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

```
int main() {
  vector<string> words1 = findAllWords(54321234);
  vector<string> words2;
  words2 = findAllWords(54321234);
  cout << “done!” << endl;
}

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

Run the starter code to find the correct answer!

Move Semantics

recap

New Overarching Topic!

What do we mean by a resource?

- Memory (handled by pointers)
- Files (handled by streams)
- CPU time (handled by locks)
- Networks (handled by sockets)

We will focus on the most important: memory, and a bit into multithreading at the very end.

The principles we'll see about memory can be applied to other resources.

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.

Member Initialization List

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

- Faster. Why construct, then immediately reassign?
- Members might be a non-assignable type.

```
template <typename T>
vector<T>::vector<T>() :
     _size(0), _capacity(kInitialSize),
  _elems(new T[kInitialSize]) { }
```

What's wrong with the default generated copy constructor?

```
template <typename T>
vector::vector<T>(const vector<T>& other) :
     _size(other._size),
     _capacity(other._capacity),
     _elems(other._elems) { }
```

Copying a pointer != copying the thing it's pointing to.

Copy constructor: the correct implementation

Instead of constructing _elems then reassigning, directly construct.

```
template <typename T>
vector::vector<T>(const vector<T>& other) :
      _size(other._size),
      _capacity(other._capacity),
      _elems(new T[other._capacity]) {

      std::copy(other._elems,
                other._elems + other._size, _elems);
}
```

Copy it ourselves!

Copy assignment: the correct implementation

```
template <typename T>
vector<T>& vector<T>::operator=(const vector<T>& other)
     if (&other == this) return *this;
     _size = other._size;
     _capacity = other._capacity;
     T* new_elems = new T[other._capacity];

     std::copy(other._elems,
               other._elems + other._size, new_elems);
     delete [] _elems;
     _elems = new_elems;
     return *this;
}
```

Avoid self assignment!

The copy operations must perform the following tasks.

Copy Constructor

- Use initializer list to copy members where copy constructor does the correct thing.
  - int, other objects, etc.

- Manually copy all members where assignment does not work.
  - pointers to heap memory
  - non-copyable things

Copy Assignment

- Clean up any resources in the existing object about to be overwritten.

- Copy members using direct assignment when assignment works.

- Manually copy members where assignment does not work.

You can prevent copies from being made by explicitly deleting these operations.

```
class PasswordManager {
public:
  PasswordManager();
  ~PasswordManager();
  // other methods
  PasswordManager(const PasswordManager& rhs) = delete;
  PasswordManager& operator=(const PasswordManager& rhs) = delete;

private:
   // other stuff
}
```

You can ask the compiler to generate default versions of special member functions.

```
class StreamMedian {
public:
  StreamMedian();
  ~StreamMedTraStreamMediancker();
  // other methods
  StreamMedian(const StreamMedian& rhs) = default;
  StreamMedian& operator=(const StreamMedian& rhs) = default;

private:
   // other stuff
}
```

Technically they're automatically generated either way, but...
(1) never hurts to be explicit about your intent
(2) if you declare any copy/move operations, default ones not created

Rule of Zero

If the default special member functions work, then use the default ones and don't declare your own.

Reason: the compiler is smarter than you and won't make mistakes :)

Rule of Three

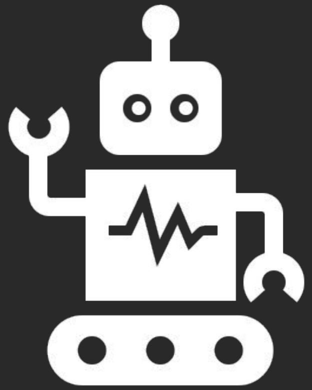
If you explicitly define (or delete) a copy constructor, copy assignment, or destructor, you should define (or delete) all three.

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

motivation

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

```
int main() {
  vector<string> words1 = findAllWords(54321234);
  vector<string> words2;
  words2 = findAllWords(54321234);
  cout << "done!" << endl;
}

vector<string> findAllWords(size_t size) {
  vector<string> words(size, "Ito");
  return words;
}
```



Demo

Printing all calls to special member functions

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

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

Quiz: how many times is each special member function called (without copy elision)?

```
vector<string> findAllWords(size_t size) {
  vector<string> words(size, "Ito");
  return words;
}
```

Fill constructor

Copy constructor
(return value)

Destructor
for words

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

```
int main() {
  vector<string> words1 = findAllWords(54321234);


  vector<string> words2;


  words2 = findAllWords(54321234);

}
```

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

Copy constructor

```
int main() {
  vector<string> words1 = findAllWords(54321234);
```

Destructor (return value)

```
  vector<string> words2;
```

Default constructor

Copy assignment

```
  words2 = findAllWords(54321234);
```

Destructor (return value)

```
}
```

Destructor x 2 (words1, words2)

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)

Quiz: how many times is each special member function called (with copy elision)?

```
vector<string> findAllWords(size_t size) {
  vector<string> words(size, "Ito");
  return words;
}
```

Quiz: how many times is each special member function called (with copy elision)?

```
vector<string> findAllWords(size_t size) {
  vector<string> words(size, "Ito");
  return words;
}
```

Fill constructor

Copy constructor
(elided)

Destructor
for words

Quiz: how many times is each special member function called (with copy elision)?

```
int main() {
  vector<string> words1 = findAllWords(54321234);


  vector<string> words2;


  words2 = findAllWords(54321234);

}
```

Quiz: how many times is each special member function called (with copy elision)?

Copy constructor
(elided)

```
int main() {
  vector<string> words1 = findAllWords(54321234);

  vector<string> words2;

  words2 = findAllWords(54321234);

}
```

Destructor
(return value)

Default constructor

Copy
assignment

Destructor
(return value)

Destructor x 2
(words1, words2)

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?

```
int main() {
  vector<string> words1 = findAllWords(54321234);

  vector<string> words2;

  words2 = findAllWords(54321234);
}
```

Copy constructor

Destructor
(return value)

Destructor
(return value)

Copy
assignment

Let's try to move the vectors instead.

Move constructor

```
int main() {
  vector<string> words1 = findAllWords(54321234);

  vector<string> words2;

  words2 = findAllWords(54321234);
}
```

Move
assignment

Another example!

Copy constructor

```
int main() {
  vector<string> words1 = findAllWords(54321234);

  vector<string> words2 = words1;


  words1.push_back("Everything is fine!");
}
```

Copy constructor

Can we always use the move constructor?

Move constructor

```
int main() {
  vector<string> words1 = findAllWords(54321234);

  vector<string> words2 = words1;

  words1.push_back("Everything is fine!");
}
```

Move constructor

The array was stolen from words1…that's bad!

Move constructor

```
int main() {
  vector<string> words1 = findAllWords(54321234);

  vector<string> words2 = words1;
```

Move constructor

```
  words1.push_back("Everything is NOT fine!");
}
```

How do we distinguish between these cases?

Move constructor

```
int main() {
  vector<string> words1 = findAllWords(54321234);

  vector<string> words2 = words1;


  words1.push_back("Everything is fine!");
}
```

Copy constructor

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

Move constructor

```
int main() {
  vector<string> words1 = findAllWords(54321234);


  vector<string> words2 = words1;
```

Move constructor

```


  // You: I promise to never use words1 again.
}
```

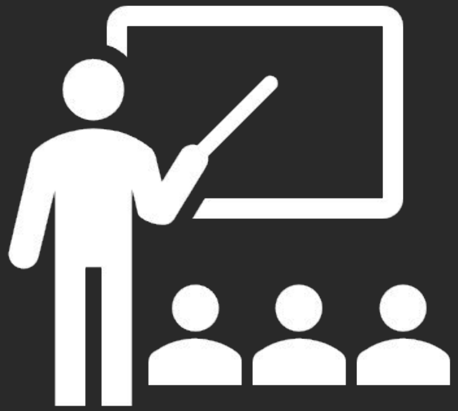
Where we are going!

- 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)

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, …

Which expressions to the right of the assignment are r-values?

Ignore the left hand side - they have to be l-values.

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

auto v4 = v1 + v2;                  // D
auto v5 = v1 += v4;                 // E
size_t size = v.size();             // F
val = static_cast<int>(size);       // G
v1[1] = 4*i;                        // H
ptr = &val;                         // I
v1[2] = *ptr;                       // J
```

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!

+ returns a copy to a temporary.

cast returns a copy of size.

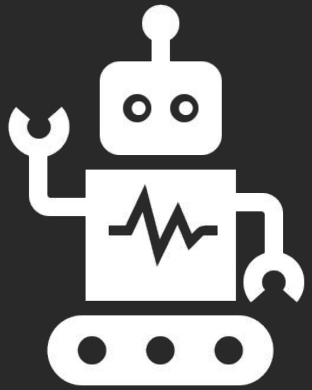
Value type differences: lifetimes

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
```



Key Idea

l-value have a name/identity, and live until its scope is over
r-values are temporary, live until the next line

Questions

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

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);                  // A
rref(v1);                  // B
lref(v2 + v3);             // C
clref(v2 + v3);            // D
rref(v2 + v3);             // E
```

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);                  // OKAY: l binds to l
rref(v1);                  // BAD: r no bind to l
lref(v2 + v3);             // BAD: l no bind to r
clref(v2 + v3);            // OKAY: cl binds to r
rref(v2 + v3);             // OKAY: r binds to r
```

Flashback to Week 1 of CS 106B

- One caveat: in order to pass a variable by reference, you need to actually have a variable. The following does not work, for our example above:

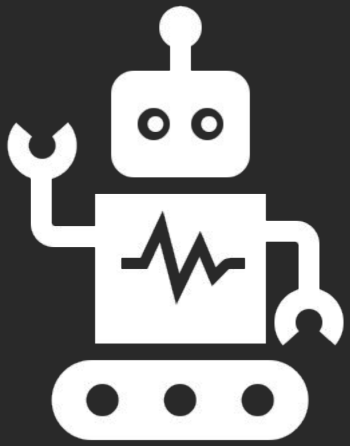
```
doubleValueWithRef(15); // error! cannot pass a literal value by reference
```

Compiler error:

```
../all-examples.cpp:135:5: error: no matching function for call to 'doubleValueWithRef'
  doubleValueWithRef(15);
  ^~~~~~~~~~~~~~~~~~
../all-examples.cpp:11:6: note: candidate function not viable: expects an l-value for 1st argument
void doubleValueWithRef(int &x);
   ^
1 error generated.
```

Challenge Question

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



l or r-value?

Challenge Question

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

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;
```

`vector<int>&&` → `vector<int>`

`v4` → `v1 + v2`

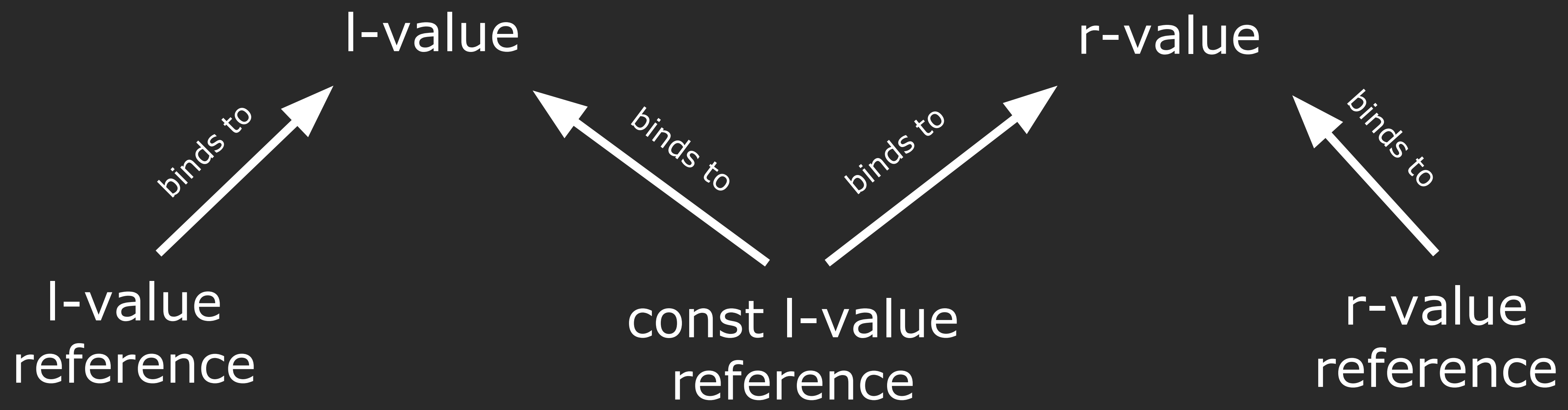
l-value → r-value

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.



Key Idea

l-value references bind only to l-values
r-value references bind only to r-values
const l-value references bind to anything

Questions

Announcements

Logistics

- Assignment 2 is officially released!
- Starter code available on…
  - Class Website (Qt Creator version)
  - Github (direct files -> command line or other IDE)
- Lots of ways to get help! OH, assignment walkthroughs, etc.

Logistics

- Like assignment 1, we'll have 2 checkpoints to help you stay on track.
- Checkpoint 1 is soft, but checkpoint 2 is strict (i.e. we'll require late days).
- The first checkpoint requires ~15 lines of code.
- There is a hard deadline: Week 10, Wednesday, at 11:59 pm.

Checkpoint 1 due next Tuesday

- Required:
  - Milestone 0: learn about hashing, read the starter code
  - Milestone 1: finish rehash()
  - Milestone 2: implement operators, fix const-correctness
  - Milestone 3: implement special member functions + move semantics
  - Milestone 4: answer short answer questions

- Optional:
  - Milestone 5: implement advanced constructors
  - Milestone 6: implement an iterator class

Back to Move Semantics

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.

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.

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.

not safe to move
copy instead

l-value

safe to move

r-value

`T& other`

`const T& other`

`T&& other`

Everything in one picture.

not safe to move
copy instead

l-value

safe to move

r-value

`T& other`

`const T& other`

`T&& other`

Right now your copy constructor
works for both l and rvalues

Everything in one picture.

not safe to move
copy instead

l-value

safe to move

r-value

Compiler prefers T&& for
rvalues if implemented

`T& other`

`const T& other`

`T&& other`

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

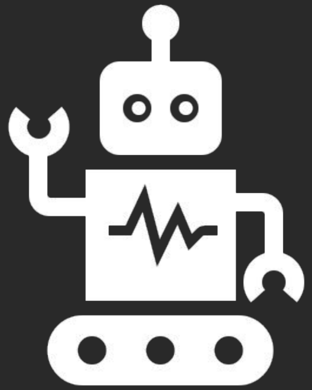
Function signatures of all our special member functions.

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

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

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



Example

Move constructor

Move constructor

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

```
vector(vector<T>&& other) :
   _elems(other._elems),
   _size(other._size),
   _capacity(other._capacity) {

   other._elems = nullptr;
   // optional - set other._size/_capacity to 0
}
```

Super important! Otherwise you'll get the double free we saw last time.

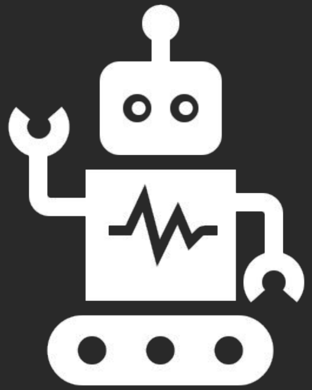
Key steps for a move constructor

- 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

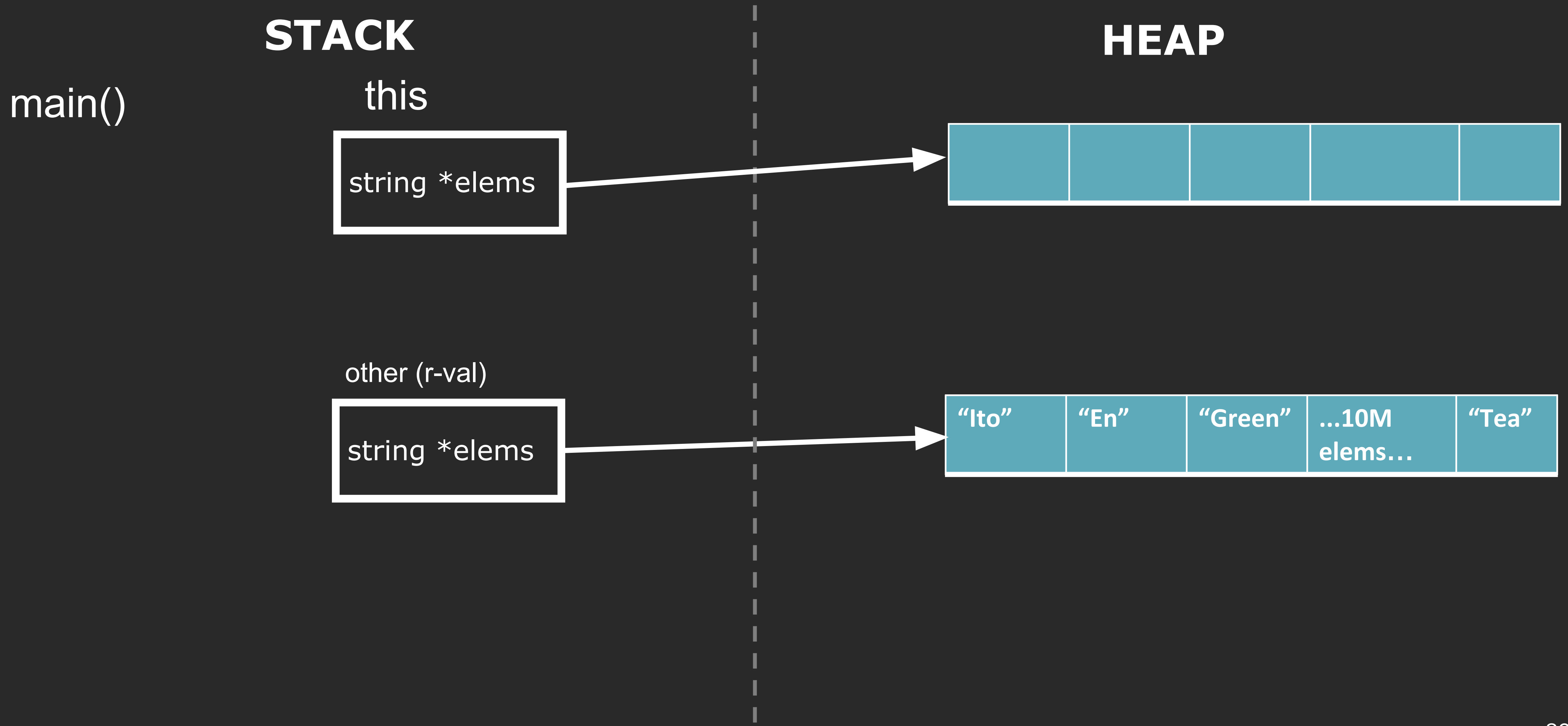
- Check self-assignment.
- Free up resources held by this.
- Transfer the contents of other to this.
    - Move instead of copy whenever possible!
- Leave other in an undetermined but valid state
    - Normally: set it to the default value of class



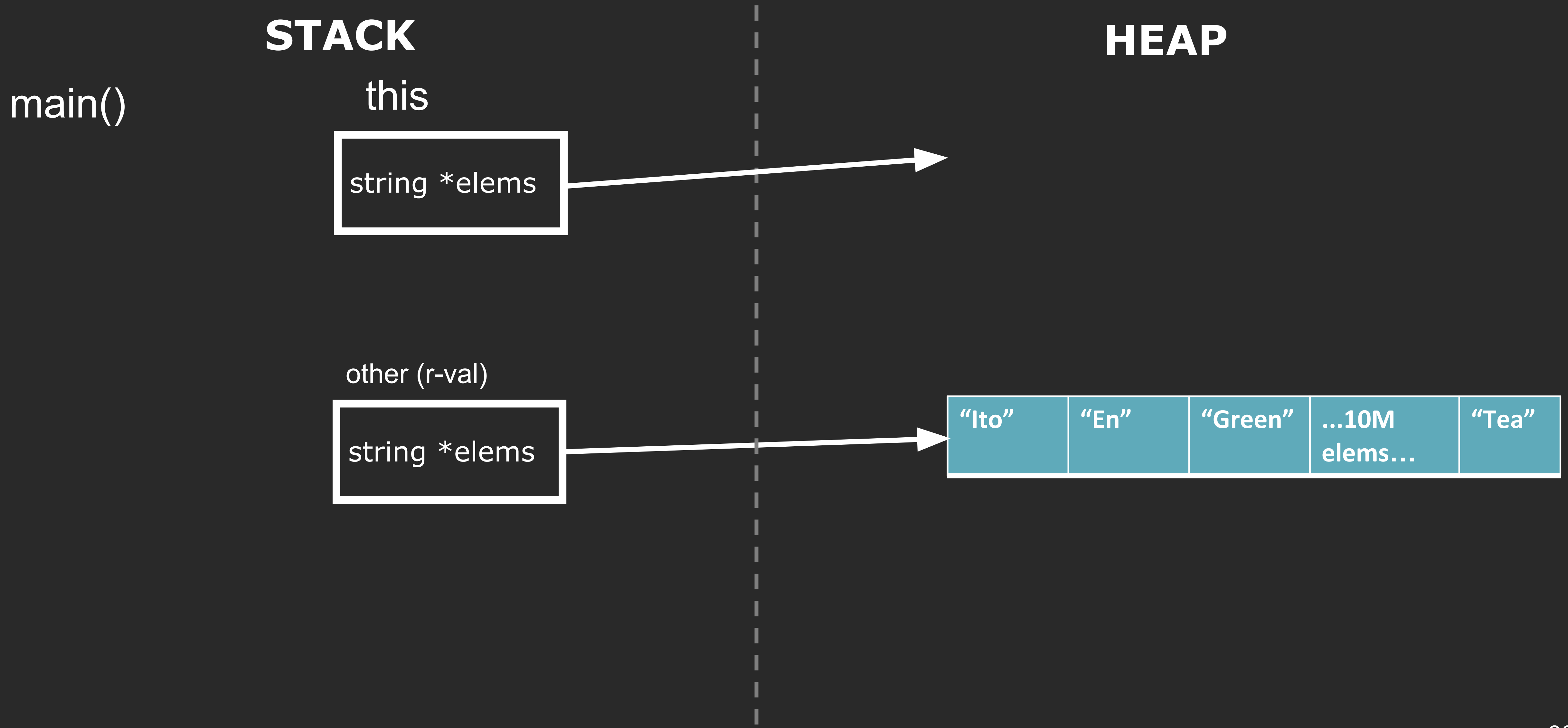
Example

Move assignment

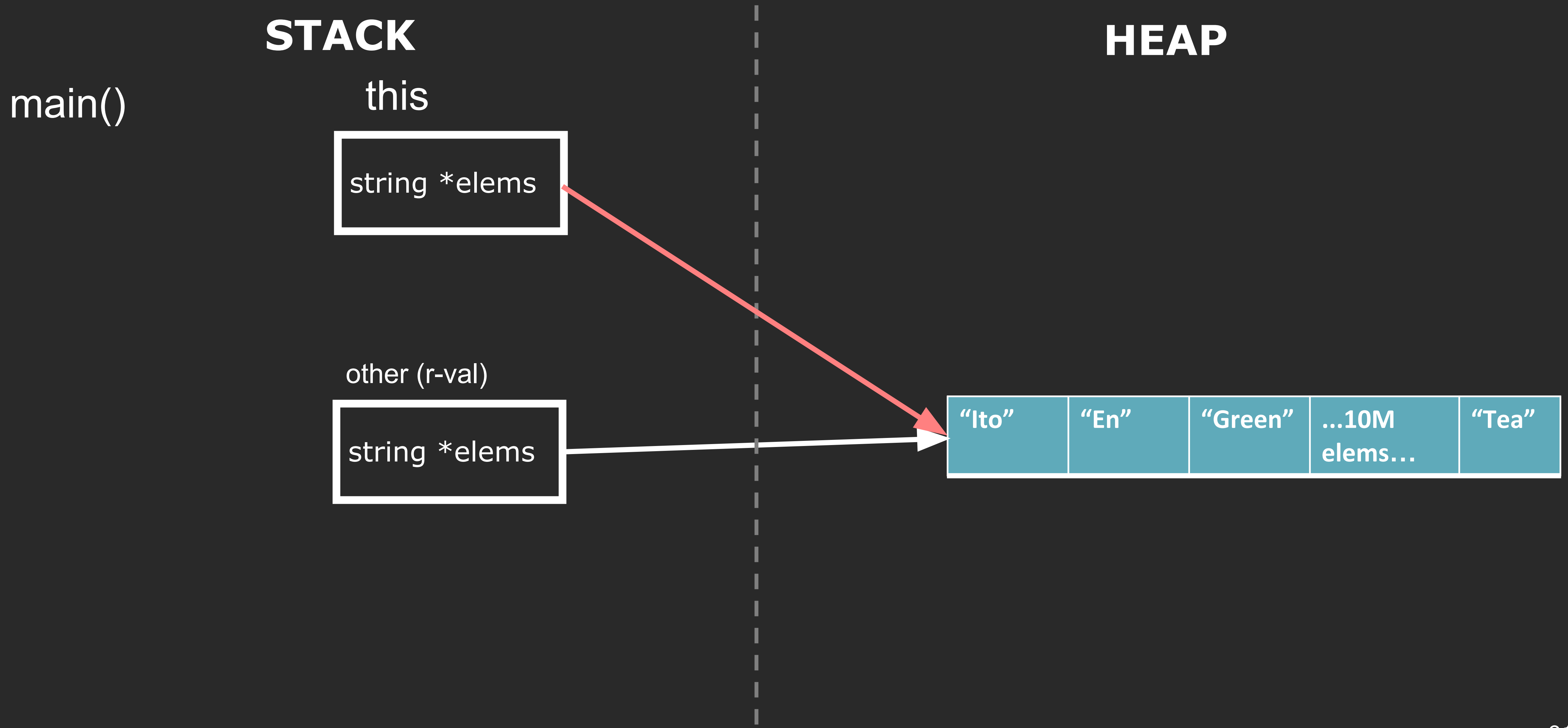
Simulation of move assignment.



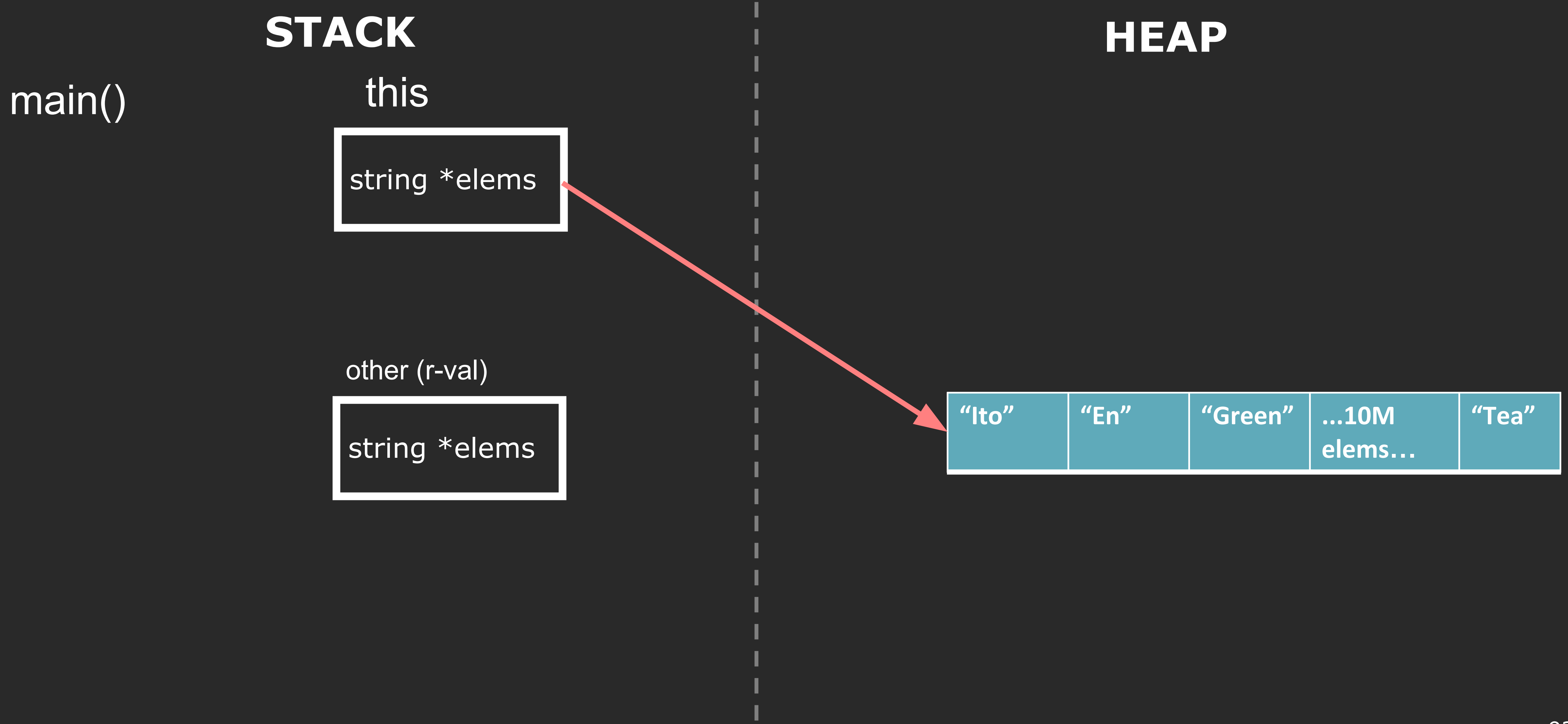
Simulation of move assignment.



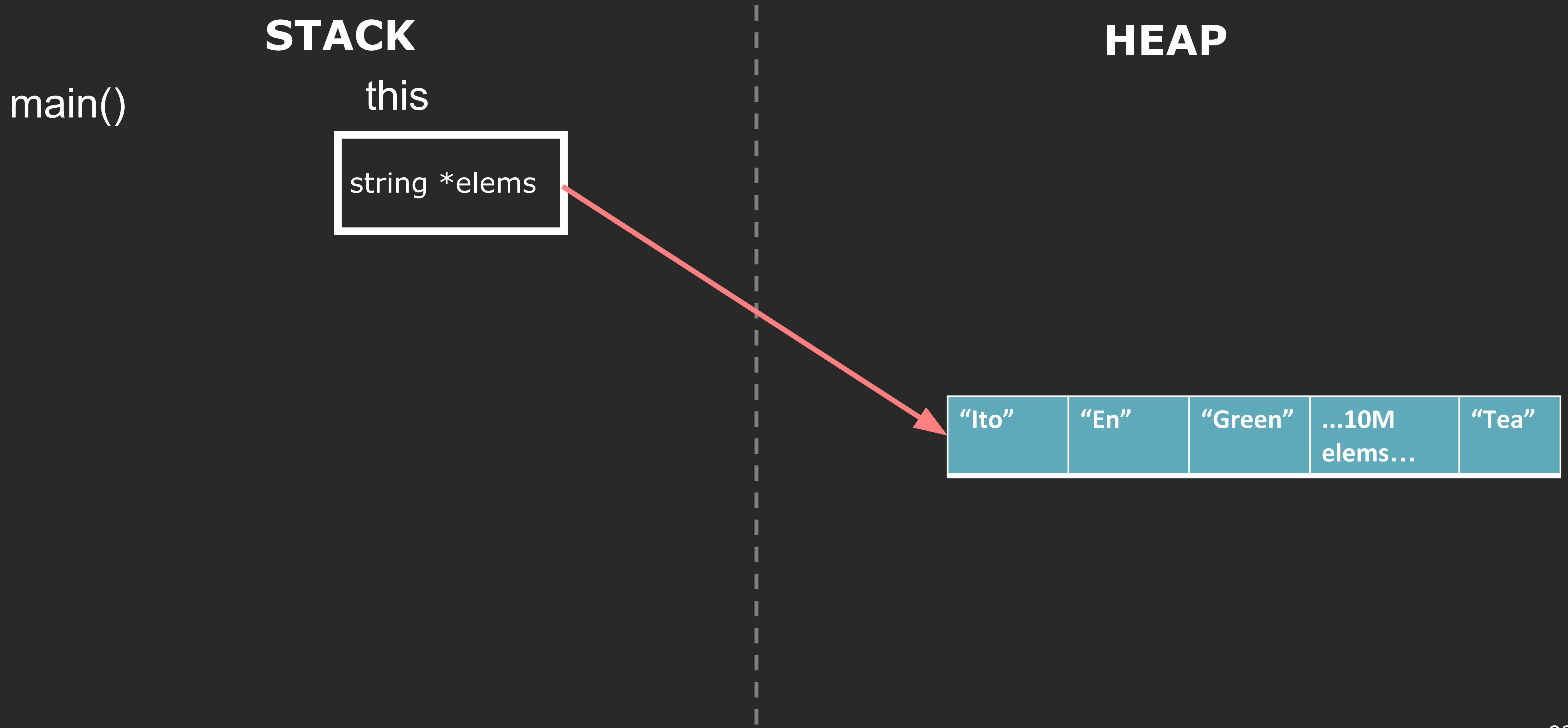
Simulation of move assignment.



Simulation of move assignment.



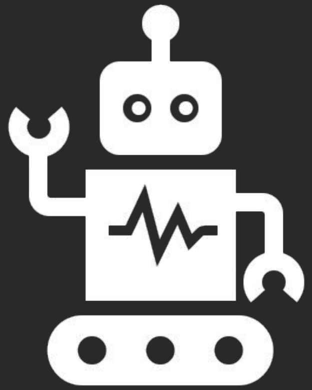
Simulation of move assignment.



Move assignment

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

```
vector<T>& operator=(vector<T>&& rhs) {
  if (this != &rhs) return *this;
   delete[] elems;
   _size = rhs._size;
   _capacity = rhs._capacity;
   _elems = rhs._elems;
   rhs._elems = nullptr;

  return *this;
}
```



Key Idea

Copy operations accept as input a l-value reference
Move operations accept as input a r-value reference
The compiler chooses which based on whether you have an l-value or an r-value.

Questions

This is a small problem in our code.

Did we actually move
all the members?

You can ask the compiler to generate default versions of special member functions.

```
class StreamMedian {
public:
  StreamMedian();
  ~StreamMedian();
  // other methods
  StreamMedian(StreamMedian&& other);
  StreamMedian& operator=(StreamMedian&& rhs);

private:
   vector<int> _elems;
   Compare _comp;
}
```

Move or copy assignment?

```
StreamMedian& StreamMedian<T>::operator=(StreamMedian&& rhs) {
   if (this != &rhs) {
      // no freeing needed

      _elems = rhs._elems;




   }
   return *this;
}
```

Move or copy assignment?

```
StreamMedian& StreamMedian<T>::operator=(StreamMedian&& rhs) {
   if (this != &rhs) {
      // no freeing needed

      _elems = rhs._elems;


   }
   return *this;
}
```

move or copy?

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 incorrect: we are still copying!

```
StreamMedian& StreamMedian<T>::operator=(StreamMedian&& rhs) {
   if (this != &rhs) {
      // no freeing needed

      _elems = rhs._elems;


   }
   return *this;
}
```

these are l-values!

This is incorrect: we are still copying!

```
StreamMedian& StreamMedian<T>::operator=(StreamMedian&& 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

The fix: cast it to an r-value.

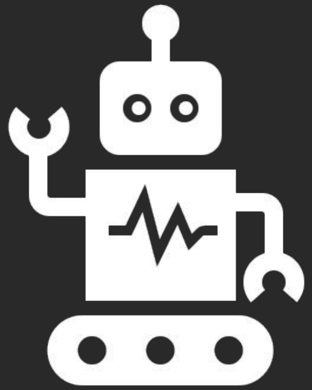
```
StreamMedian& StreamMedian<T>::operator=(StreamMedian&& rhs) {
  if (this != &rhs) {
    // no freeing needed

    _elems = static_cast<vector<T>&&>(rhs._elems);


  }
  return *this;
}
```

now it is!

std::move!

```
StreamMedian& StreamMedian<T>::operator=(StreamMedian&& rhs) {
    if (this != &rhs) {
        // no freeing needed

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



    }
    return *this;
}
```

better template version
in the STL!



Example

std::move-ing all the members

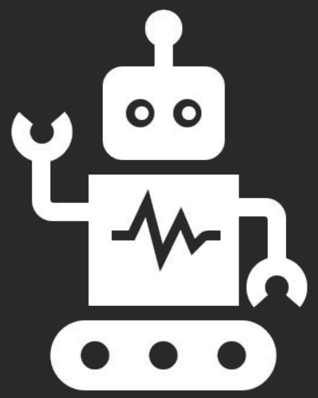
Move constructor
(now it's perfect and idiomatic)

```
vector(vector<T>&& other) :
   _elems(std::move(other._elems)),
   _size(std::move(other._size)),
   _capacity(std::move(other._capacity)) {

   other._elems = nullptr;

}
```

Move assignment

(now it's perfect and idiomatic)

```
vector<T>& operator=(vector<T>&& rhs) {
  if (this != &rhs) return *this;
   delete[] elems;
   _size = std::move(rhs._size);
   _capacity = std::move(rhs._capacity);
   _elems = std::move(rhs._elems);
   rhs._elems = nullptr;

  return *this;
}
```



Key Idea

You must explicitly move each member inside your move operation, otherwise your move operation actually copies.

Questions

std::move

Unconditional cast to an r-value.

std::move

Unconditional cast to an r-value.

Yes...there is a conditional cast!

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

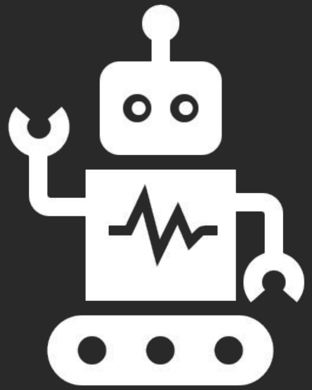
std::move

Honestly a way better name…

```
_elems = std::r_value_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 become an r-value.



Key Idea

std::move turns anything into an r-value
std::move doesn't move anything -
a move operation (ctor or assignment) actually does.

Questions

What did that fix?

```
int main() {
  vector<string> words1 = findAllWords(54321234);

  vector<string> words2;

  words2 = findAllWords(54321234);
}
```

r-value

r-value

What did that fix?

Move constructor

```
int main() {
  vector<string> words1 = findAllWords(54321234);

  vector<string> words2;
```

Move assignment

```
  words2 = findAllWords(54321234);
}
```

What did that fix?

```
int main() {
  vector<string> words1 = findAllWords(54321234);


  vector<string> words2 = words1;




  words1.push_back("Everything is fine!");
}
```

r-value

l-value

What did that fix?

Move constructor

```
int main() {
  vector<string> words1 = findAllWords(54321234);

  vector<string> words2 = words1;


  words1.push_back("Everything is fine!");
}
```

Copy constructor

What did that fix?

```
int main() {
  vector<string> words1 = findAllWords(54321234);

  vector<string> words2 = std::move(words1);
```

r-value

```
  // I promise to not use words1 before reassigning it
}
```

What did that fix?

```
int main() {
  vector<string> words1 = findAllWords(54321234);

  vector<string> words2 = std::move(words1);

  // I promise to not use words1 before reassigning it
}
```

Move constructor

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

```
int main() {
  vector<string> words1 = findAllWords(54321234);
  vector<string> words2;
  words2 = findAllWords(54321234);
  cout << “done!” << endl;
}

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

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

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

Fill constructor

Copy constructor
(elided)

Destructor
for words

Demo

Printing and timing all calls to special member functions

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

```
int main() {
  vector<string> words1 = findAllWords(54321234);

  vector<string> words2;

  words2 = findAllWords(54321234);

}
```

Move constructor (elided)

Destructor (return value)

Default constructor

Move assignment

Destructor (return value)

Destructor x 2 (words1, words2)

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

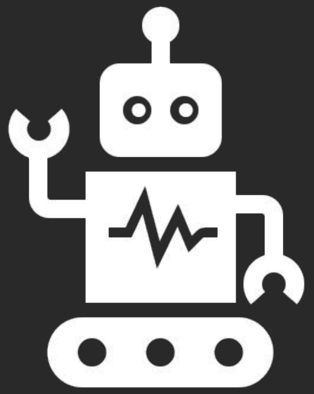
applications

Our push_back function before.

```
template <typename T>
void vector<T>::push_back(const T& element) {
   elems[_size++] = element;
}
```

Gets a new r-value sibling!!

```
template <typename T>
void vector<T>::push_back(const T& element) {
    elems[_size++] = element;
}

template <typename T>
void vector<T>::push_back(T&& element) {
    elems[_size++] = std::move(element);
}
```



Deep Dive into STL

r-value references in the wild

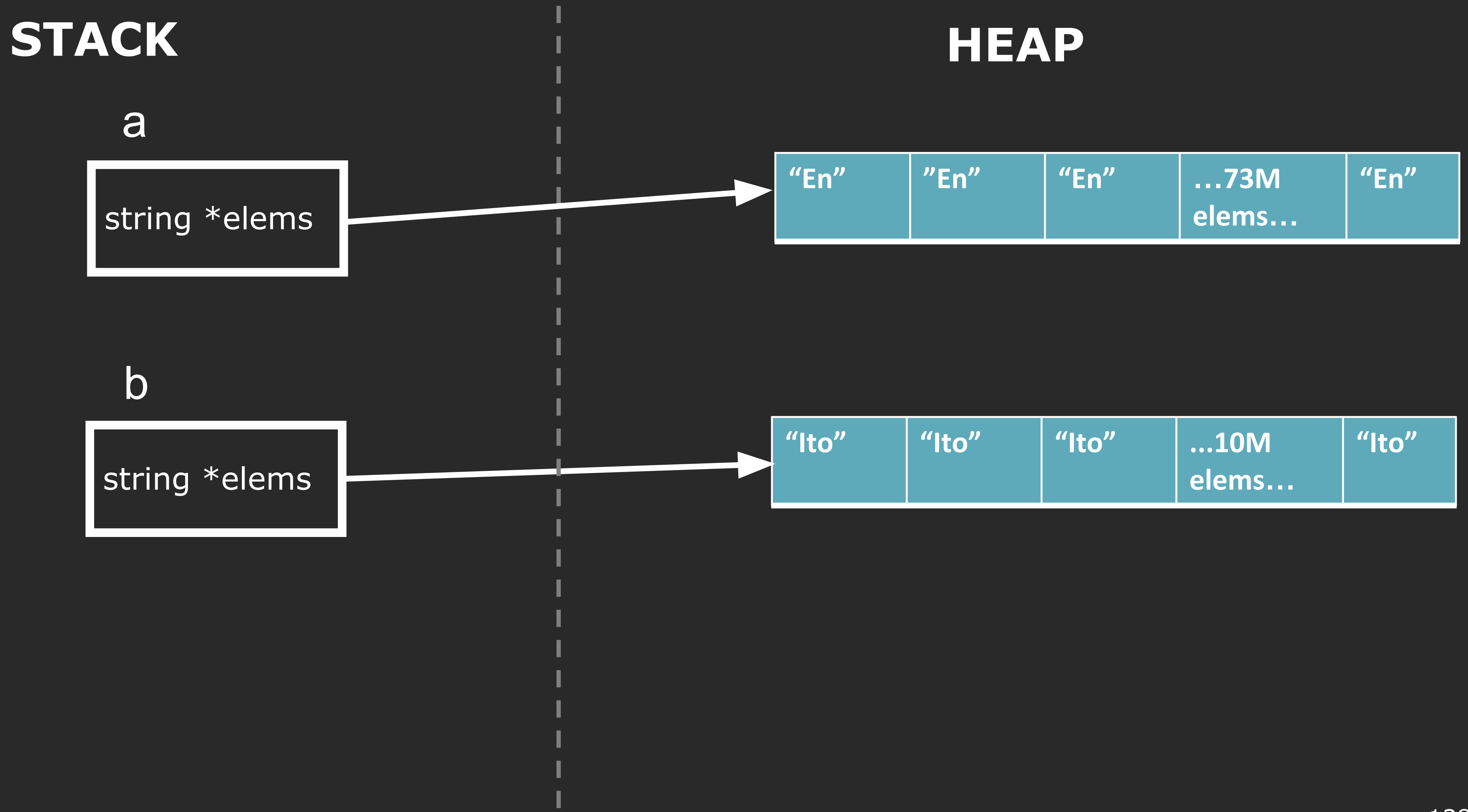
Questions

Your task: write a generic swap function.

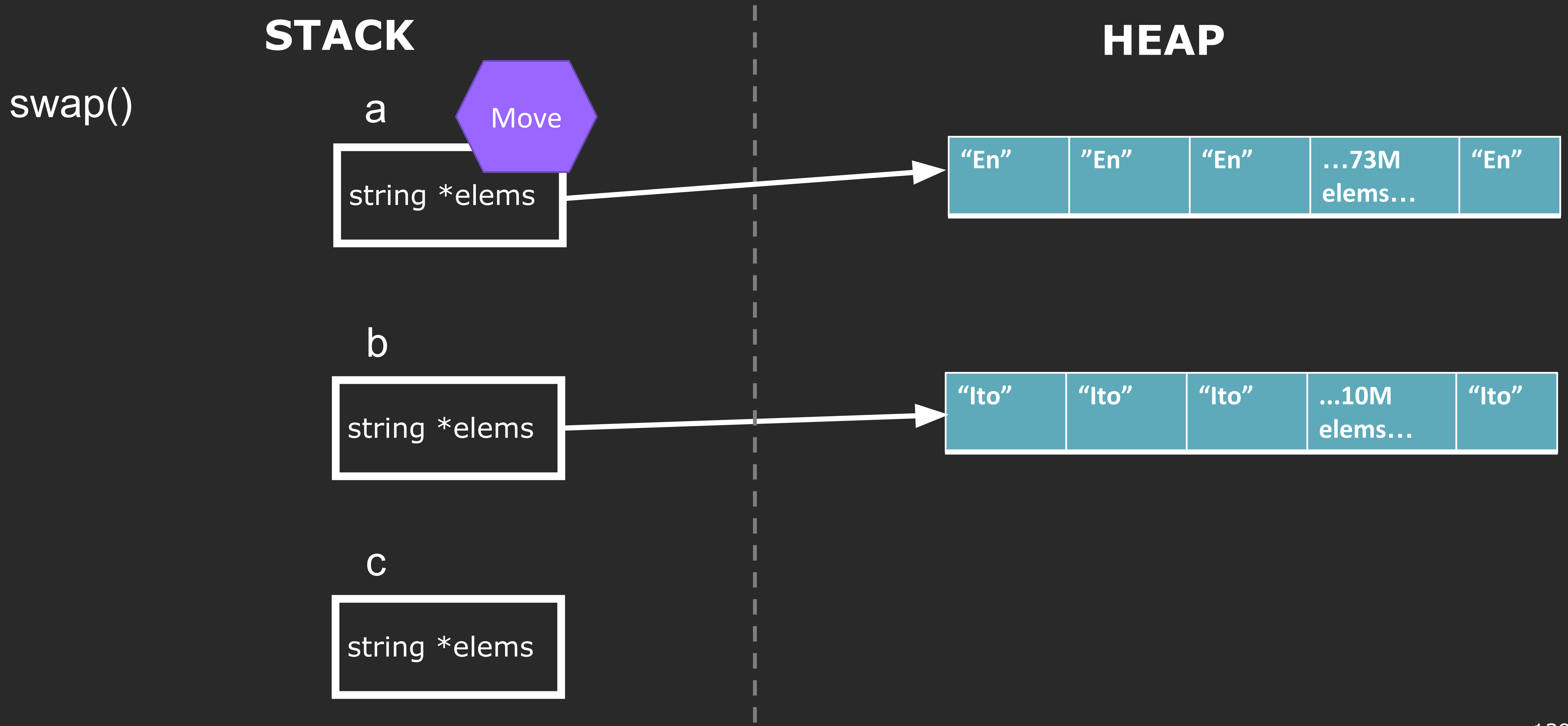
```
int main() {
  vector<string> v1("En", 73837463);
  vector<string> v2("Ito", 10000000);
  swap(v1, v2);


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

Simulation of move assignment.

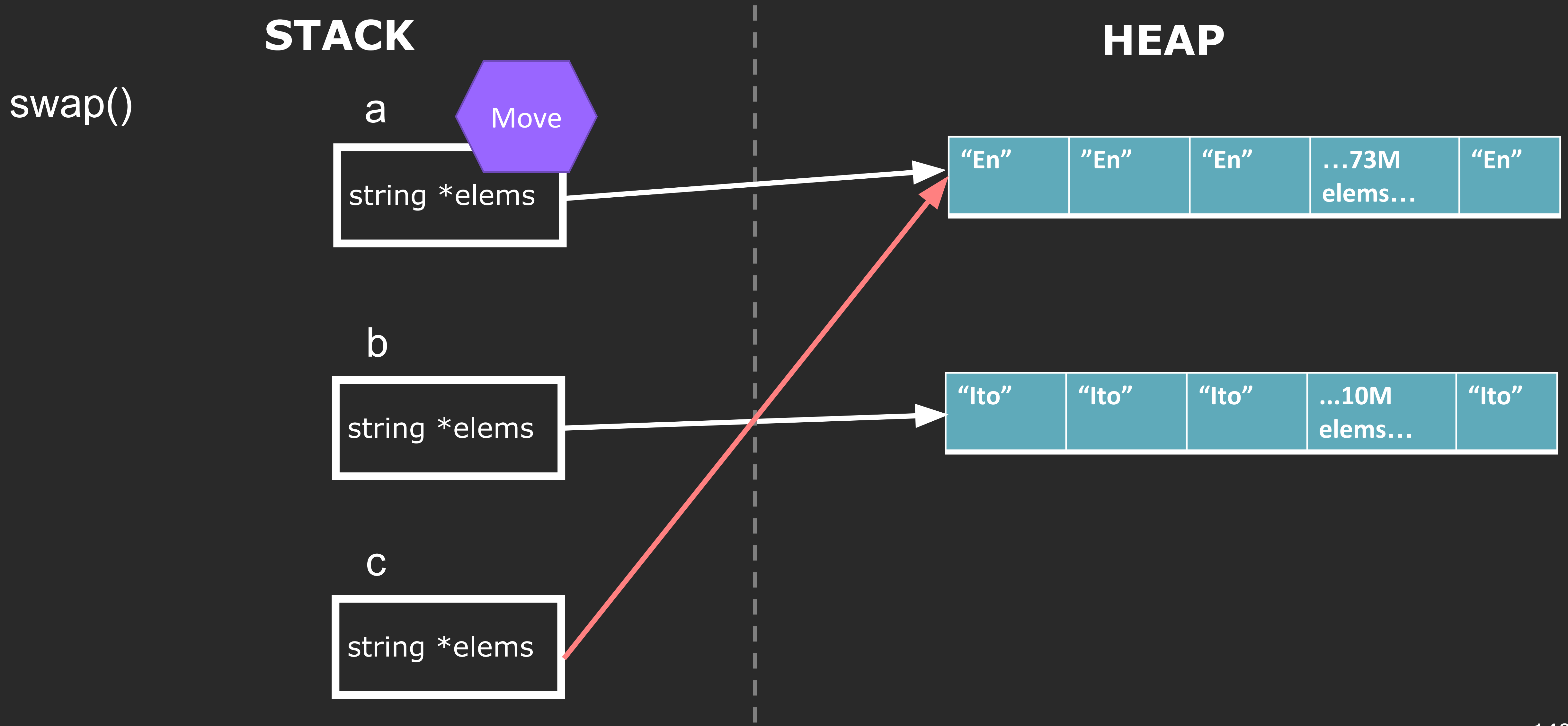
swap()



Simulation of move assignment.

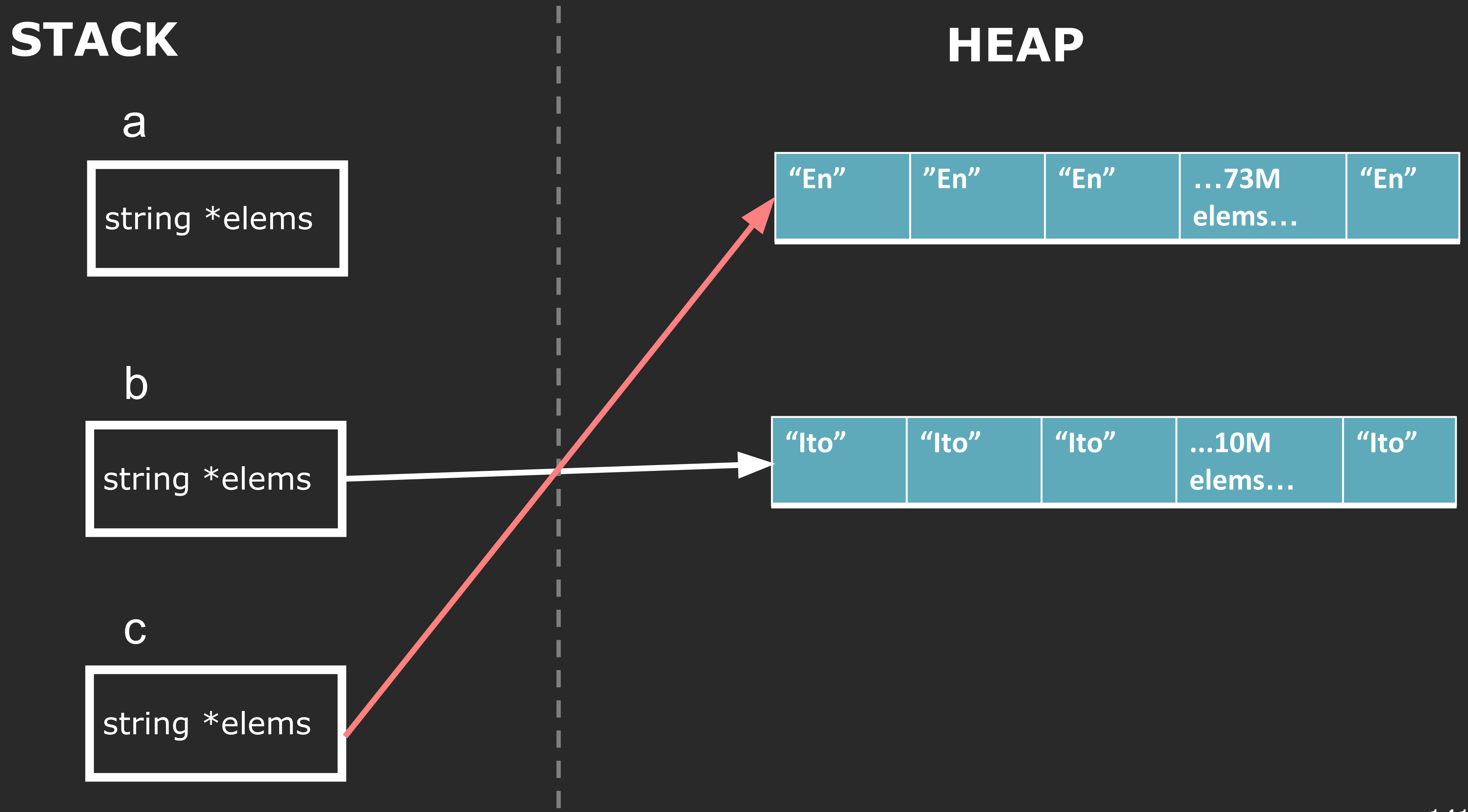


Simulation of move assignment.



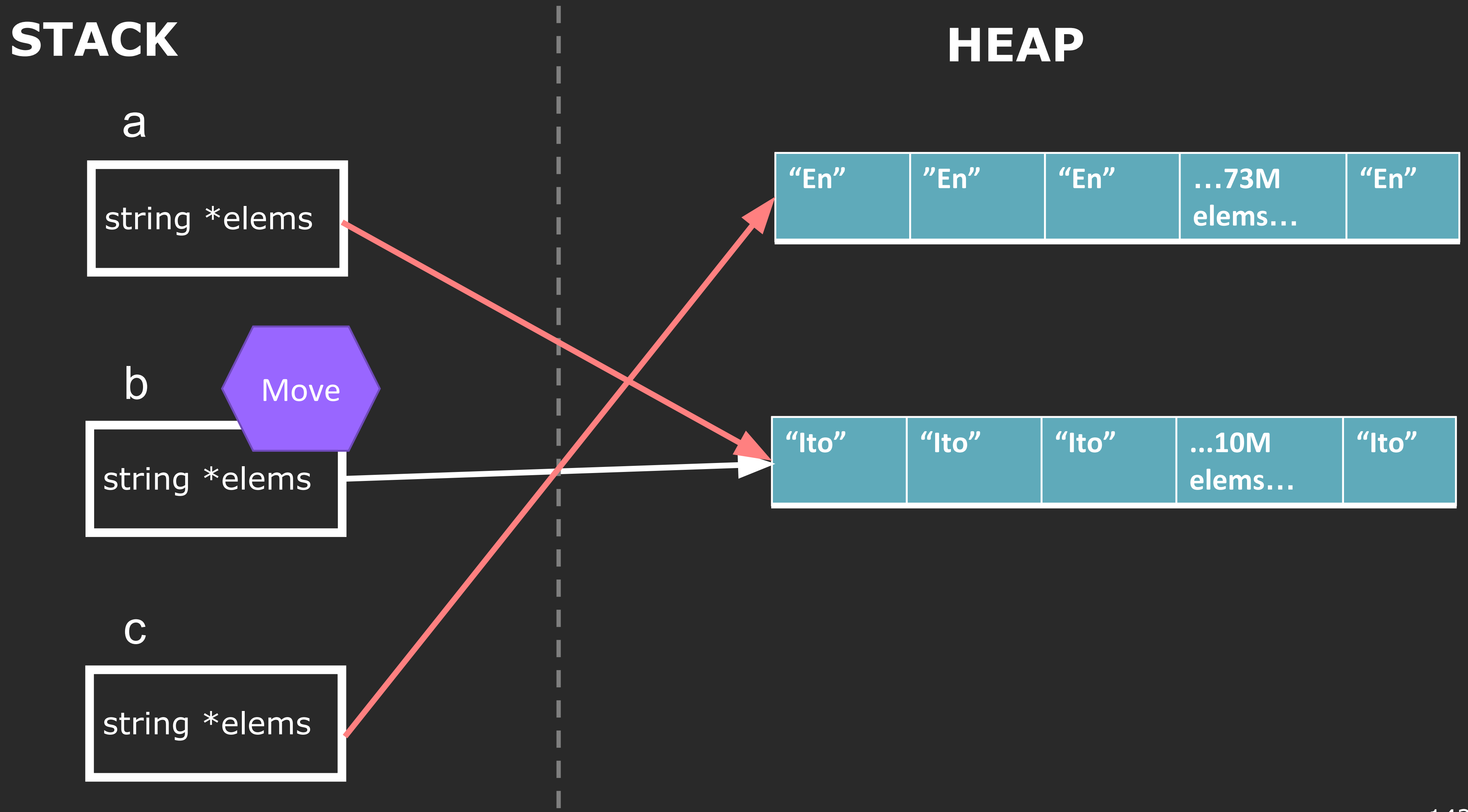
Simulation of move assignment.

swap()



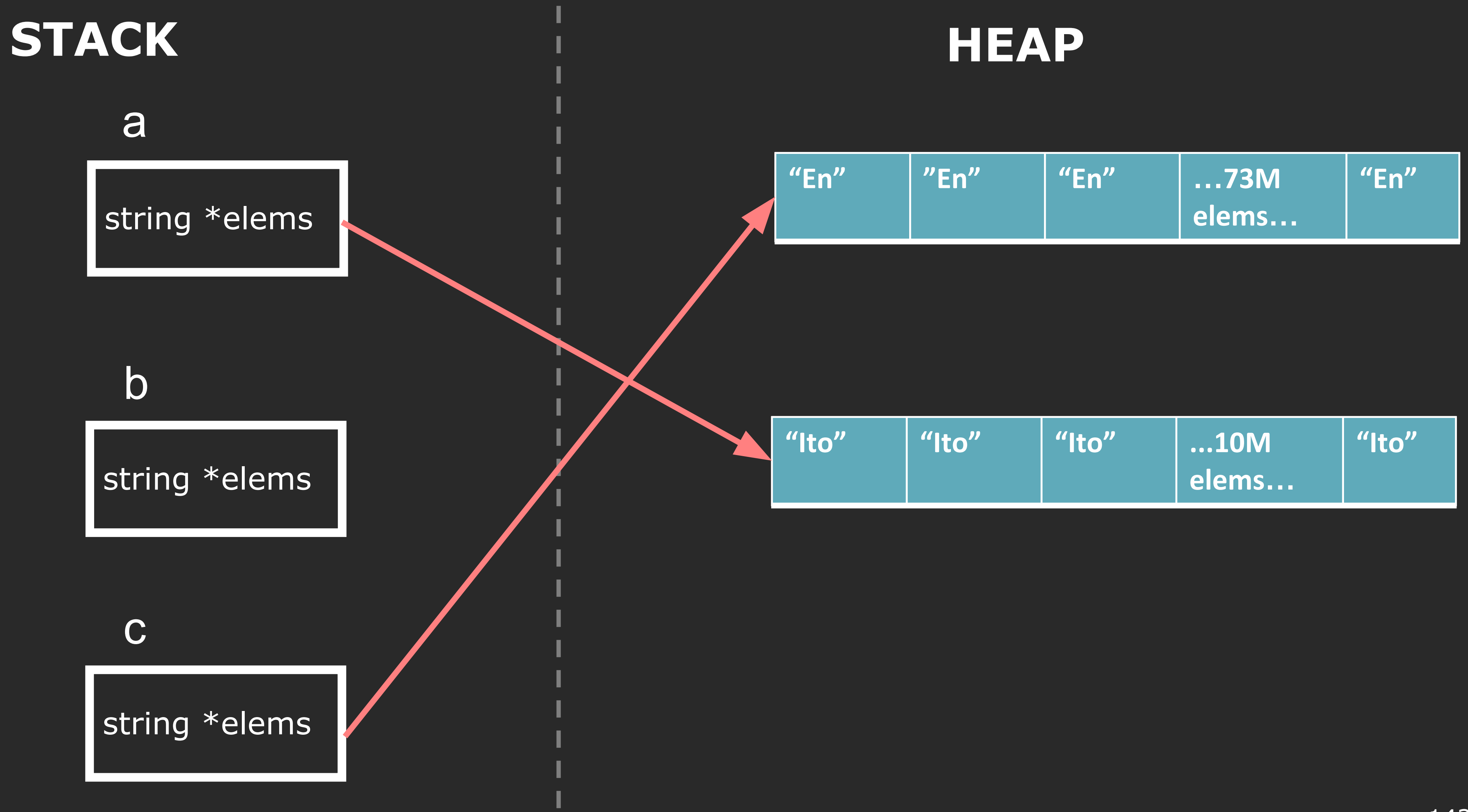
Simulation of move assignment.

swap()

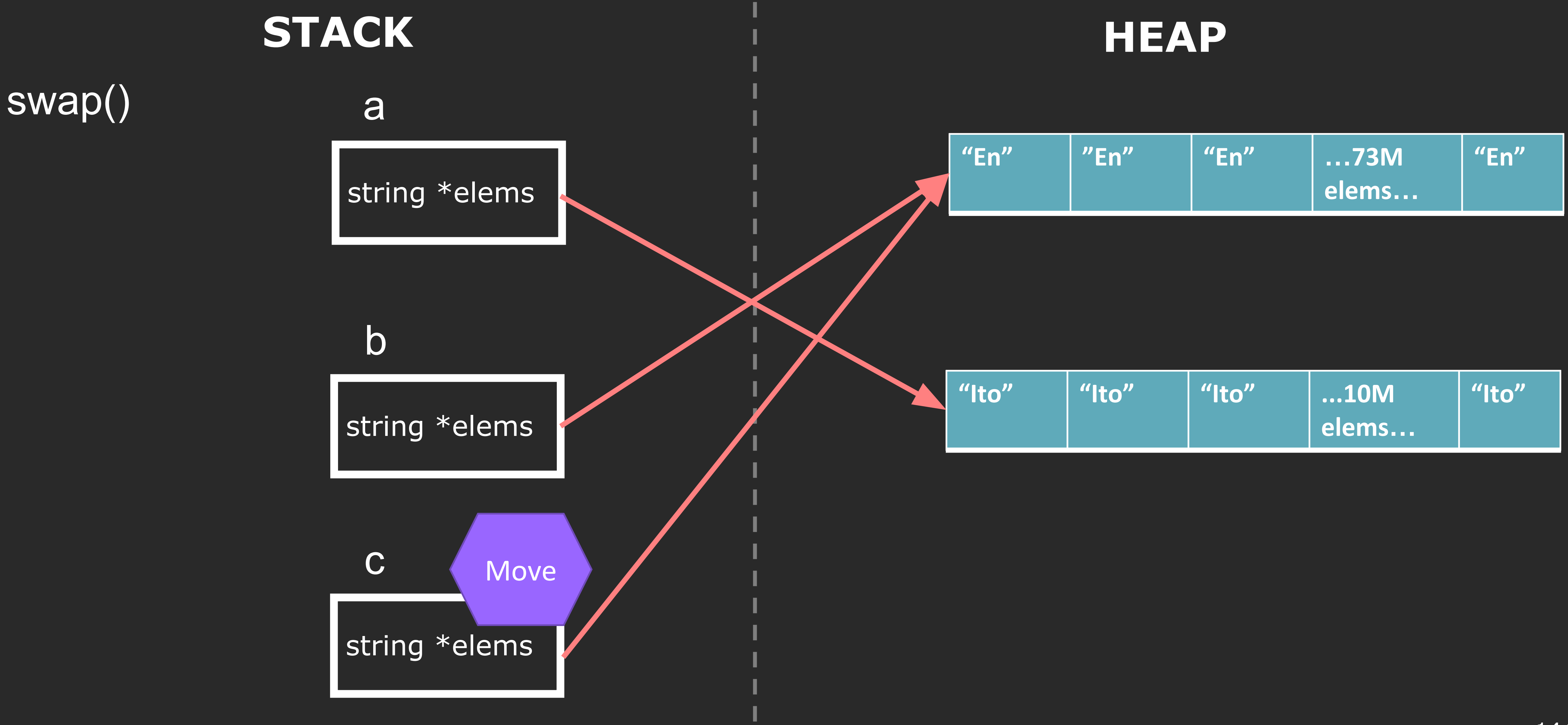


Simulation of move assignment.

swap()

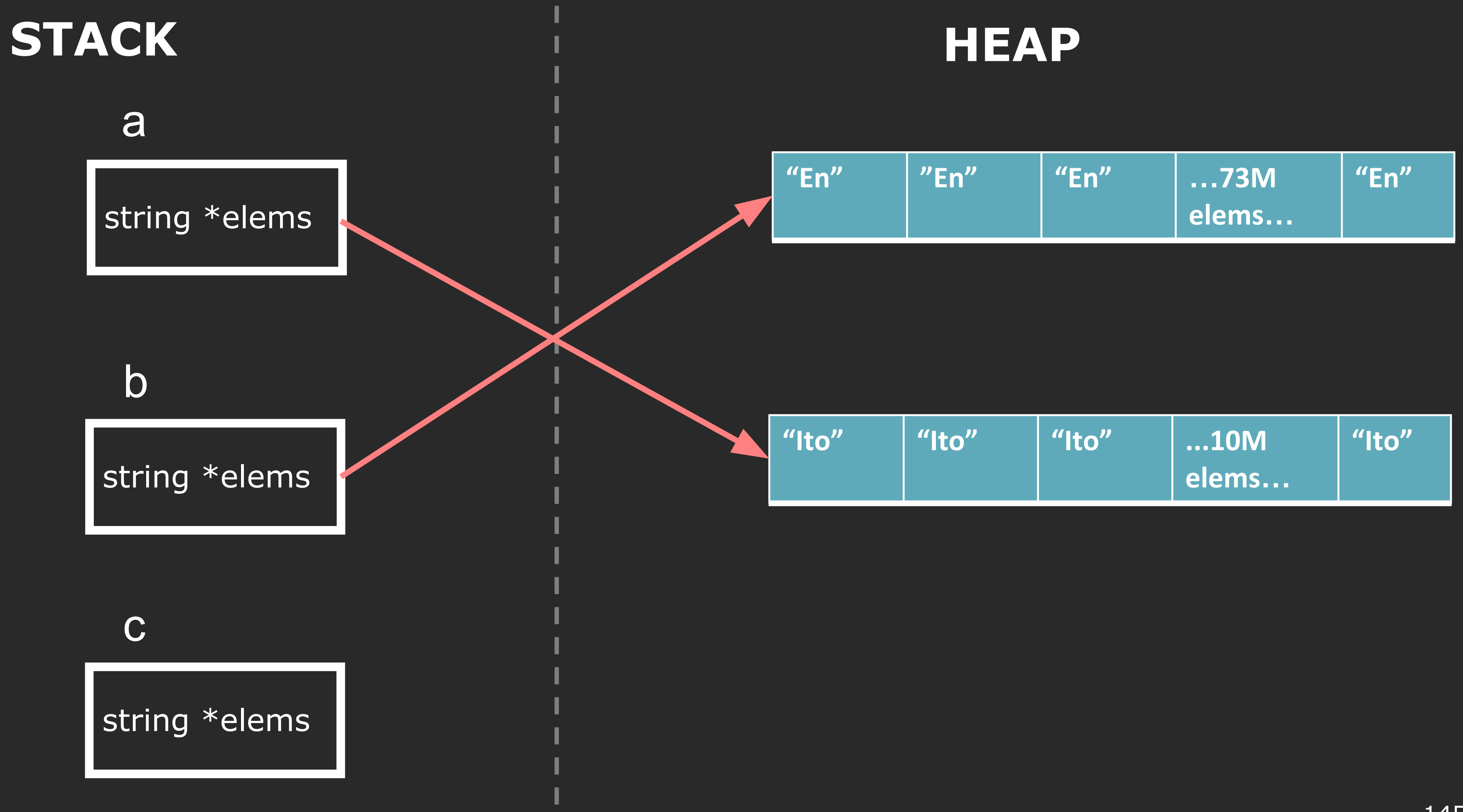


Simulation of move assignment.

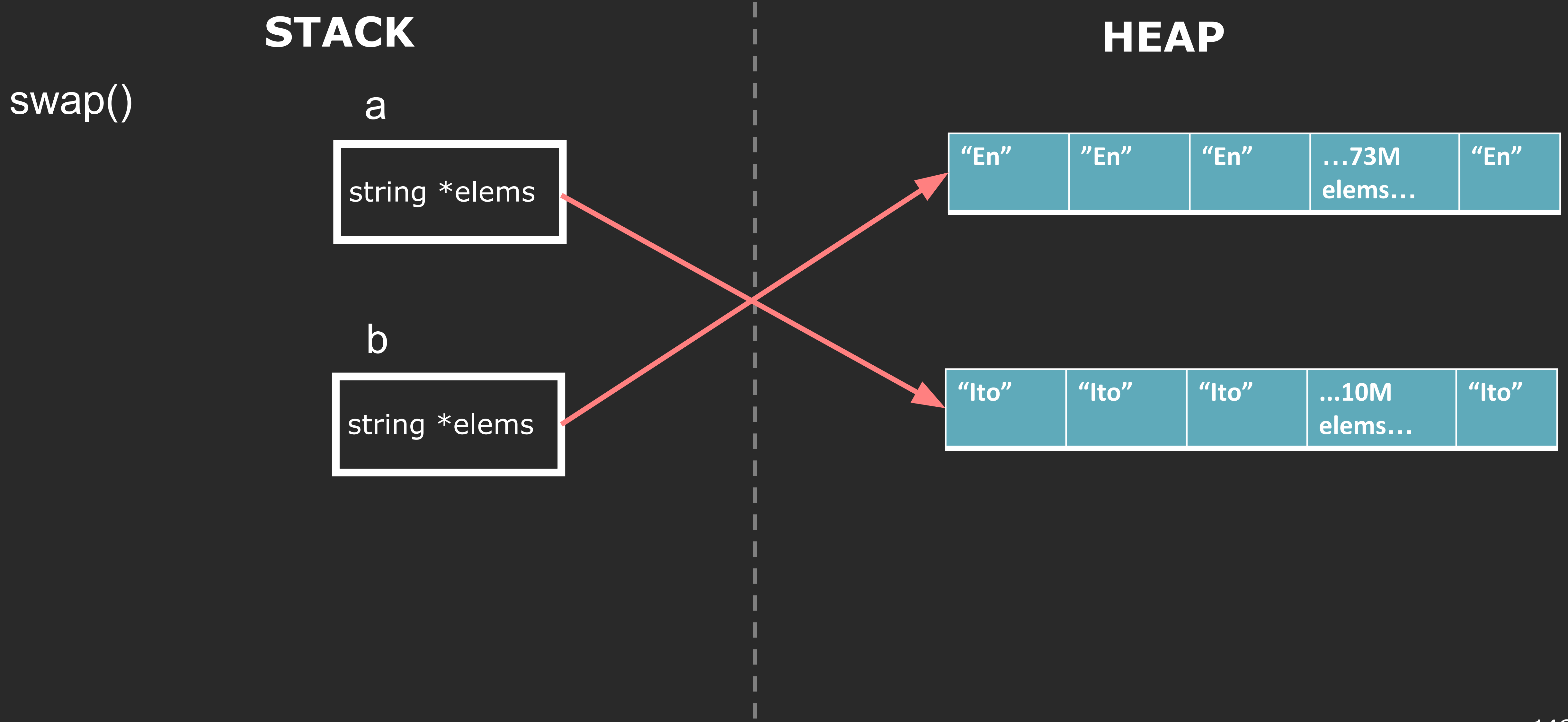


Simulation of move assignment.

swap()

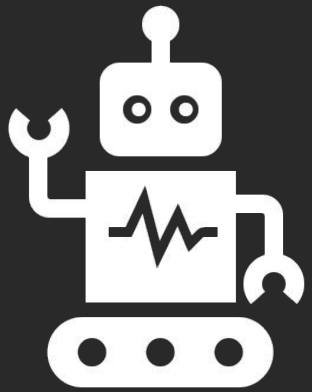


Simulation of move assignment.



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
```



Key Idea

move semantics is important besides the special member functions.

Questions

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

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

Fill constructor

Move constructor

Destructor for words

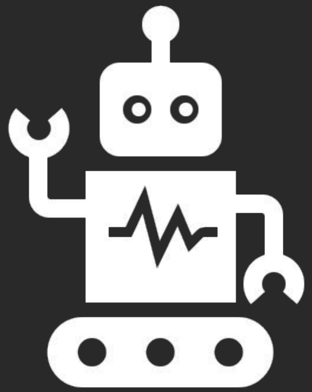
The compiler is great at optimizing return values. Don’t interfere with it.

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

Fill constructor

Copy constructor
(elided)

Destructor
for words



Key Idea

You will almost **never** see r-value references as the return value.

Questions

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.

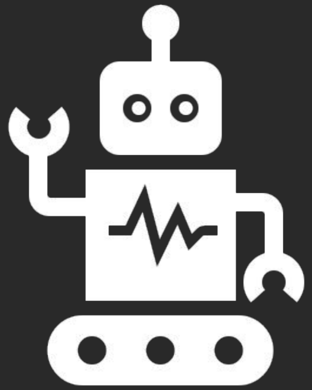
The fact that you defined one of these means
one of your members has ownership issues
that need to be resolved.

Rule of Zero

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

You can default these operations explicitly!

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



Key Idea

If possible, default all special member functions.
Otherwise, define/delete all five.

Questions

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

Common patterns with special member functions.

Type 1: default everything!
    (classes which don't require external resources, preferred!)

Type 2: declare all five special member functions, and move operations are different from copy operations.
    (classes which require external resources - e.g. memory)

Type 3: delete the copy operations, only have move operations and dtor
    (classes where the external resource can't be copied - e.g. files)

Type 4: delete both move and copy, only have the destructor
    (classes where moving a resource will destroy its integrity - e.g. mutexes)

perfect forwarding and emplace_back

We definitely won't have time for this…

but vote for it as the final topic!

How do you implement make_pair?

This is a simplification - there are things called reference_wrappers that are way beyond the scope of CS 106L.

```
template <typename T1, typename T2>
std::pair<T1, T2> make_pair(const T1& a, const T2& b) {

    return pair<T1, T2>{a, b};

}
```

How do you implement make_pair?

This is a simplification - there are things called reference_wrappers that are way beyond the scope of CS 106L.

```
template <typename T1, typename T2>
std::pair<T1, T2> make_pair(const T1& a, const T2& b) {

  return pair<T1, T2>{a, b};

}
```

Problem: what if your parameters are r-values?

One solution: overload every possible combination of l and r-value reference parameters.

```
template <typename T1, typename T2>
std::pair<T1, T2> make_pair(const T1& a, const T2& b);

template <typename T1, typename T2>
std::pair<T1, T2> make_pair(T1&& a, const T2& b);

template <typename T1, typename T2>
std::pair<T1, T2> make_pair(const T1& a, T2&& b);

template <typename T1, typename T2>
std::pair<T1, T2> make_pair(T1&& a, T2&& b);
```

Good luck implementing make_tuple!

Technically - what we declared aren't r-value references.

```
template <typename T1, typename T2>
std::pair<T1, T2> make_pair(T1&& a, T2&& b) {

   return pair<T1, T2>{a, b};

}
```

The difference here is that T1 and T2 are template parameters here, unlike before.

Universal (forwarding) references: references that bind to both l and r-values.

```
template <typename T1, typename T2>
std::pair<T1, T2> make_pair(T1&& a, T2&& b) {

   return pair<T1, T2>{a, b};

}
```

The difference here is that T1 and T2 are template parameters here, unlike before.

Problem: we are forwarding the parameters as l-values.

```
template <typename T1, typename T2>
std::pair<T1, T2> make_pair(T1&& a, T2&& b) {

    return pair<T1, T2>{a, b};

}
```

Can we use std::move here? No!
What if the parameters were l-values
before the function?

The fix: use std::forward.

```
template <typename T1, typename T2>
std::pair<T1, T2> make_pair(T1&& a, T2&& b) {

    return pair<T1, T2>{std::forward<T1>(a),
                        std::forward<T2>(b)};

}
```

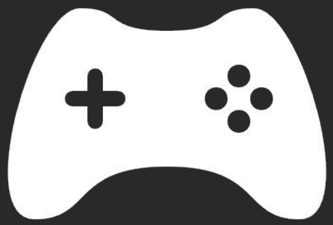
Conditional cast to an r-value, depending whether a and b were r-values before the function call.

move semantics, part 2

How would you implement make_tuple?

What exactly is emplace_back?

Vote for this in the final lecture!



Next time

RAII
(the single most important C++ idiom)

One slide summary of today

- l-value has identity, lives by scope, r-value does not, lives to next line
- r-value reference binds only to r-values, const l-value ref binds to all
- move operation uses r-value reference as parameter so it is called with r-values only. Copy operations use const l-value references.
- std::move is an unconditional cast to r-value. Does not actually move!
- std::move each member in your move operations
- move semantics appears in many places! e.g. swap
- rule of five: declare/delete all five. rule of zero: default if possible