resource
   number = other._number; //move resource from other
   other._number = nullptr; //set other to valid by empty state
   return *this;
}
```

Class A move assignment operator

```
A a1{3}; //use constructor
A a2{a1}; //use copy constructor
vector<A> v;
v.push_back(A{4}); //use move constructor
A a3{5};
a3 = a2; //use copy assignment operator
a3 = A{6}; //use move assignment operator
```

Class A move semantics

```
class A {
public:
   A(int number); //constructor
   ~A(); //destructor
   A(const A& other); //copy constructor
   A(A&& other); //move constructor
   A& operator=(const A& other); //copy assignment operator
   A& operator=(A&& other); //move assignment operator
private:
   int * _number;
};
```



Agenda

- What is the goal with move semantics?
- What is a rvalue?
- What is a rvalue reference?
- What is move semantics?
- **Forcing move semantics**
- Move semantics and compiler optimization

Forcing move semantics

Move function performs cast to rvalue reference

```
A a1{1};
A a2{move(a1)}; //a2 _number is set to value 1,
                //a1 _number is set to nullptr
```

Forcing move semantics

The typical use case for move is within move constructors and move assignment operator

```
//move assignment operator of class B
B& B::operator=(B&& other)
{
    c = move(other.c);
    d = move(other.d);
    e = move(other.e);
}
```

Agenda

- What is the goal with move semantics?
- What is a rvalue?
- What is a rvalue reference?
- What is move semantics?
- Forcing move semantics
- **Move semantics and compiler optimization**
- Exercises

Move semantics and compiler optimization

Compiler optimization reduces occasions when move semantics are actually used

```
A a1 = A{1}; //copy elision optimization


A factory()
{
   A a3{3};
   return a3; //return value optimization
}
A a2 = factory();
```

Agenda

- What is the goal with move semantics?
- What is a rvalue?
- What is a rvalue reference?
- What is move semantics?
- Forcing move semantics
- Move semantics and compiler optimization
- **Exercises! :)**

Exercises! :)

https://github.com/jmossberg/rvalues-move-semantics-cpp