Note about naming in lecture slides

Today, all instances of "vector" refer to the vector we wrote in namespace mycollection, not std::vector.

We will either refer to this as mycollection::vector, or simply put using namespace mycollection above without using namespace std.

Quiz Question

What is the output of the following code snippet?

```
int a = 1;
int* ptr = &a;
int* copy = ptr;
*copy = 2;
cout << *ptr << endl;
```

Special Member Functions

Where we're going...

CS 106B has covered the absolute barebones of C++ class design. We'll cover the rest.

We will cover the rest:

- template classes
- const correctness
- operator overloading
- **special member functions**
- move semantics
- RAII

New Overarching Topic!

Last few classes were all about making your classes more flexible through templates, operators, and const correct functions.

The next three lectures are all about resource management. How can objects of your class acquire, use, copy, and release resources?

What do we mean by a resource?

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.

Key questions we will answer today

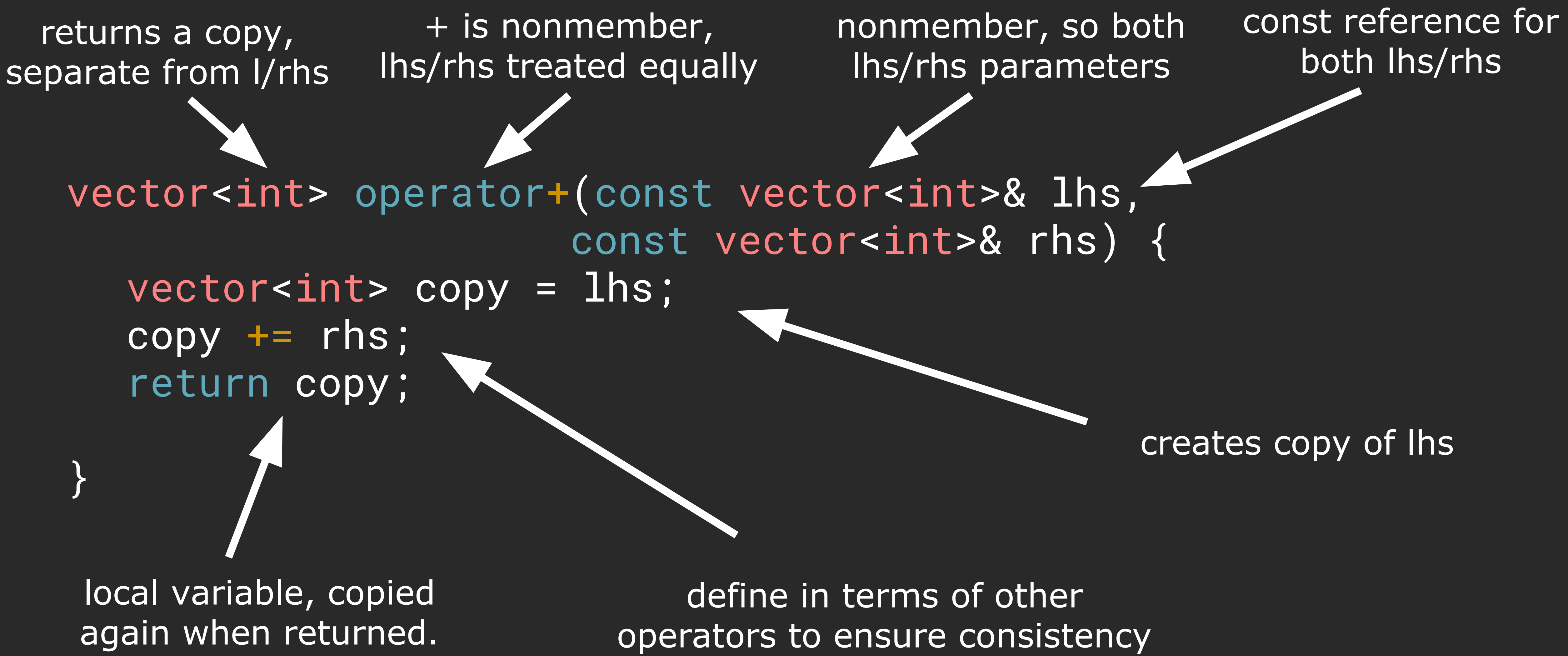
- What are special member functions? When are they called?
- When should we declare a special member function?
- When should we not declare a special member function?
- How can we make our special member functions more efficient?

Game Plan

- construction vs. assignment
- member initializer list
- copy operations
- default and delete
- rule of three
- copy elision

Operator review!

Design considerations when overloading operators!



```
vector<int> operator+(const vector<int>& lhs,
                      const vector<int>& rhs) {
   vector<int> copy = lhs;
   copy += rhs;
   return copy;

}
```

returns a copy, separate from l/rhs

+ is nonmember, lhs/rhs treated equally

nonmember, so both lhs/rhs parameters

const reference for both lhs/rhs

creates copy of lhs

local variable, copied again when returned.

define in terms of other operators to ensure consistency

Member vs. Non-member

MEMBER

1. Must: [], (), ->, =
2. Should: unary operators (++)
3. Both sides not equally treated (+=)

NON-MEMBER

1. lhs is prewritten type (<<)
2. Binary symmetric operator (+, ==, <)
3. Prefer non-friends to friends.

Both const and non-const members declared!

non-const reference, can be written over

must be member

called by non-const objects

```
T& vector<T>::operator[](size_t index) {
   // the same thing, or optionally static/const_cast trick
}

const T& vector<T>::operator[](size_t index) const {
   return _elems[index];
}
```

const reference cannot be written over

called by const objects

Principle of Least Astonishment (POLA)

“If a necessary feature has a high astonishment factor,
it may be necessary to redesign the feature”.

Summary of POLA

Operator semantics are very important!

- Should this be a member or a non-member (friend or not?)
- Should the parameters be const or not?
- Should the function be const or not?
- Should the return value be a reference or a const reference?
- What is the convention for overloading that operator?

Prefix vs. Postfix

returns reference to *this

unary operator, implement as member

non-const, since we change iterator position

```
iterator& iterator::operator++();    // prefix

iterator iterator::operator++(int);  // postfix
```

returns a copy of original pointer

used to distinguish between pre/postfix

Why support iterators?

```
vector<string> vec(3, “Hello”);
std::sort(vec.begin(), vec.end());
for (const auto& val : vec) {
  cout << val << ‘\n’;
}
```

iterators must support these operators

```
auto iter = v.begin();    // copy constructor
copy = iter;          // copy assignment
if (iter == copy)       // (in)equality
*iter;           // dereference
++iter;            // prefix increment
copy++;            // postfix increment
```

random access iterators are even more powerful

```
iter += 3;           // compound add
iter + 3;          // iter + size_t
iter1 - iter2;       // iter - iter
if (iter <= copy)      // comparison
--iter;            // prefix decrement
copy--;            // postfix decrement
```

How are iterator implemented?

A class that stores a "position" as well as information needed to iterate over another container.

Overloads a lot of operators!

We decided to not teach iterator implementation this quarter, but you can choose it as a final topic!

Example

Demonstrating the problem with the assignment operator.

Special Member Functions

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.

Which special member function is called with each line?

```
vector<int> function(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2);
  vector<int> vec6{};
  vector<int> vec7{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- 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.

Copy constructor when you pass by value***

***One exception - ask after class if interested.

```
vector<int> function(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2);
  vector<int> vec6{};
  vector<int> vec7{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- 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.

Default constructor creates empty vector.

```
vector<int> function(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2);
  vector<int> vec6{};
  vector<int> vec7{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- 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.

Fill constructor, creates vector {0, 0, 0}

See documentation for what the fill constructor is!

```
vector<int> function(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2);
  vector<int> vec6{};
  vector<int> vec7{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- 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.

Initializer_list constructor: creates vector {3}

See documentation - if declared, initializer list constructor has priority over other constructors.

```
vector<int> function(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2);
  vector<int> vec6{};
  vector<int> vec7{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- 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.

Trick question: this is a function declaration.

You'll (probably) get a compiler error. This is called C++'s most vexing parse.

```
vector<int> function(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2);
  vector<int> vec6{};
  vector<int> vec7{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- 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.

Copy constructor: object created as copy of another existing object.

```
vector<int> function(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2);
  vector<int> vec6{};
  vector<int> vec7{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- 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.

Default constructor!

```
vector<int> function(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2);
  vector<int> vec6{};
  vector<int> vec7{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- 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.

Copy constructor, even if existing object is a temporary object!

*** see next lecture for move constructors

```
vector<int> function(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2);
  vector<int> vec6{};
  vector<int> vec7{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- 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.

Copy constructor: vec8 is newly constructed.

Not a copy assignment! vec8 is not an existing object.

```
vector<int> function(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2);
  vector<int> vec6{};
  vector<int> vec7{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- 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.

Copy assignment: vec8 is existing object.

```
vector<int> function(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2);
  vector<int> vec6{};
  vector<int> vec7{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- 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.

Copy constructor: copies vec8 to location outside of function scope

There are smart optimizations around this that we'll talk about.

```
vector<int> function(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2);
  vector<int> vec6{};
  vector<int> vec7{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- 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.

Destructors on all vectors (except the return value) are called.

```
vector<int> function(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2);
  vector<int> vec6{};
  vector<int> vec7{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- 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.

Which one is better?

```
vector<T> vec;
vec = {1, 2, 3, 4, 5};
```

```
vector<T> vec = {1, 2, 3, 4, 5};
```

Which one is better?

```
vector<T> vec;
vec = {1, 2, 3, 4, 5};
// default constructor, then copy assignment
// internally two arrays have to be freed in total


vector<T> vec = {1, 2, 3, 4, 5};
// copy constructor
```

Member Initialization List

```
template <typename T>
vector<T>::vector<T>() {
   _size = 0;
   _capcity = kInitialSize;
   _elems = new T[kInitialSize];
}
```

Members are first default constructed.

Then each member is reassigned.

Member Initialization List

```
template <typename T>
vector<T>::vector<T>() {
   _size = 0;
   _capcity = kInitialSize;
   _elems = new T[kInitialSize];
}
```

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

Directly construct each member with starting value.

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]) { }
```

Example

Adding member initialization lists to our vector and StreamingMedianTracker classes.

Concept Check

Why use a member initialization list?

In which special member functions can you use a member initialization list?

Concept Check

Faster (directly construct rather than construct then reassign). Sometimes is the only way that works.

Anything that is a constructor. Doesn't have to be a special member function.

Key Idea

Use member initialization lists for all of your constructors!

Example

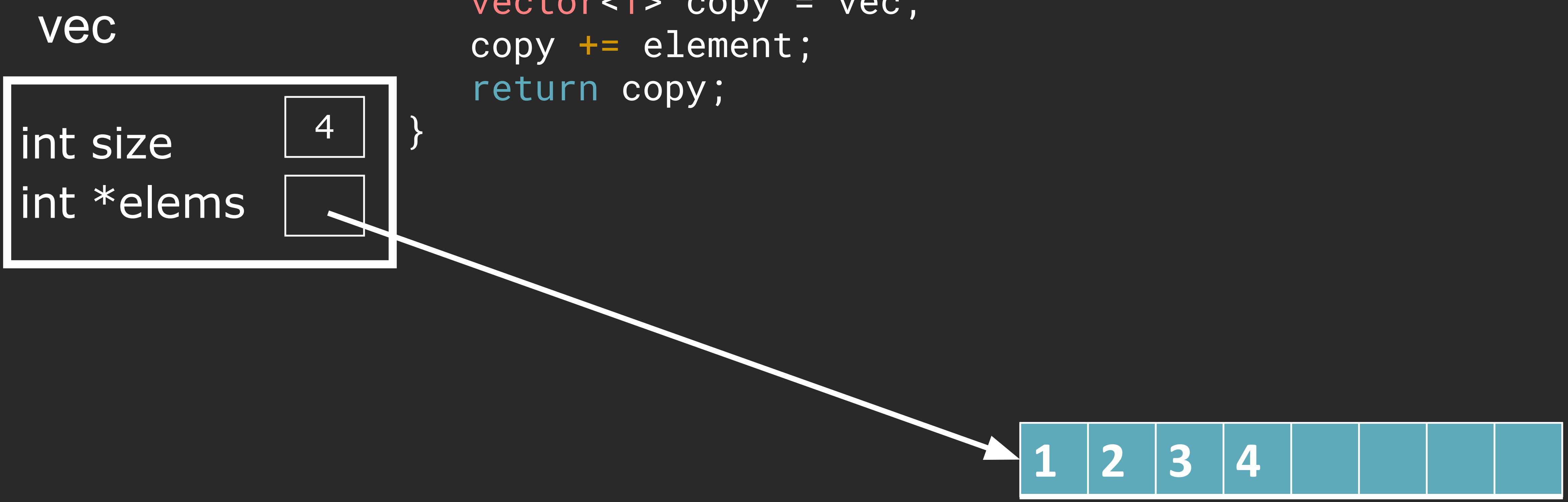
Let's run our current vector and see it crash!

I lied…this code doesn’t actually work.

```
template <typename T>
vector<T> operator+(const vector<T>& vec,
                    const T& elem) {
  vector<T> copy = vec;
  copy += element;
  return copy;
}
```

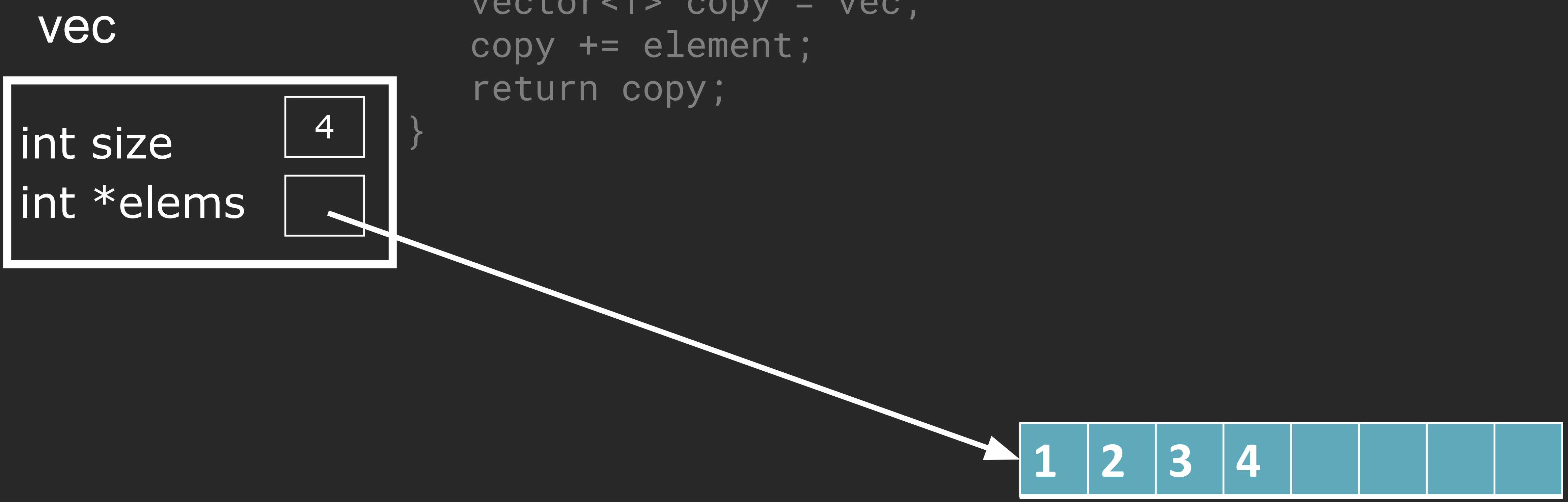
Copy is not as simple as copying each member.

```
template <typename T>
vector<T> operator+(const vector<T>& vec, const T& elem) {
    vector<T> copy = vec;
    copy += element;
    return copy;
}
```



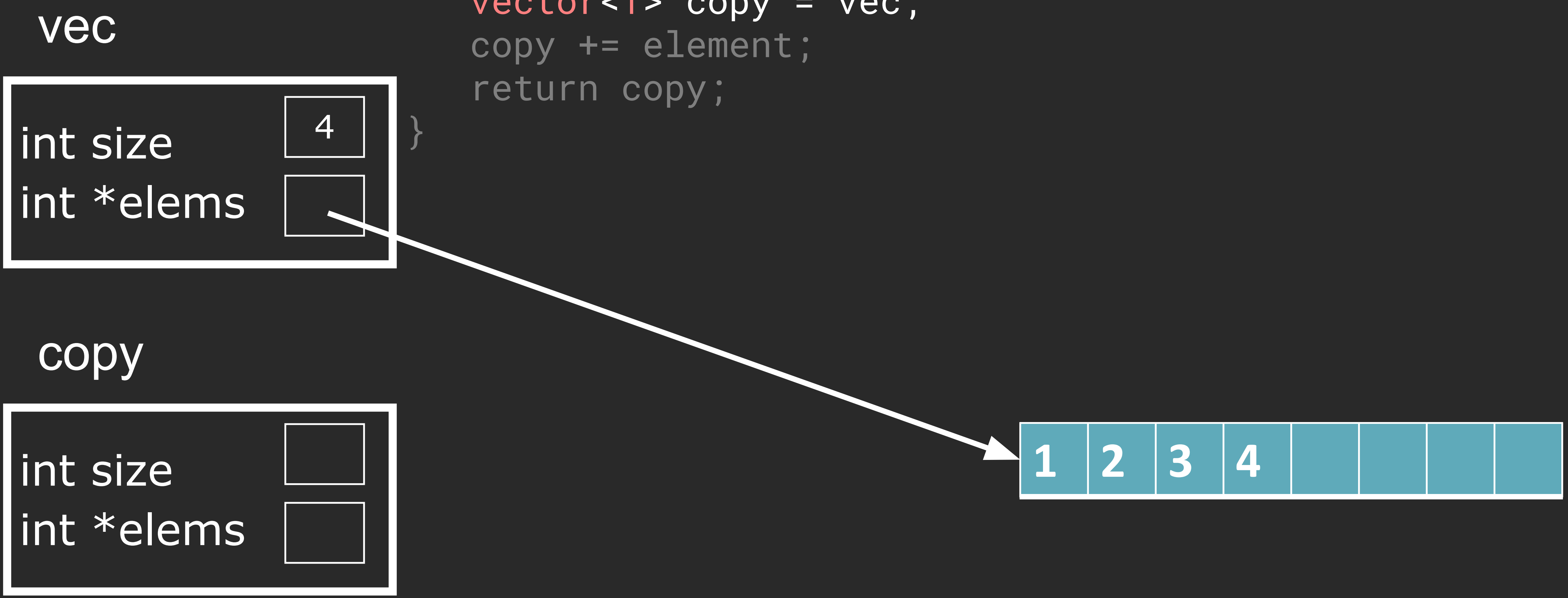
Copy is not as simple as copying each member.

```
template <typename T>
vector<T> operator+(const vector<T>& vec, const T& elem) {
   vector<T> copy = vec;
   copy += element;
   return copy;
}
```



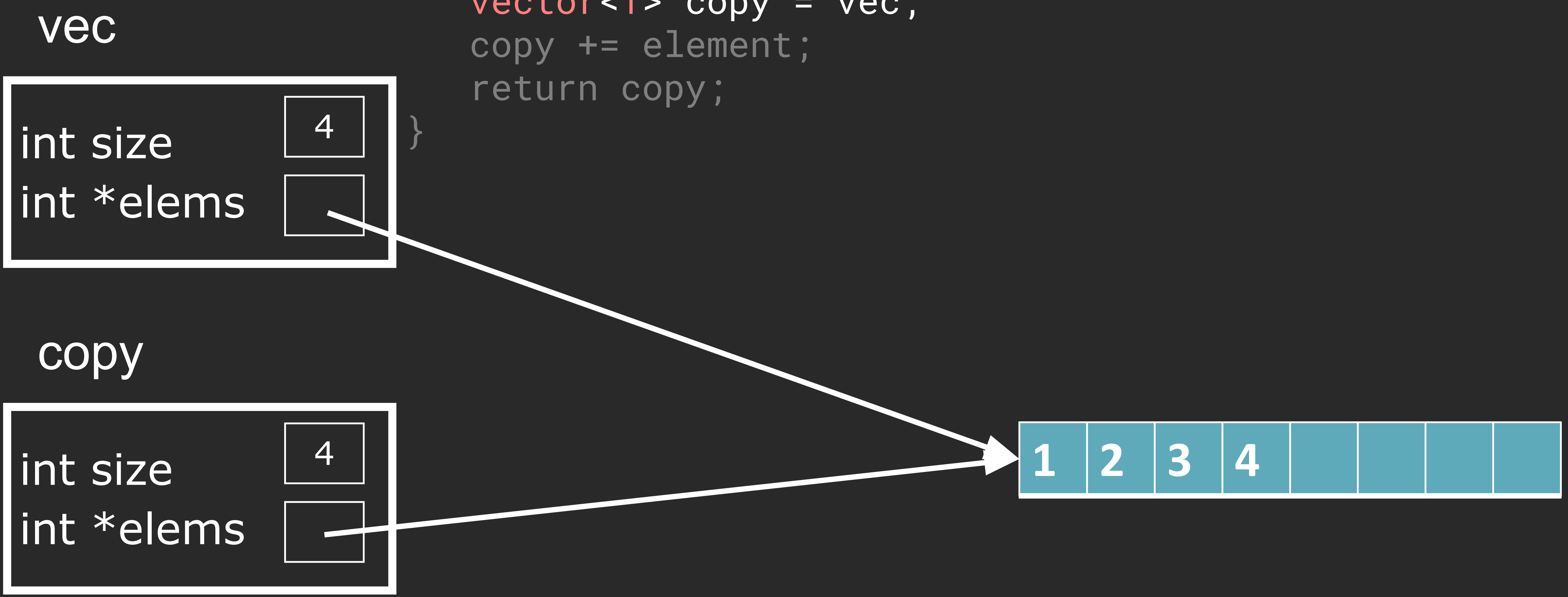
Copy is not as simple as copying each member.

```
template <typename T>
vector<T> operator+(const vector<T>& vec, const T& elem) {
    vector<T> copy = vec;
    copy += element;
    return copy;
}
```



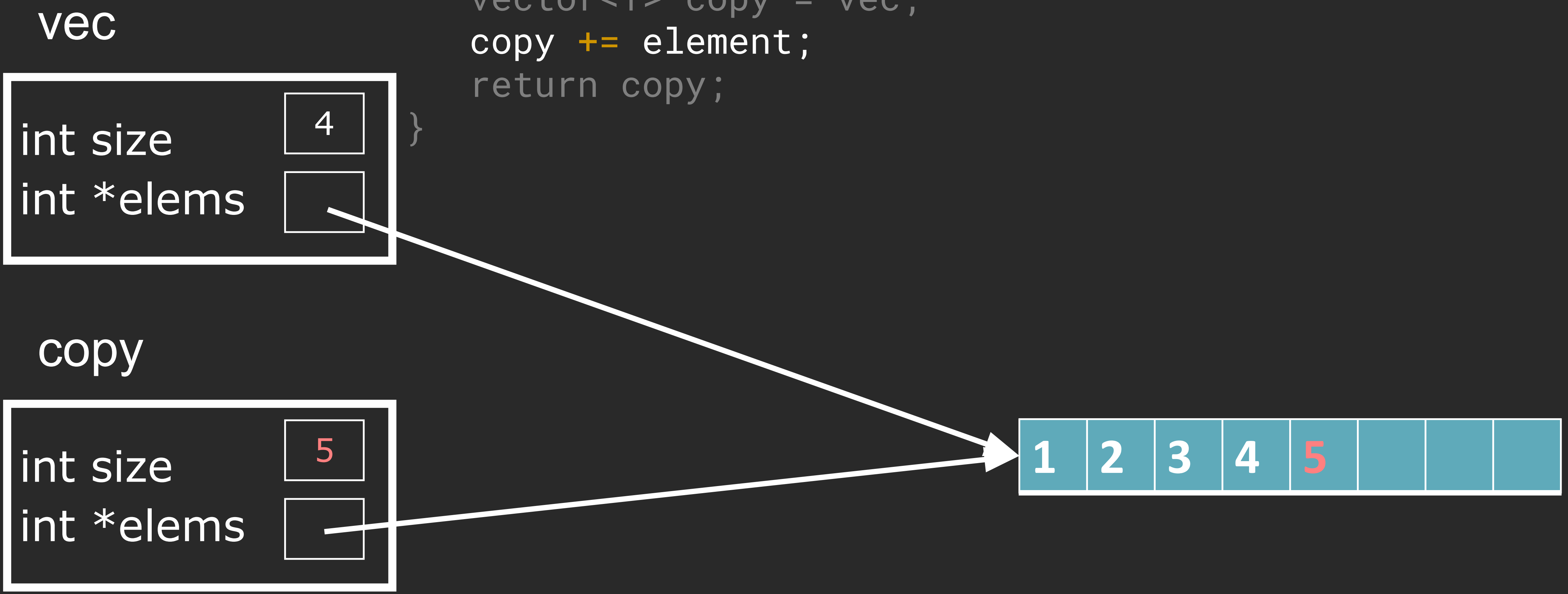
Copy is not as simple as copying each member.

```
template <typename T>
vector<T> operator+(const vector<T>& vec, const T& elem) {
    vector<T> copy = vec;
    copy += element;
    return copy;
}
```



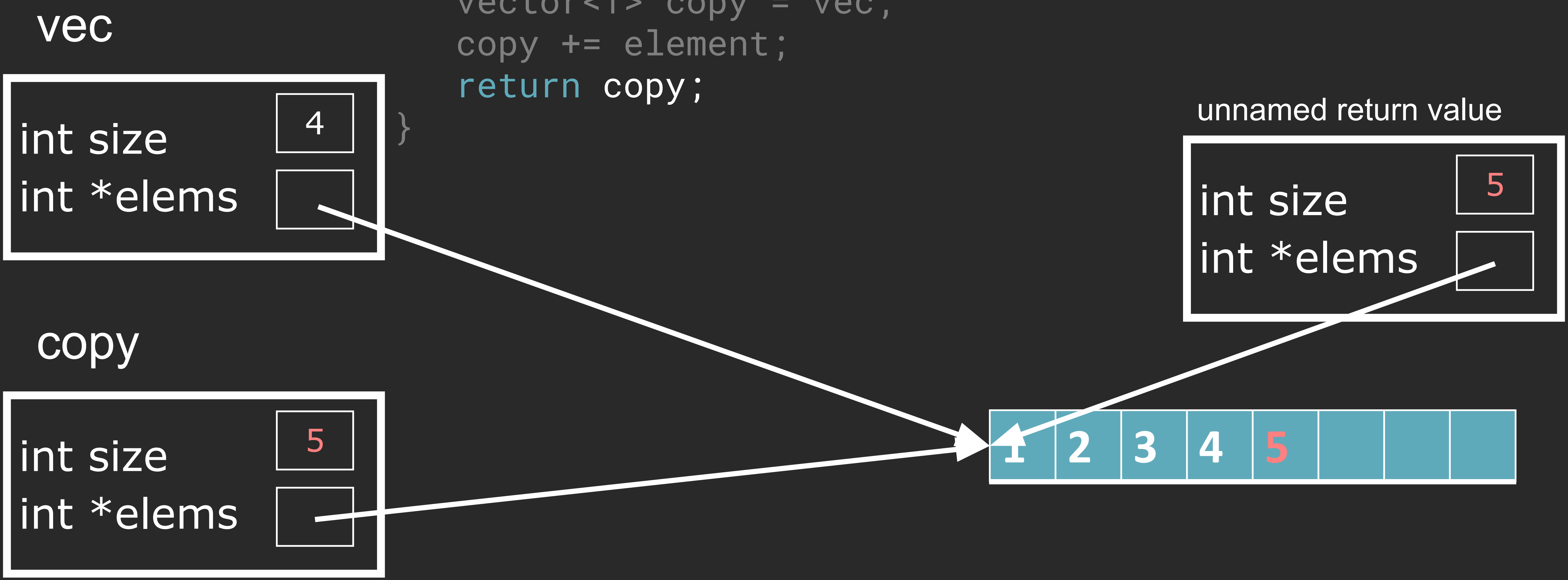
Copy is not as simple as copying each member.

```
template <typename T>
vector<T> operator+(const vector<T>& vec, const T& elem) {
    vector<T> copy = vec;
    copy += element;
    return copy;
}
```



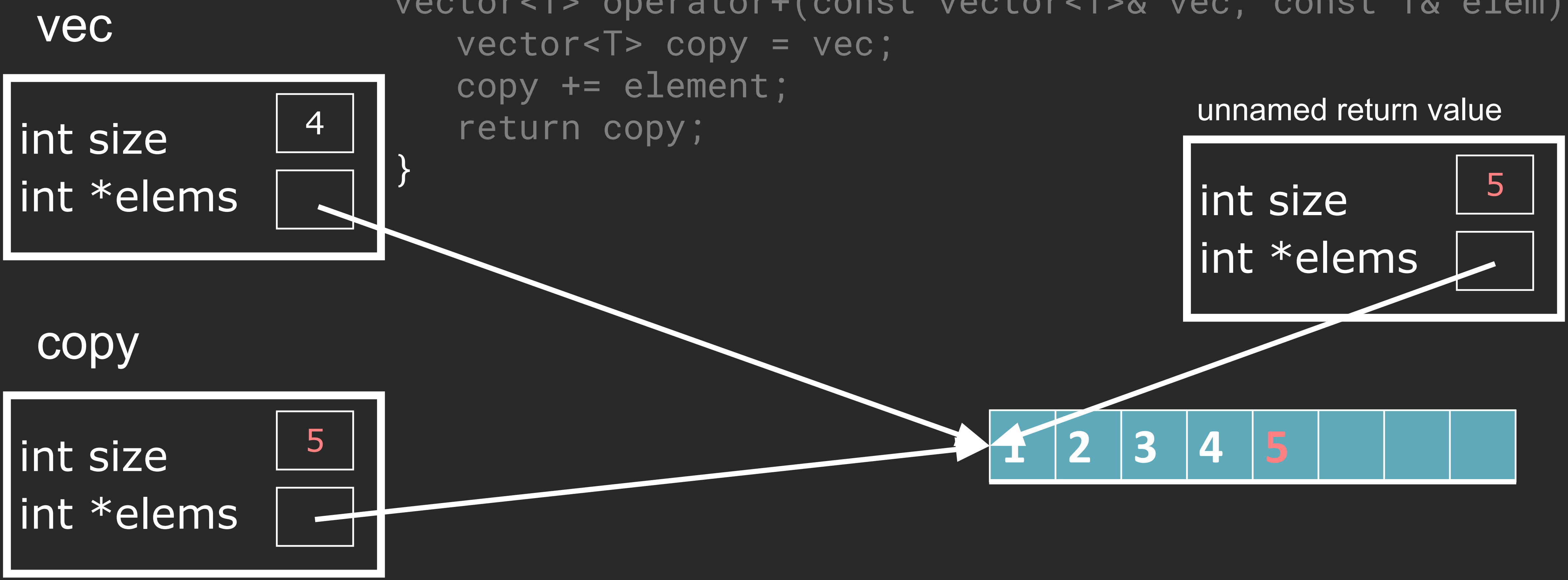
At return statement, copy of local variable made.

```
template <typename T>
vector<T> operator+(const vector<T>& vec, const T& elem) {
   vector<T> copy = vec;
   copy += element;
   return copy;
}
```



When the vectors go out of scope, their destructor tries to free the array.

```
template <typename T>
vector<T> operator+(const vector<T>& vec, const T& elem) {
   vector<T> copy = vec;
   copy += element;
   return copy;
}
```



vec

int size 4

int *elems

copy

int size 5

int *elems

unnamed return value

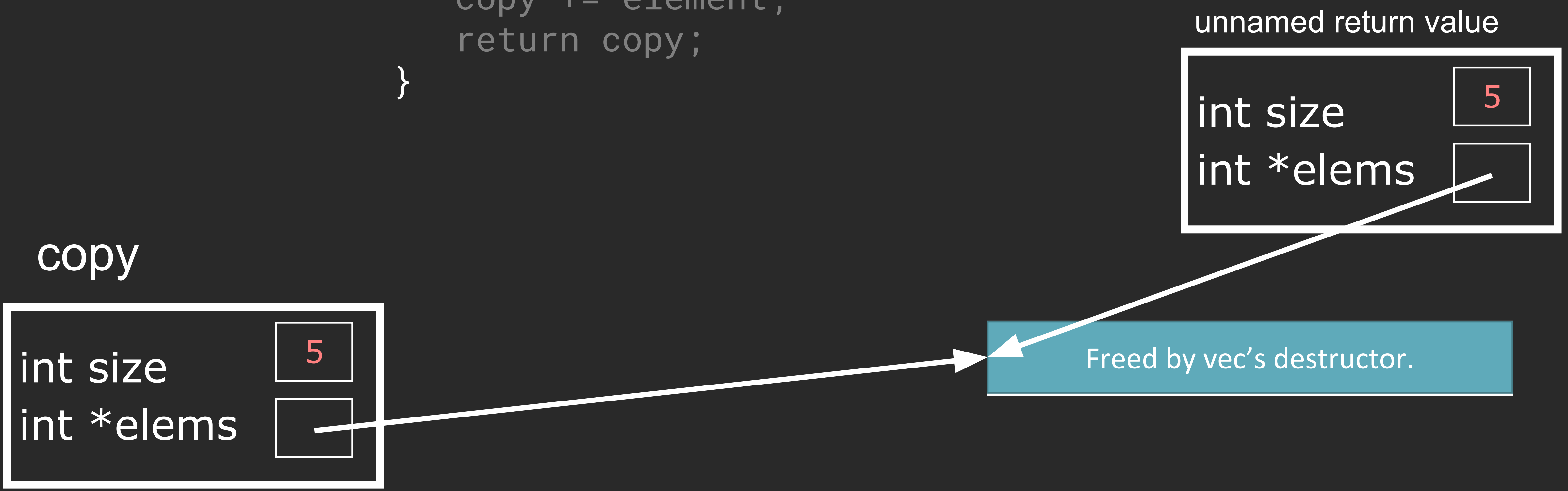
int size 5

int *elems

1 2 3 4 5

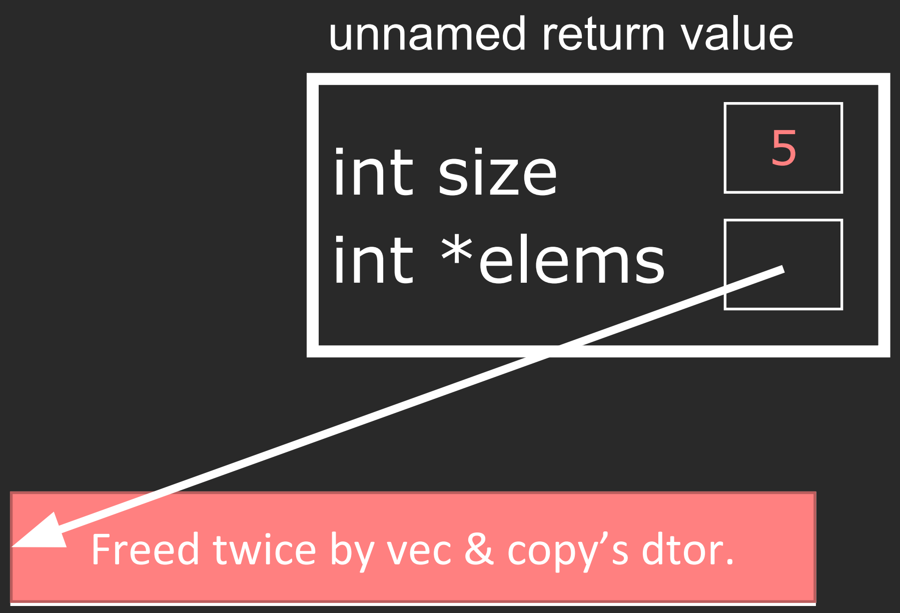
When the vectors go out of scope, their destructor tries to free the array.

```
template <typename T>
vector<T> operator+(const vector<T>& vec, const T& elem) {
   vector<T> copy = vec;
   copy += element;
   return copy;
}
```



When the vectors go out of scope, their destructor tries to free the array.

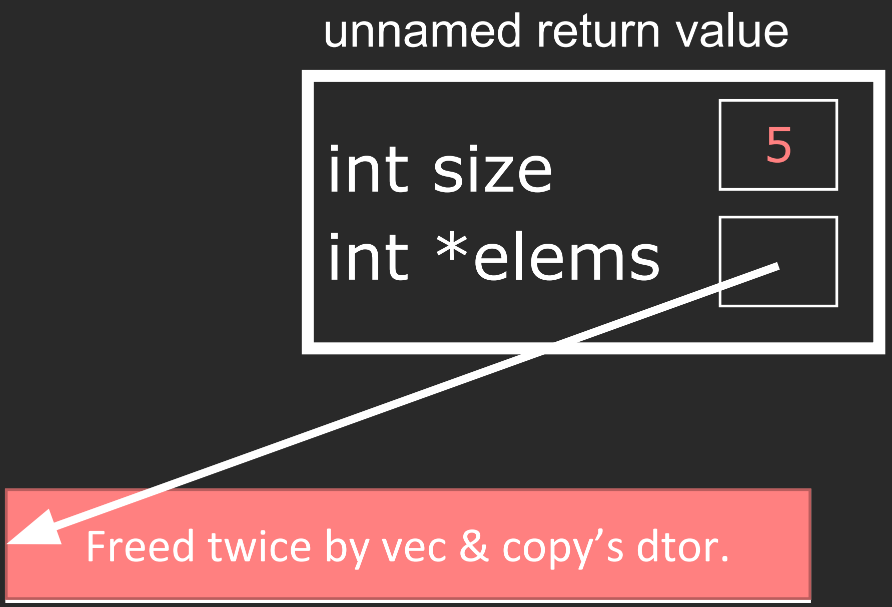
```
template <typename T>
vector<T> operator+(const vector<T>& vec, const T& elem) {
   vector<T> copy = vec;
   copy += element;
   return copy;
}
```



Freed twice by vec & copy’s dtor.

Problems: double free and return value has a dangling pointer.

```
template <typename T>
vector<T> operator+(const vector<T>& vec, const T& elem) {
   vector<T> copy = vec;
   copy += element;
   return copy;
}
```



Freed twice by vec & copy’s dtor.

The problem is this copy operation.

```
template <typename T>
vector<T> operator+(const vector<T>& vec, const T& elem) {
    vector<T> copy = vec;
    copy += element;
    return copy;
}
```

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.

Concept Check

Why are all the _elem pointers of each copy pointing to the same array?

Concept Check

When you copy a pointer, you copy the addresses you saved in the variable, not the actual array the pointer is pointing to.

Key Idea

Take special care copying members which are handles (eg. pointers) to other resources (eg. memory).
Are you actually copying the resource or the handle?

copy operations

From this point forward, "special member functions" do not include the default ctor.

- 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.

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.

How do we fix this?

The other ones are fine!

```
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.

How do we fix this?

The other ones are fine!

```
template <typename T>
vector::vector<T>(const vector<T>& other) :
     _size(other._size,
     _capacity(other._capacity),
     _elems(other._elems) {
     _elems = new T[other._capacity];
     std::copy(other._elems,
               other._elems + other._size, _elems);
}
```

Copy it ourselves!

How do we fix this?

```
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);
}
```

Instead of constructing _elems then reassigning, directly construct.

Copy it ourselves!

What's wrong with the default assignment?

```
template <typename T>
vector<T>& vector<T>::operator=(const vector<T>& other)
     _size = other._size;
     _capacity = other._capacity;
     _elems = other._elems;




}
```

What's wrong with the default assignment?

```
template <typename T>
vector<T>& vector<T>::operator=(const vector<T>& other)
     _size = other._size;
     _capacity = other._capacity;
     _elems = other._elems;




}
```

We're copying a pointer again!

There's still a problem!

```
template <typename T>
vector<T>& vector<T>::operator=(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!

Memory leaks!

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

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

}
```

What about the old array that
_elems pointed to?

Use a temporary pointer, copy things over, free the old array, then reassign the pointer.

```
template <typename T>
vector<T>& vector<T>::operator=(const vector<T>& other)
     _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;
}
```

What about the old array that _elems pointed to?

Use a temporary pointer, copy things over, free the old array, then reassign the pointer.

```
template <typename T>
vector<T>& vector<T>::operator=(const vector<T>& other)
    _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;
}
```

Return reference to the vector itself.

Be careful about an edge case: self-assignment.

```
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!

Concept Check

What are the things you have to do in an assignment operator?

Be careful about an edge case: self-assignment.

```
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;
}
```

Check for self-assignment.

```
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;
}
```

Copy all members that can be automatically copy assigned.

```
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;
}
```

Think about when you are freeing existing members.

```
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;
}
```

Manually copy all other members.

```
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;
}
```

Return a reference to itself.

```
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;
}
```

Concept Check

What are the things you have to do in an assignment operator?

1. Check for self-assignment.
2. Make sure you to when to free existing members.
3. Copy assign each automatically assignable member.
4. Manually copy all other members.
5. Return a reference to itself (that was just reassigned).

Key Idea

In assignment, you must free the existing resources of the object before you overwrite it!

2-min stretch break!

Announcements

Logistics

- Summary of Projects in CS 106L
  - Optional project 1 (GraphViz): already released
  - Optional project 2 (Interviews): will release later this week
  - Required project 1 (Wiki Racer): part A/B released week 3/4, due week 6
  - Required project 2 (HashMap): released week 7, due week 10 Sunday
  - Optional project 3 (Gap Buffer): released week 8
  - Optional project 4 (K-d Trees): released week 9

- Each required project is around 120 lines of code
- Assignment 2 formally released on Wednesday. You can see a preview on Github right now.

Assignment 2: HashMap

- You will be extending a minimal implementation of a HashMap and applying every single concept from our class design unit.
- You can do the assignment with a partner!
- There will be some background reading, coding, and short answer questions.

Assignment 2: HashMap

- Default project: HashMap
  - You can use Qt Creator to work on the project.
  - You can use the command line to work on the project.
  - You can use any other IDE you want, but we'll only formally support Qt Creator and the command line.
  - Choose the one that causes you the least headache.
  - A test harness containing 16 test cases is provided, which you must pass to receive credit.

Assignment 2: HashMap

- Required:
  - Milestone 0: learn about hashing, read the starter code
  - Milestone 1: finish two functions erase() and 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

Assignment 2: HashMap

- Custom project: your choice!
  - Design a class on anything that interests you. For example, more advanced data structure, a data processing system, a better web scraper, etc.
  - Must be a template class that is const-correct, supports operators, correct copying and move semantics.
  - Will certainly be *much* more work than the default project.
  - We can help with C++-specific things, but our support overall will be fairly limited.
  - Make a post on Piazza to get approval first!
  - Grading will focus mostly on design/effort rather than strictly on correctness.

Back to Special Member Functions

default + delete

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
}
```

The default copy constructor for our StreamingMedian class works!

```
template <typename T>
StreamMedian::StreamMedian<T>(const StreamMedian<T>& other) :
     _elems(other._elems), _comp(other._comp) {



}
```

Type of _elems is our own vector class, which has a good copy constructor now!

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

Key Idea

You can delete/default each special member function.
Allows you to control when they can be called.

Questions

rule of zero / rule of three

Rule of Zero

If the default copy constructor, assignment, and destructor work, then use the default ones and don't declare your own.

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

Our declaration was technically incorrect...

```
template <typename T>
StreamMedian::StreamMedian<T>(const StreamMedian<T>& other) :
     _elems(other._elems) {



}
```

We forgot to add some optimizations
(e.g. move semantics)

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 :)

Concept Check: rule of zero

When would you not need to declare your copy constructor, assignment, or destructor?

Concept Check: rule of zero

When would you not need to declare your copy constructor, assignment, or destructor?

When all your members are…
(1) primitive types
(2) types which have correct copy operations and destructor.

Concept Check: rule of zero

Which types have the correct copy operations and destructors?

Every STL library is correct.

Clients of your class will assume *your* class is correct. Great power = great responsibility.

When do you need to write your own special member functions?

When the default one generated by the compiler does not work.

Most common reason: ownership issues
A member is a handle on a resource outside of the class.
(eg: pointers, mutexes, filestreams)

Rule of Three

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

What’s the rationale?

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.

Questions

Key Idea

If possible, don't declare special member functions.
Otherwise, you have an ownership issue, so make sure you
resolve them correctly.

problems with copying

How many vectors are created?

```
int main() {
  vector<string> words;
  words = findAllWords(“words.txt”);
  // print words
}

vector<string> findAllWords(const string& filename) {
  vector<string> words;
  // read from filename using an ifstream
  return words;
}
```

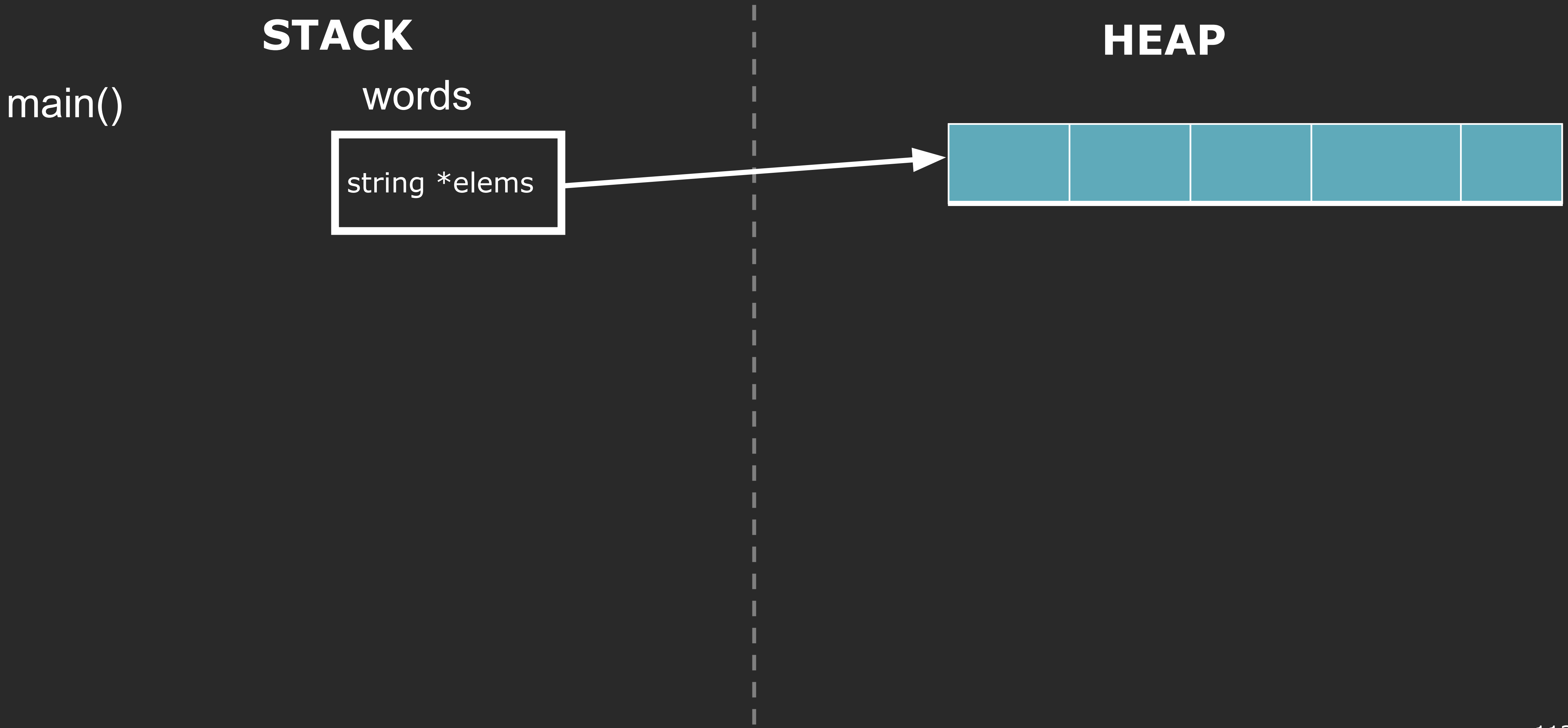
How many vectors are created?

STACK

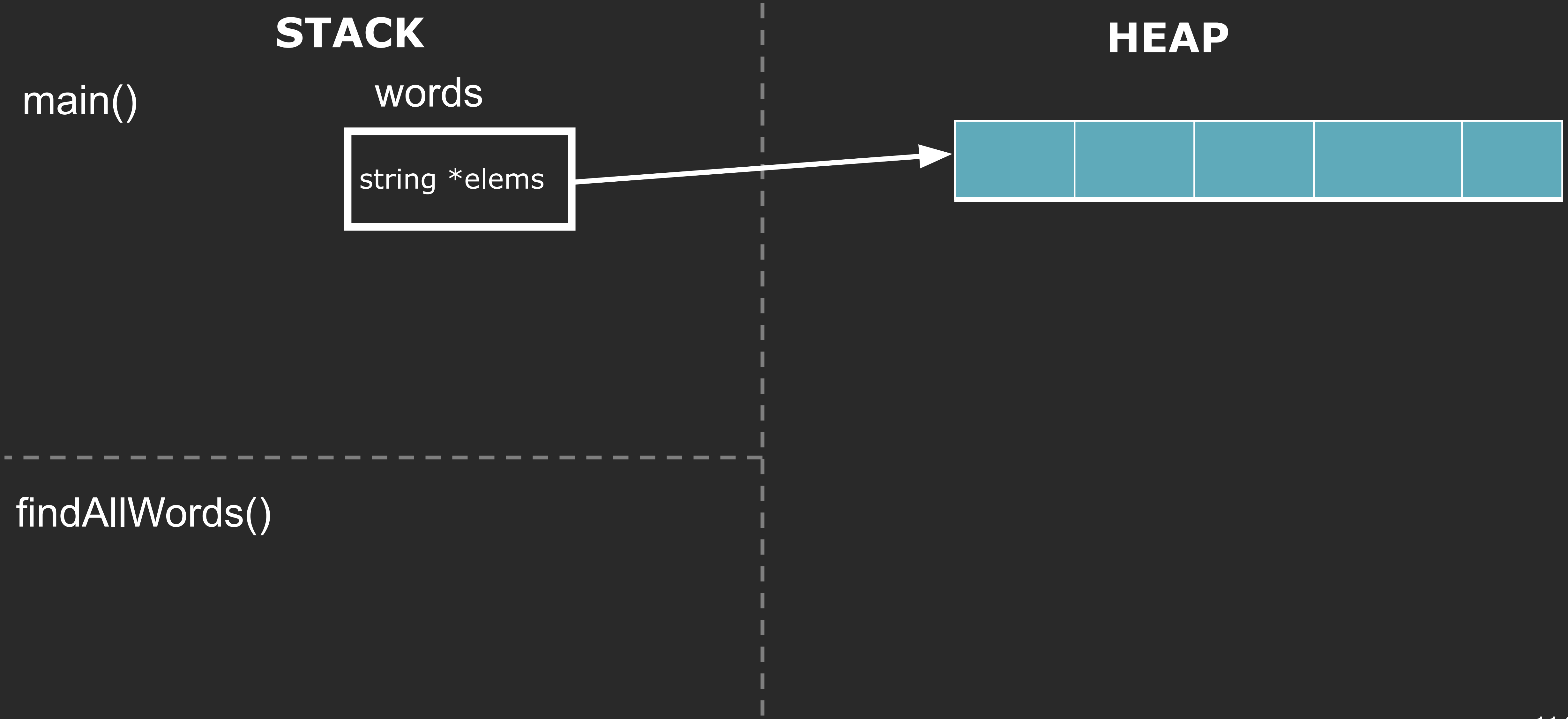
HEAP

main()

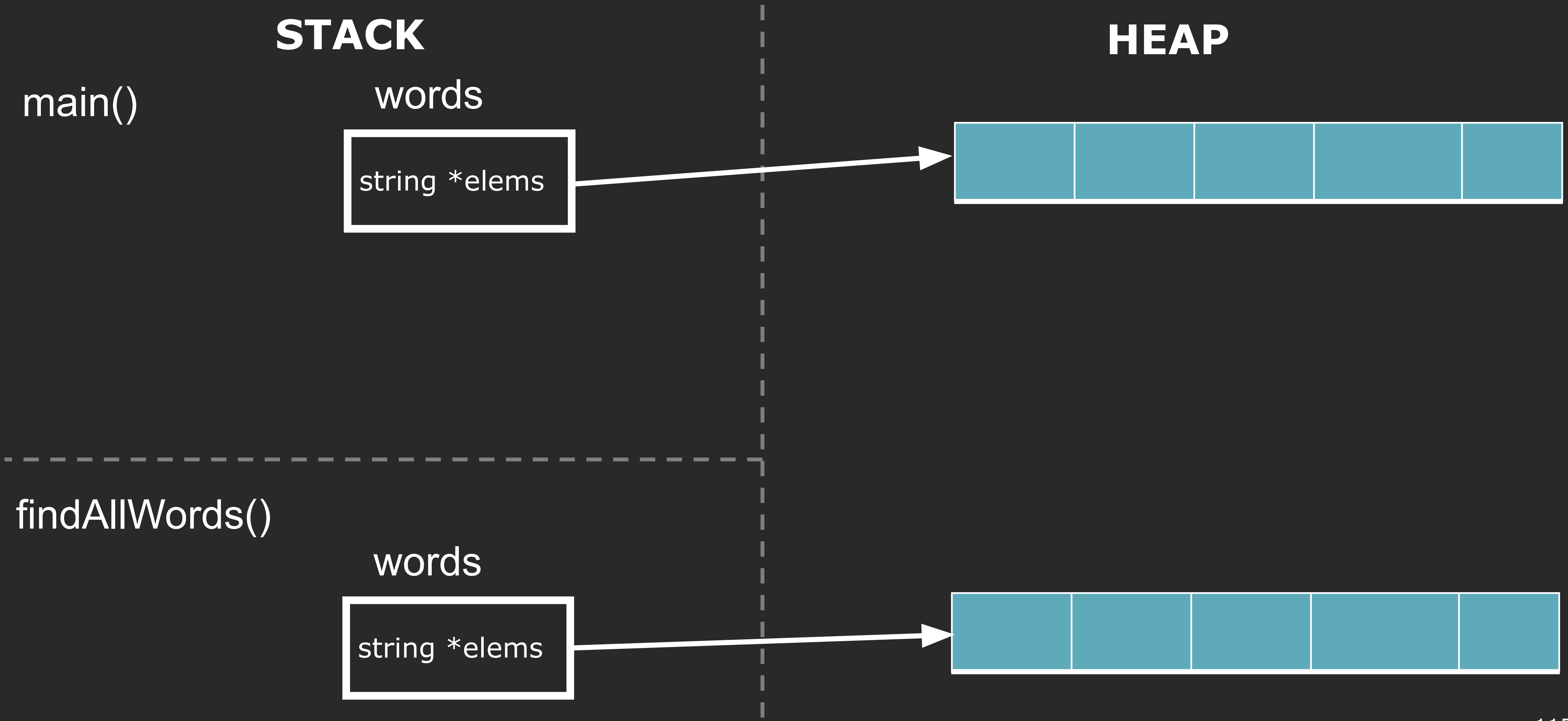
How many vectors are created?



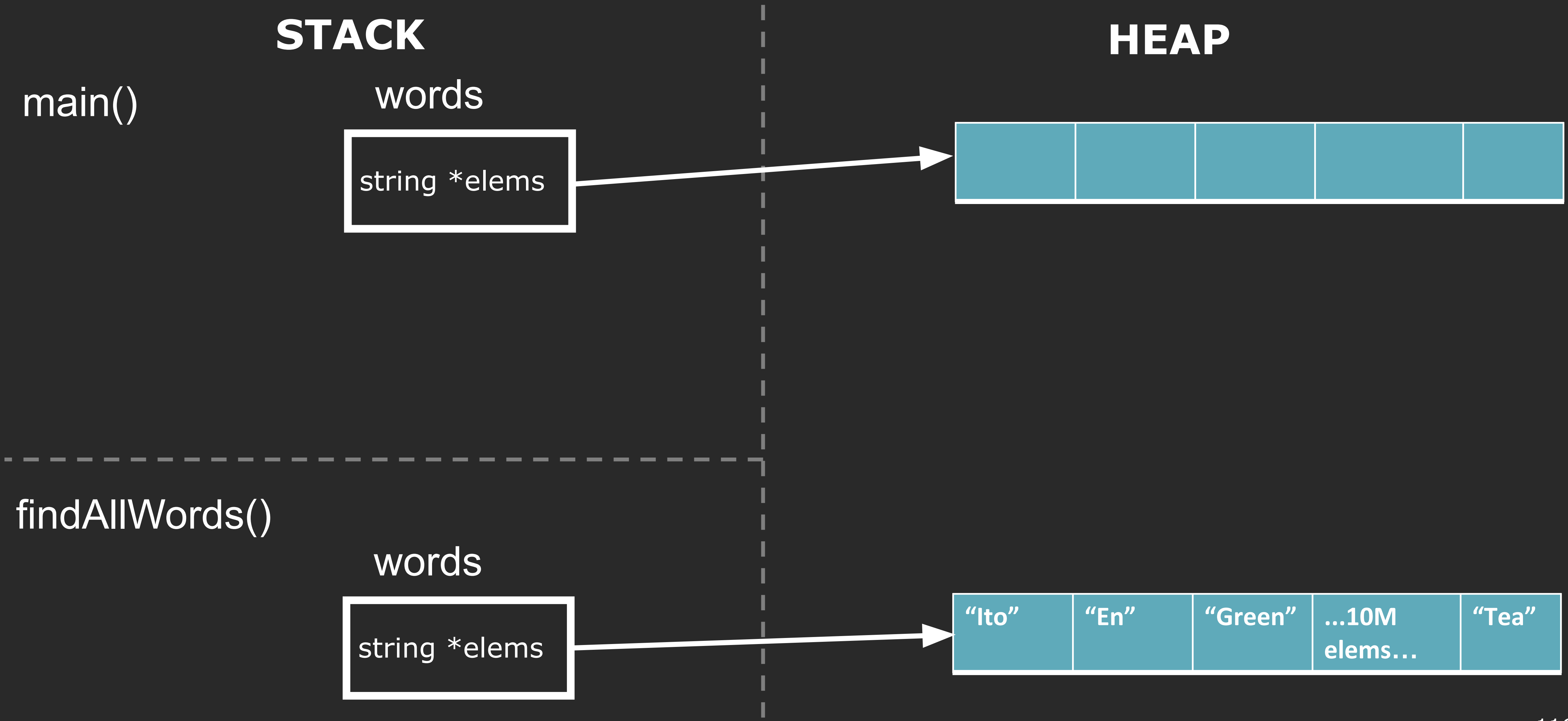
How many vectors are created?



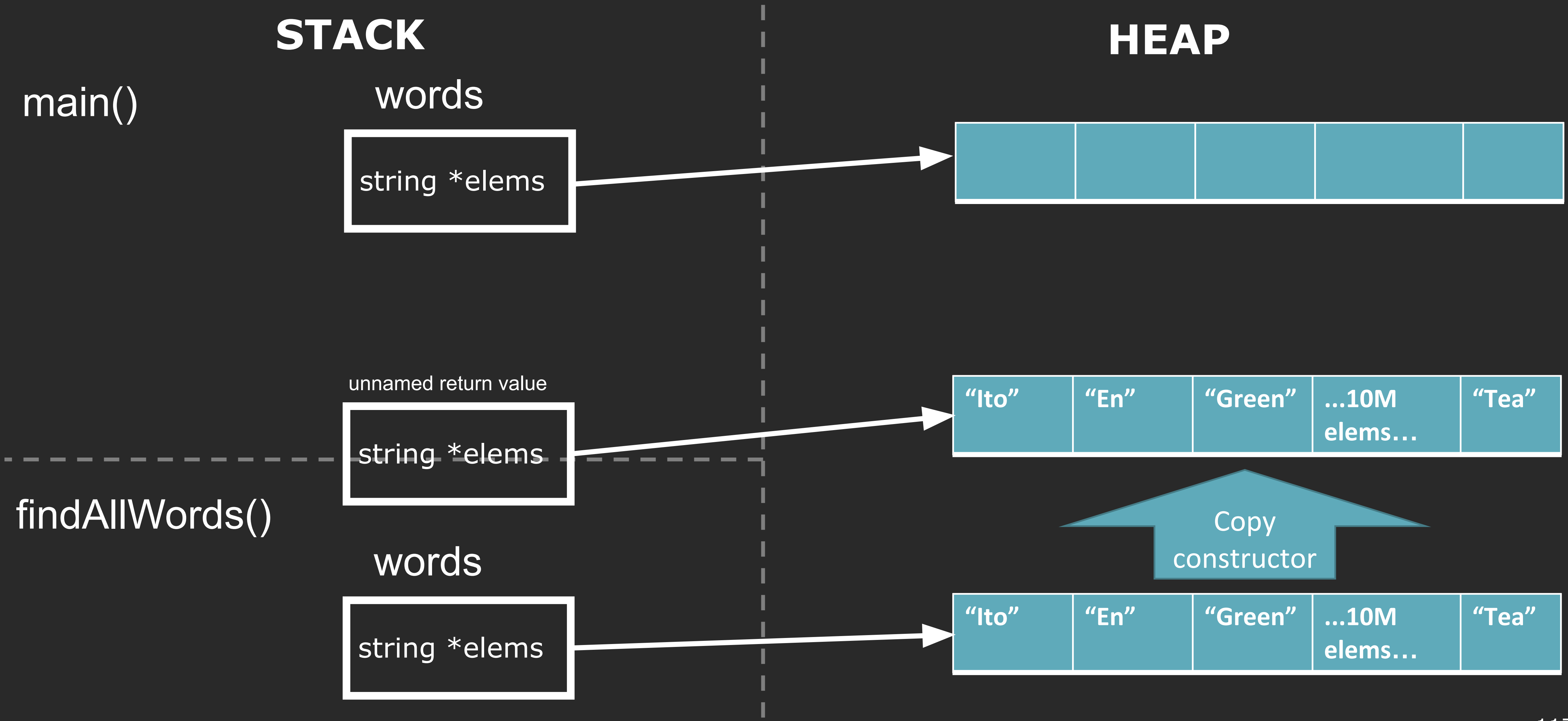
How many vectors are created?



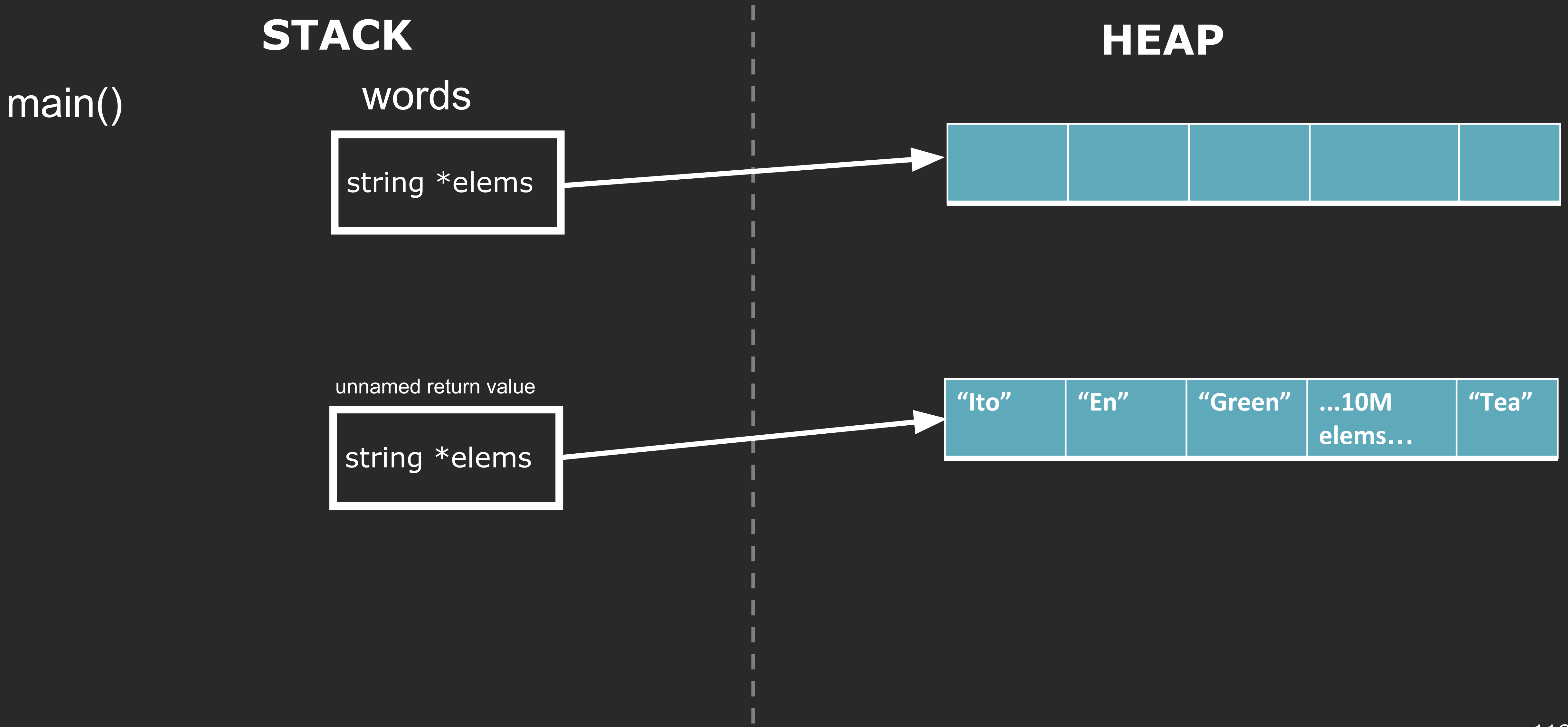
How many vectors are created?



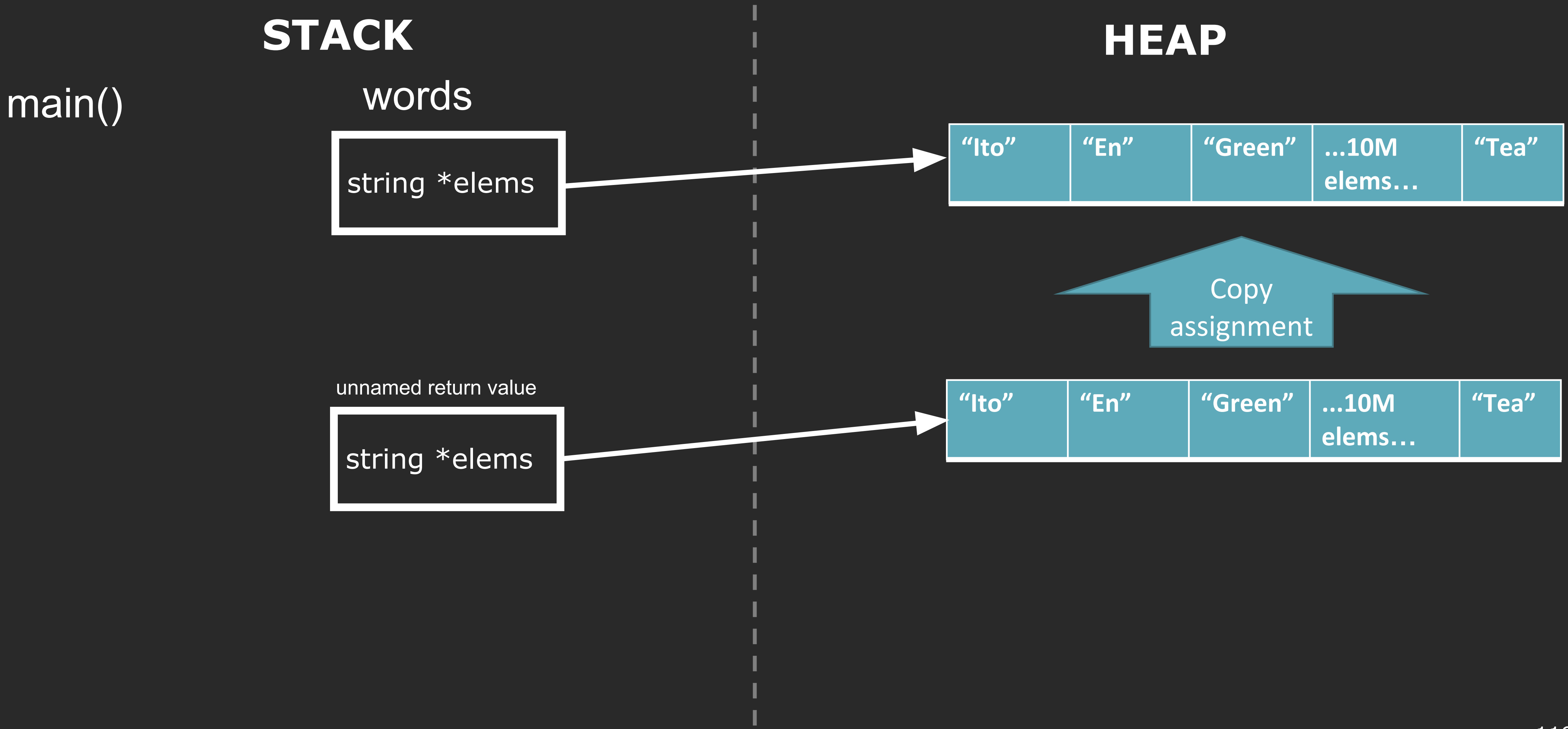
How many vectors are created?



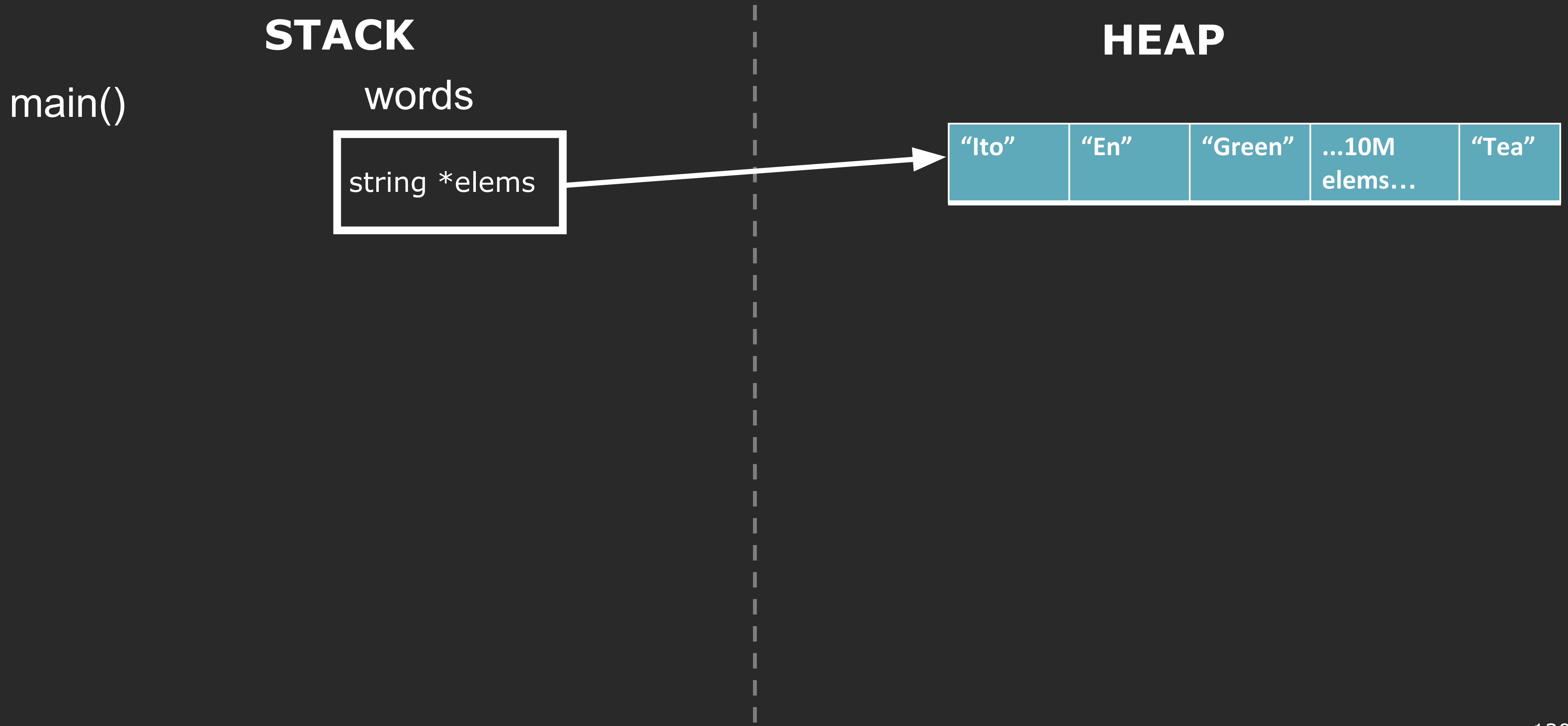
How many vectors are created?



How many vectors are created?



How many vectors are created?



STACK

HEAP

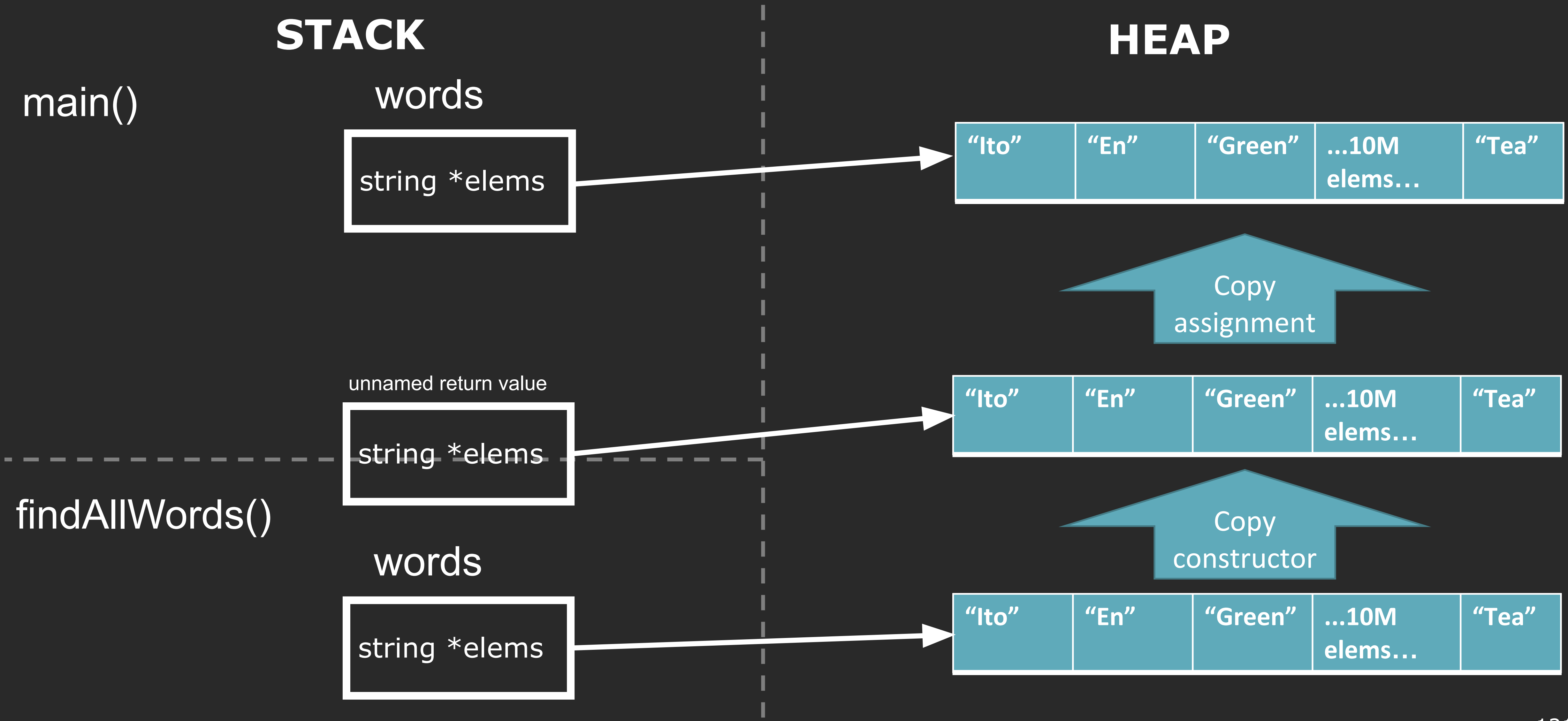
main()

words

string *elems

“Ito” | “En” | “Green” | ...10M elems… | “Tea”

That is a lot of copies.



copy elision and return value optimization (RVO)

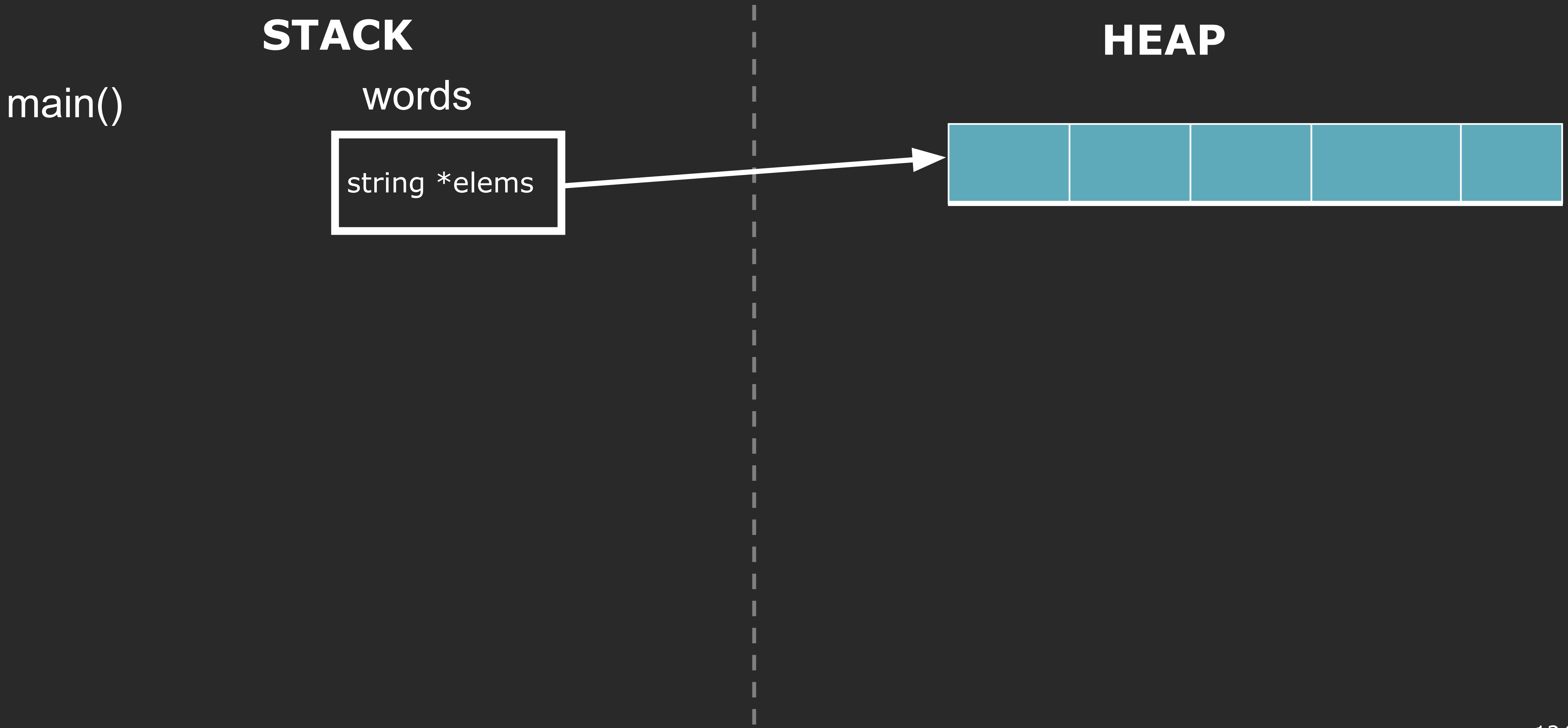
In practice: copy elision.

STACK

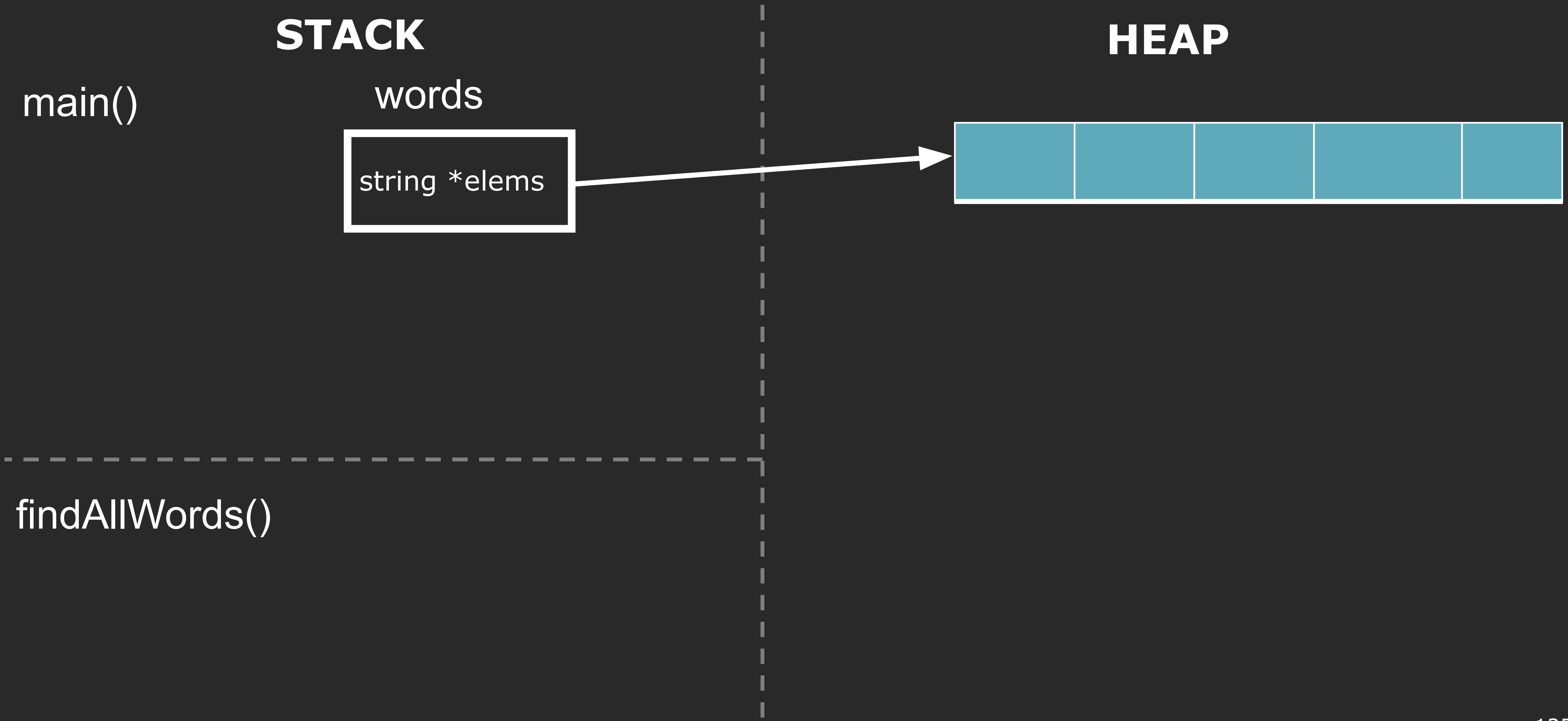
main()

HEAP

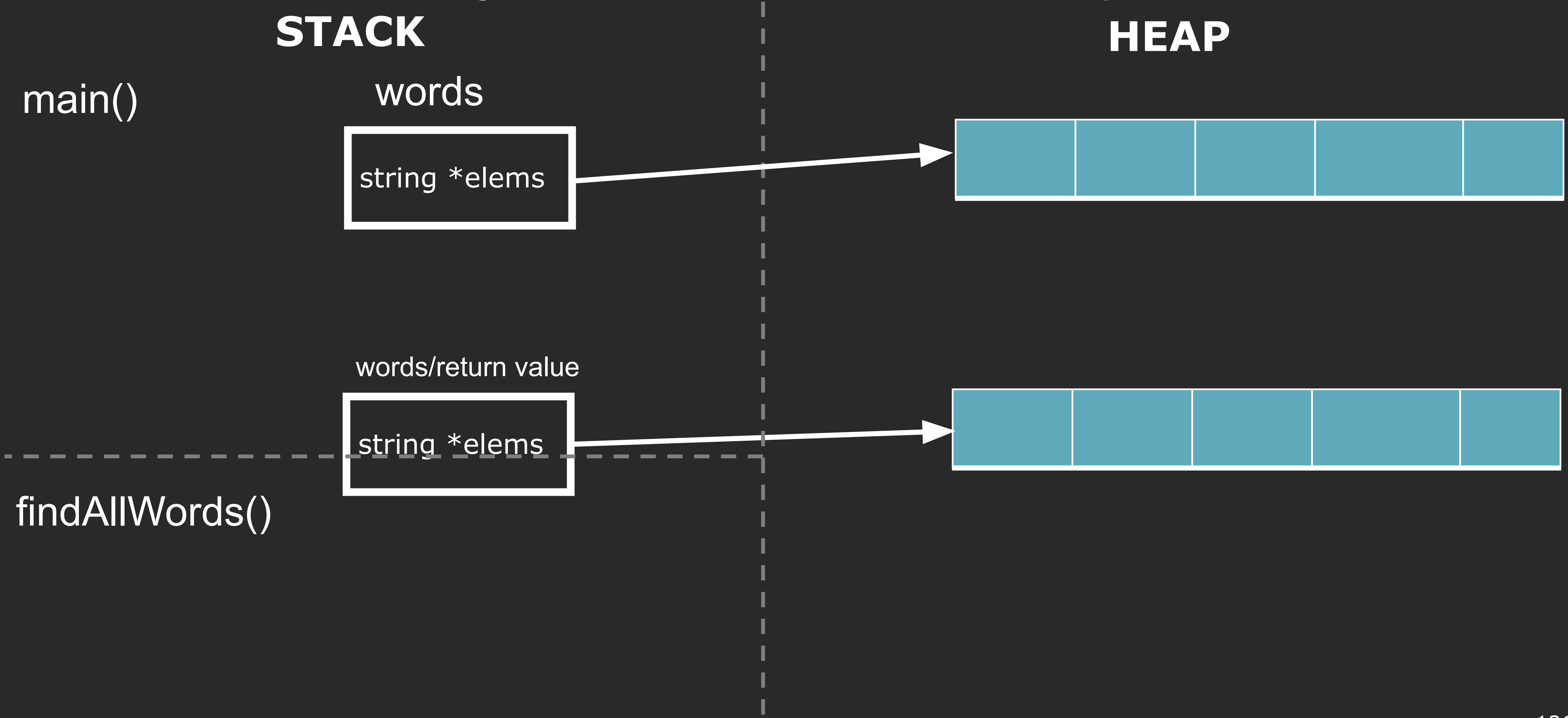
In practice: copy elision.



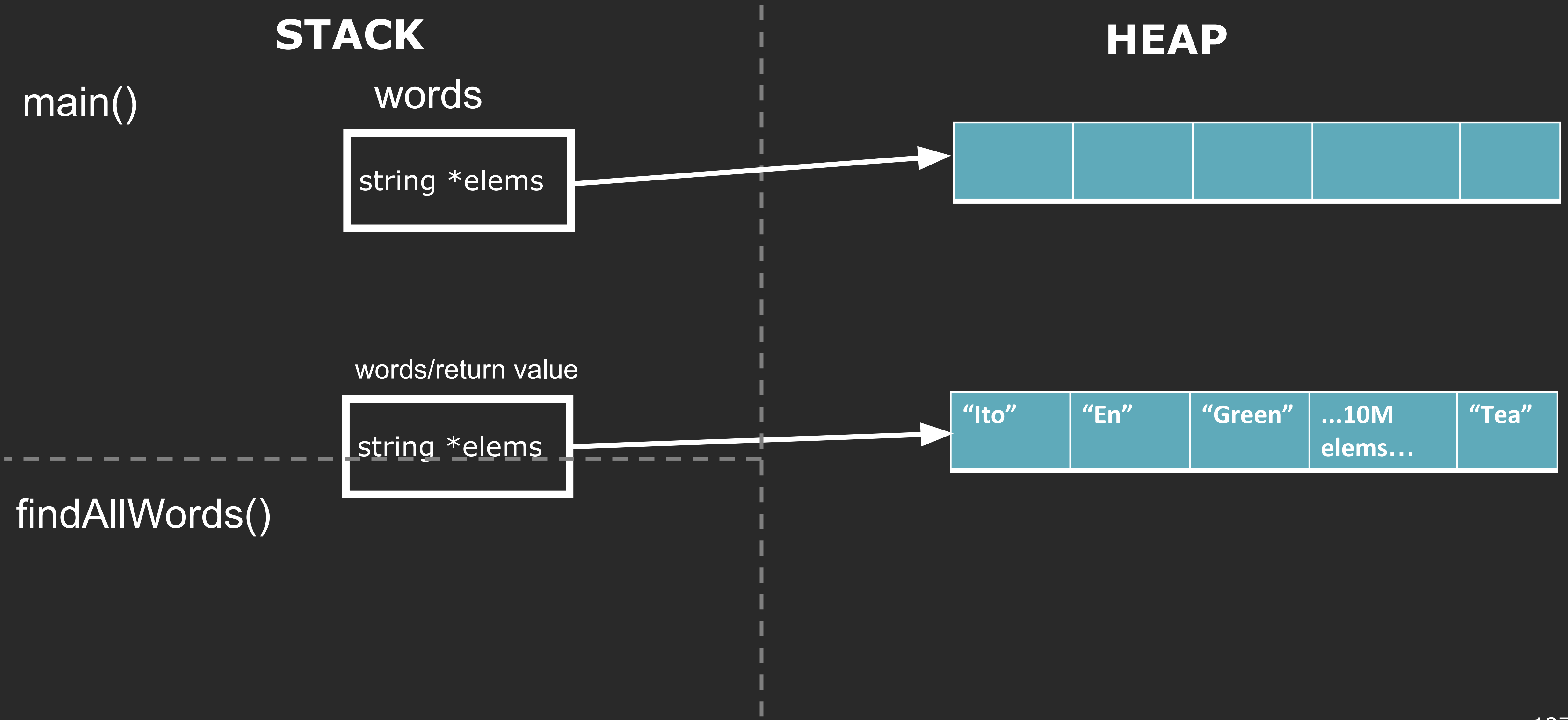
In practice: copy elision.



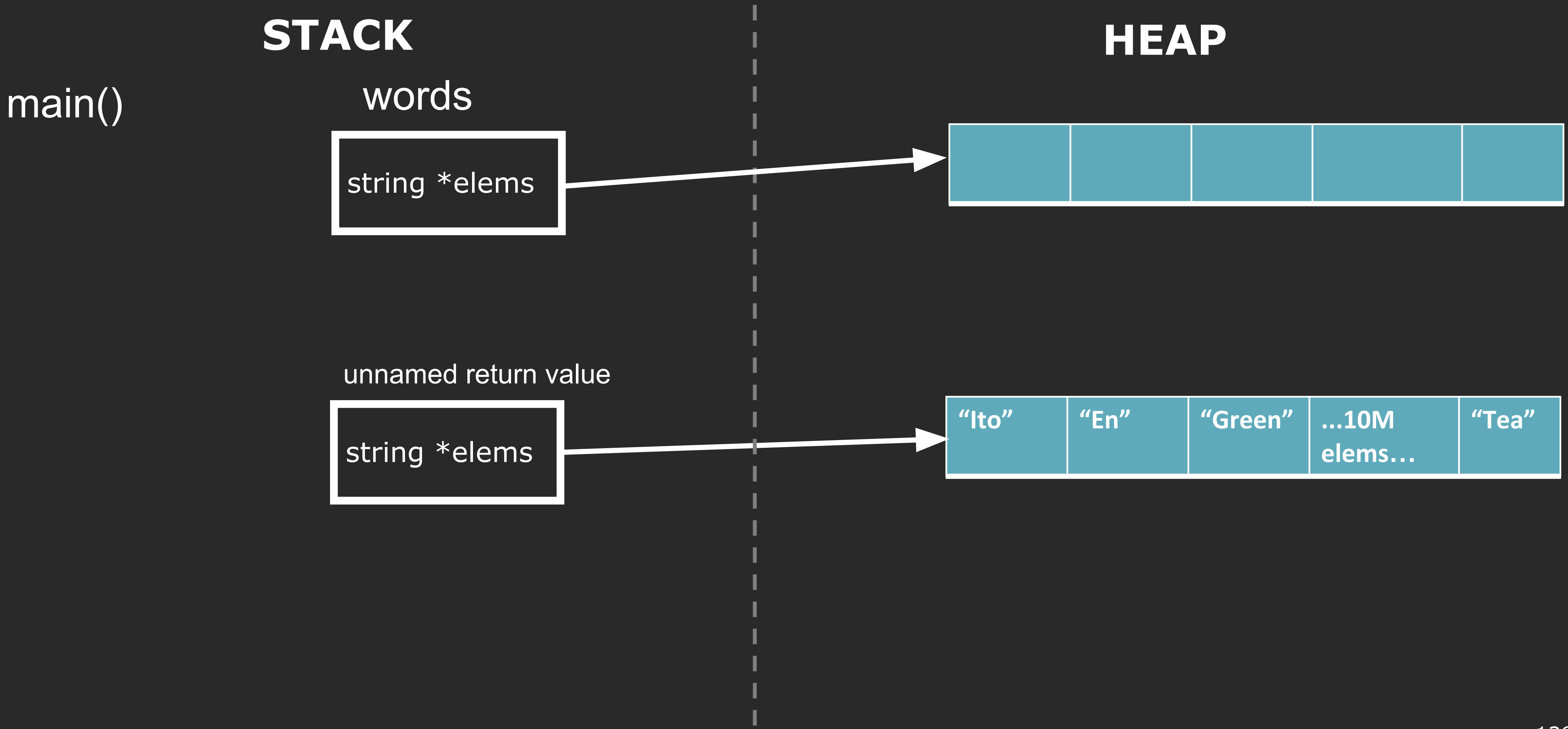
Compiler: “I know this vector is going to be returned, so I’ll just create it in main’s space.



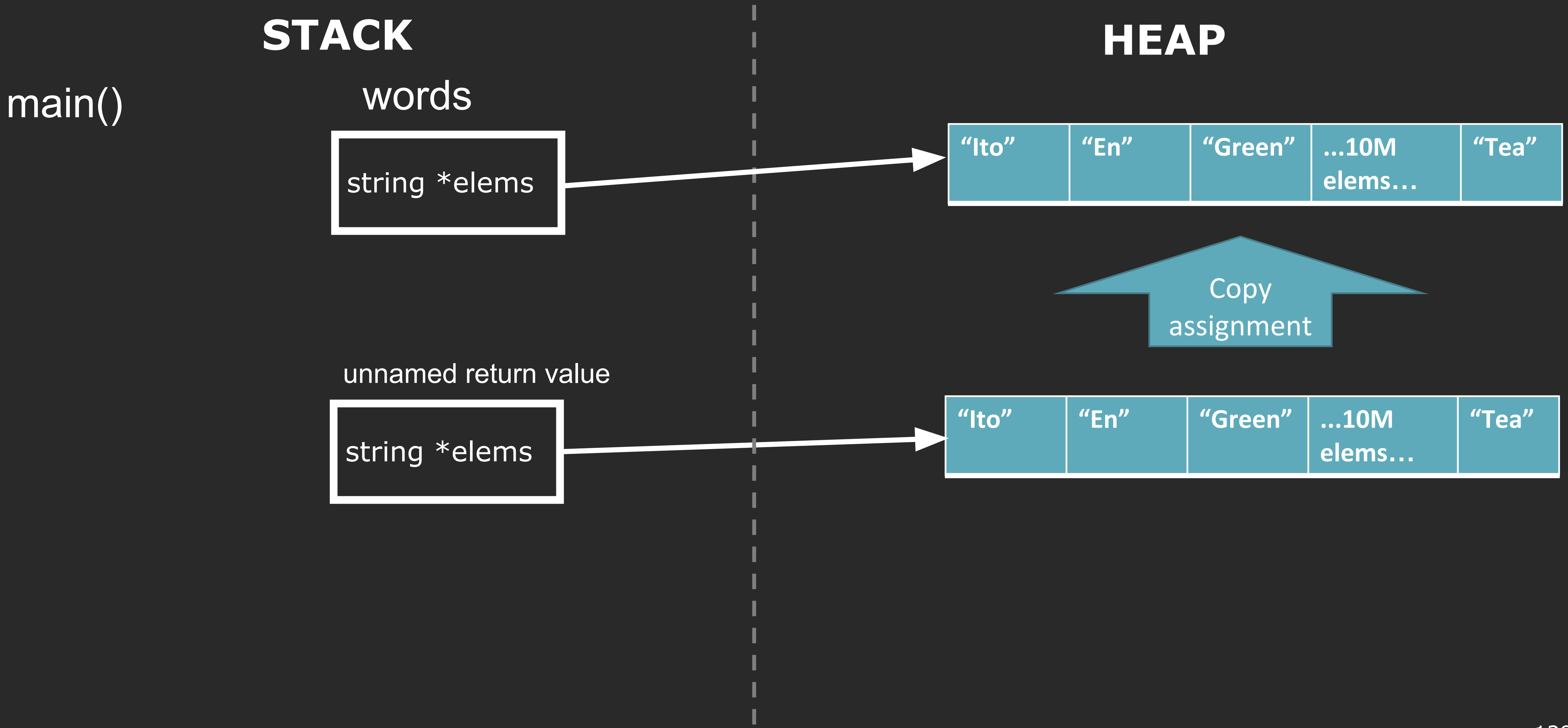
The words are still added as normal.



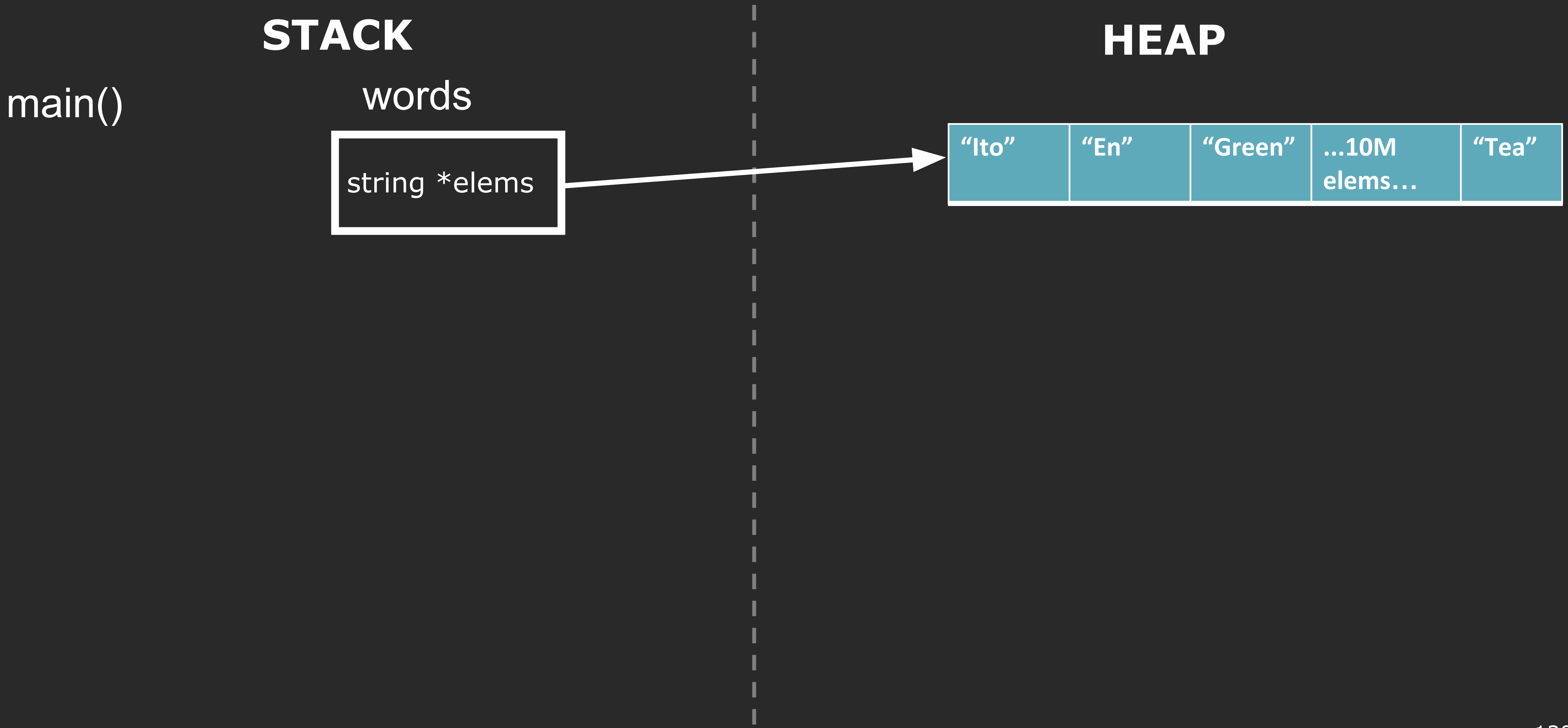
We get to skip the copy constructor call for the return.



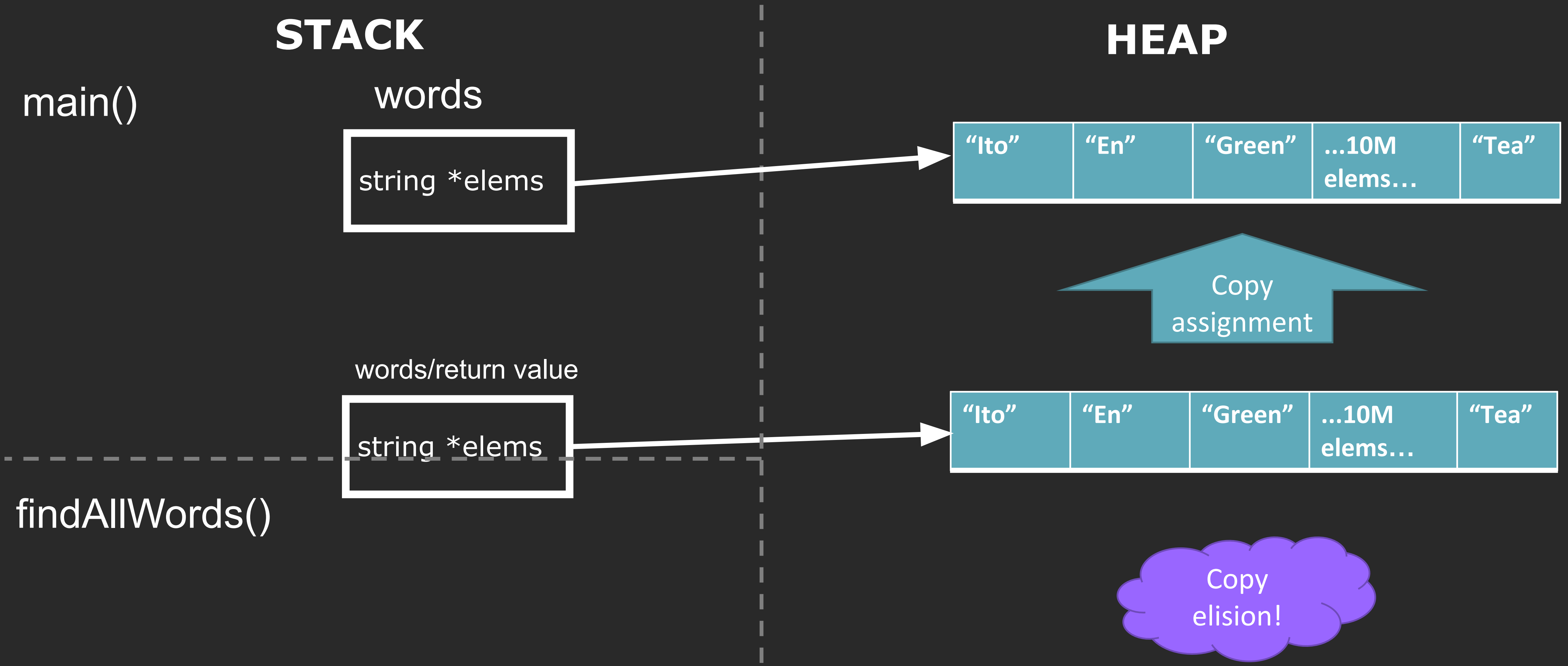
We get to skip the copy constructor call for the return.



We get to skip the copy constructor call for the return.



We get to skip the copy constructor call for the return.



Questions

Key Idea

Copy elision is an optimization that the compiler makes to skip unnecessary copies.

Copy elision is guaranteed by compiler in C++17.

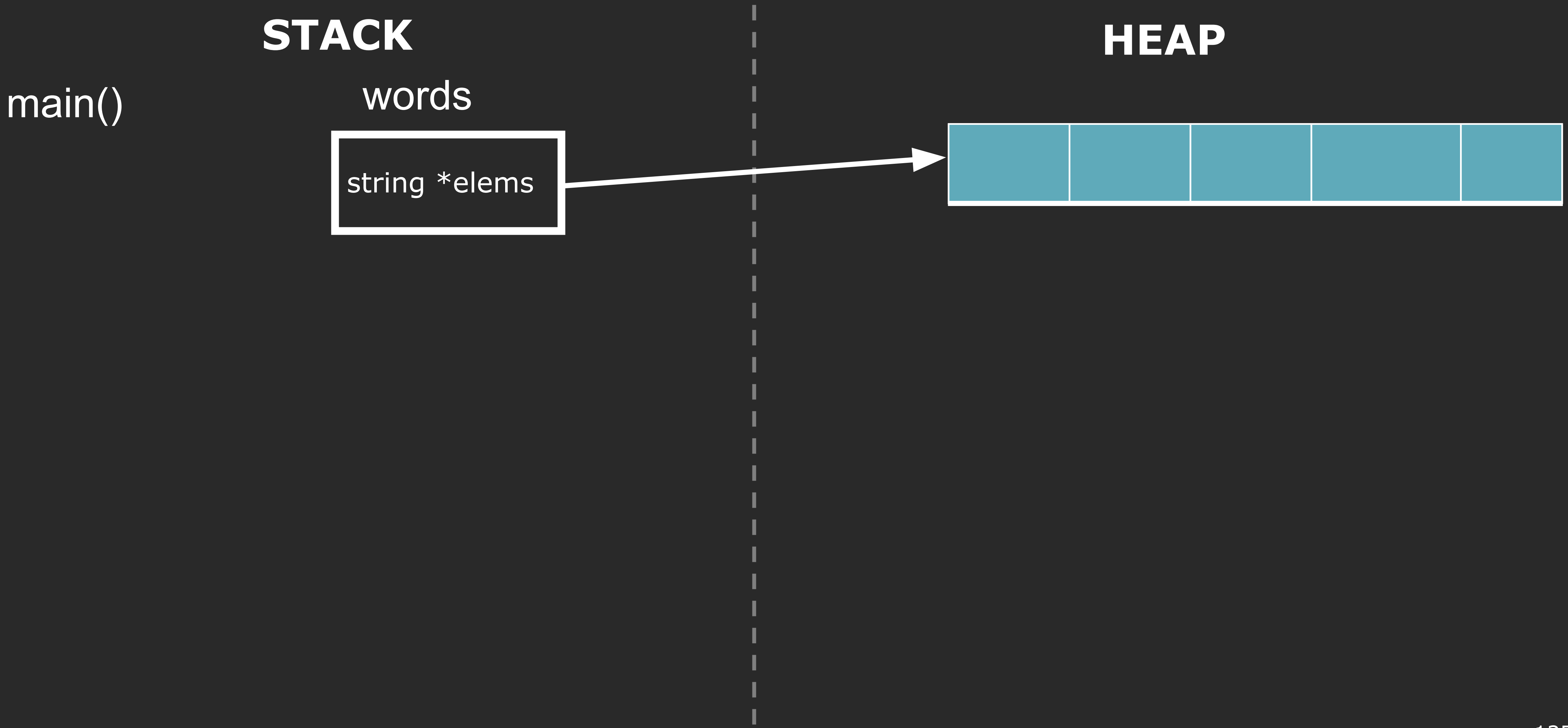
Here’s a more clever idea?

STACK

main()

HEAP

Here’s a more clever idea?



STACK

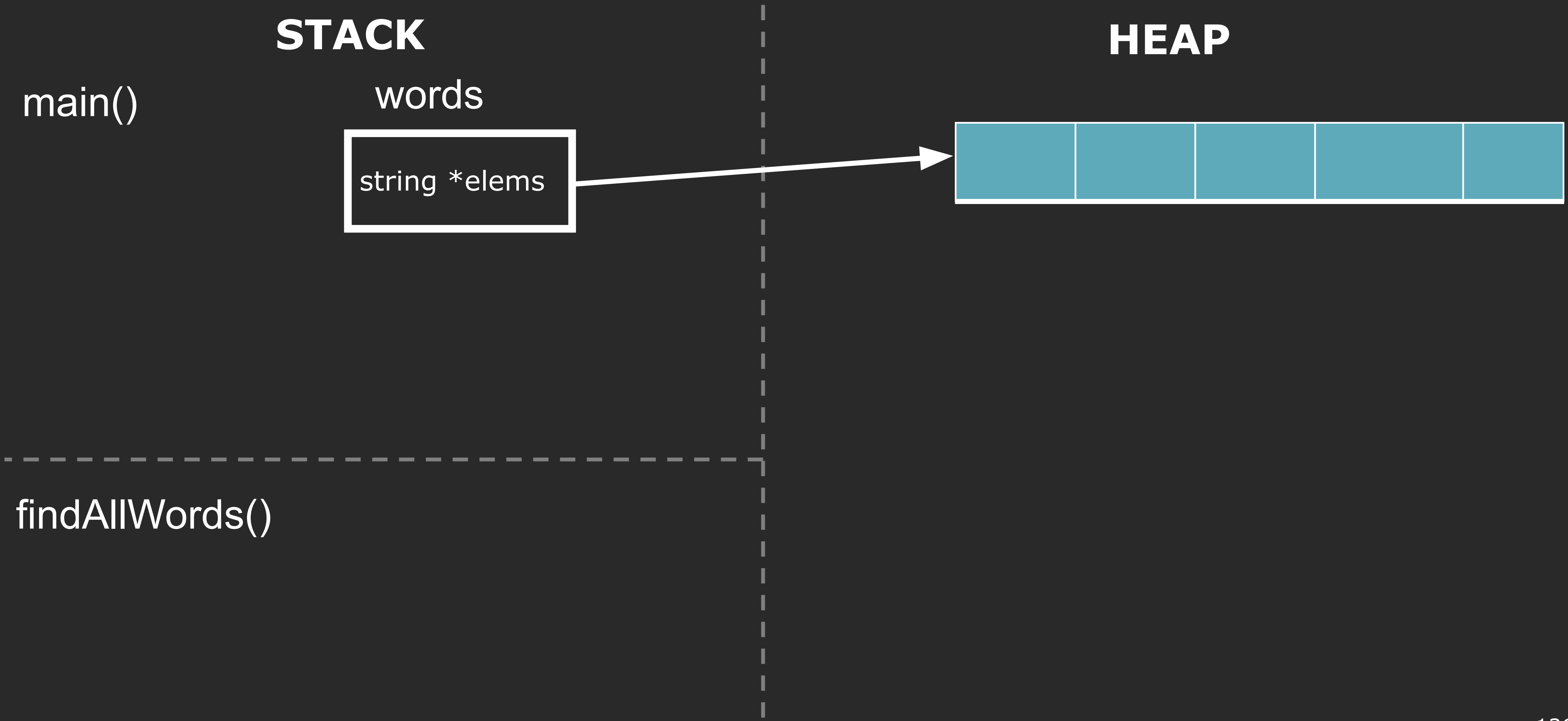
HEAP

main()

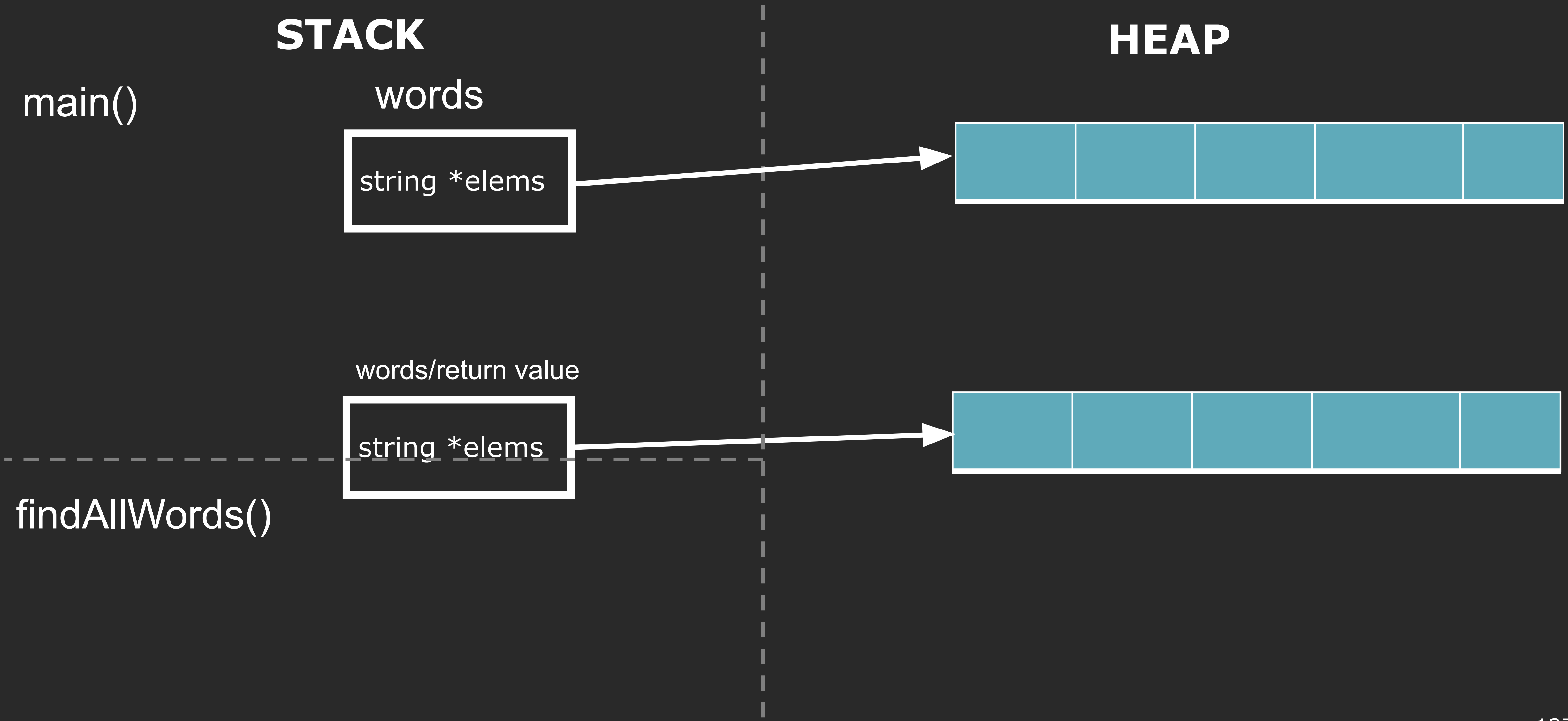
words

string *elems

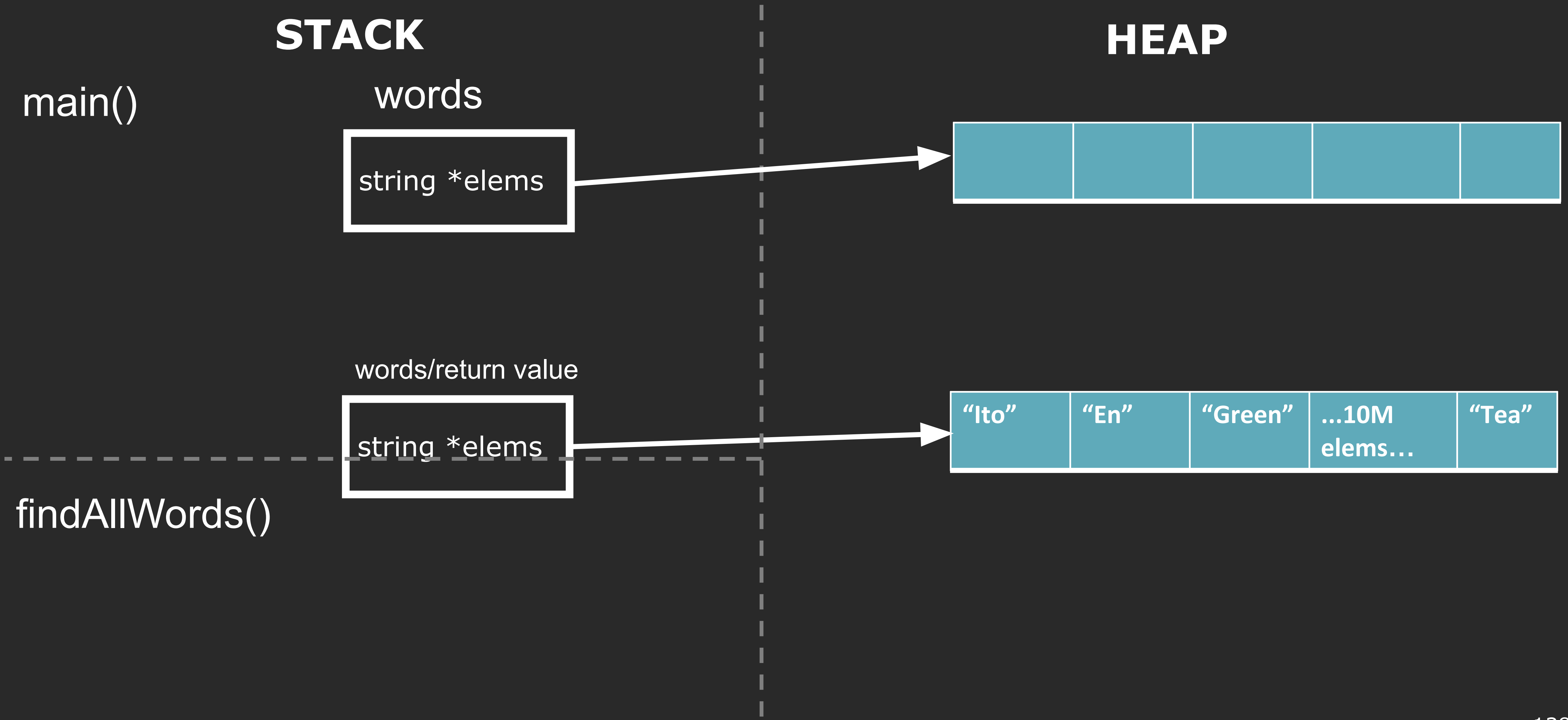
Here’s a more clever idea?



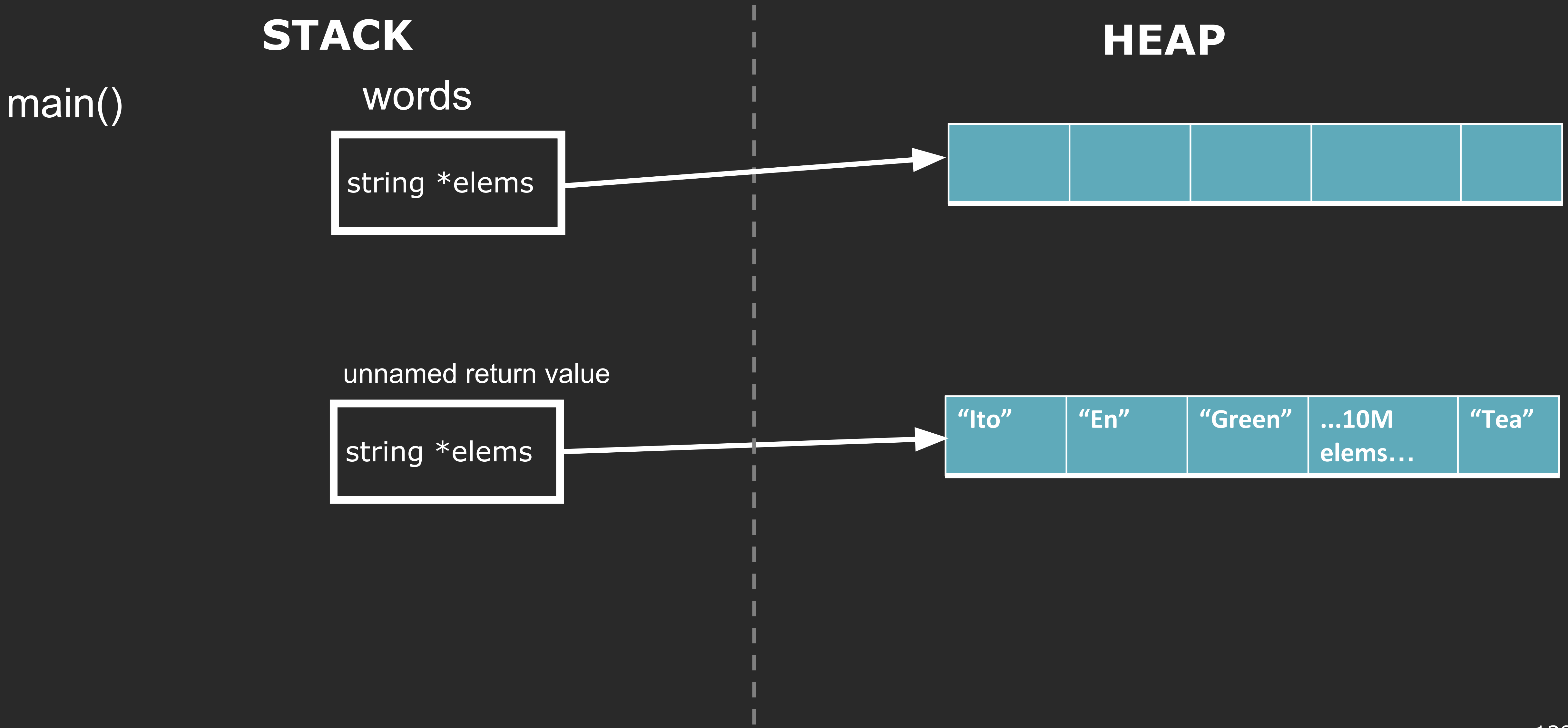
Here’s a more clever idea?



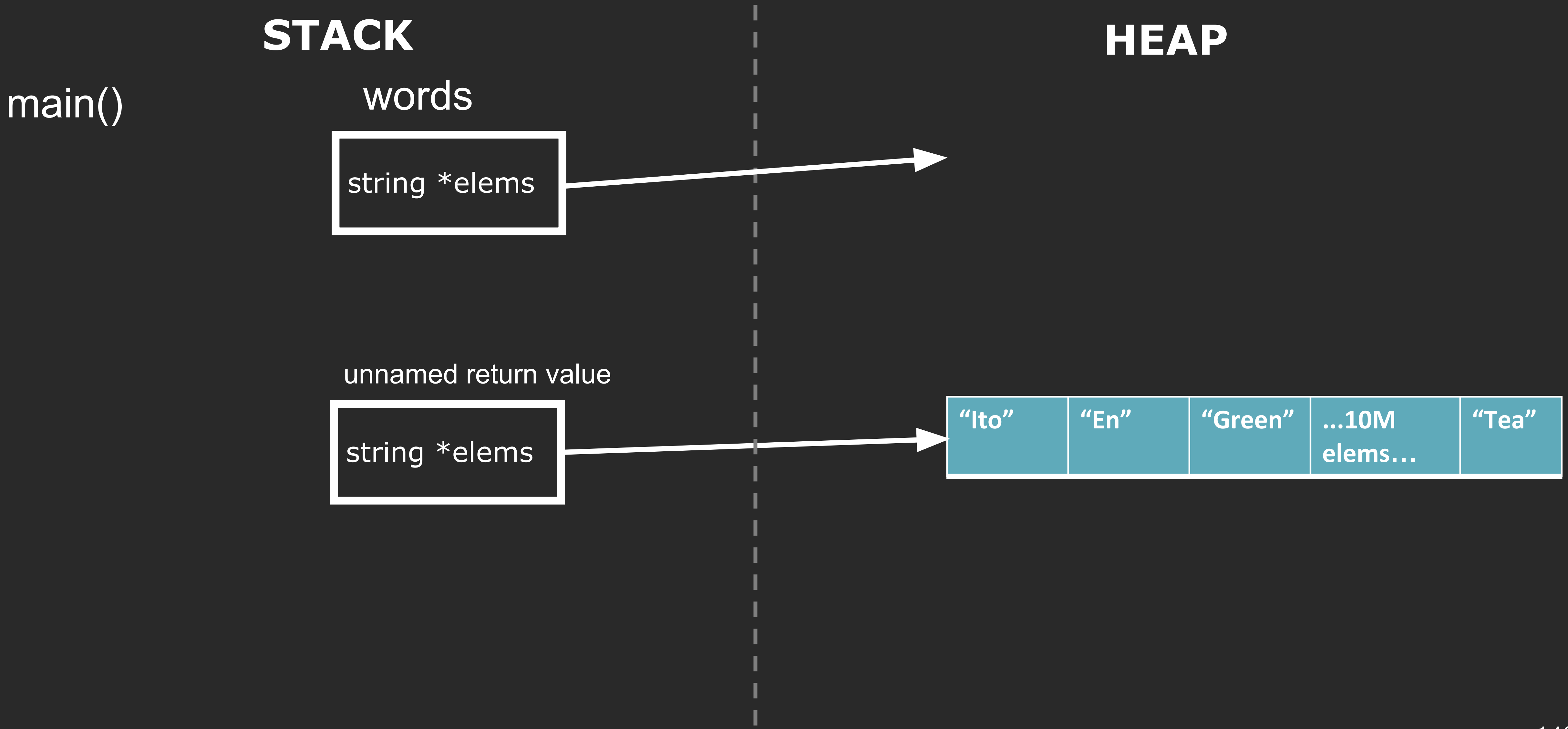
Here's a more clever idea?



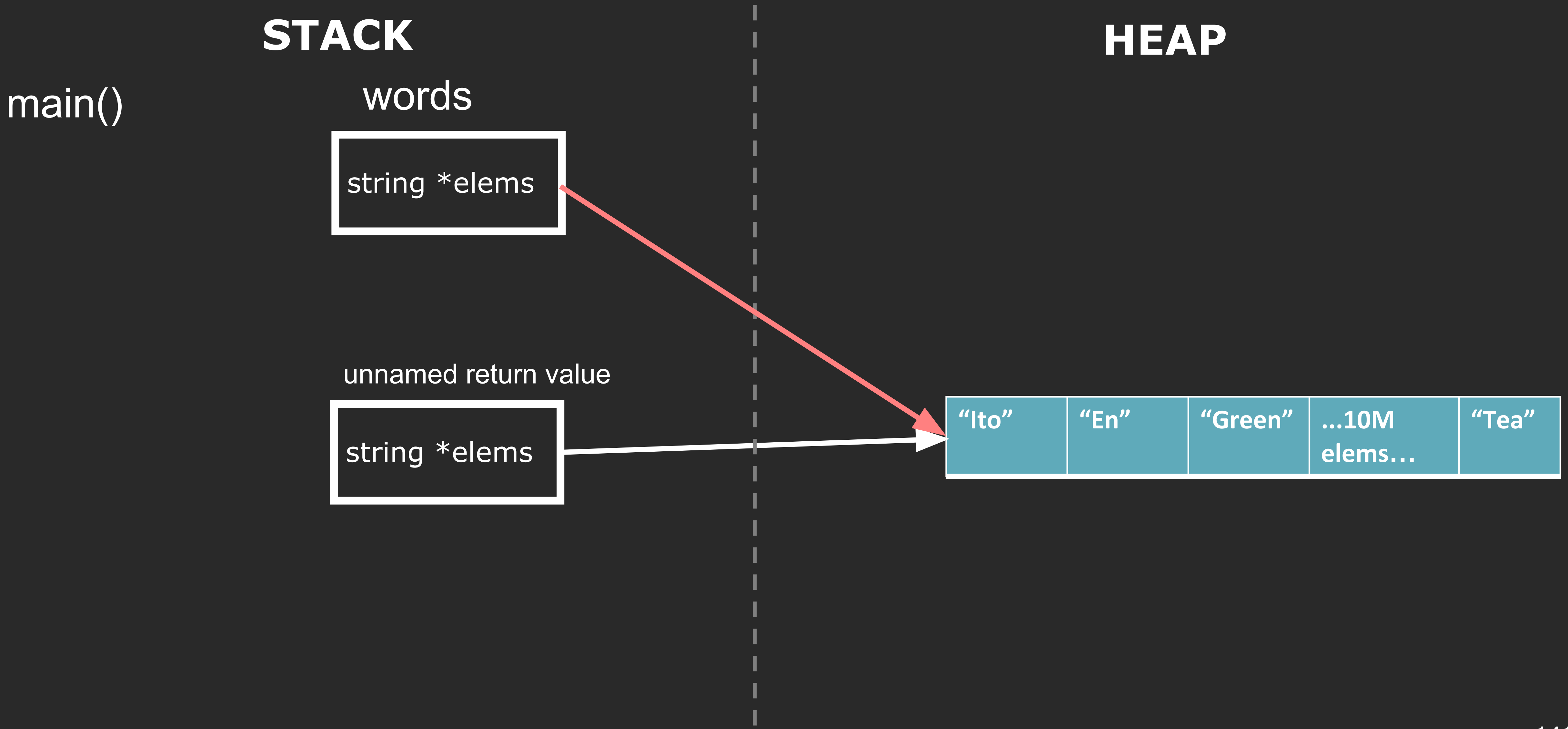
Here’s a more clever idea?



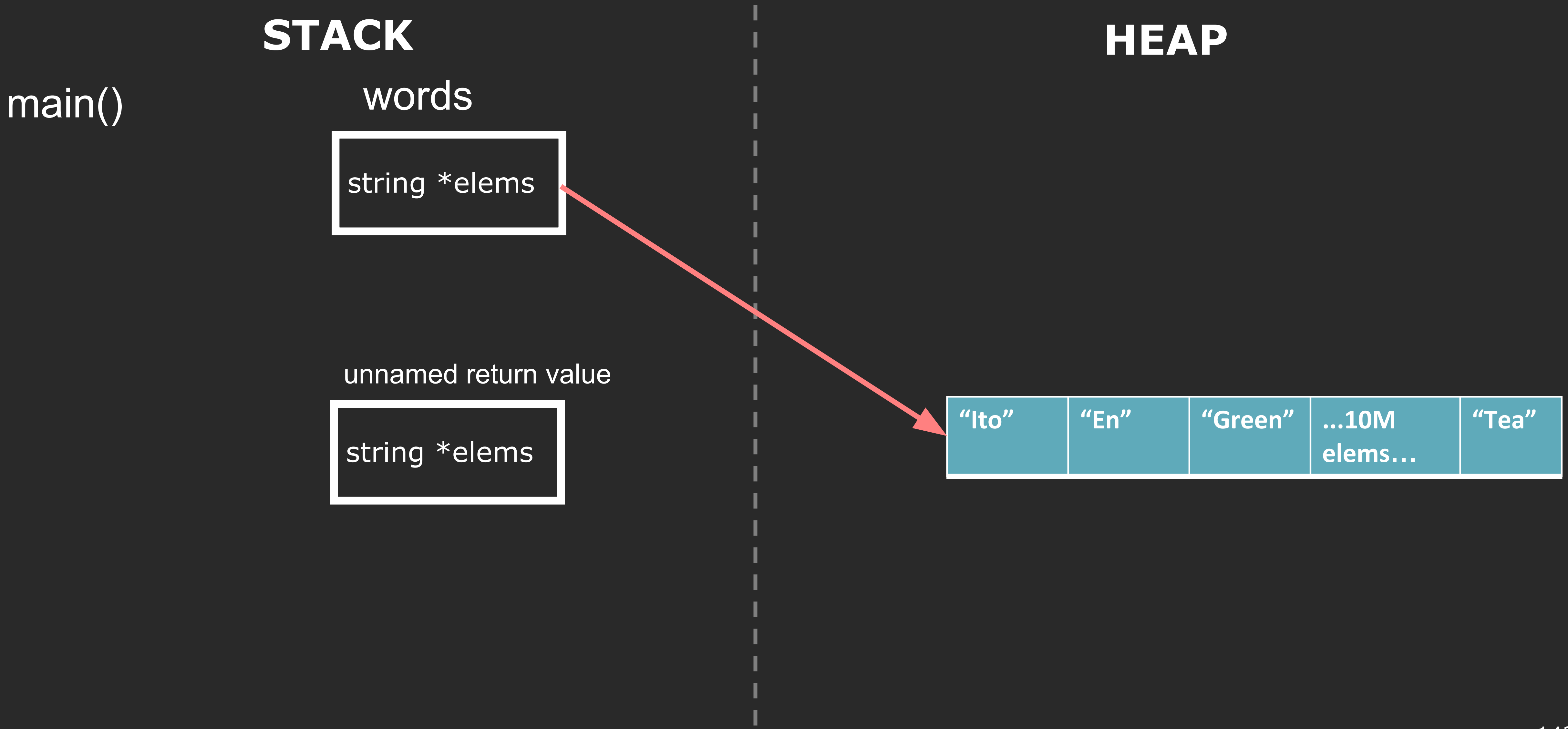
Let’s get rid of that empty array.



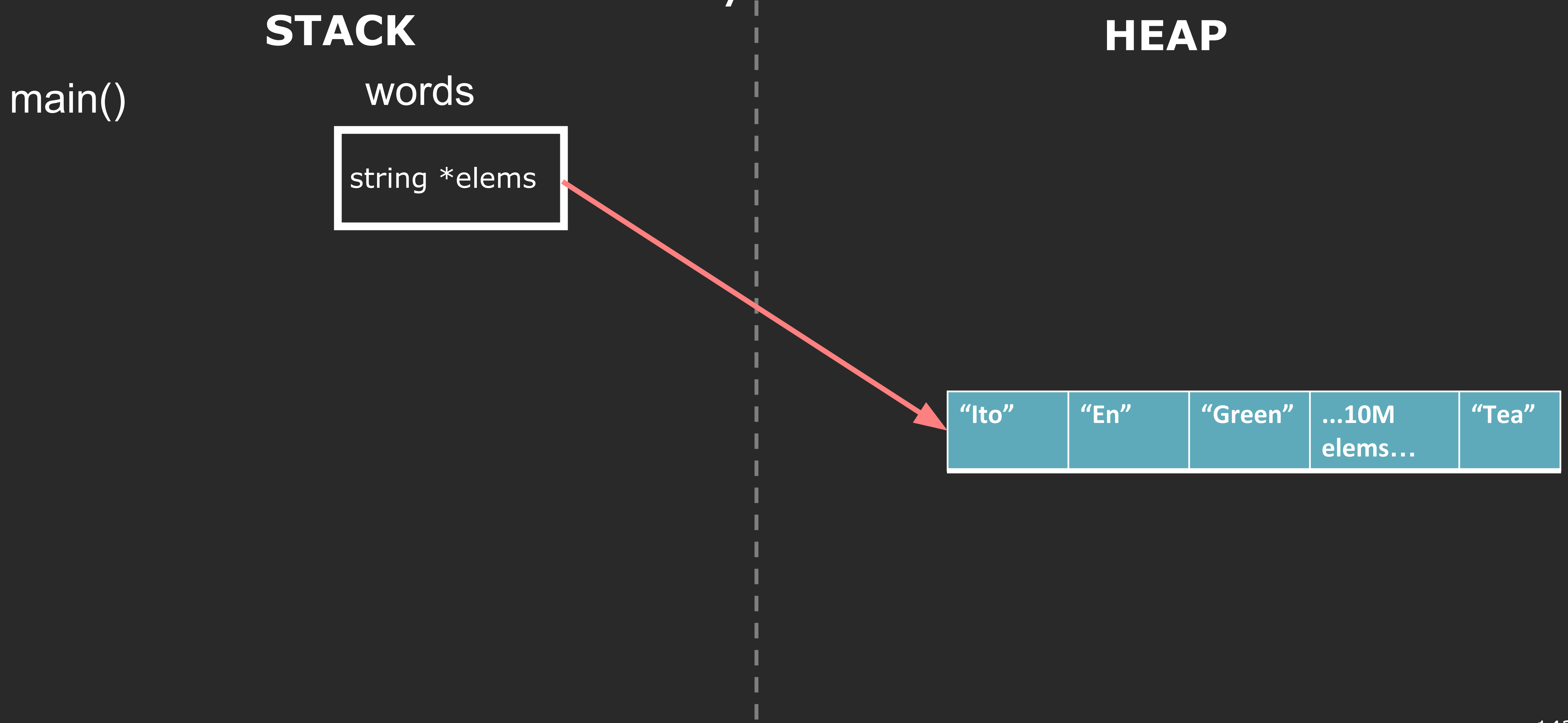
Steal the array of the unnamed return value.



Evict the unnamed return value’s claim over the array.



The return value is temporary, so it will be gone on the very next line.



STACK

HEAP

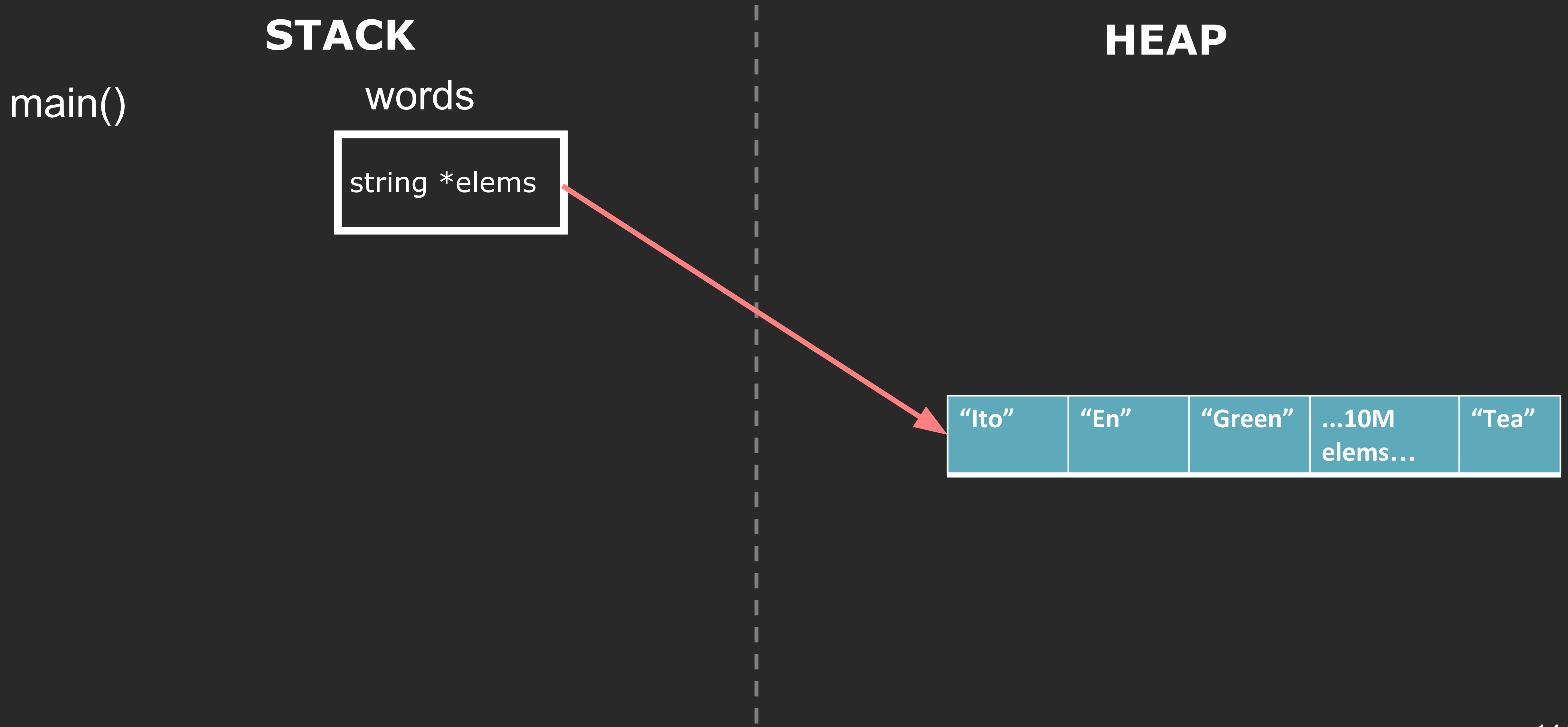
main()

words

string *elems

“Ito”	“En”	“Green”	…10M elems…	“Tea”

Zero unnecessary copies!!!



Why this isn't super easy to do...

- When are we allowed to just "evict" another object's claim over some resource (memory)?
- We'll have a long discussion about what a "temporary" value is and why that is relevant.

Next time

Move Semantics

One slide summary of today

- constructor creates object, assignment overwrites existing object.
- use member initializer lists over direct assignment in ctor.
- compiler generated copy does not work on handles to resources (eg. pointers). you must manually copy the resource yourself.
- for assignment: check self-assign, free resources (memory)
- you can delete a special member function so it can't be called. using default asks compiler to create the default one.
- rule of 0: if default copy is correct, don't define any copy/dtor.
- rule of 3: if you define one special member function, define all three.
- compiler does some optimizing, but we'll do more next time