Move Semantics

CS 106L Winter 2020 – Avery Wang and Anna Zeng

Announcements

- Fill out the survey on Piazza for an extra late day. Also vote on final lecture topics.
- I will stop at exactly 2:20 PM today. Let’s do an open Q&A today about the assignment.
- You should be done with part 6 already. Get started if you haven’t already!

Reminders

- We'll make sure the full lecture code is posted after lecture.
- There's no starter code today – just the full lecture code.

recap

Member Initialization List

q?

Prefer to use member initialization list, which constructs each member with given value.

- Faster. Why construct, then reassign?
- Some types can’t be reassigned (=delete)

```
template <typename T>
MyVector<T>::MyVector<T>() :
          logicalSize(0), allocatedSize(kInitialSize),
          elems(new T[kInitialSize]) { }
```

Special member functions are (usually) automatically generated by the compiler.

- Default construction: object created with no parameters.
- Copy construction: object is created as a copy of an existing object.
- Copy assignment: existing object replaced as a copy of another existing object.
- Destruction: object destroyed when it is out of scope.

Constructor and destructor for MyVector<T>.

```
// constructor using member initialization list
MyVector<T>::MyVector() :
          logicalSize(0),
          allocatedSize(kInitialSize),
          elems(new T[allocatedSize]) {
}

// destructor
MyVector<T>::~MyVector() {
     delete [] elems;
}
```

The copy operations must perform the following tasks.

Copy Constructor

- Copy members using initializer list when assignment works.
- Deep copy members where assignment does not work.

Copy Assignment

- Clean up any resources in the existing object about to be overwritten.
- Copy members using initializer list when assignment works.
- Deep copy members where assignment does not work.

Copy constructor copies each member, creating deep copy when necessary.

```
MyVector<T>::MyVector(const MyVector<T>& other) :
            logicalSize(other.logicalSize),
            allocatedSize(other.allocatedSize),
            elems(new T[allocatedSize]) {

    std::copy(other.begin(), other.end(), begin());
}
```

Copy assignment copies each member, replacing existing object.

```
// can't use initializer list – not a constructor!
MyVector<T>& MyVector::operator=(const MyVector<T>& rhs) {
   if (this != &rhs) {
      delete [] elems;
      logicalSize = rhs.logicalSize;
      allocatedSize = rhs.allocatedSize;
      elems = new T[allocatedSize];
      std::copy(rhs.begin(), rhs.end(), begin());
   }
    return *this;
}
```

What members need to be deep copied?

Any member that are handles to external resources.

- Handles to memory (pointers)
- Handles to files (filestreams)
- Handles to locks (mutexes)

Rule of Three

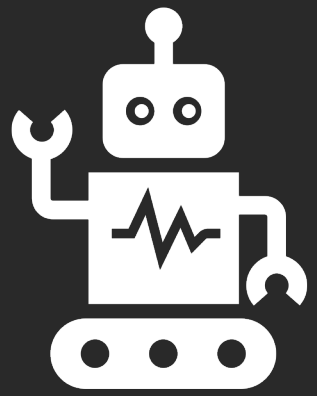
If you explicitly define (or delete)
a copy constructor, copy assignment, or destructor,
you should define (or delete) all three.

The fact that you defined one of these means
one of your members has ownership issues
that need to be resolved.

motivation

Quick quiz: how many times is each special member function called (with and without copy elision)?

```
int main() {
  MyVector<string> names1 = findAllWords(54321234);
  MyVector<string> names2;
  names2 = findAllWords(54321234);
  cout << “done!” << endl;
}

MyVector<string> findAllWords(size_t size) {
  MyVector<string> names(size, “Ito”);
  return names;
}
```



Demo

Printing all calls to special member functions

Quick quiz: how many times is each special member function called (with and without copy elision)?

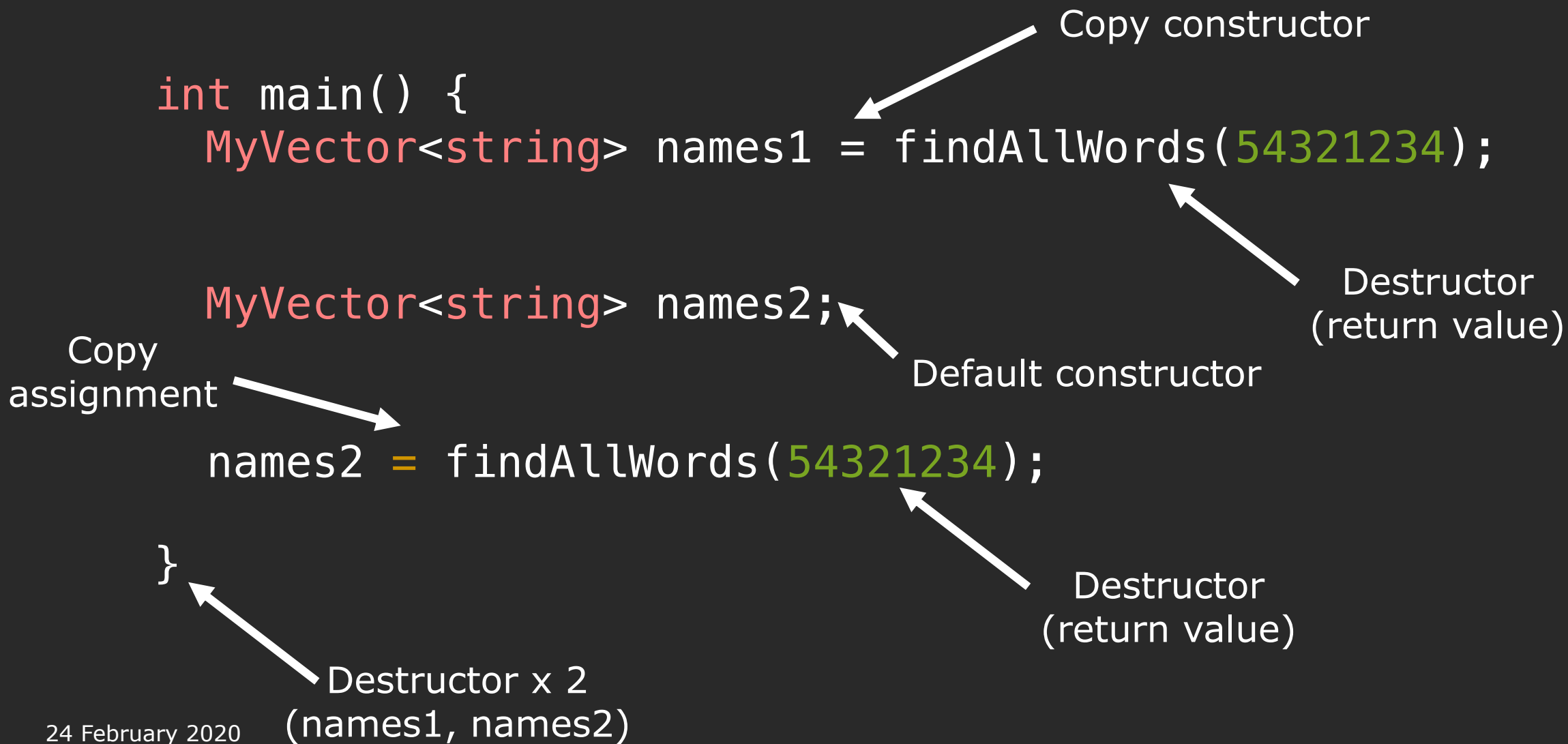
```
MyVector<string> findAllWords(size_t size) {
  MyVector<string> names(size, “Ito”);
  return names;
}
```

Fill constructor

Copy constructor
(return value)

Destructor
for names

Quick quiz: how many times is each special member function called (with and without copy elision)?



Counts without copy elision.

`findAllWords`

- Fill constructor x 1
- Copy constructor x 1
- Destructor x 1

Counts without copy elision.

`main`

- Copy constructor x 1
- Default constructor x 1
- Copy assignment x 1
- `findAllWords` x 2
  - Fill constructor x 1
  - Copy constructor x 1
  - Destructor x 1
- Destructor x 4

Counts without copy elision.

`main`

- Copy assignment x 1
- Copy constructor x 3
- Default constructor x 1
- Destructor x 6
- Fill constructor x 2

copy elision and return value optimization (RVO)

Quick quiz: how many times is each special member function called (with and without copy elision)?

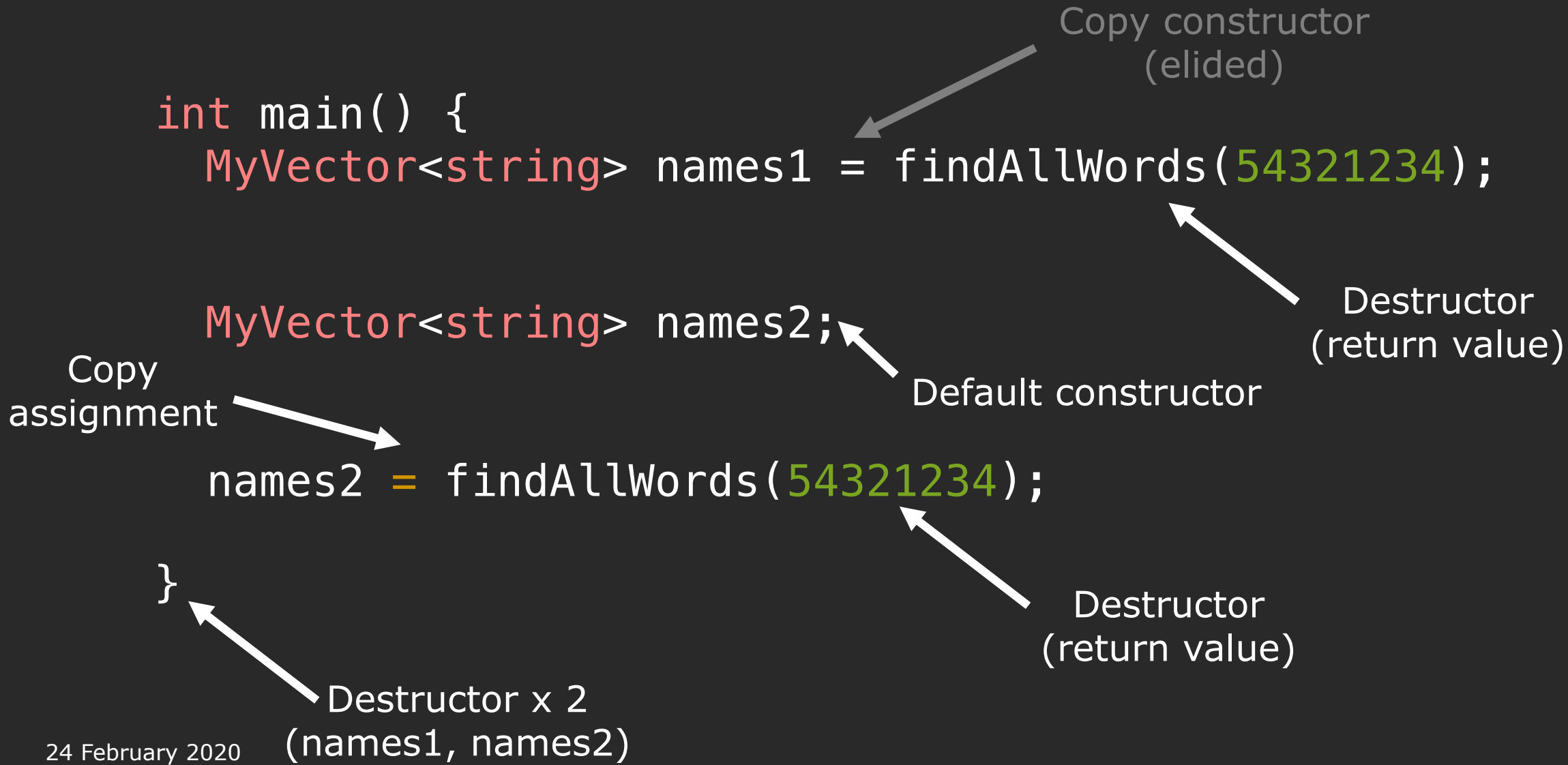
```
MyVector<string> findAllWords(size_t size) {
  MyVector<string> names(size, “Ito”);
  return names;
}
```

Fill constructor

Copy constructor (elided)

Destructor for names

Quick quiz: how many times is each special member function called (with and without copy elision)?



Counts with copy elision.

`findAllWords`

- Fill constructor x 1
- Copy constructor x 1
- Destructor x 1

Counts with copy elision.

`main`

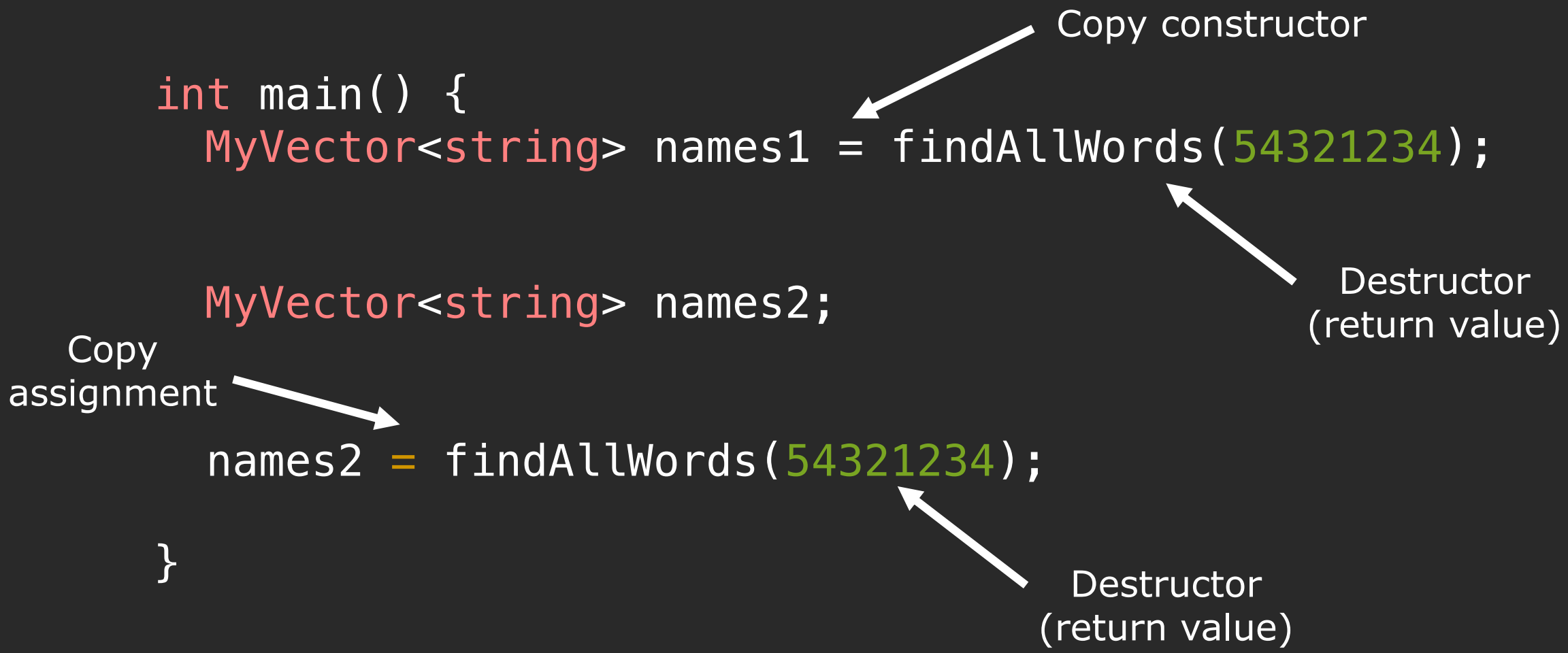
- Copy constructor x 1
- Default constructor x 1
- Copy assignment x 1
- `findAllWords` x 2
  - Fill constructor x 1
  - Copy constructor x 1
  - Destructor x 1
- Destructor x 1

Counts with copy elision.

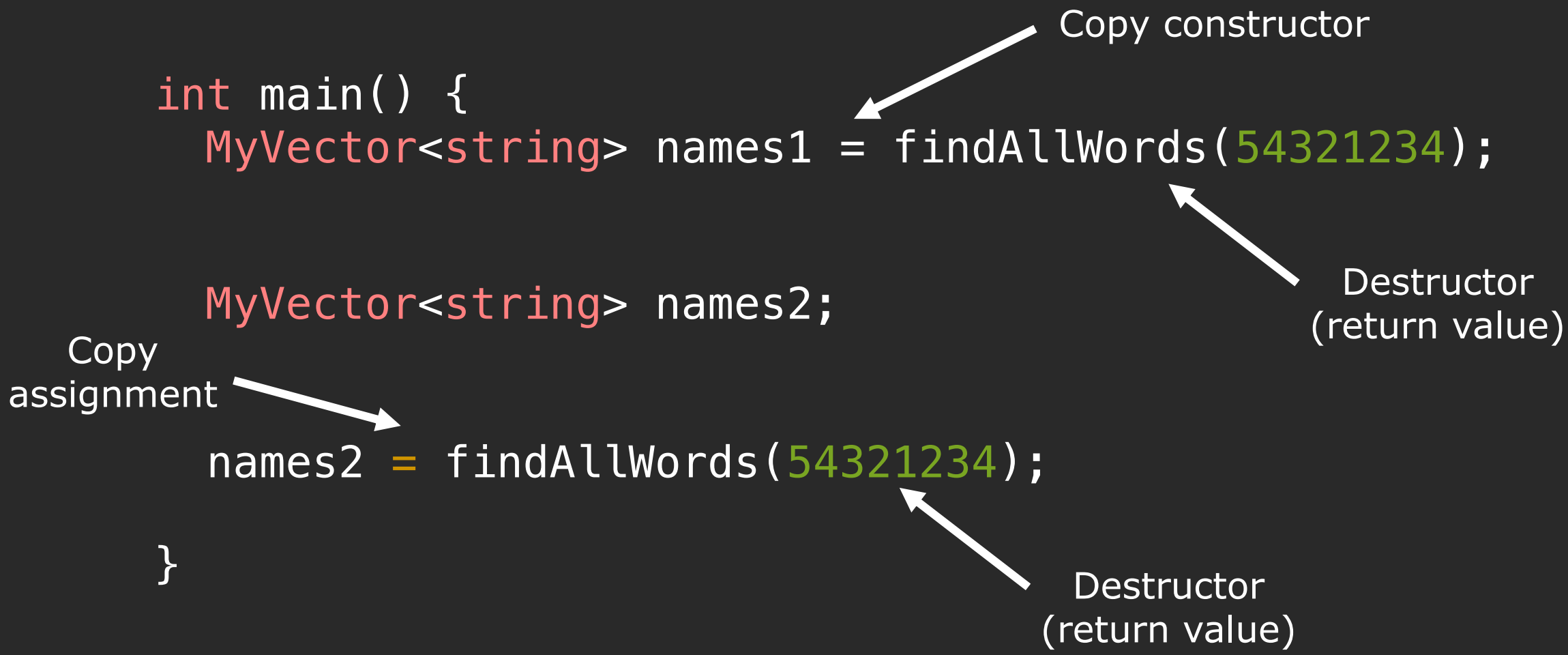
`main`

- Copy assignment x 1
- Copy constructor x 3
- Default constructor x 1
- Destructor x 3
- Fill constructor x 2

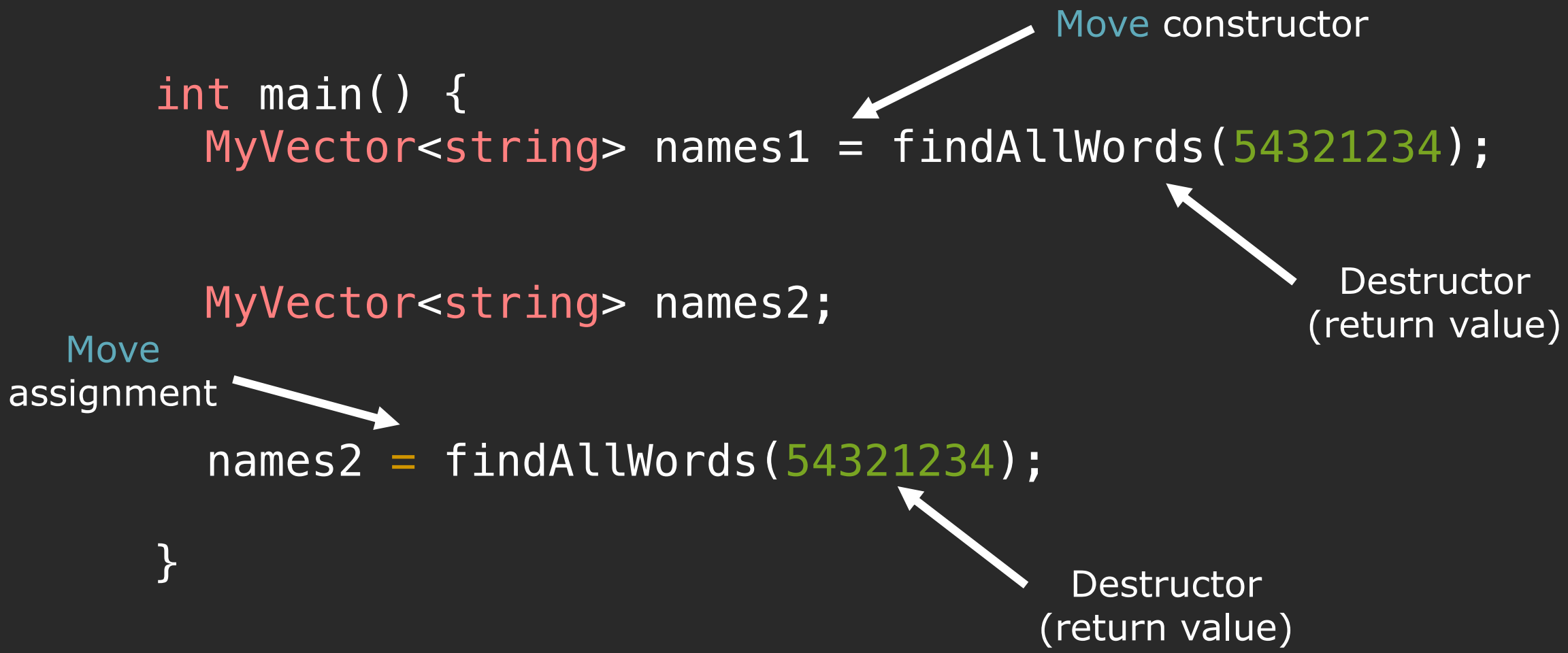
Can we do better?



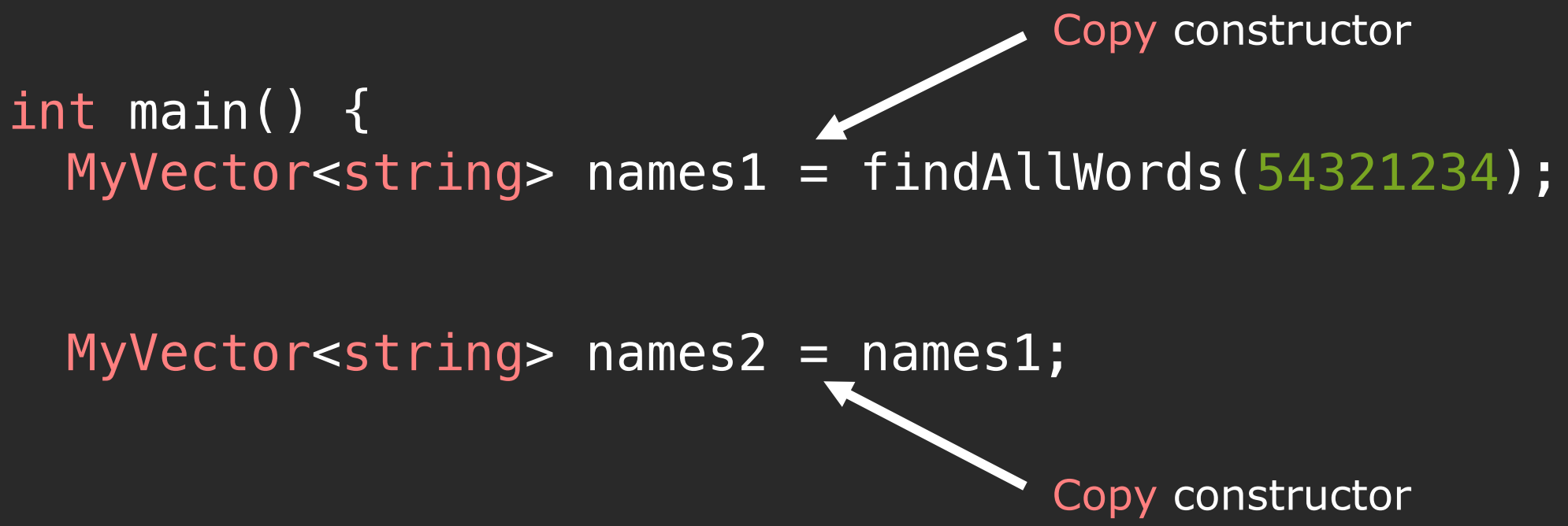
What a shame – we copy then destruct.



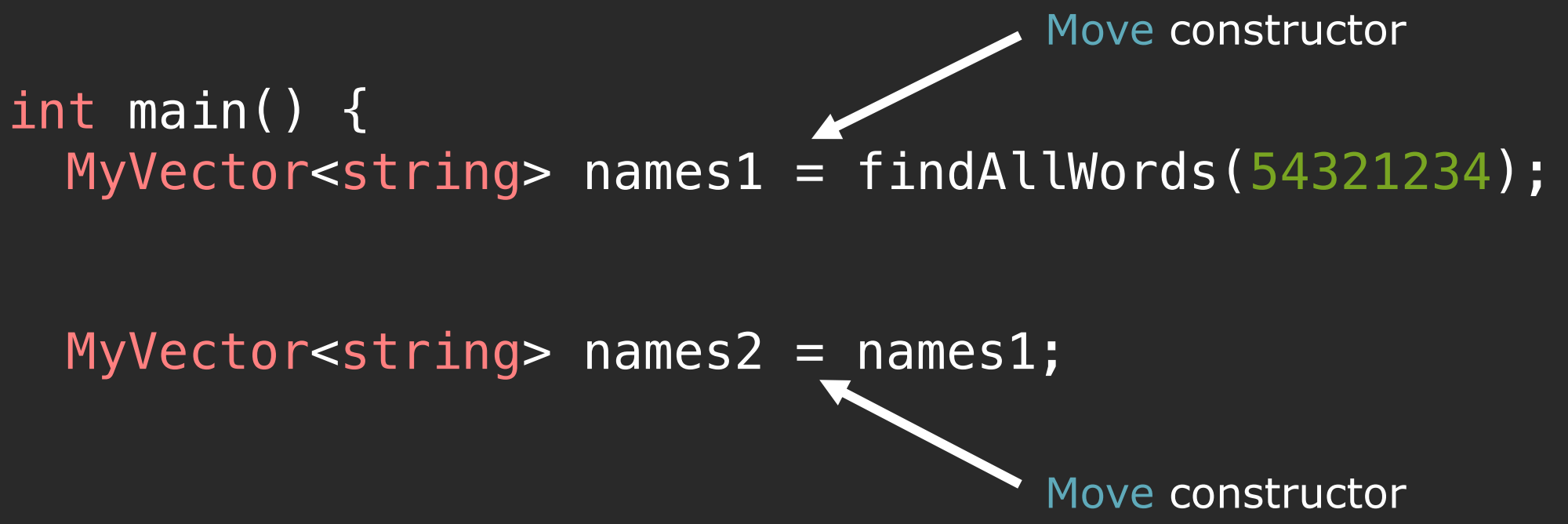
Let’s move the wasted array to our vector!



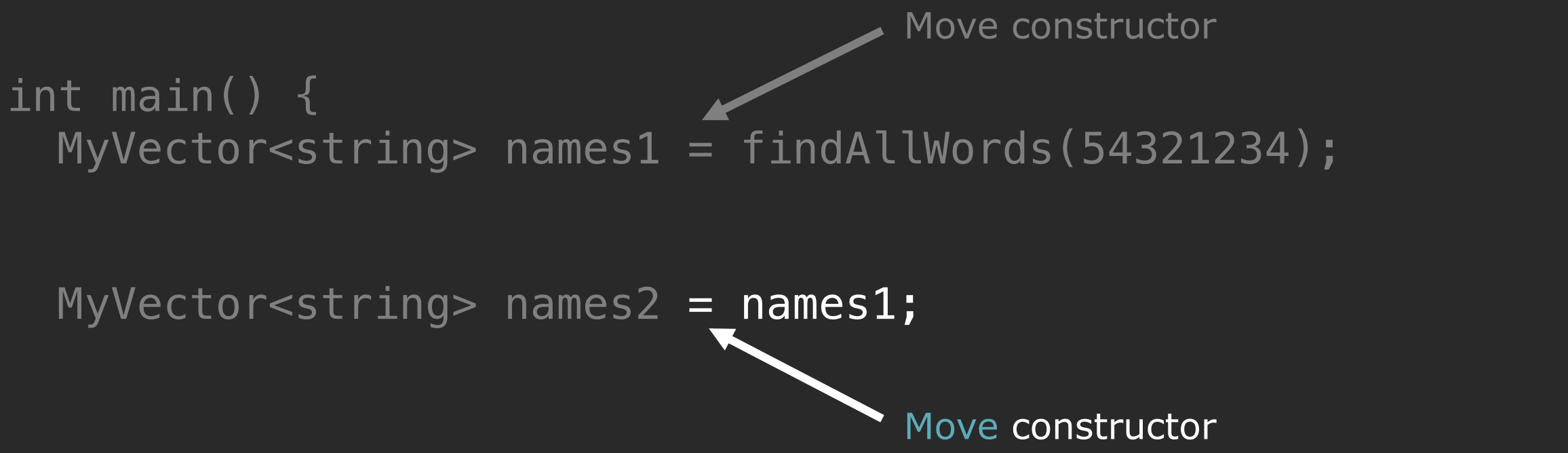
Another example!



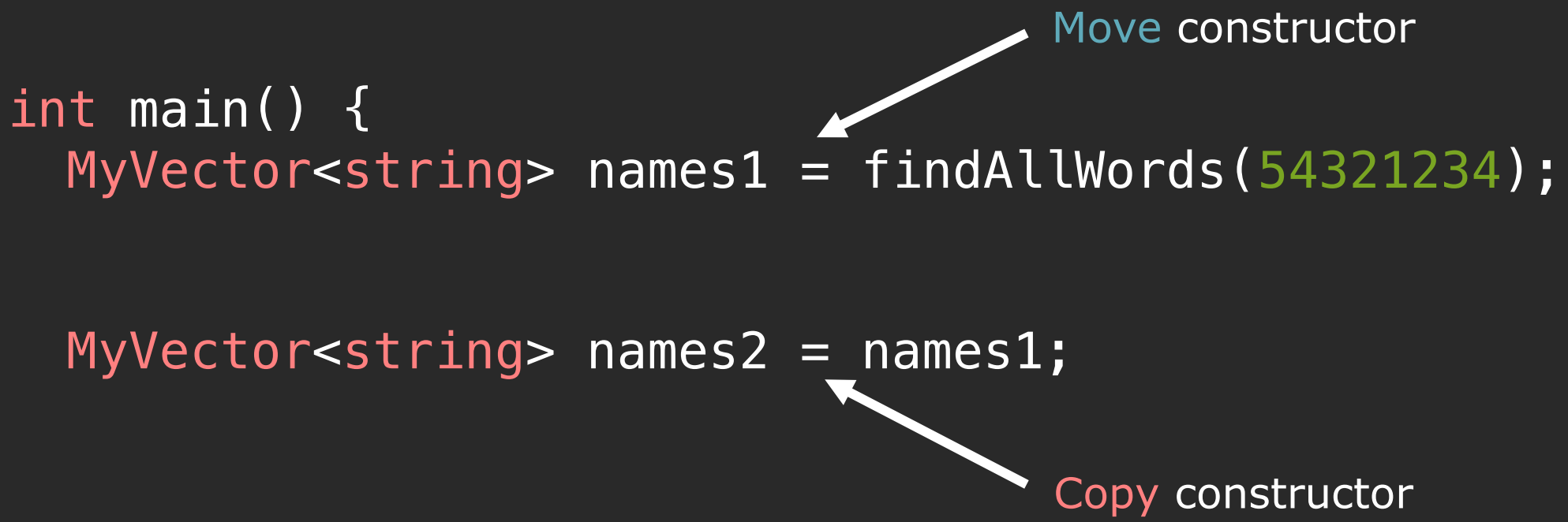
Can we always use the move constructor?



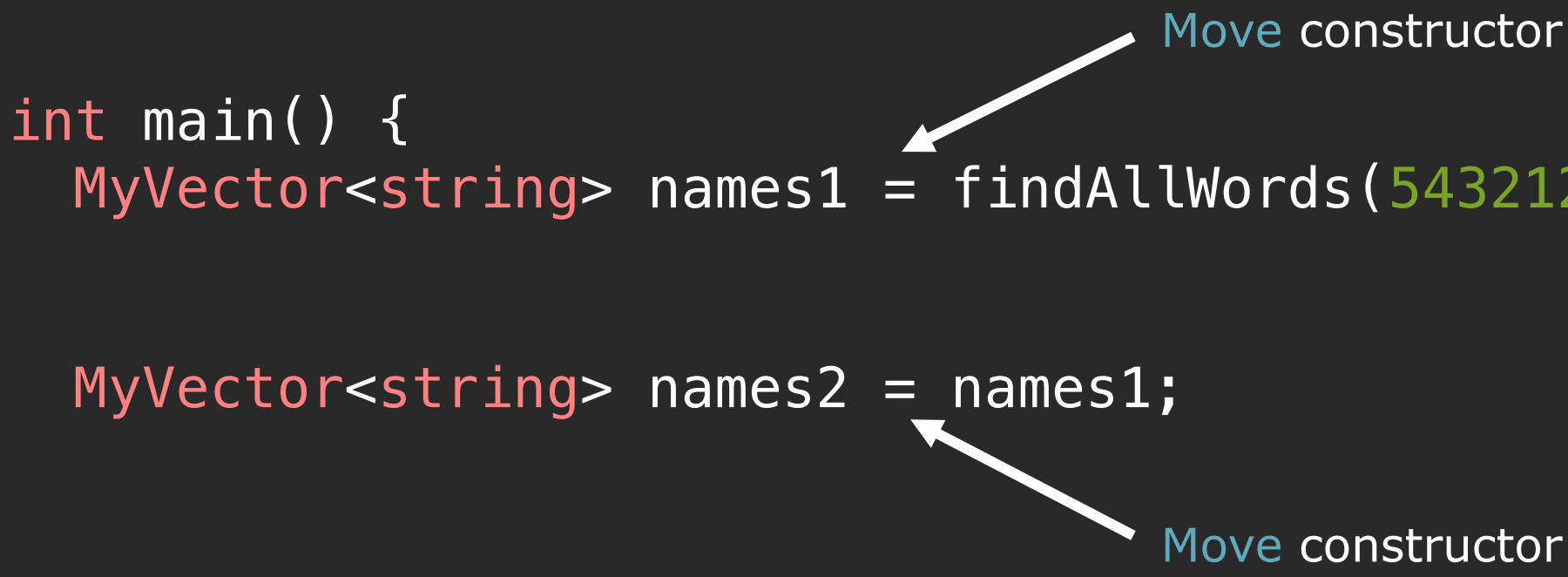
The array was stolen from names1…that's bad!



Hmm…how do we distinguish between these cases?



Also…Can we force the move constructor to be called?



Where we are going!

- How do we distinguish between when we CAN and CANNOT move?
- How do we actually move?
- Can we force a move to occur?
- How do we apply move?
- Can we apply this to templates?

Where we are going!

- How do we distinguish between when we CAN and CANNOT move? = l vs. r-values
- How do we actually move? = implementation
- Can we force a move to occur? = std::move
- How do we apply move? = swap and insert
- Can we apply this to templates? = perfect forwarding

Game Plan

- lvalues vs. rvalues
- move constructor and assignment
- std::move
- swap and insert
- perfect forwarding

lvalues and rvalues

Note: this is a simplification of a complicated topic!

Value Categories: l-value vs. r-value

An l-value is an expression that has a name (identity).

- can find address using address-of operator (&var)

An r-value is an expression that does not have a name (identity).

- temporary values
- cannot find address using address-of operator (&var)

区分方法：1. &(expression)可以找到一个地址/name；2. 假设法：令(expression)为l-value，看能不能给expression赋值：(expression) = value ？ 3. note: l-value可右可左，r-value只能右边，所以能放在左边被赋值一定是l-value，一定不是r-value。4. 函数返回值：return by value: r-value(temp); return by reference: l-value(alias to a real var)

Intuitive definition of l vs. r-values
(this was technically the definition until 2011)

An l-value is an expression that can appear either left or right of an assignment.***

An r-value is an expression that can appear only on the right of an assignment.***

***technically there are these weird things called gl-values, pr-values, x-values, …

Examples: where are the r-values?

```
int val = 2;
int* ptr = 0x02248837;
vector<int> v1{1, 2, 3};

auto v4 = v1 + v2;
auto v5 = v1 += v4;
size_t size = v.size();
val = static_cast<int>(size);
v1[1] = 4*i;
ptr = &val;
v1[2] = *ptr;
```

Here are all the r-values!

```
int val = 2;
int* ptr = 0x02248837;
vector<int> v1{1, 2, 3};

auto v4 = v1 + v2;
auto v5 = v1 += v4;
size_t size = v.size();
val = static_cast<int>(size);
v1[1] = 4*i;
ptr = &val;
v1[2] = *ptr;
```

Don’t have a name!

linked to operator function

+ returns a copy to a temporary.

cast returns a copy of size.

Value type differences: lifetimes

An l-value's lifetime is decided by scope.

An r-value's lifetime ends on the very next line (unless you purposely extend it!)

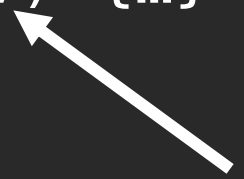
Value type differences: lifetimes

```
{
    vector<int> v1{1, 2, 3};


    auto v4 = v1 + v2;
    // copy constructor for v4
    // destructor on v1 + v2


} // destructor called on v1 and v4
```

Review: what is a reference?

A reference is an alias to an already existing object.

```
int main() {
      vector<int> vec;
      changeVector(vec);
}

void changeVector(vector<int>& v) {…}
```



v is another name
for vec.

Value References: l vs. r-value reference

An l-value reference can bind to an l-value.

An r-value reference can bind to an r-value.

Value References: l vs. r-value reference

An l-value reference can bind to an l-value.

```
auto& ptr2 = (ptr += 3);
```

An r-value reference can bind to an r-value.

```
auto&& v4 = v1 + v2;
```

Value References: l vs. r-value reference

An l-value reference can bind to an l-value.

```
auto& ptr2 = (ptr += 3);
```

returns l-value ref to *this

++a => a += 1

a++ => a+1

An r-value reference can bind to an r-value.

```
auto&& v4 = v1 + v2;
```

returns copy, which is r-value.

Value References: l vs. r-value reference

An l-value reference can bind to an l-value.

```
auto& ptr2 = (ptr += 3);
```

A const l-value reference can bind to either l or r-value.

An r-value reference can bind to an r-value.

```
auto&& v4 = v1 + v2;
```

Value References: l vs. r-value reference

An l-value reference can bind to an l-value.

```
auto& ptr2 = (ptr += 3);
```

A const l-value reference can bind to either l or r-value.

```
const auto& ptr2 = (ptr += 3);
const auto&& v4 = v1 + v2;
```

An r-value reference can bind to an r-value.

```
auto&& v4 = v1 + v2;
```

Which ones cause compiler errors?

```
void lref(vector<int>& v);
void clref(const vector<int>& v);
void rref(vector<int>&& v);
// BTW: no one uses crref


vector<int> v1 = v2 + v3;
lref(v1);
rref(v1);
lref(v2 + v3);
clref(v2 + v3);
rref(v2 + v3);
```

Which ones cause compiler errors?

```
void lref(vector<int>& v);
void clref(const vector<int>& v);
void rref(vector<int>&& v);
// BTW: no one uses crref
```

r-reference作用：1. r部分：声明处理的对象是一个r-value（区分于l- value）；2. reference部分：可以用于pass by reference

```
vector<int> v1 = v2 + v3;
lref(v1);                   // l-ref binds to l-v
rref(v1);                   // r-ref no bind to l-v
lref(v2 + v3);              // l-ref no bind to r-v
clref(v2 + v3);             // cl-ref binds to r-v
rref(v2 + v3);              // r-ref binds to r-v
```

non-const l-reference
不能配r-value，要改r-value的值要明说，用r- value reference；但const l-reference因为不会改变r-value的值

Challenge Question

```
const auto&& v4 = v1 + v2;
```

l or r-value?

Challenge Question

```
const auto&& v4 = v1 + v2;
```

bind to an r–value

An r-value reference is an alias to an r-value

BUT the r-value reference itself is an l-value

Challenge Question

```
const auto&& v4 = v1 + v2;
```

l r

vector<int>&&

v4

l-value

vector<int>

v1 + v2

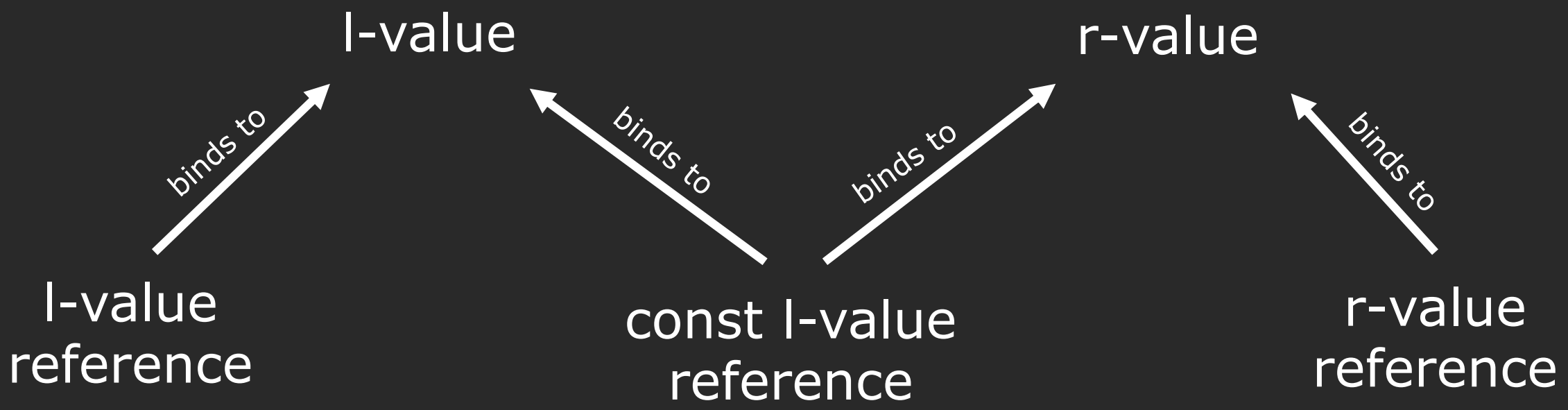
r-value

Recite this one more time.

An r-value reference is an alias to an r-value

BUT the r-value reference itself is an l-value

Everything in one picture.



move operations

This is pretty conceptually intense.

Please stop me at any time if you have questions!

Why r-values are key to move semantics.

An object that is an l-value is NOT disposable.

一次性

An object that is an r-value is disposable.

Why r-values are key to move semantics.

An object that is an l-value is NOT disposable, so you can copy* from, but definitely cannot move from.

An object that is an r-value is disposable.

*there exists some objects that can’t be copied (eg. stream)

Why r-values are key to move semantics.

An object that is an l-value is NOT disposable, so you can copy* from, but definitely cannot move from.

An object that is an r-value is disposable, so you can either copy* or move from.

Why?

*there exists some objects that can’t be copied (eg. stream)

Why r-values are key to move semantics.

An object that is an l-value is NOT disposable, so you can copy* from, but definitely cannot move from.

An object that is an r-value is disposable, so you can either copy* or move from.

Key insight: if an object might potentially be reused, you cannot steal its resources.

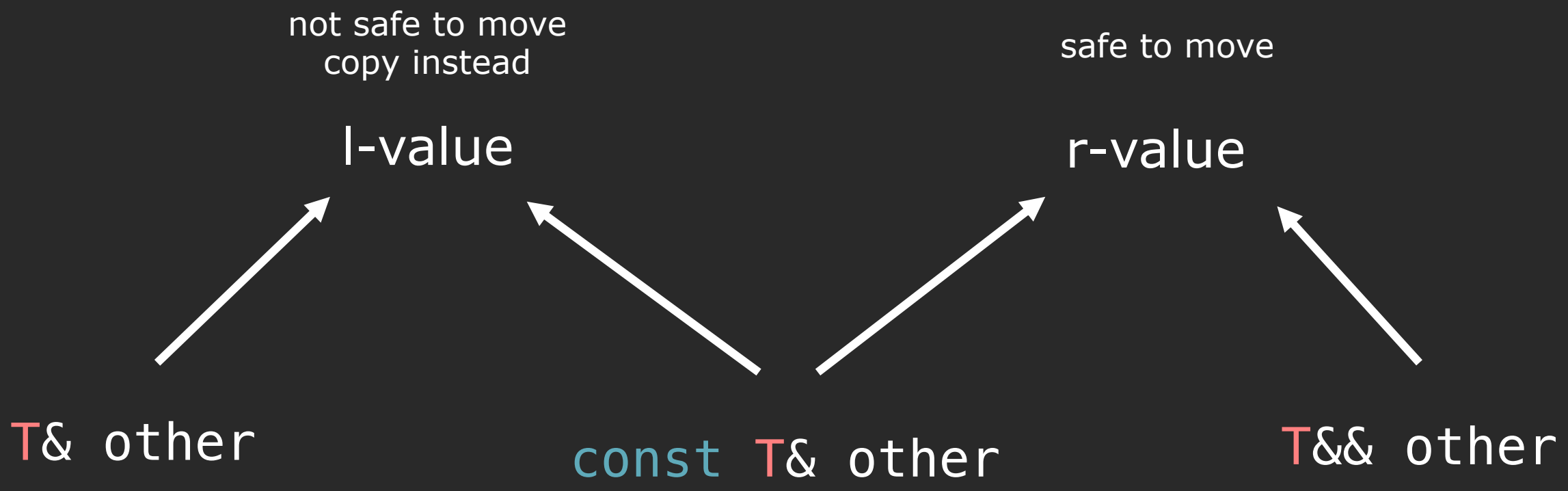
会重复使用，最好不要用move来steal，l-value用std::move强制转r-value一定要保证后面不会再用到该l-value所对应的对象

Welcome the two new special member functions!

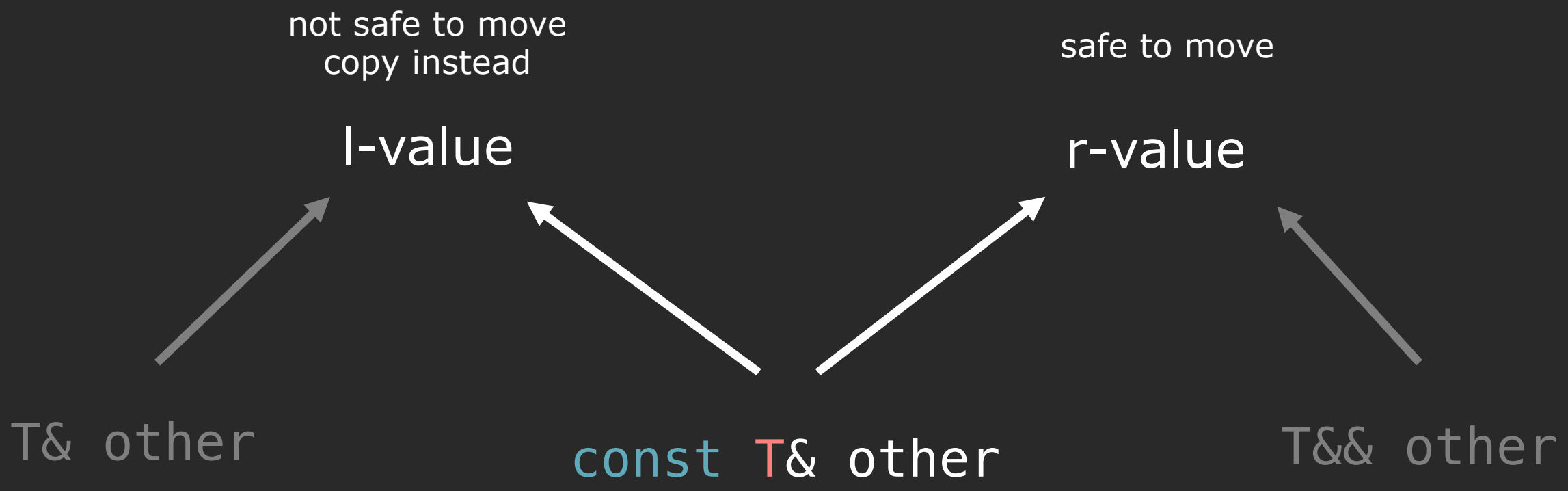
- Default constructor
- Copy constructor (create new from existing l-value)
- Copy assignment (overwrite existing from existing l-value)
- Destructor

- Move constructor (create new from existing r-value)
- Move assignment (overwrite existing from existing r-value)

Everything in one picture.

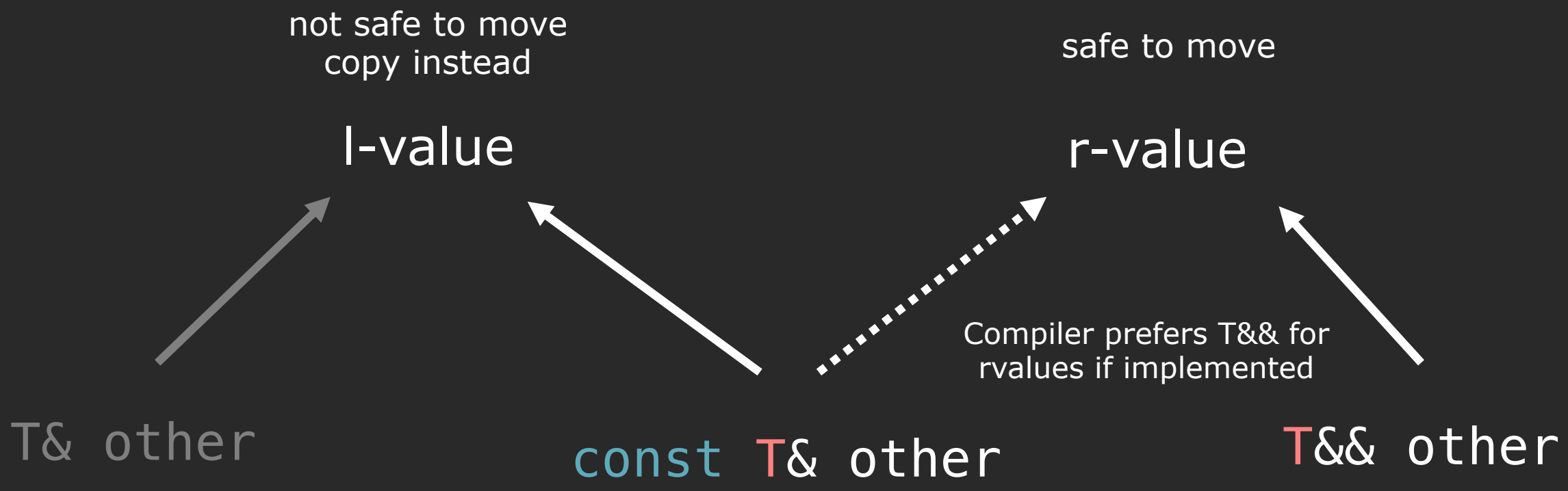


Everything in one picture.



Right now your copy constructor
works for both l and rvalues

Everything in one picture.



Now we will implement a constructor taking an r-value reference.

Function signatures of all our special member functions.

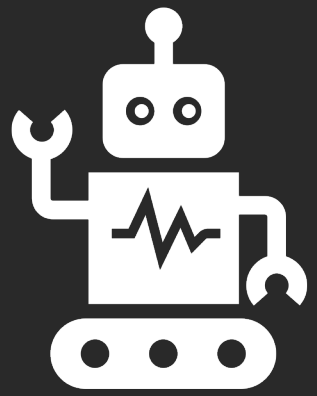
```
MyVector();
MyVector(const MyVector<T>& other);
MyVector<T>& operator=(const MyVector<T>& rhs);
~MyVector();

MyVector(MyVector<T>&& other);
MyVector<T>& operator=(MyVector<T>&& rhs);
```

normal

Key steps for a move constructor

一个字：偷steal：1. 先other的东西转移transfer过来；2. 再把other的东西扫荡干净（才算偷：我有了，你没有了）

- Transfer the contents of other to this.
  - Move instead of copy whenever possible!
- Leave other in an undetermined but valid state
  - Normally: set it to the default value of class



Example

Move constructor

Move constructor

(warning: this is not perfect…we'll come back to this!)

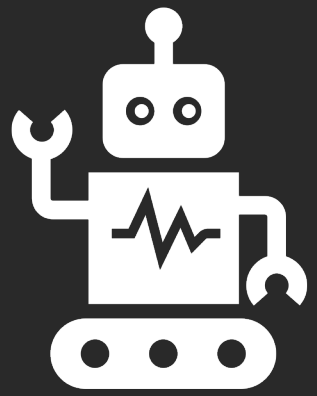
```
MyVector(MyVector<T>&& other) :
      elems(other.elems),
      logicalSize(other.logicalSize),
      allocatedSize(other.allocatedSize) {

      other.elems = nullptr;

}
```

Key steps for a move constructor

- Transfer the contents of other to this.
  - Move instead of copy whenever possible!
- Leave other in an undetermined but valid state
  - Normally: set it to the default value of class

Key steps for a move assignment

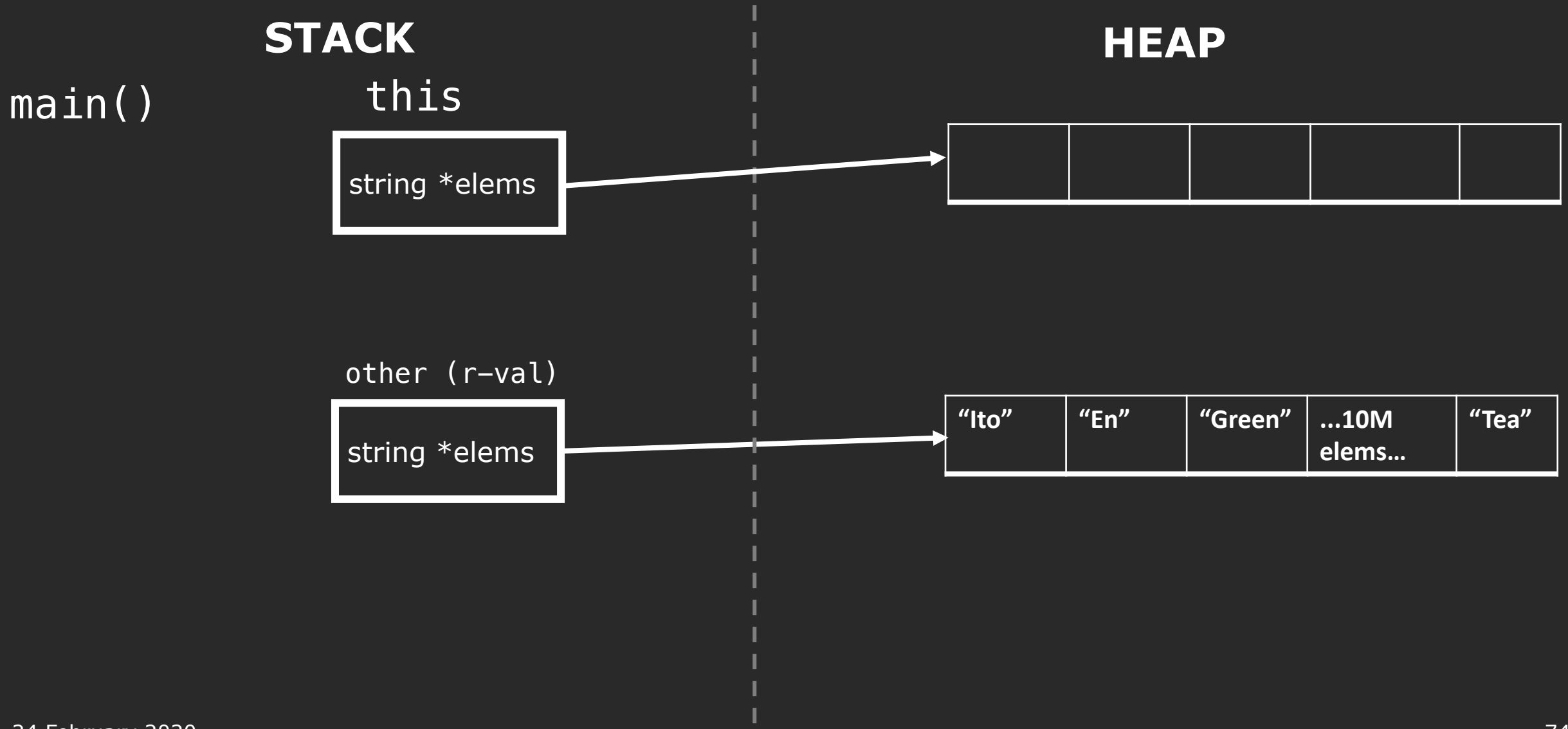
- Check self-assignment.
- Free up resources held by `this`.
- Transfer the contents of other to this.
  - Move instead of copy whenever possible!
- Leave other in an undetermined but valid state
  - Normally: set it to the default value of class



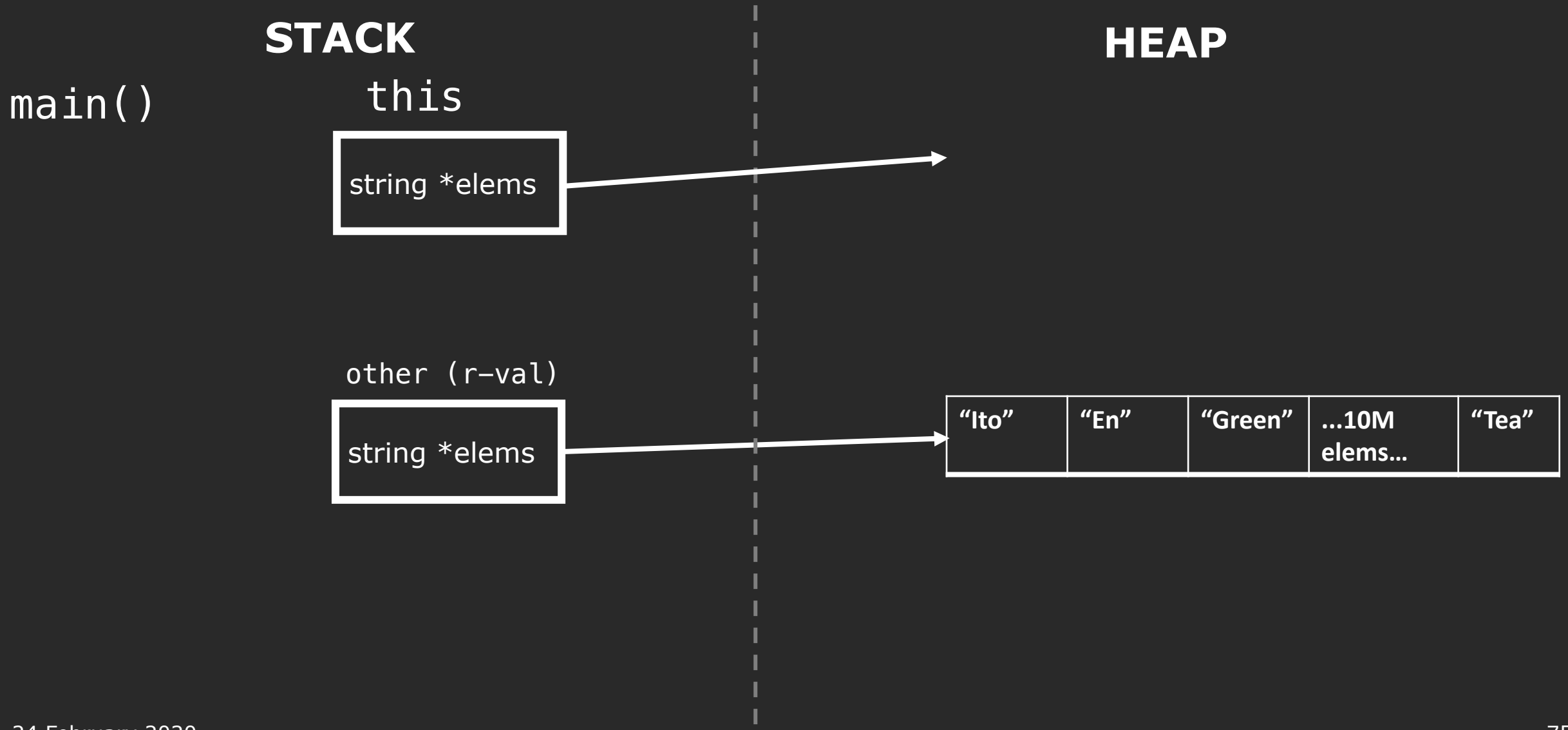
Example

Move assignment

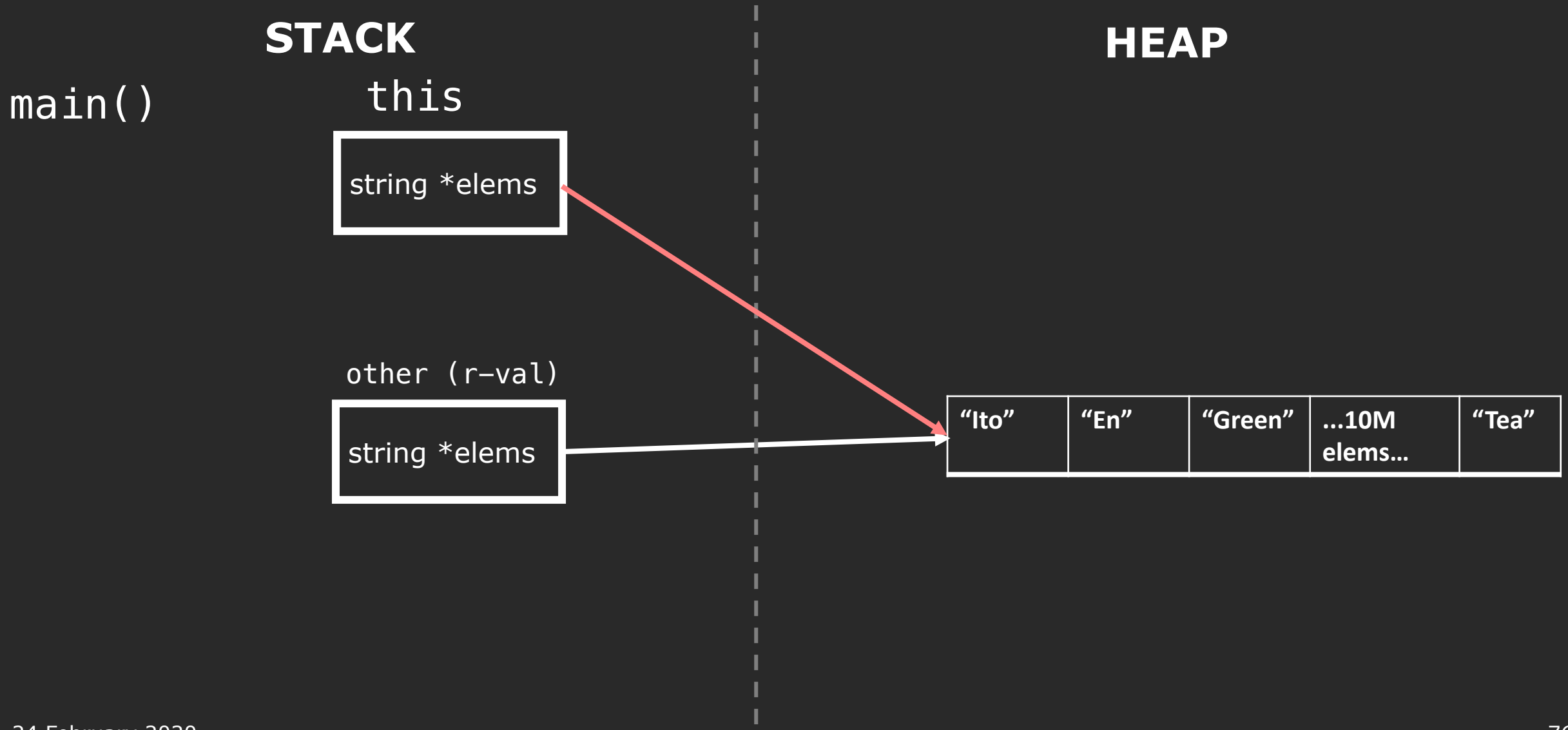
Simulation of move assignment.



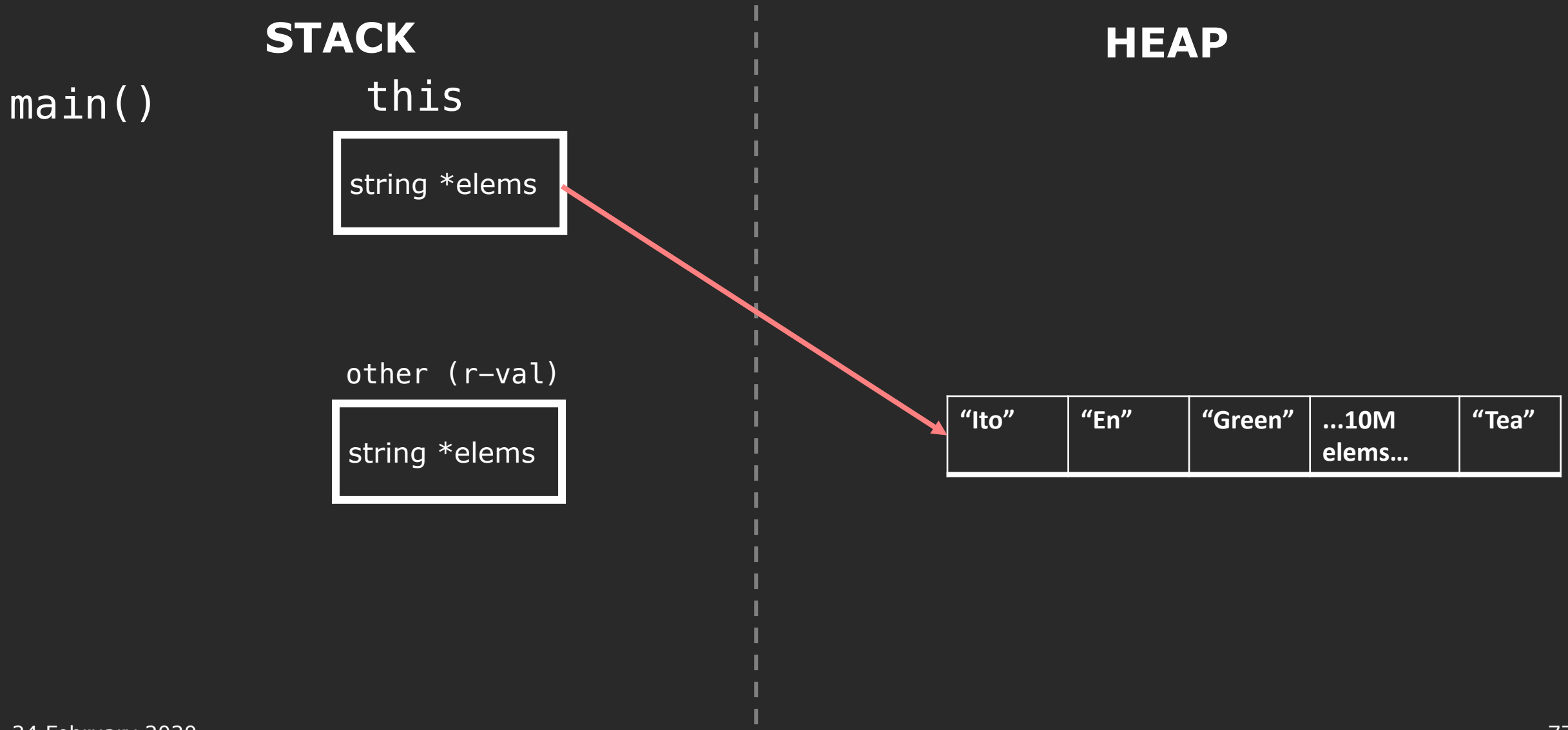
Simulation of move assignment.



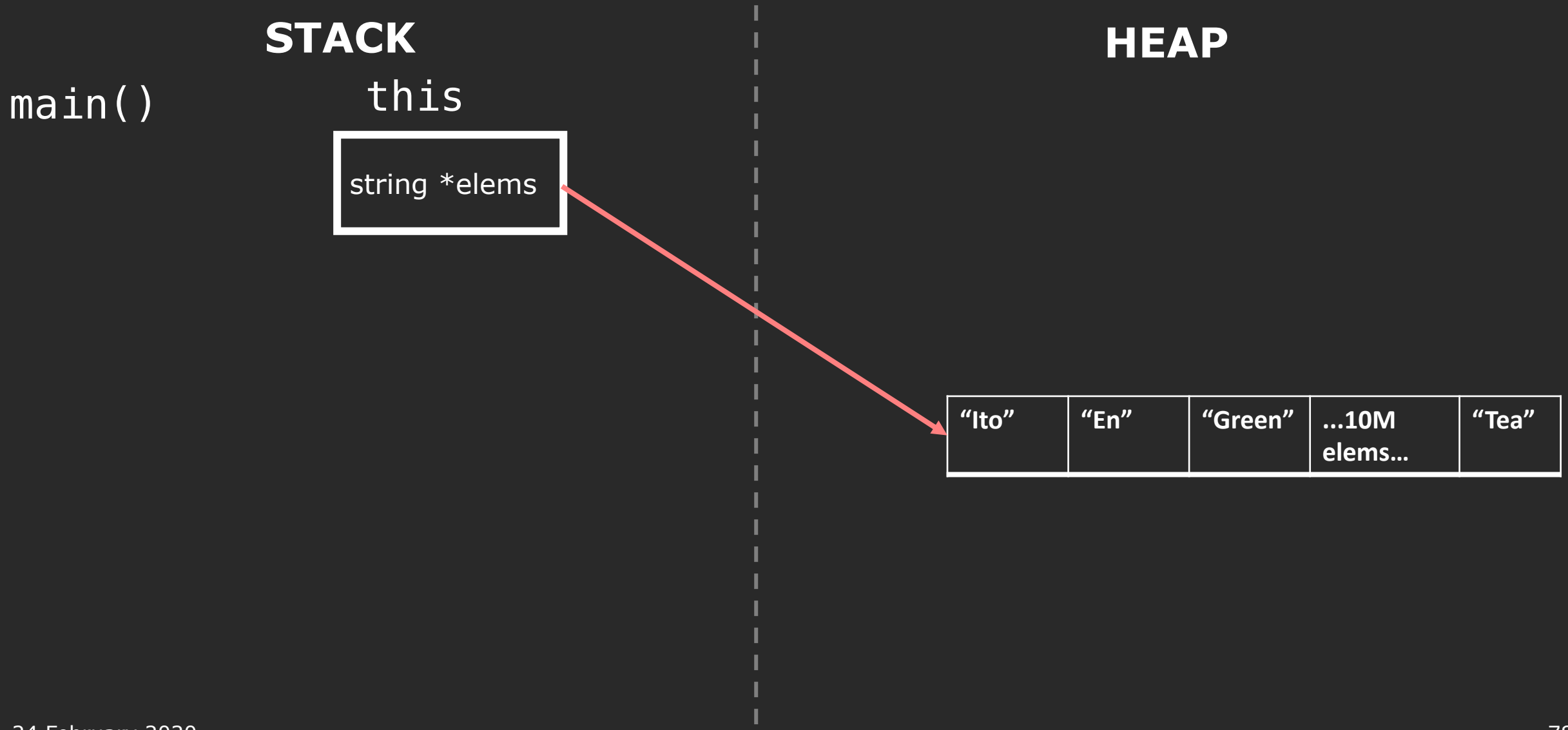
Simulation of move assignment.



Simulation of move assignment.



Simulation of move assignment.



Move assignment

(warning: this is not perfect…we'll come back to this!)

```
MyVector<T>& operator=(MyVector<T>&& rhs) {
  if (this == &rhs) {
     delete[] elems;
     allocatedSize = rhs.allocatedSize;
     logicalSize = rhs.logicalSize;
     elems = rhs.elems;
     rhs.elems = nullptr;
  }
  return *this;
}
```

This is a small problem in our code.

Did we actually move
all the members?

Consider this other example…from CS 106B!

```
class RandomBag {
public:
     RandomBag();

     void add(int value);
     int removeRandom();

private:
     vector<int> elems;
}
```

Move or copy assignment?

```
RandomBag& RandomBag::operator=(RandomBag&& rhs) {
    if (this != &rhs) {
        // no freeing needed

        elems = rhs.elems;

    }
    return *this;
}
```

l–value;

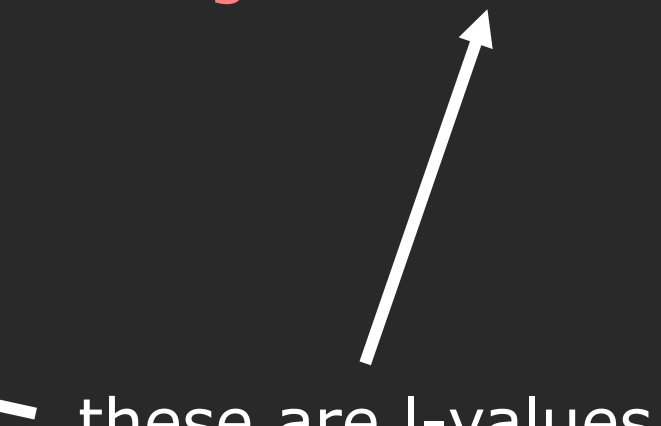
Recite this one more time.

An r-value reference is an alias to an r-value

BUT the r-value reference itself is an l-value

This is copy assignment! How inefficient!

```
RandomBag& RandomBag::operator=(RandomBag&& rhs) {
    if (this != &rhs) {
        // no freeing needed

        elems = rhs.elems;



    }
    return *this;
}
```



This is copy assignment! How inefficient!

```
RandomBag& RandomBag::operator=(RandomBag&& rhs) {
    if (this != &rhs) {
        // no freeing needed

        elems = rhs.elems;


    }
    return *this;
}
```

can we force this to be an r-value?

rhs is going to be in undetermined state anyways

This is copy assignment! How inefficient!

```
RandomBag& RandomBag::operator=(RandomBag&& rhs) {
    if (this != &rhs) {
        // no freeing needed

        elems = static_cast<RandomBag&&>(rhs.elems);



    }
    return *this;
}
```

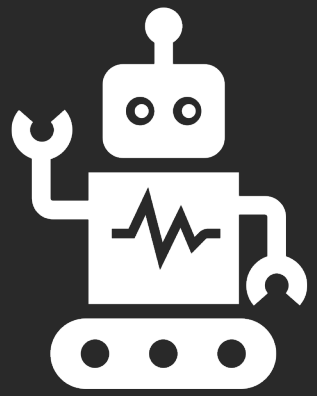
Now it is!

This is copy assignment! How inefficient!

```
RandomBag& RandomBag::operator=(RandomBag&& rhs) {
    if (this != &rhs) {
        // no freeing needed

        elems = std::move(rhs.elems);


    }
    return *this;
}
```

returns a r–value of the element

Templatized version
in the standard library



Example

std::move-ing all the members

Move constructor
(now it’s perfect and idiomatic)

```
MyVector<T>(MyVector<T>&& other) :
      elems(std::move(other.elems)),
      logicalSize(std::move(other.logicalSize)),
      allocatedSize(std::move(other.allocatedSize)) {

      other.elems = nullptr;

}
```

Move assignment
(now it’s perfect and idiomatic)

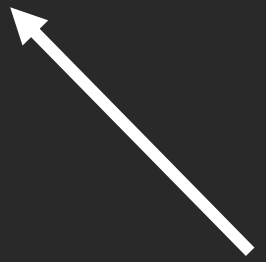
```
MyVector<T>& operator=(MyVector<T>&& rhs) {
  if (this == &rhs) {
     delete[] elems;
     allocatedSize = std::move(rhs.allocatedSize);
     logicalSize = std::move(rhs.logicalSize);
     elems = std::move(rhs.elems);
     rhs.elems = nullptr;
  }
  return *this;
}
```

std::move

Unconditional cast to an r-value.

std::move

Unconditional cast to an r-value.

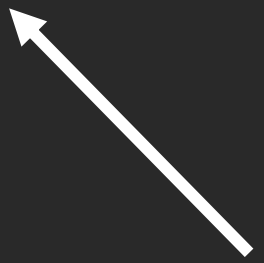


Yes...there is a conditional cast that is even more poorly named!

std::move

Poorly named things in C++, part 5

```
std::move(rhs.elems);
```

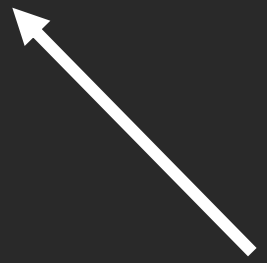


std::move itself doesn’t move anything

std::move

Poorly named things in C++, part 5

```
elems = std::move(rhs.elems);
```



The real move happens during assignment

std::move

Honestly a way better name…

```
elems = std::rvalue_cast(rhs.elems);
```

std::move summary

- When declaring move operations, make sure to std::move all members.
- std::move doesn't really move anything
- Call std::move to force anything to an r-value.

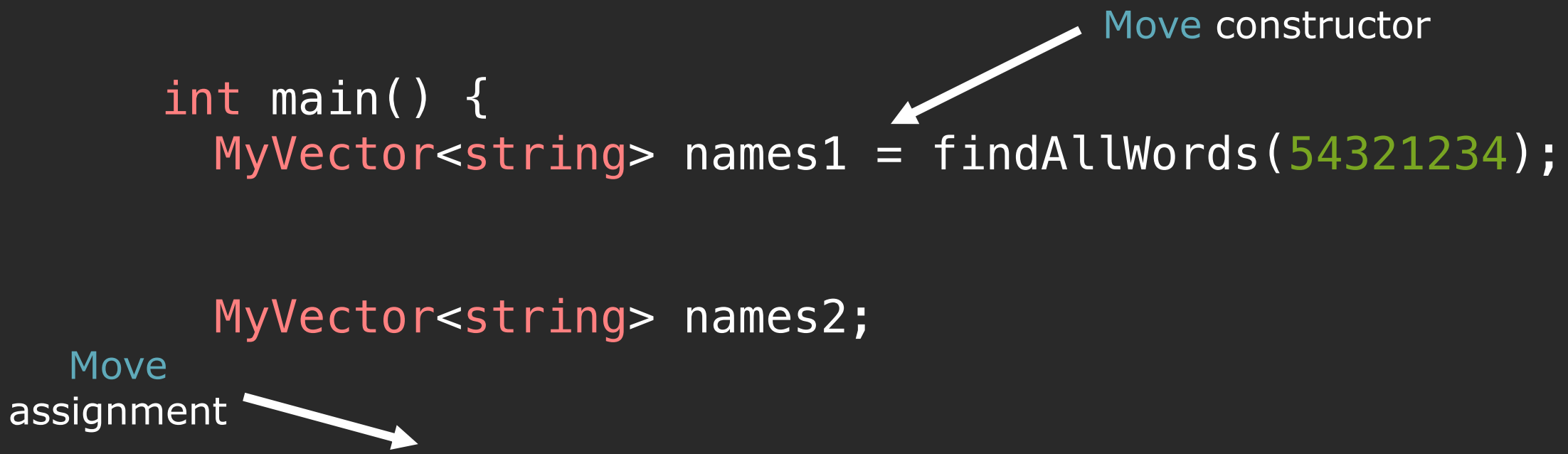
What did that fix?

```
int main() {
  MyVector<string> names1 = findAllWords(54321234);

  MyVector<string> names2;

  names2 = findAllWords(54321234);
}
```

r-value

r-value

What did that fix?



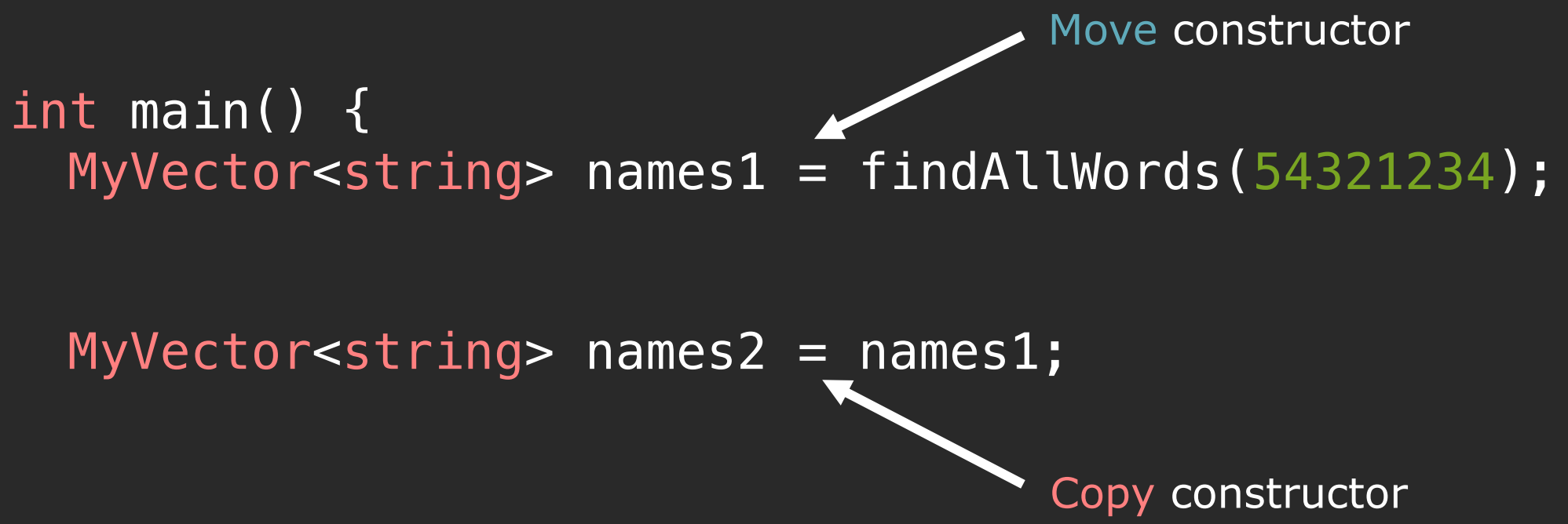
What did that fix?

```
int main() {
  MyVector<string> names1 = findAllWords(54321234);

  MyVector<string> names2 = names1;

  names1.push_back("Everything is fine!");
}
```

r-value

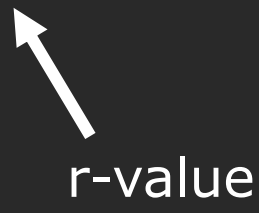
l-value

What did that fix?



What did that fix?

```
int main() {
  MyVector<string> names1 = findAllWords(54321234);

  MyVector<string> names2 = std::move(names1);

  // I promise to not use names1 before reassigning it
}
```



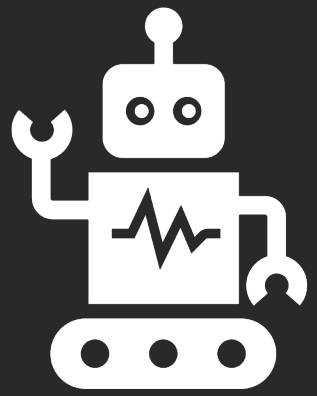
What did that fix?

```
int main() {
  MyVector<string> names1 = findAllWords(54321234);

  MyVector<string> names2 = std::move(names1);

  // I promise to not use names1 before reassigning it
}
```

Move constructor

Final quiz: how many times is each special member function called (with copy elision and move semantics)?

```
int main() {
  MyVector<string> names1 = findAllWords(54321234);
  MyVector<string> names2;
  names2 = findAllWords(54321234);
  cout << “done!” << endl;
}

MyVector<string> findAllWords(size_t size) {
  MyVector<string> names(size, “Ito”);
  return names;
}
```

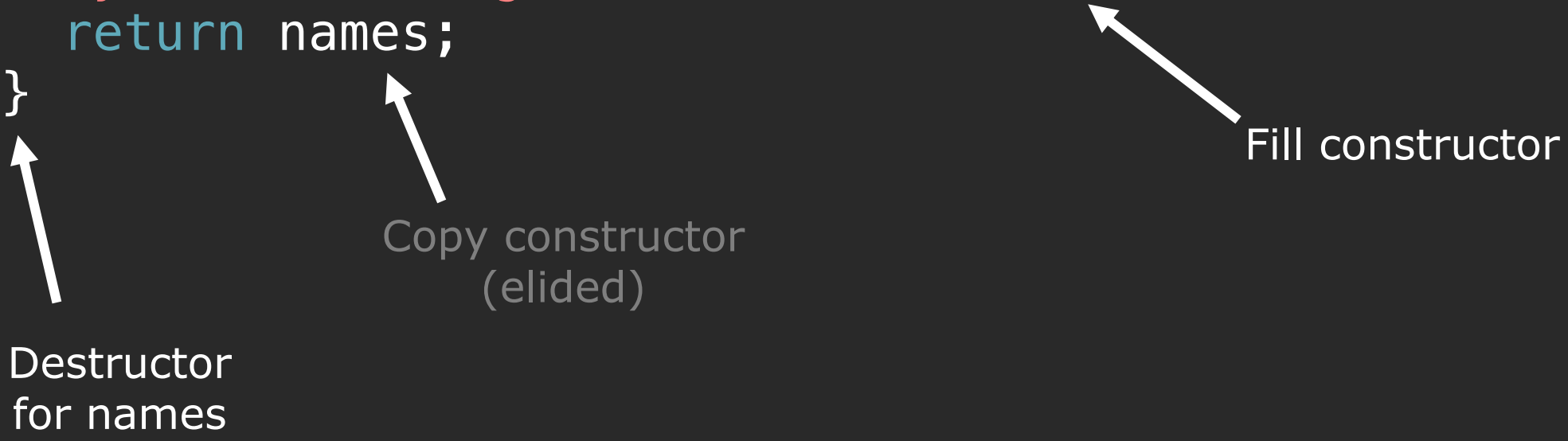


Demo

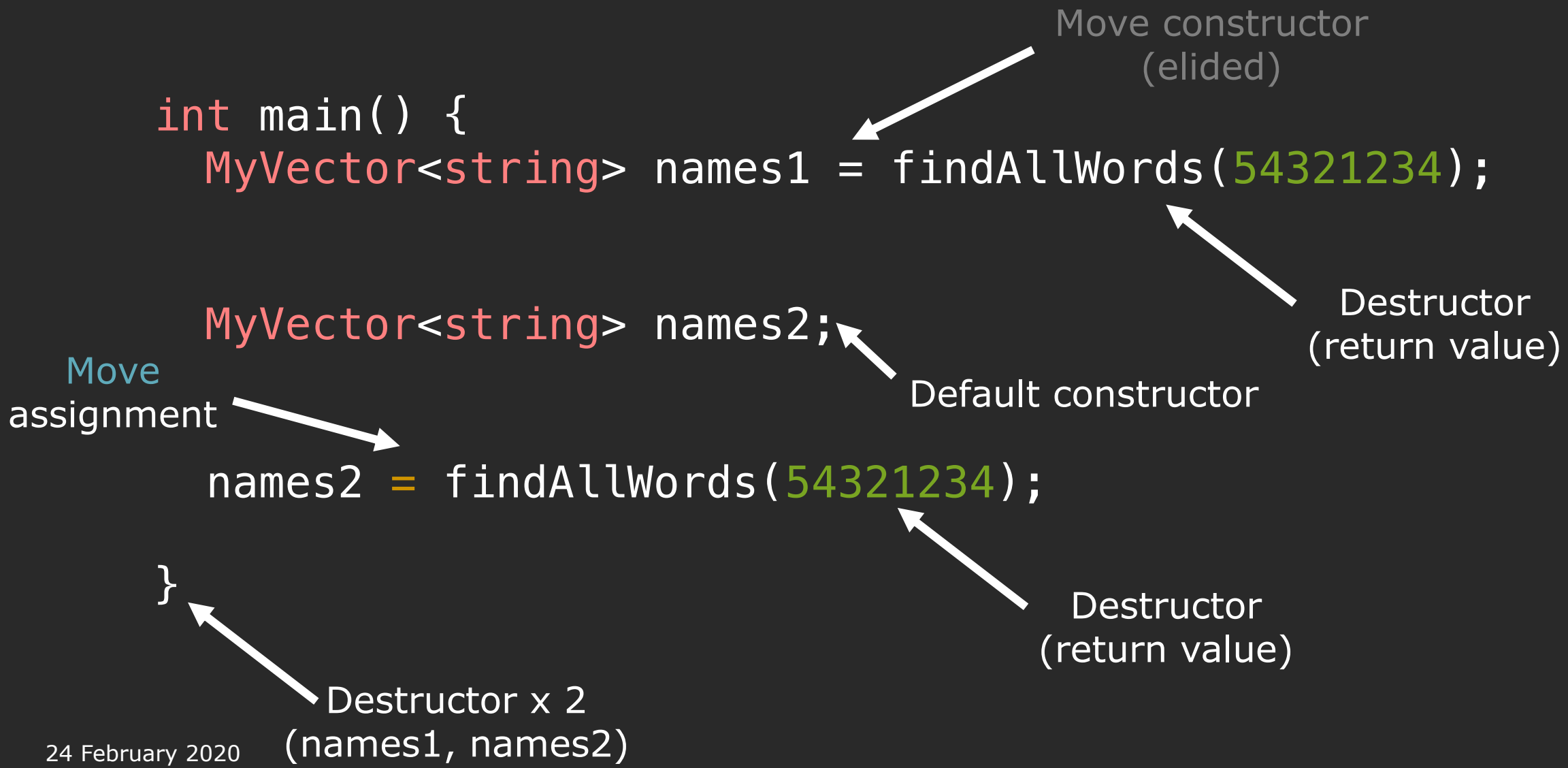
Printing and timing all calls to special member functions

With copy elision and move semantics.

```
MyVector<string> findAllWords(size_t size) {
  MyVector<string> names(size, “Ito”);
  return names;
}
```



With copy elision and move semantics.



Counts with copy elision.

`findAllWords`

- Fill constructor x 1
- Copy constructor x 1
- Destructor x 1

Counts with copy elision.

`main`

- Move constructor x 1
- Default constructor x 1
- Move assignment x 1
- `findAllWords` x 2
  - Fill constructor x 1
  - Copy constructor x 1
  - Destructor x 1
- Destructor x 1

Counts with copy elision.

`main`

- Move assignment x 1
- Copy constructor x 3
- Default constructor x 1
- Destructor x 3
- Fill constructor x 2

applications

Our push_back function before.

反正element push完之后就不会用了，干脆用move转进去

```
template <typename T>
void MyVector<T>::push_back(const T& element) {
     if (size() == allocatedSize) resize(…);
     elems[logicalSize++] = element;
}
```

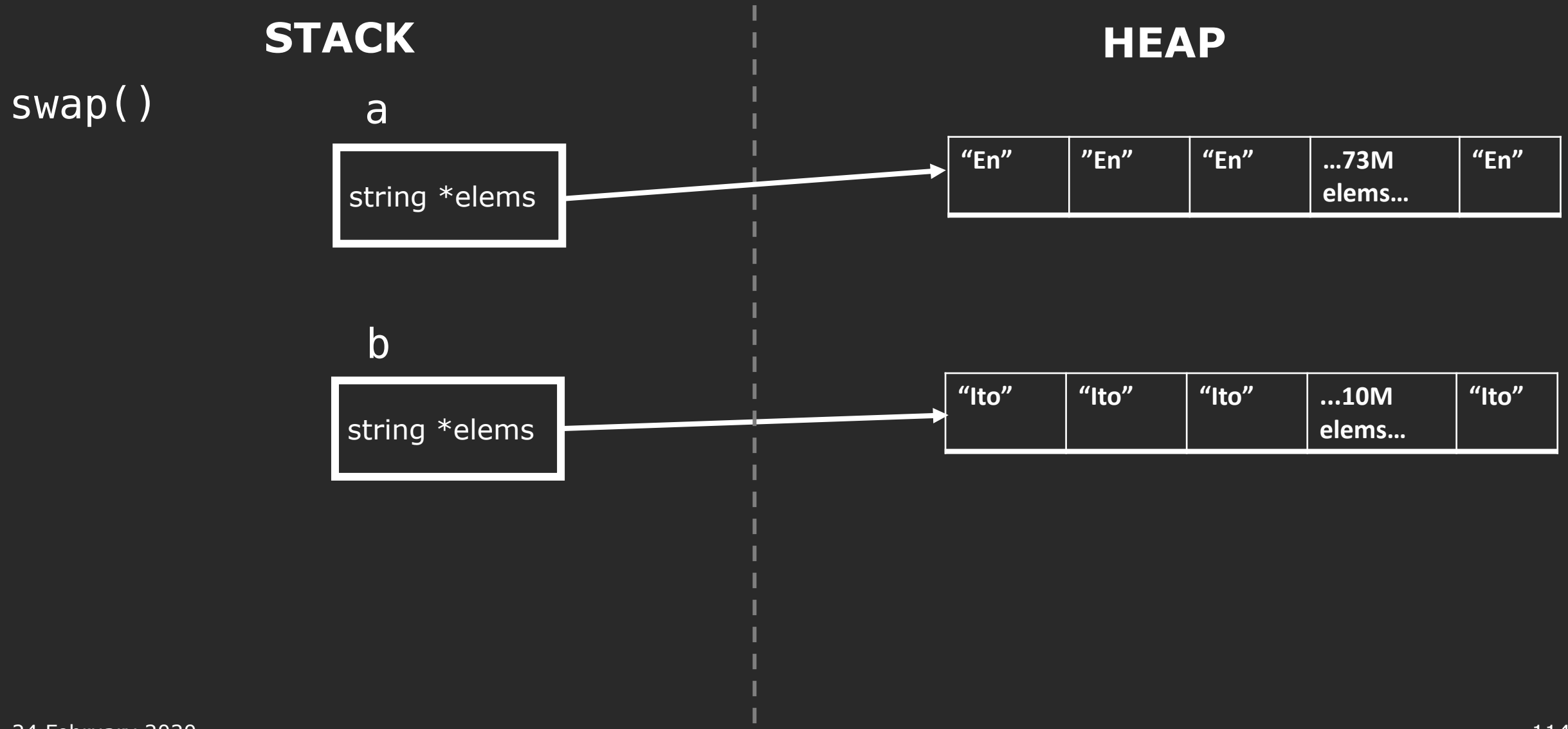
Our push_back function now supports an r-value overload.

```
template <typename T>
void MyVector<T>::push_back(T&& element) {
     if (size() == allocatedSize) resize(…);
     elems[logicalSize++] = std::move(element);
}
```

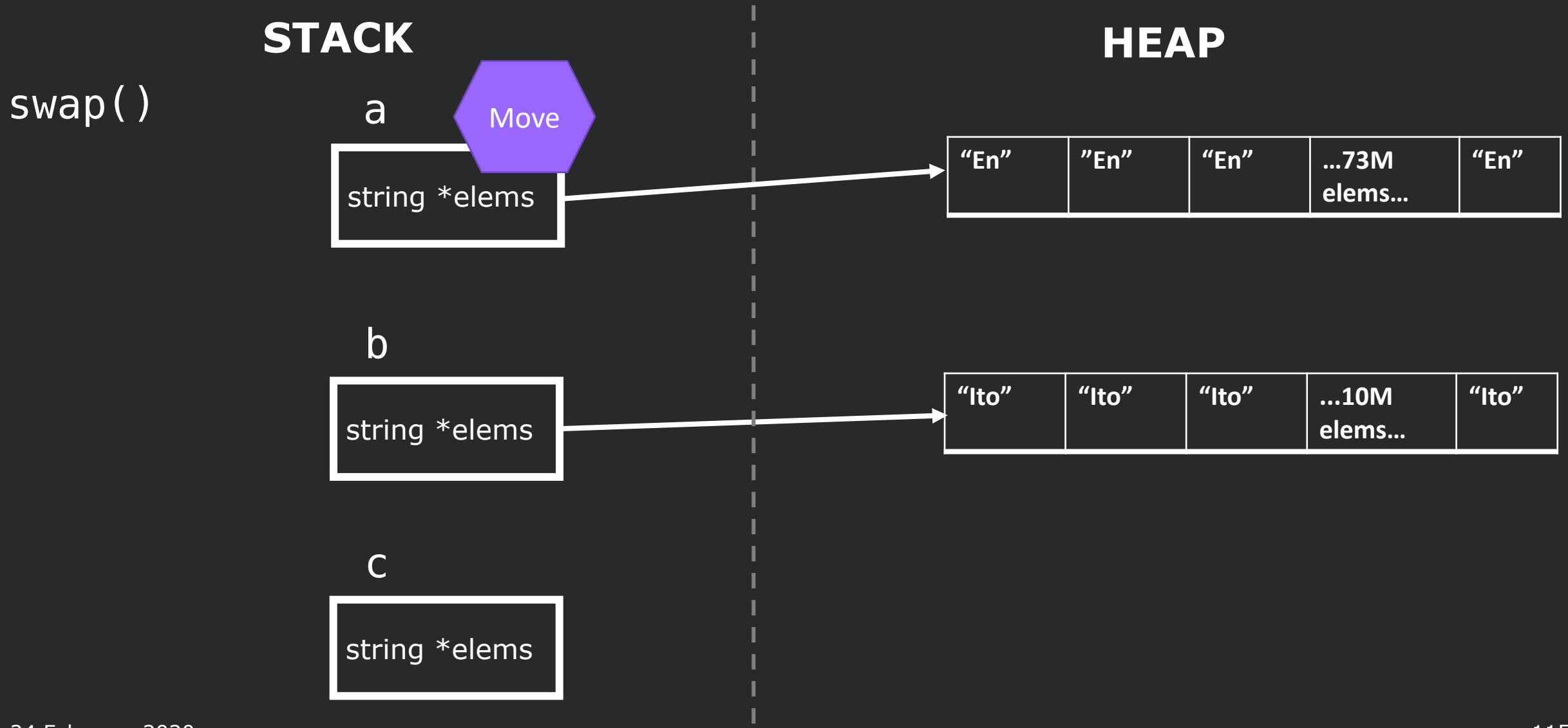
Your task: write a generic swap function.

```
int main() {
  vector<string> v1(“En”, 73837463);
  vector<string> v2(“Ito”, 10000000);
  swap(v1, v2);


  Patient patient1{“Anna”, 2};
  Patient patient2{“Avery”, 3};
  swap(patient1, patient2);
}
```

Simulation of move assignment.

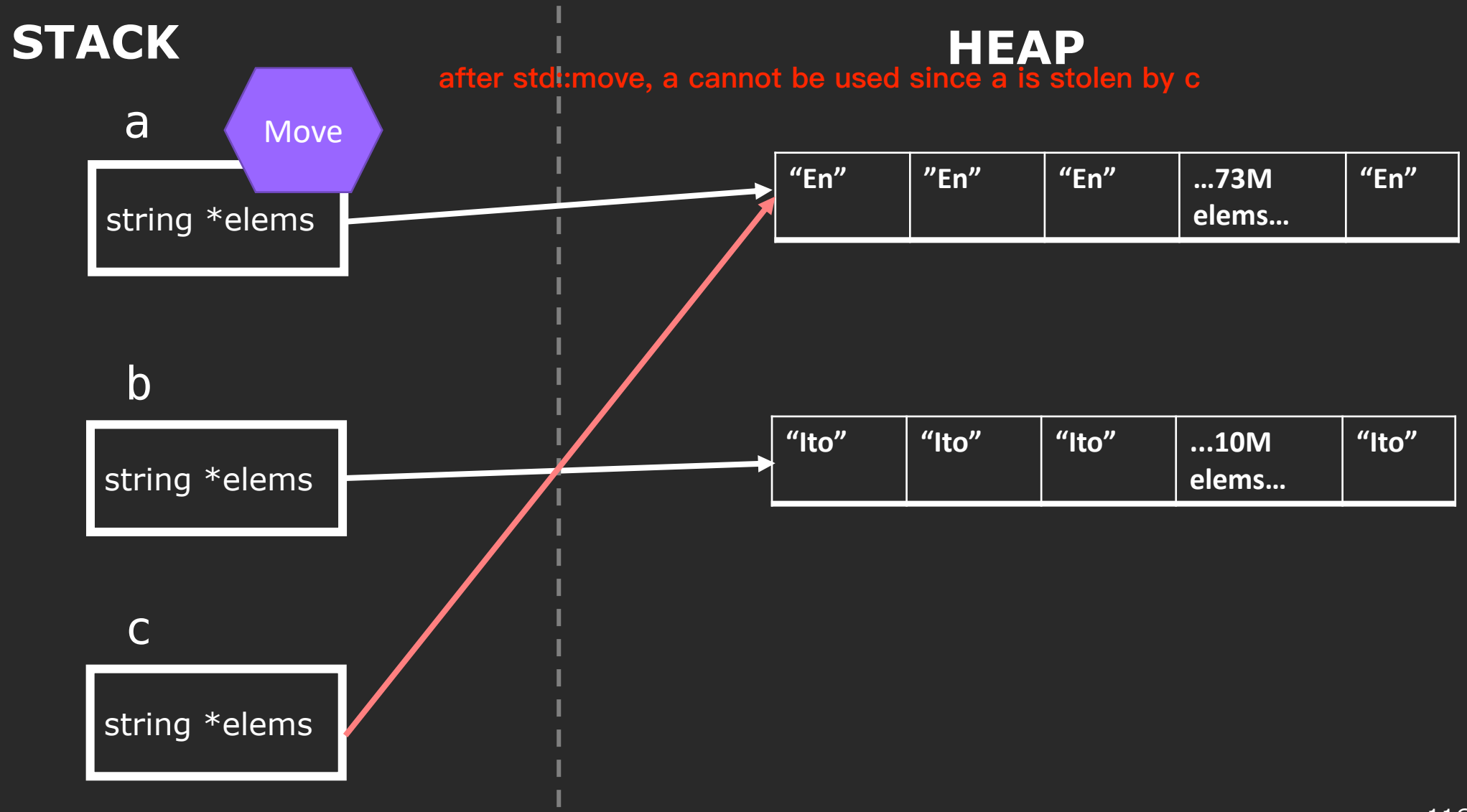


Simulation of move assignment.

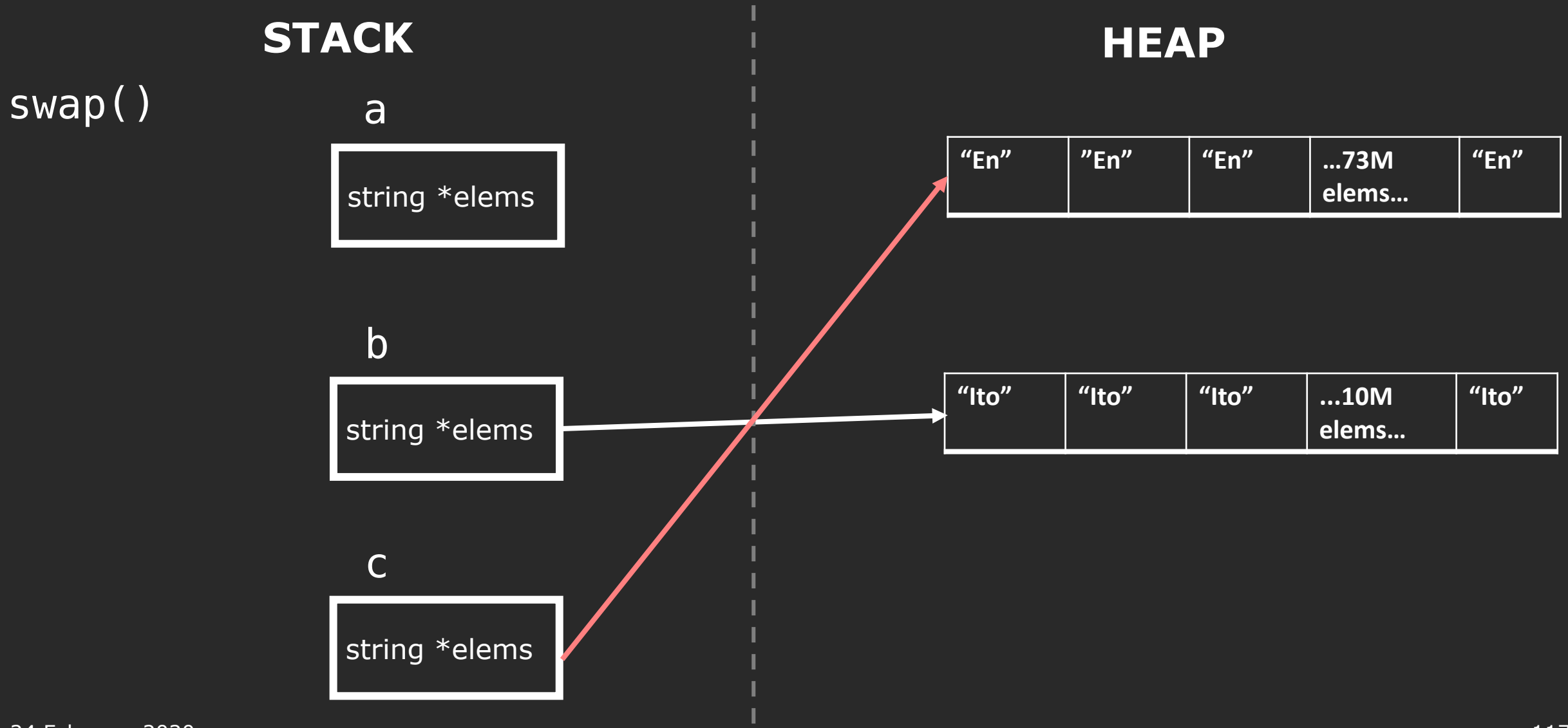


Simulation of move assignment.

swap()

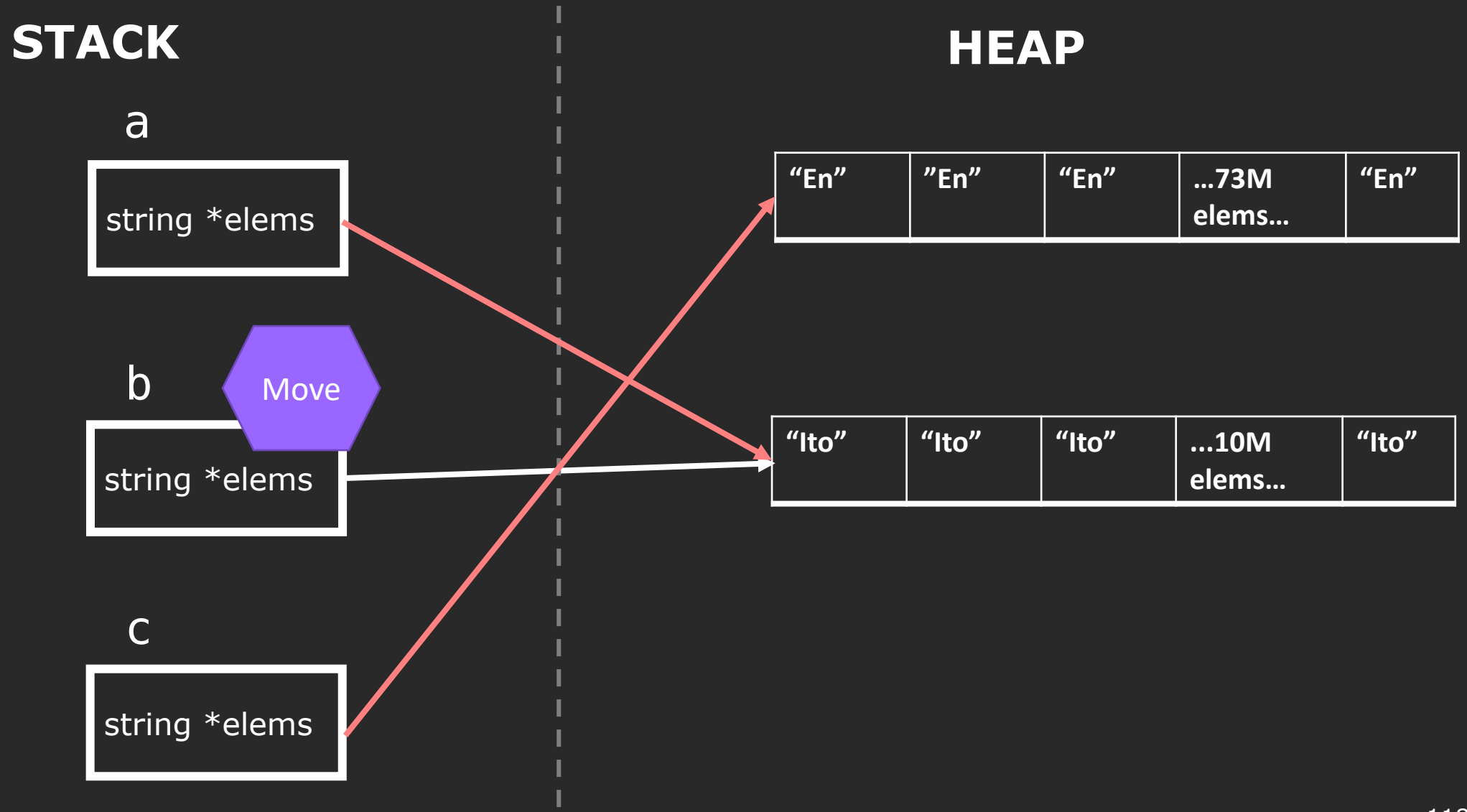


Simulation of move assignment.

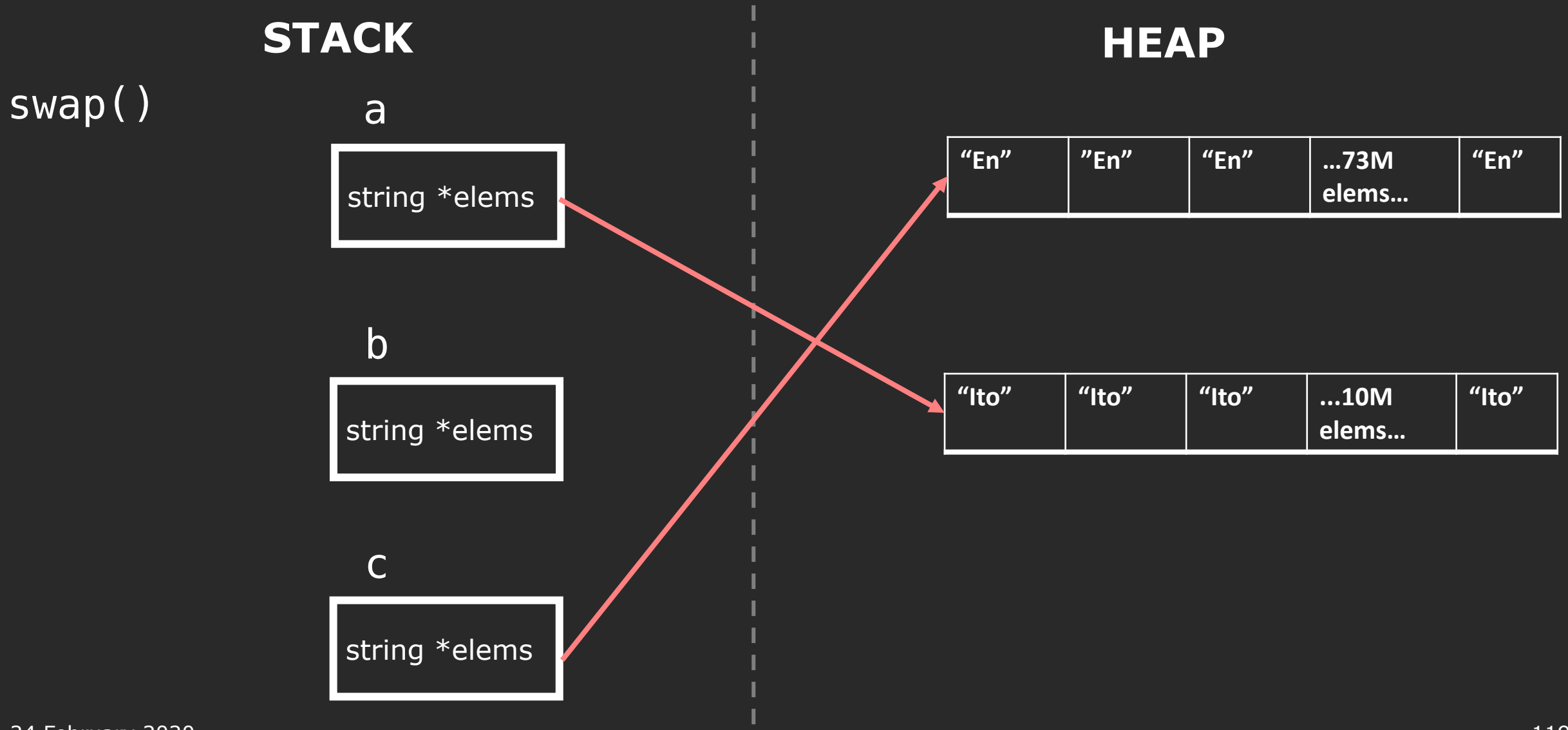


Simulation of move assignment.

swap()

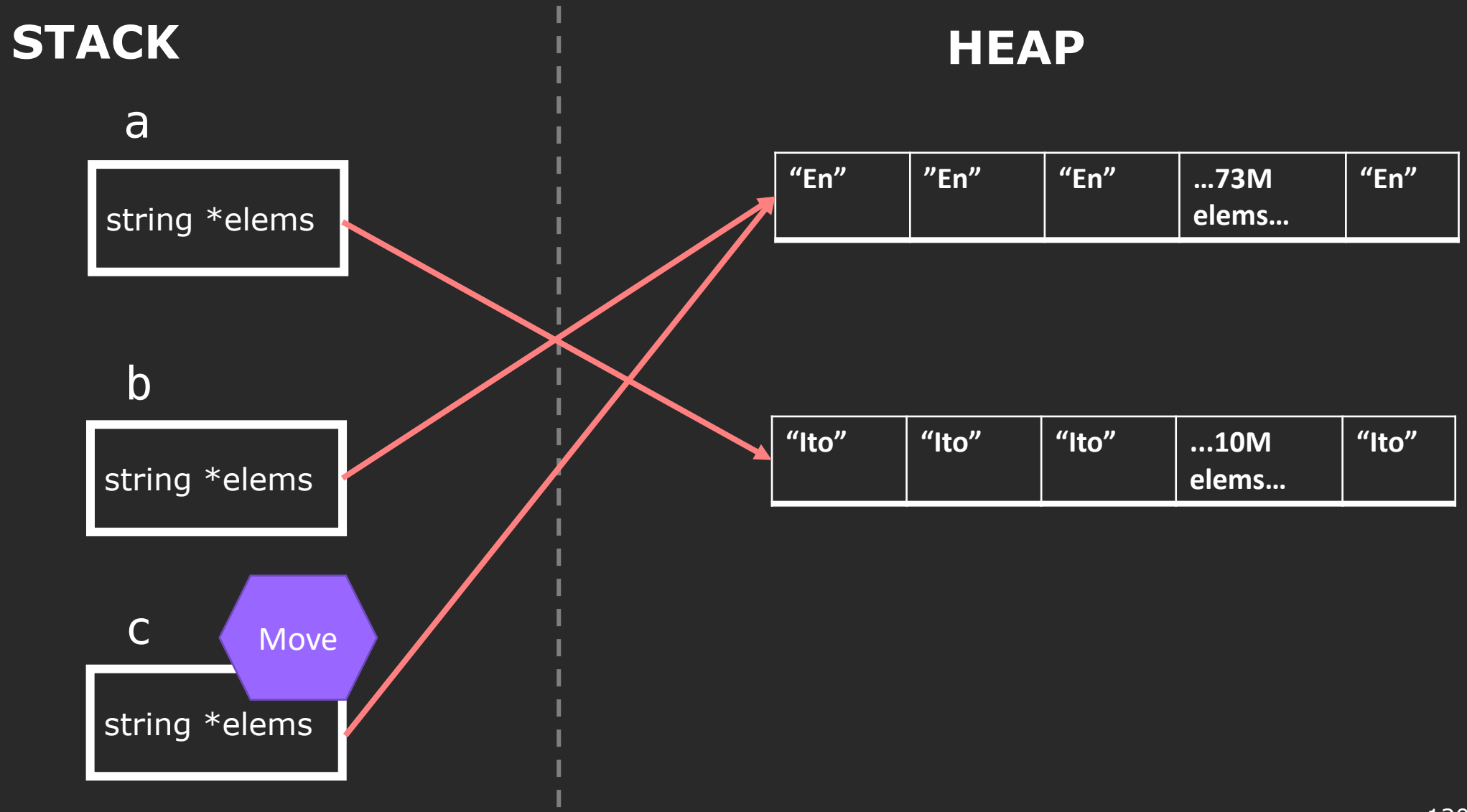


Simulation of move assignment.

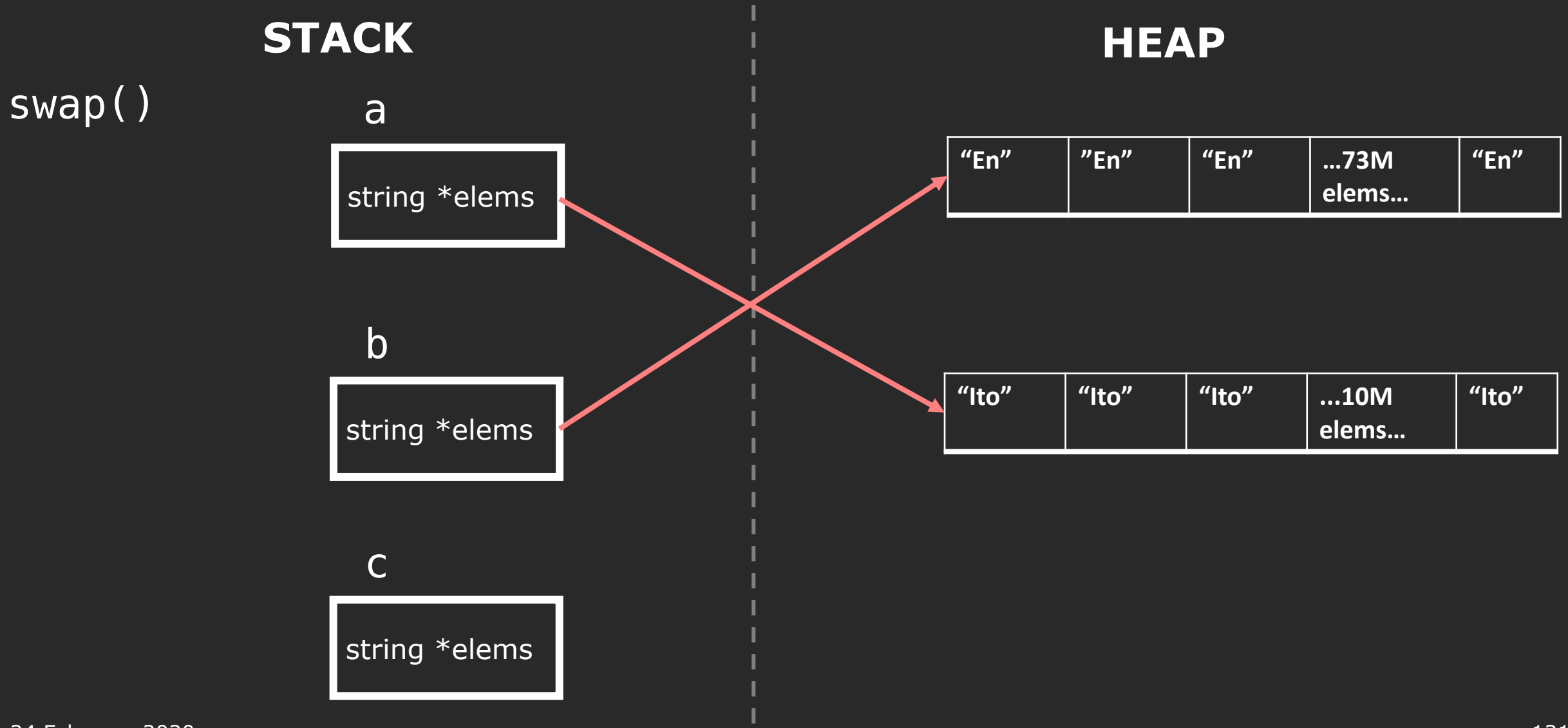


Simulation of move assignment.

swap()

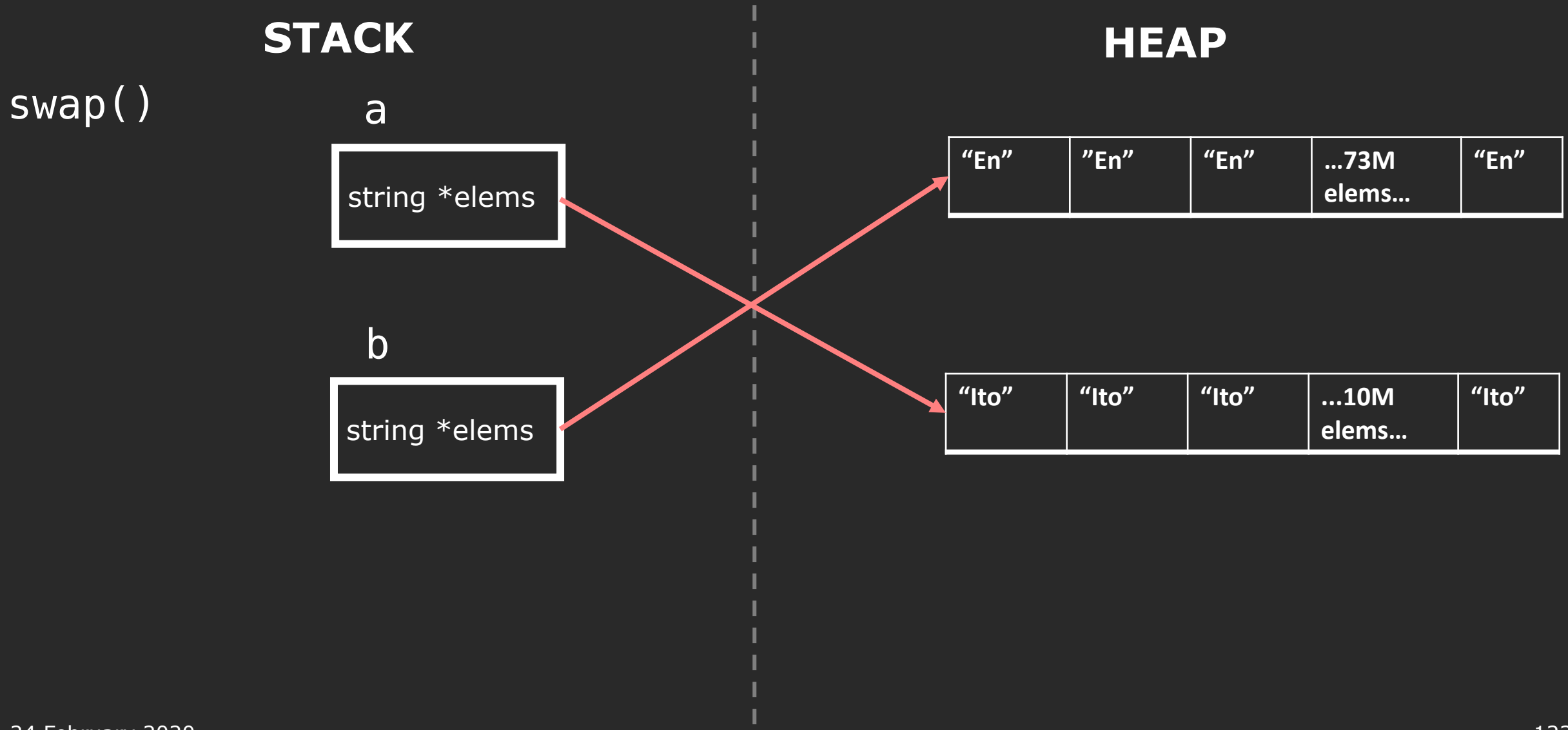


Simulation of move assignment.



Simulation of move assignment.

swap()

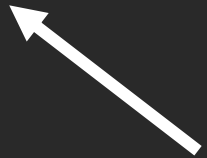


Non-idiomatic use (do not use!)
std::moving the return value

```
MyVector<string>&& findAllWords(size_t size) {
  MyVector<string> names(size, “Ito”);
  return std::move(names);
}
```

Destructor
for names

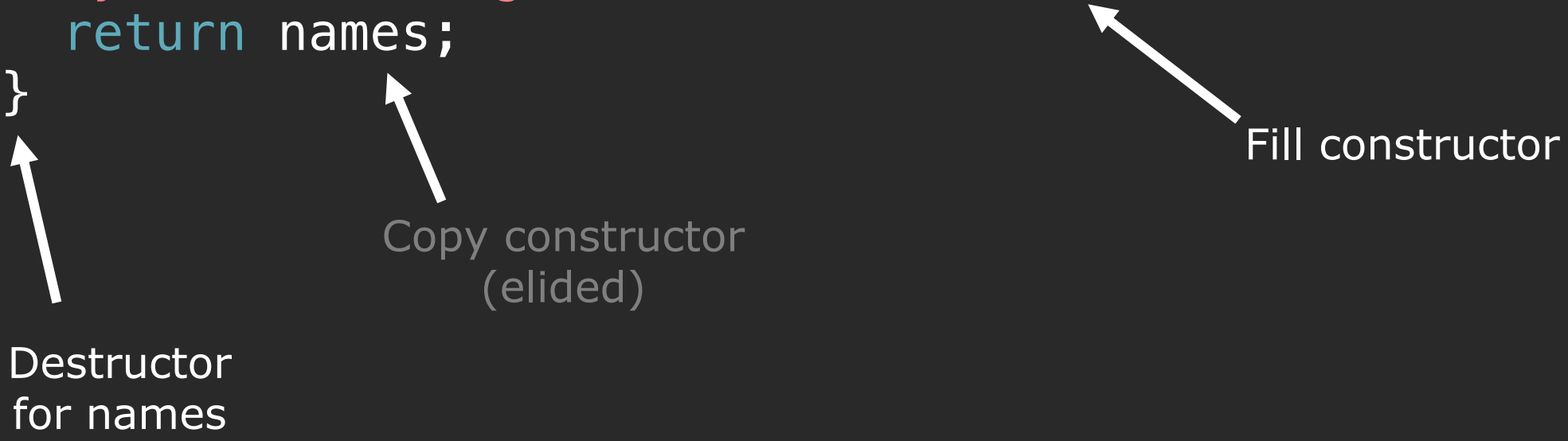
Move constructor



Fill constructor

The compiler is great at optimizing return values. Don’t interfere with it.

```
MyVector<string> findAllWords(size_t size) {
  MyVector<string> names(size, “Ito”);
  return names;
}
```



Your task: write a generic swap function.

```
template <typename T>
void swap(T& a, T& b) noexcept {
    T c(std::move(a)); // move constructor
    a = std::move(b);  // move assignment
    b = std::move(c);  // move assignment
}

// by the way, this is std::swap
```

Rule of Five

If you explicitly define (or delete)
a copy constructor, copy assignment,
move constructor, move assignment, or destructor,
you should define (or delete) all five.

The fact that you defined one of these means
one of your members has ownership issues
that need to be resolved.

Rule of Zero

If the default operations work, then don’t define your own custom ones.

You can default these operations explicitly!

```
class RandomBag {
public:
     RandomBag();
     RandomBag(const RandomBag& other) = default;
     RandomBag(RandomBag&& other) = default;
     RandomBag& operator=(const RandomBag& rhs) = default;
     RandomBag& operator=(RandomBag&& rhs) = default;
     void add(int value);
     int removeRandom();
private:
     vector<int> elems;
}
```

Notice for RandomBag: the default operations will move all elements – avoids the bugs we made before!

```
class RandomBag {
public:
     RandomBag();
     RandomBag(const RandomBag& other) = default;
     RandomBag(RandomBag&& other) = default;
     RandomBag& operator=(const RandomBag& rhs) = default;
     RandomBag& operator=(RandomBag&& rhs) = default;
     void add(int value);
     int removeRandom();
private:
     vector<int> elems;
}
```

Rule of Zero

If the default operations work, then don’t define your own custom ones.

You are more likely to make a mistake if you define it when you don’t need to.

Be explicit – don't trust the compiler. Default or delete them yourself.

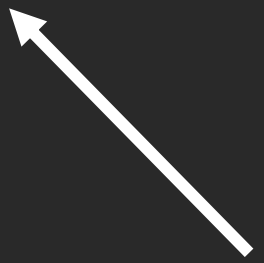
compiler implicitly declares

user declares	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

std::forward

Conditional cast to an r-value.

```
std::forward(rhs.elems);
```



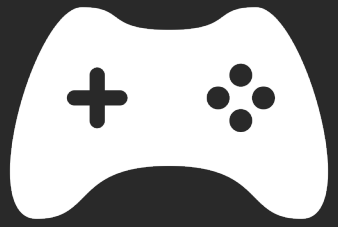
std::move itself doesn’t move anything

perfect forwarding problem and emplace_back

Never mind, there’s no way we are getting this far.

Ask me or ask on Piazza if you want to learn more!

This is also a final lecture topic if you want to learn!



Next time

RAII
(the single most important C++ idiom)