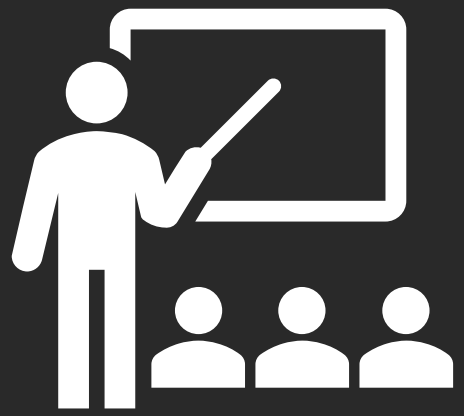
Special Member Functions

CS 106L Winter 2020 – Avery Wang and Anna Zeng

Game Plan



- construction vs. assignment
- details and delete
- rule of three/zero
- copy elision

recap

Design considerations when overloading operators!

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

```
StringVector operator+(const StringVector& lhs, const StringVector& rhs) {
        StringVector copy = lhs;
        copy += rhs;
        return copy;

}
```

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

```
string& StringVector::operator[](size_t index) {
        // static_cast/const_cast trick
}

const string& StringVector::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?

```
MyVector<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
```

copy operations

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.

create new;

- 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?

```
MyVector<int> function(MyVector<int> vec0) {
  MyVector<int> vec1;
  MyVector<int> vec2{3, 4, 5};
  MyVector<int> vec3();
  MyVector<int> vec4(vec2);
  MyVector<int> vec5{};
  MyVector<int> vec6{vec3 + vec4};
  MyVector<int> vec7 = vec4;
  vec7 = vec2;
  return vec7;
}
```

Copy constructor called, since vec0 passed by value.

```
MyVector<int> function(MyVector<int> vec0) {
  MyVector<int> vec1;
  MyVector<int> vec2{3, 4, 5};
  MyVector<int> vec3();
  MyVector<int> vec4(vec2);
  MyVector<int> vec5{};
  MyVector<int> vec6{vec3 + vec4};
  MyVector<int> vec7 = vec4;
  vec7 = vec2;
  return vec7;
}
```

pass by value

Default constructor!

```
MyVector<int> function(MyVector<int> vec0) {
  MyVector<int> vec1;
  MyVector<int> vec2{3, 4, 5};
  MyVector<int> vec3();
  MyVector<int> vec4(vec2);
  MyVector<int> vec5{};
  MyVector<int> vec6{vec3 + vec4};
  MyVector<int> vec7 = vec4;
  vec7 = vec2;
  return vec7;
}
```

Initializer List Constructor

```
MyVector<int> function(MyVector<int> vec0) {
  MyVector<int> vec1;
  MyVector<int> vec2{3, 4, 5};
  MyVector<int> vec3();
  MyVector<int> vec4(vec2);
  MyVector<int> vec5{};
  MyVector<int> vec6{vec3 + vec4};
  MyVector<int> vec7 = vec4;
  vec7 = vec2;
  return vec7;
}
```

Not a constructor – C++’s most vexing parse.

```
MyVector<int> function(MyVector<int> vec0) {
  MyVector<int> vec1;
  MyVector<int> vec2{3, 4, 5};
  MyVector<int> vec3();
  MyVector<int> vec4(vec2);
  MyVector<int> vec5{};
  MyVector<int> vec6{vec3 + vec4};
  MyVector<int> vec7 = vec4;
  vec7 = vec2;
  return vec7;
}
```

function~

Copy constructor

```
MyVector<int> function(MyVector<int> vec0) {
  MyVector<int> vec1;
  MyVector<int> vec2{3, 4, 5};
  MyVector<int> vec3();
  MyVector<int> vec4(vec2);
  MyVector<int> vec5{};
  MyVector<int> vec6{vec3 + vec4};
  MyVector<int> vec7 = vec4;
  vec7 = vec2;
  return vec7;
}
```

Default constructor

```
MyVector<int> function(MyVector<int> vec0) {
  MyVector<int> vec1;
  MyVector<int> vec2{3, 4, 5};
  MyVector<int> vec3();
  MyVector<int> vec4(vec2);
  MyVector<int> vec5{};
  MyVector<int> vec6{vec3 + vec4};
  MyVector<int> vec7 = vec4;
  vec7 = vec2;
  return vec7;
}
```

Copy constructor!

Next time…move constructor

```
MyVector<int> function(MyVector<int> vec0) {
  MyVector<int> vec1;
  MyVector<int> vec2{3, 4, 5};
  MyVector<int> vec3();
  MyVector<int> vec4(vec2);
  MyVector<int> vec5{};
  MyVector<int> vec6{vec3 + vec4};
  MyVector<int> vec7 = vec4;
  vec7 = vec2;
  return vec7;
}
```

Copy constructor (not assignment, since vec7 is newly created!)

```
MyVector<int> function(MyVector<int> vec0) {
  MyVector<int> vec1;
  MyVector<int> vec2{3, 4, 5};
  MyVector<int> vec3();
  MyVector<int> vec4(vec2);
  MyVector<int> vec5{};
  MyVector<int> vec6{vec3 + vec4};
  MyVector<int> vec7 = vec4;
  vec7 = vec2;
  return vec7;
}
```

Copy assignment (vec7 is existing vector)

```
MyVector<int> function(MyVector<int> vec0) {
  MyVector<int> vec1;
  MyVector<int> vec2{3, 4, 5};
  MyVector<int> vec3();
  MyVector<int> vec4(vec2);
  MyVector<int> vec5{};
  MyVector<int> vec6{vec3 + vec4};
  MyVector<int> vec7 = vec4;
  vec7 = vec2;
  return vec7;
}
```

Copy constructor (copies object into outside scope)

```
MyVector<int> function(MyVector<int> vec0) {
  MyVector<int> vec1;
  MyVector<int> vec2{3, 4, 5};
  MyVector<int> vec3();
  MyVector<int> vec4(vec2);
  MyVector<int> vec5{};
  MyVector<int> vec6{vec3 + vec4};
  MyVector<int> vec7 = vec4;
  vec7 = vec2;
  return vec7;
}
```

Member Initialization List

```
template <typename T>
MyVector<T>::MyVector() {
        _logicalSize = 0;
        _allocatedSize = kInitialSize;
        _elems = new T[kInitialSize];
}

template <typename T>
MyVector<T>::MyVector() :
                _logicalSize(0), _allocatedSize(kInitialSize),
        _elems(new T[kInitialSize]) { }
```

Member Initialization List

const correctness; delare and assign simultaneously

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

- Faster. Why construct, then reassign?
- Can’t reassign references, so must construct references directly with what they refer to.

```
template <typename T>
MyVector<T>::MyVector() :
                    _logicalSize(0), _allocatedSize(kInitialSize),
          _elems(new T[kInitialSize]) { }
```

Destructor

```
template <typename T>
MyVector<T>::~MyVector() {
        delete [] _elems;
}
```

std::initializer_list

```
MyVector<int> vec{1, 2, 3, 4, 5};
// same thing as
// MyVector<int> vec(std::initializer_list({1, 2, 3, 4, 5});
```

std::initializer_list

```
template <typename T>
MyVector<T>::MyVector(std::initializer_list<T> init) :
        _logicalSize(init.size()),
         _allocatedSize(max(kDefaultSize, 2 * _logicalSize)),
        _elems(new T[_allocatedSize])
{
        std::copy(init.begin(), init.end(), _elems);
}
```

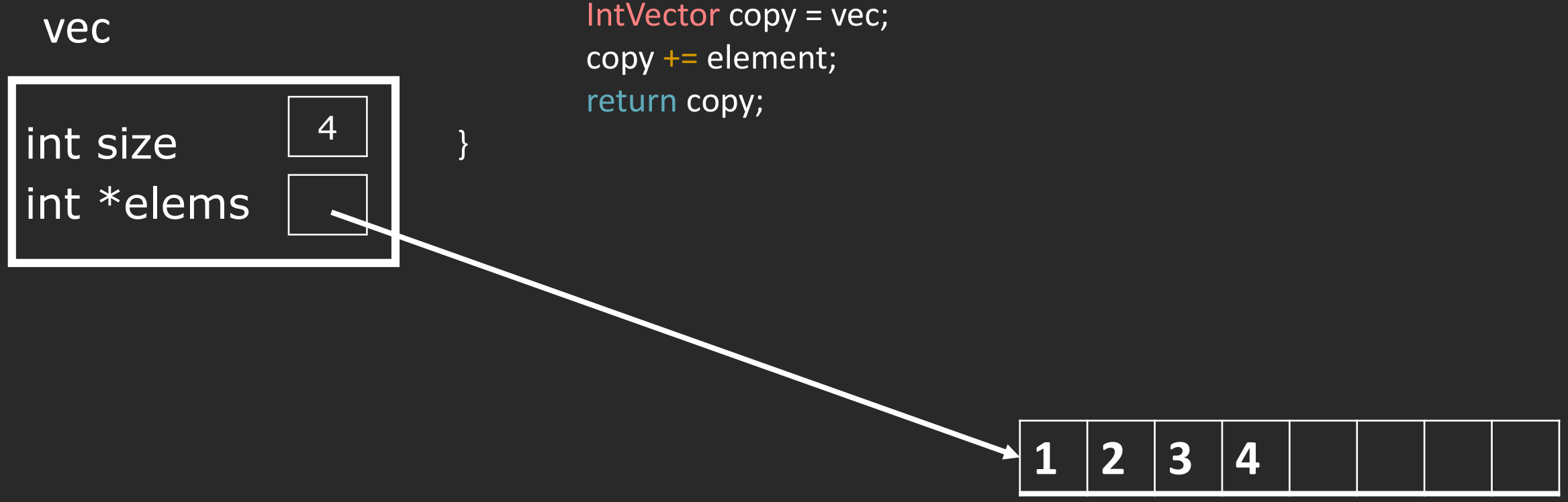
problems from last time

I lied…this code doesn't actually work.

```
IntVector operator+(const IntVector & vec, int elem) {
        IntVector copy = vec;
        copy += element;
        return copy;
}
```

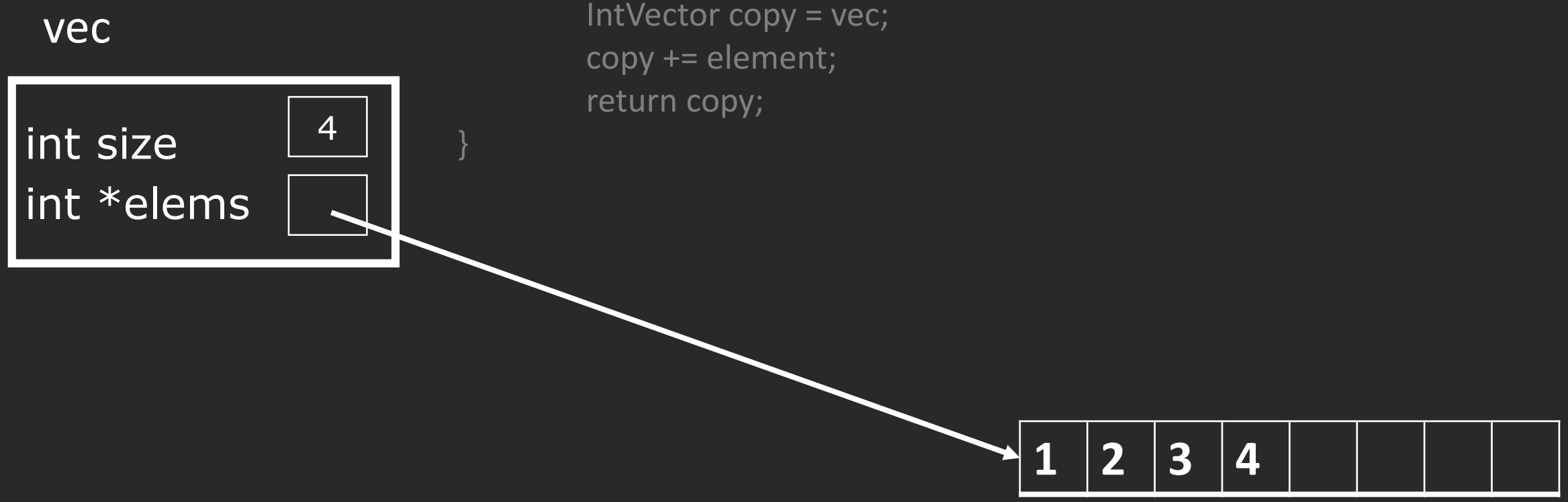
Copy is not as simple as copying each member.

```
IntVector operator+(const IntVector & vec, int elem) {
        IntVector copy = vec;
        copy += element;
        return copy;
}
```



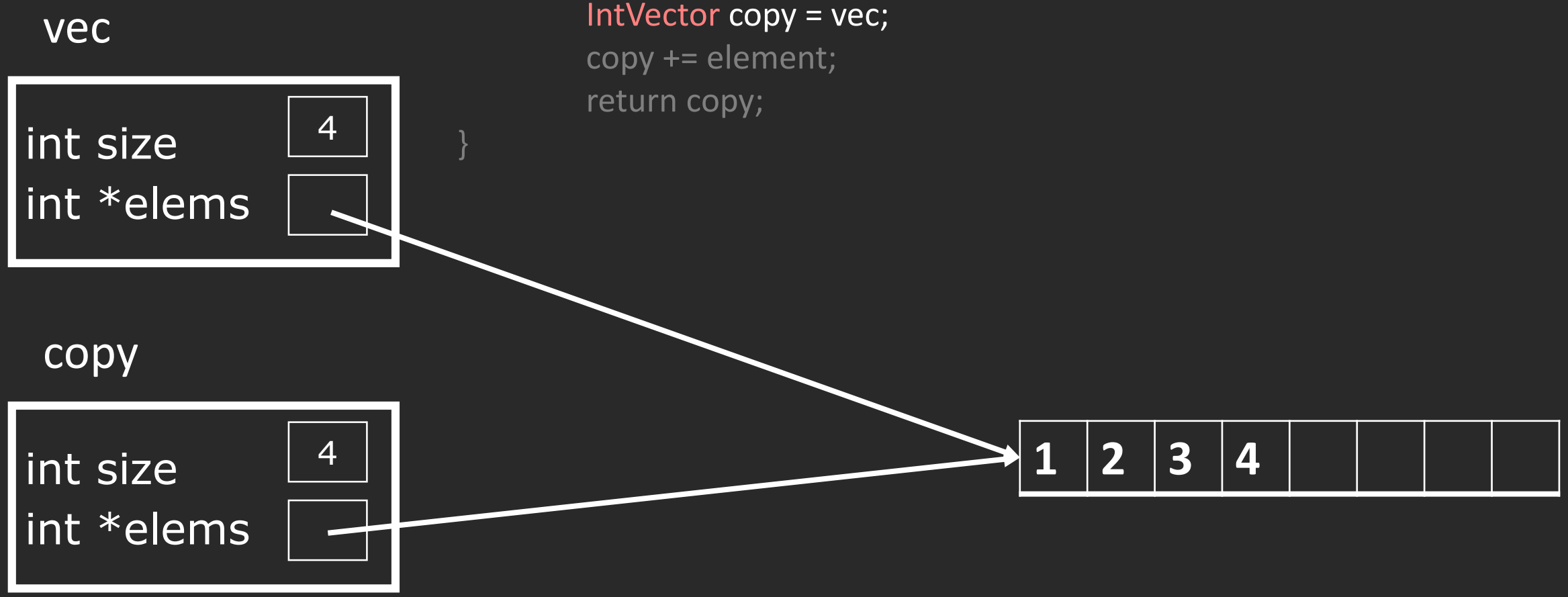
Copy is not as simple as copying each member.

```
IntVector operator+(const IntVector & vec, int elem) {
        IntVector copy = vec;
        copy += element;
        return copy;
}
```



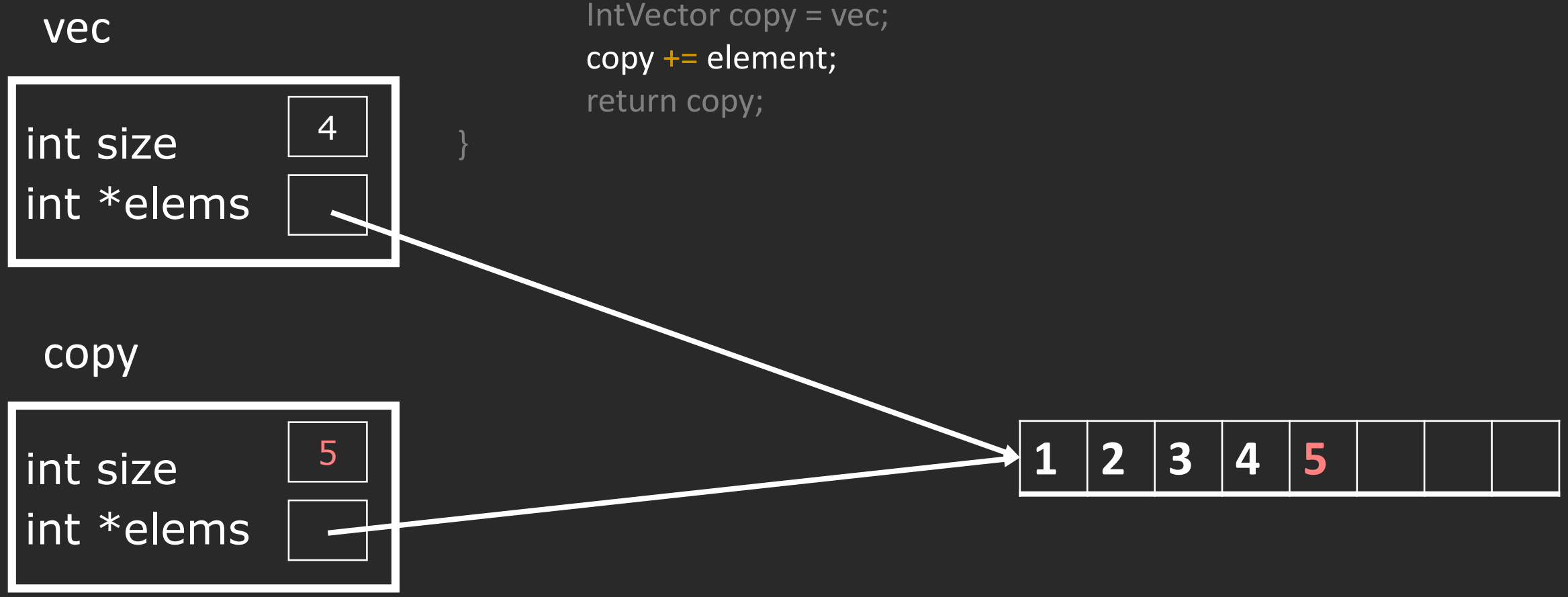
Copy is not as simple as copying each member.

```
IntVector operator+(const IntVector & vec, int elem) {
        IntVector copy = vec;
        copy += element;
        return copy;
}
```



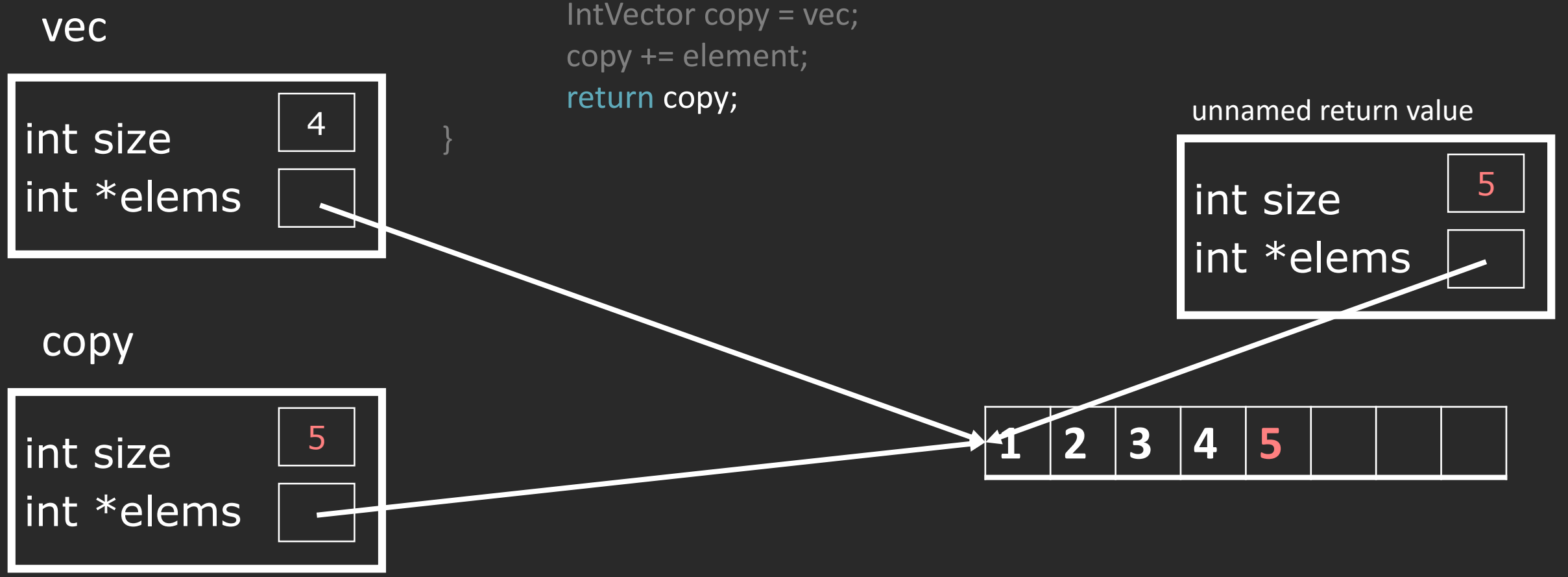
Copy is not as simple as copying each member.

```
IntVector operator+(const IntVector & vec, int elem) {
        IntVector copy = vec;
        copy += element;
        return copy;
}
```



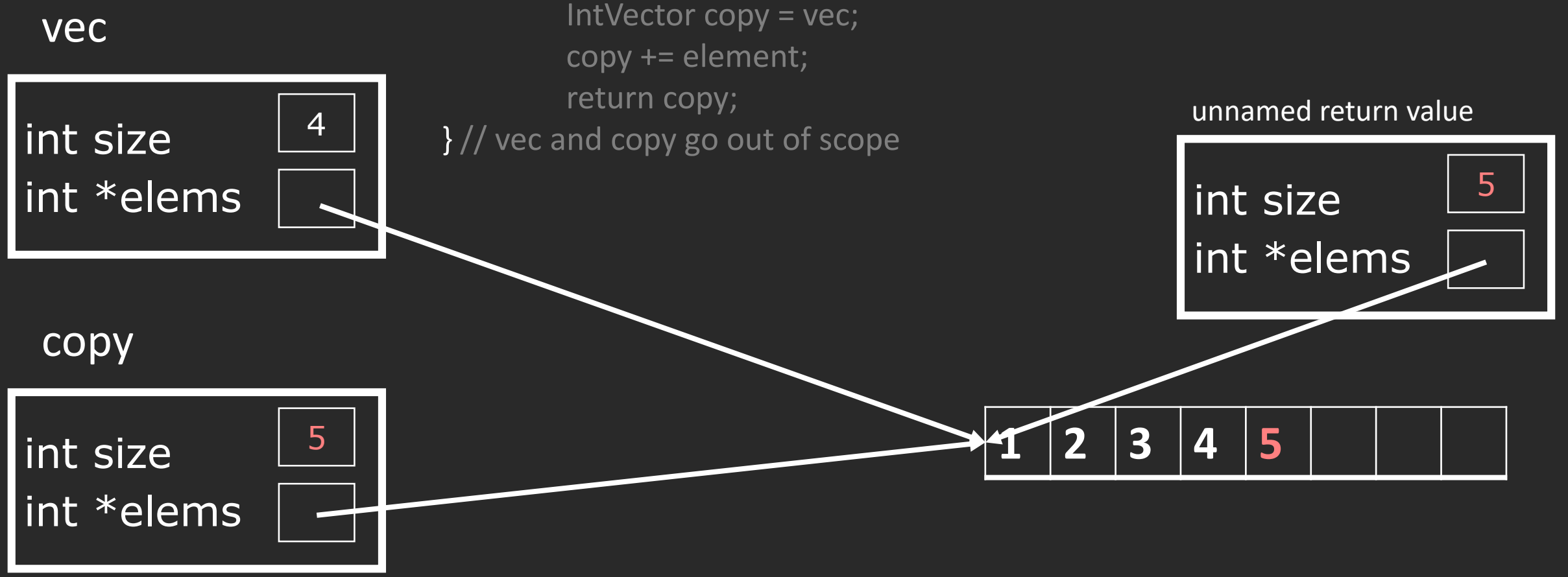
At the return statement, a copy of the local variable is made.

```
IntVector operator+(const IntVector & vec, int elem) {
        IntVector copy = vec;
        copy += element;
        return copy;
}
```



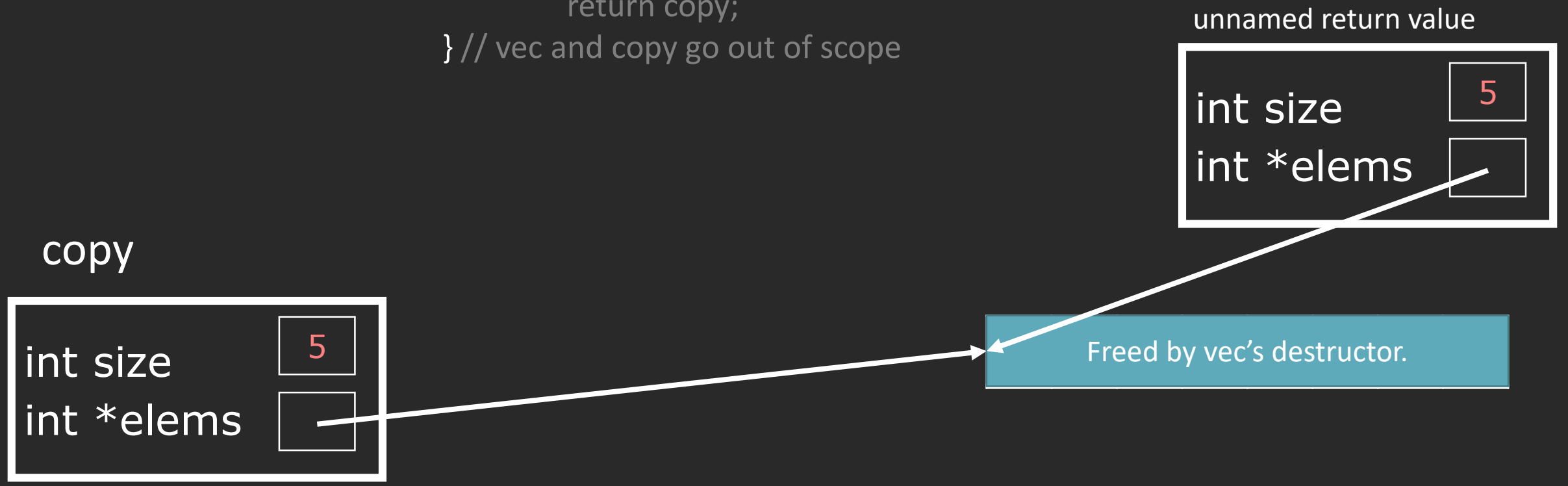
When the vectors go out of scope, their destructor tries to free the array.

```
IntVector operator+(const IntVector & vec, int elem) {
        IntVector copy = vec;
        copy += element;
        return copy;
} // vec and copy go out of scope
```



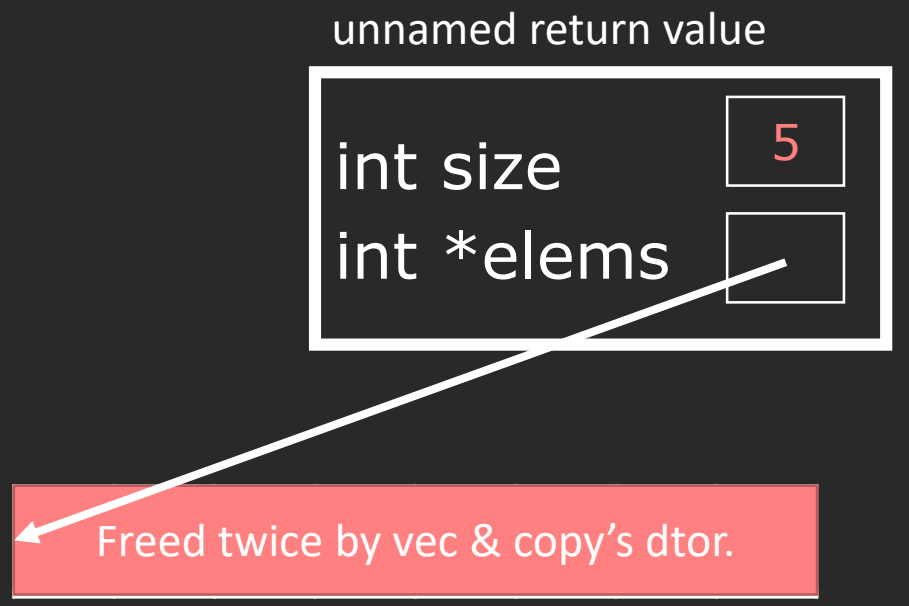
When the vectors go out of scope, their destructor tries to free the array.

```
IntVector operator+(const IntVector & vec, int elem) {
        IntVector copy = vec;
        copy += element;
        return copy;
} // vec and copy go out of scope
```



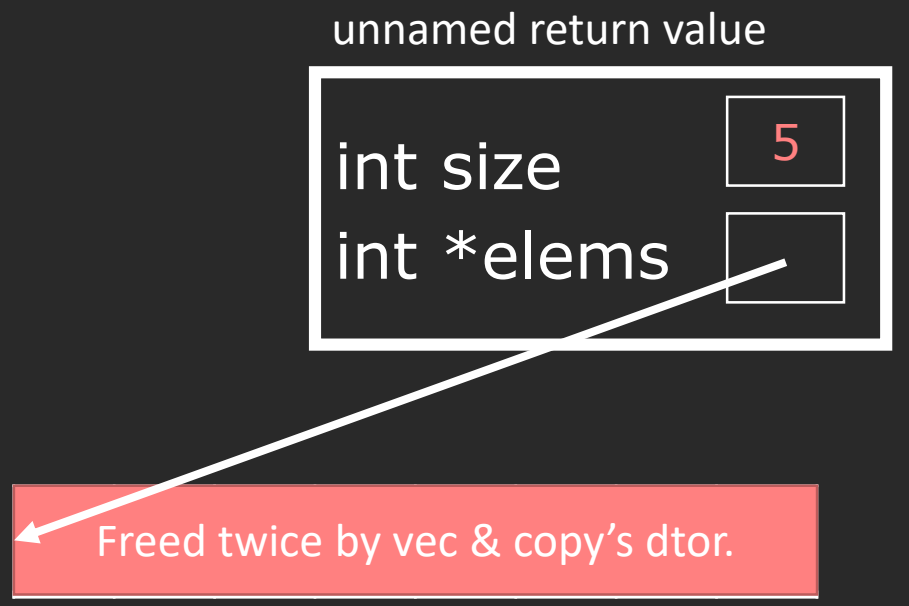
When the vectors go out of scope, their destructor tries to free the array.

```
IntVector operator+(const IntVector & vec, int elem) {
        IntVector copy = vec;
        copy += element;
        return copy;
} // vec and copy go out of scope
```



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

```
IntVector operator+(const IntVector & vec, int elem) {
        IntVector copy = vec;
        copy += element;
        return copy;
} // vec and copy go out of scope
```



The problem is this copy operation.

```
IntVector operator+(const IntVector & vec, int elem) {
        IntVector copy = vec;
        copy += element;
        return copy;
}
```

details

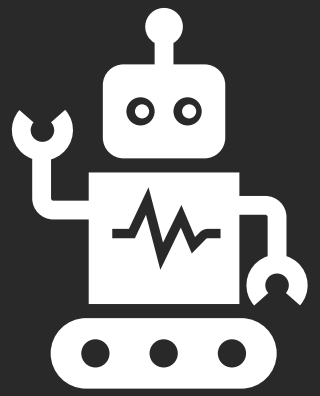
The copy operations must perform the following tasks.

Copy Constructor

- Use initializer list to copy members where assignment does the correct thing.
  - int, other objects, etc.
- Deep copy all members where assignment does not work.
  - pointers to heap memory

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.



Example

Copy constructor and copy assignment:
MyVector and Fraction.

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

```
StringVector::StringVector(const StringVector& other) :
                    _logicalSize(other._logicalSize),
            _allocatedSize(other._allocatedSize) {

            _elems = new ValueType[_allocatedSize];
            std::copy(other.begin(), other.end(), begin());
}
```

The copy assignment needs to clean up this's resources, then perform copy.

```
// can't use initializer list – not a constructor!
StringVector& StringVector::operator=(const StringVector& rhs) {
  delete [] _elems;
     _logicalSize = rhs._logicalSize;
     _allocatedSize = rhs._allocatedSize;
     _elems = new ValueType[_allocatedSize];
     std::copy(rhs.begin(), rhs.end(), begin());
     return *this;
}
```

delete：打扫干净再请客，free old memory，avoid memory leak

Careful about the edge case: self-assignment.

```
// can't use initializer list – not a constructor!
StringVector& StringVector::operator=(const StringVector& rhs) {
    if (this != &rhs) {
        delete [] _elems;
        _logicalSize = rhs._logicalSize;
        _allocatedSize = rhs._allocatedSize;
        _elems = new ValueType[_allocatedSize];
        std::copy(other.begin(), other.end(), begin());
    }
    return *this;
}
```

deleted operations

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

```
class LoggedVector {
public:
  LoggedVector(int num, int denom);
  ~LoggedVector();
  // other methods
  LoggedVector(const LoggedVector& rhs) = delete;
  LoggedVector& operator=(const LoggedVector& rhs) = delete;

private:
        // other stuff
}
```

rule of three

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.

Rule of Zero

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

problems with copying

How many vectors are created?

```
int main() {
  StringVector words;
  words = findAllWords("words.txt");
  // print words
}

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

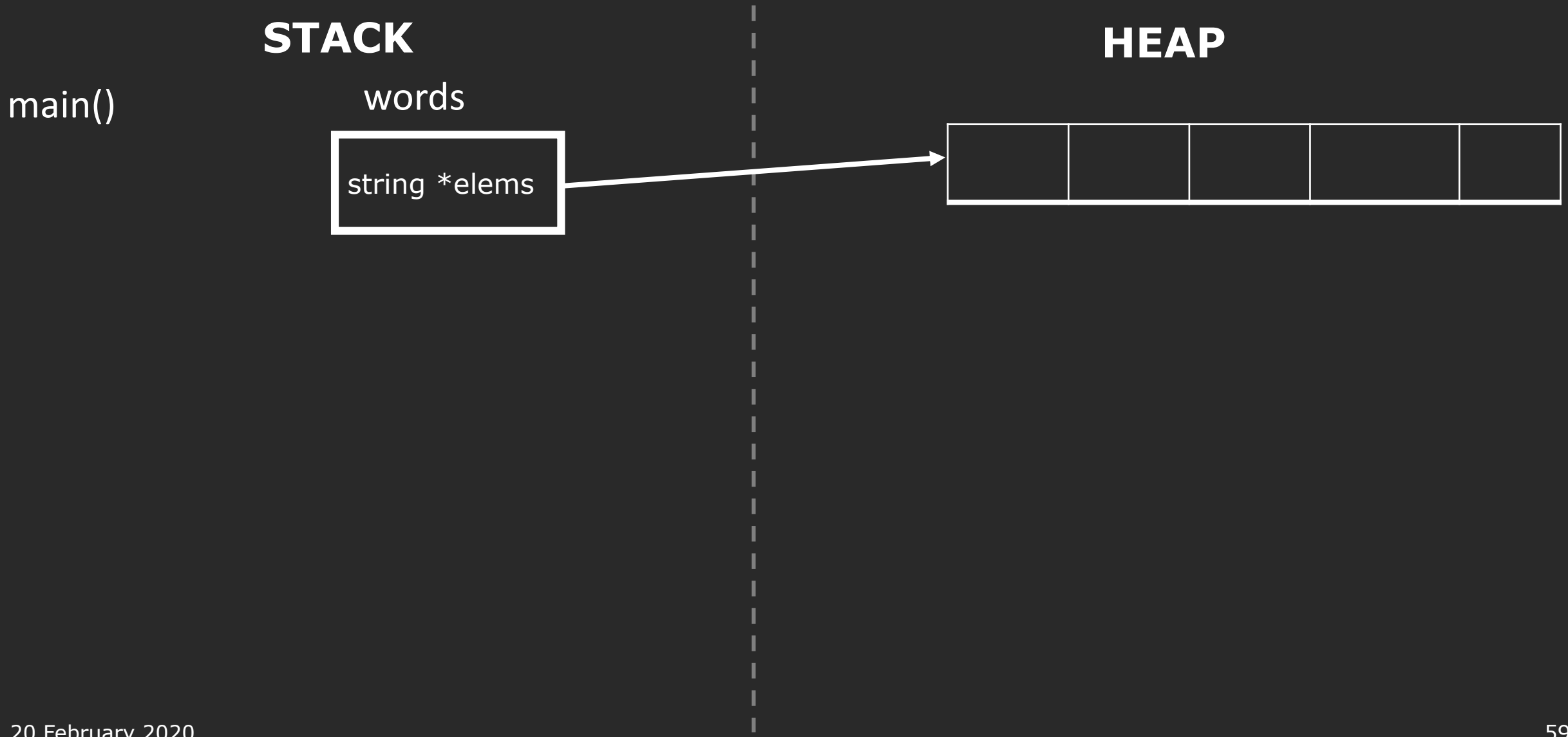
How many vectors are created?

STACK

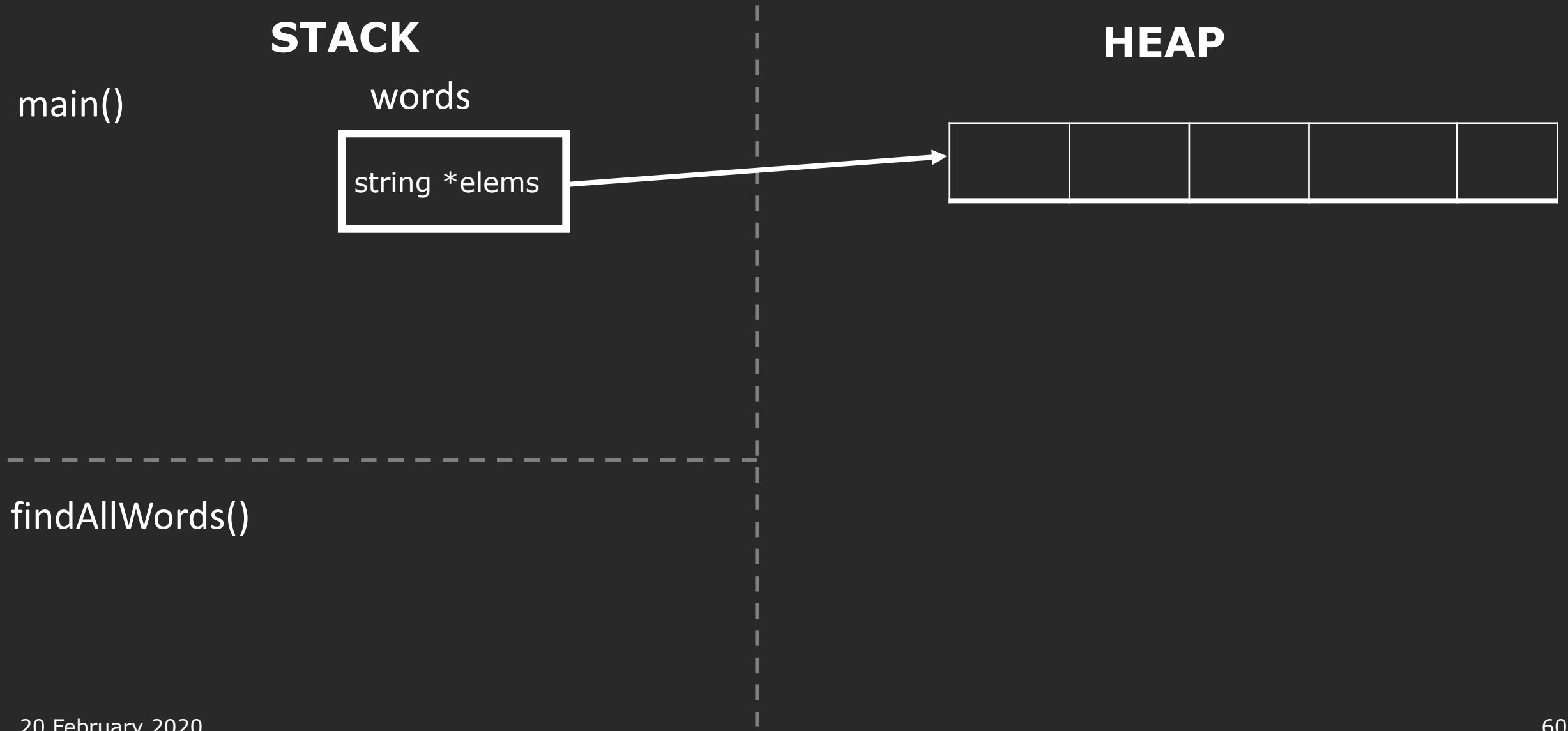
HEAP

main()

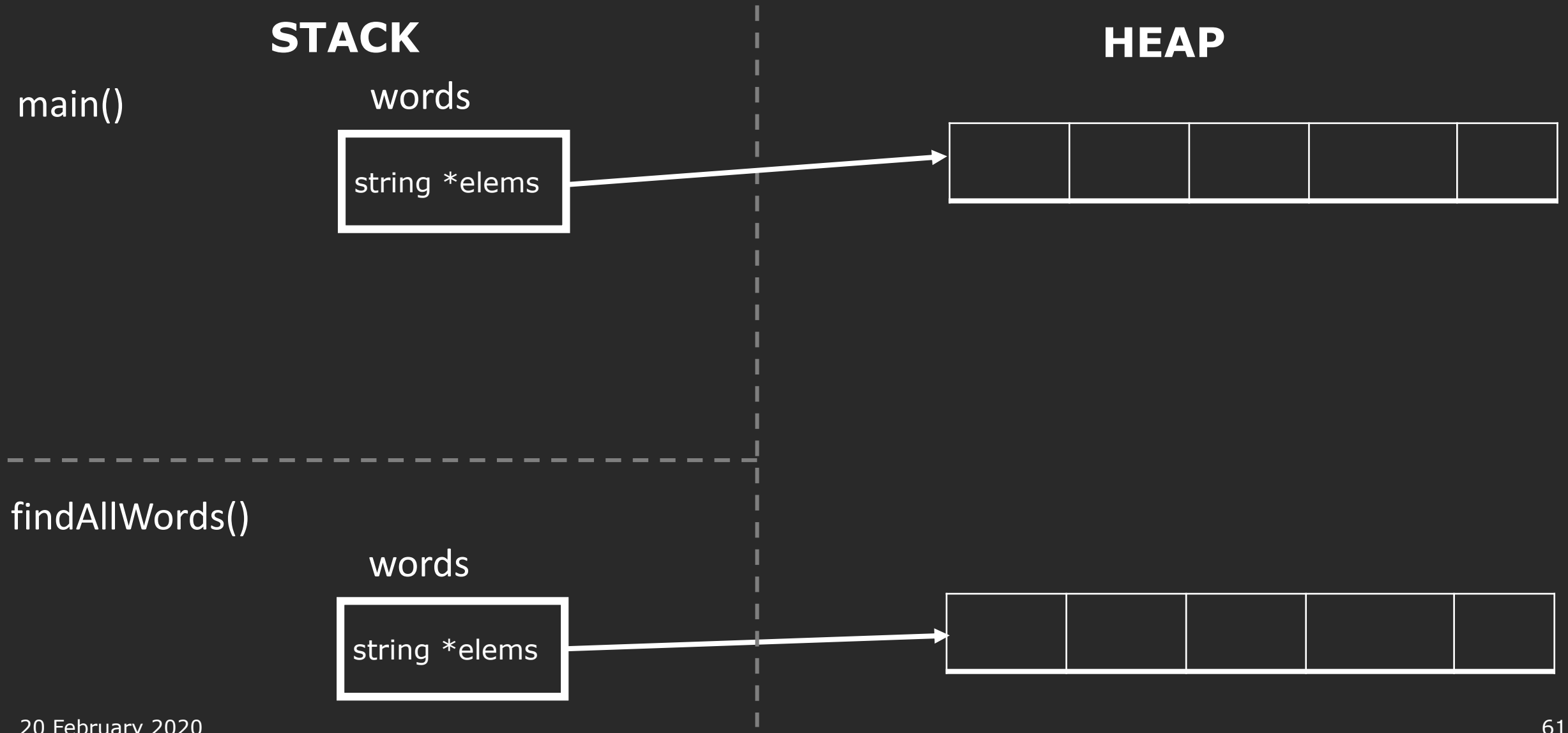
How many vectors are created?



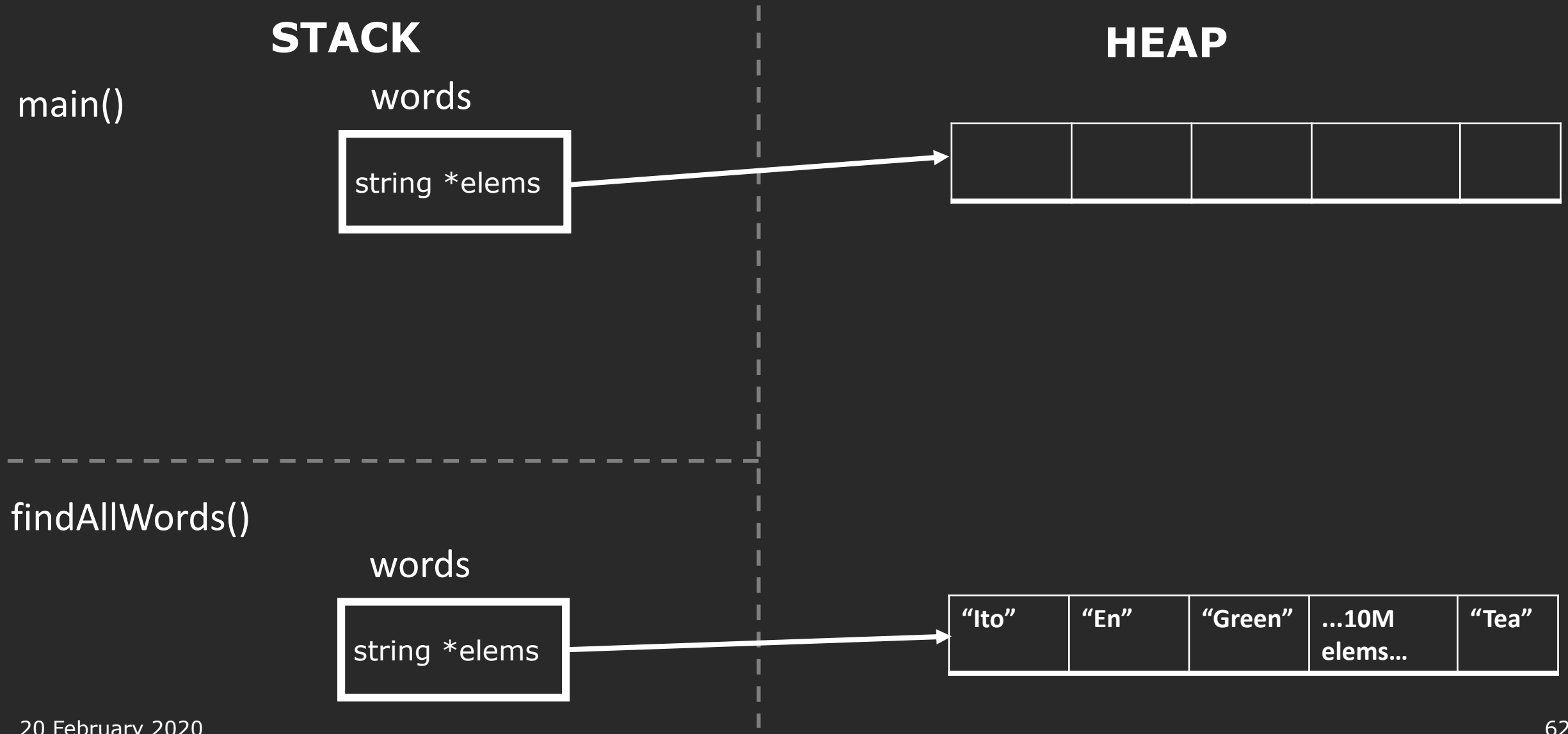
How many vectors are created?



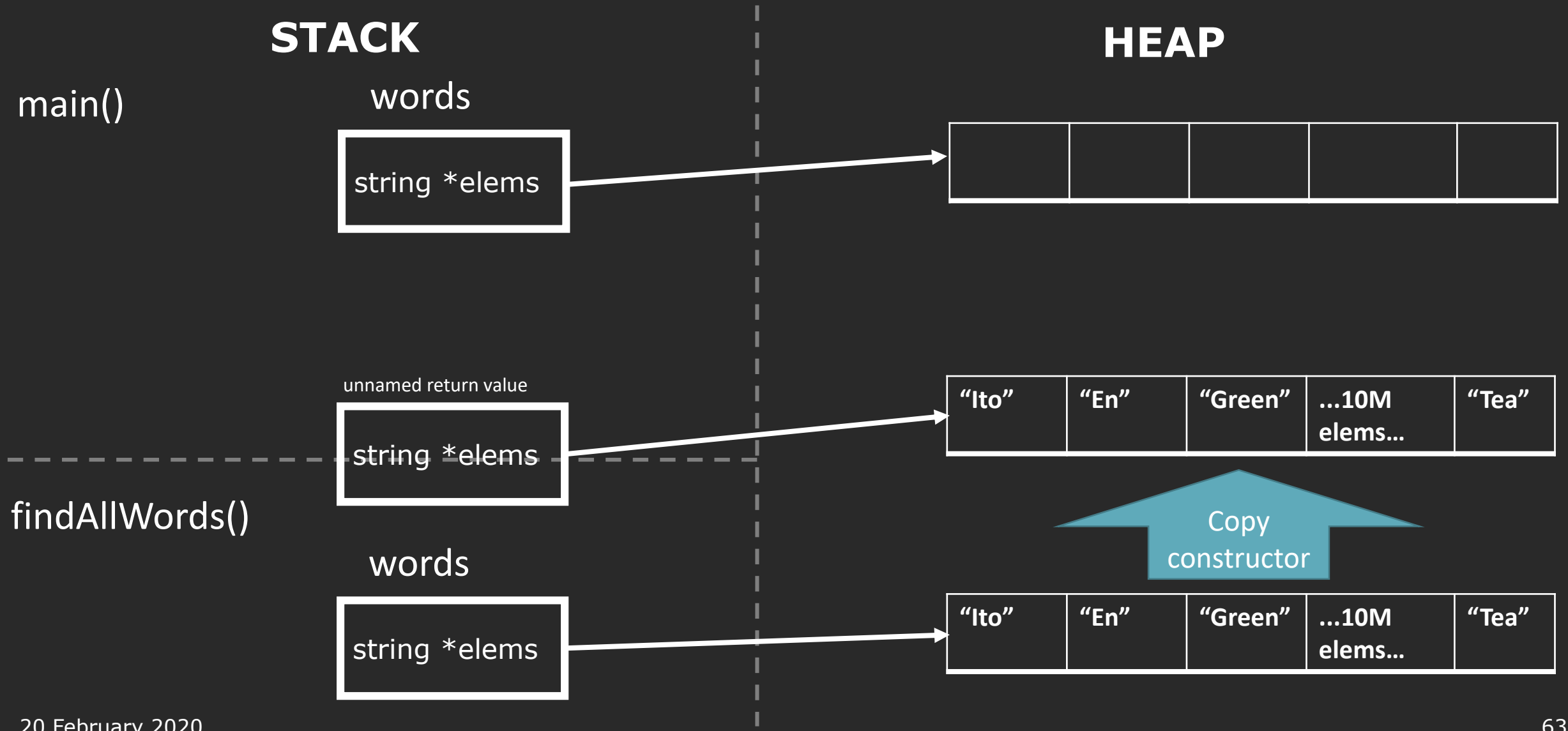
How many vectors are created?



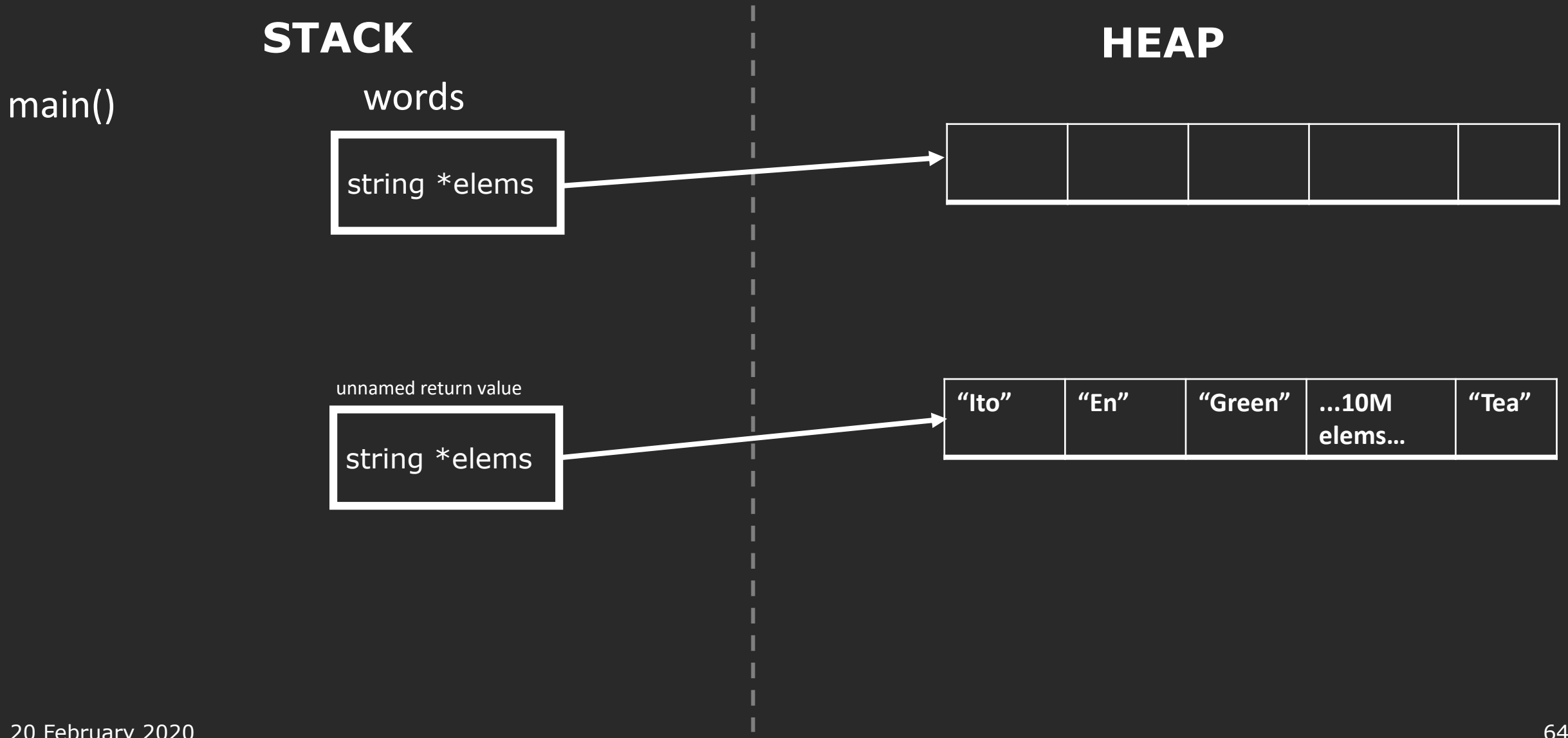
How many vectors are created?



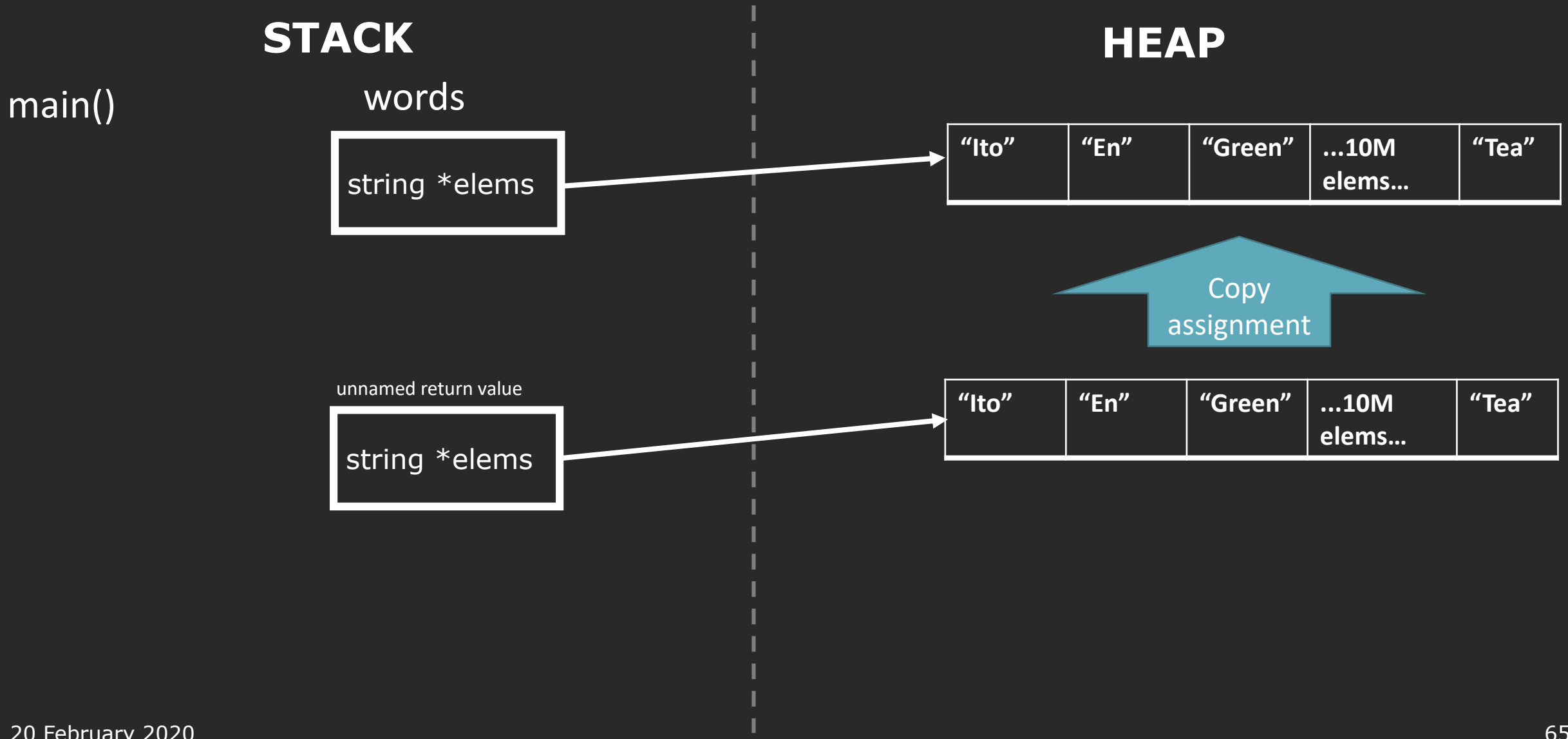
How many vectors are created?



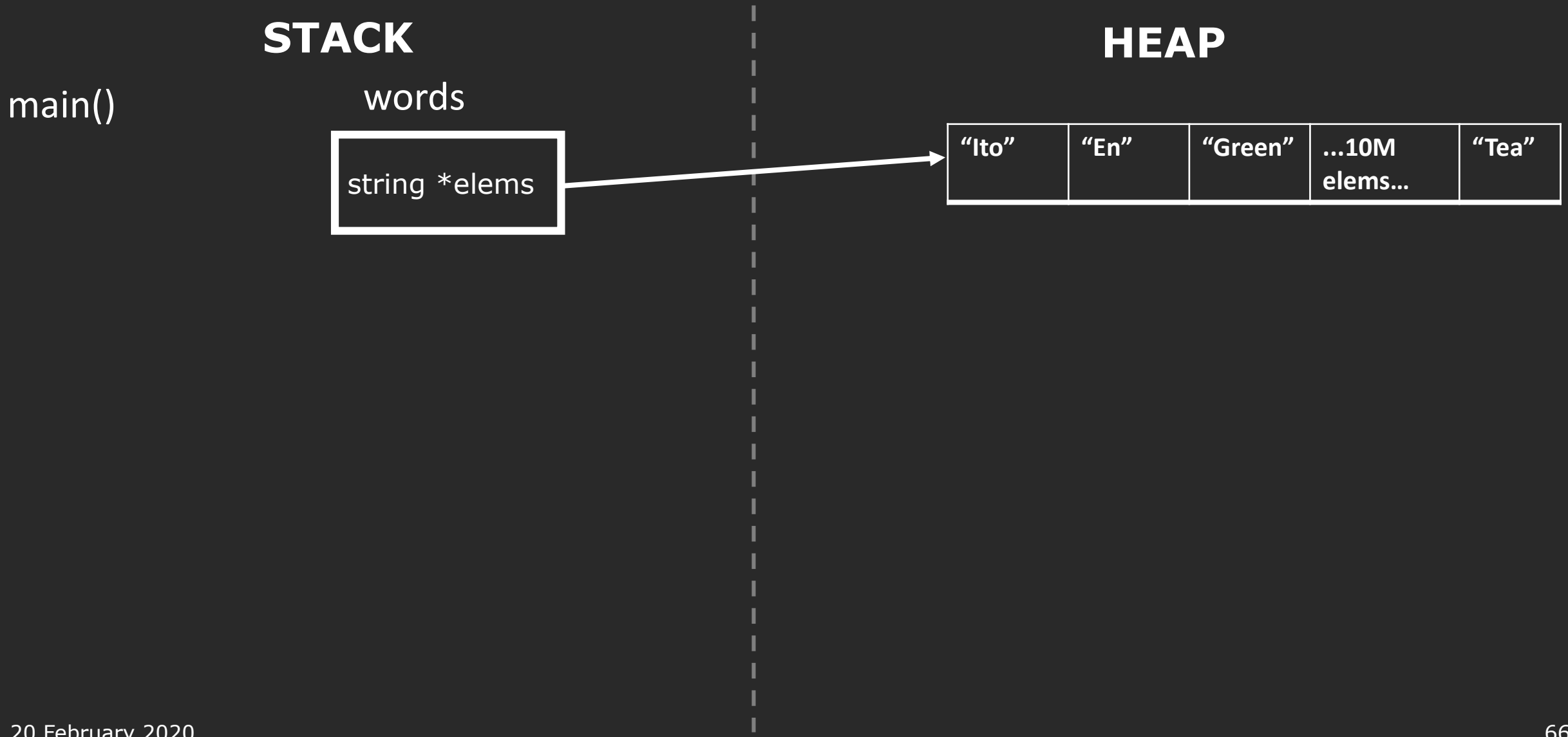
How many vectors are created?



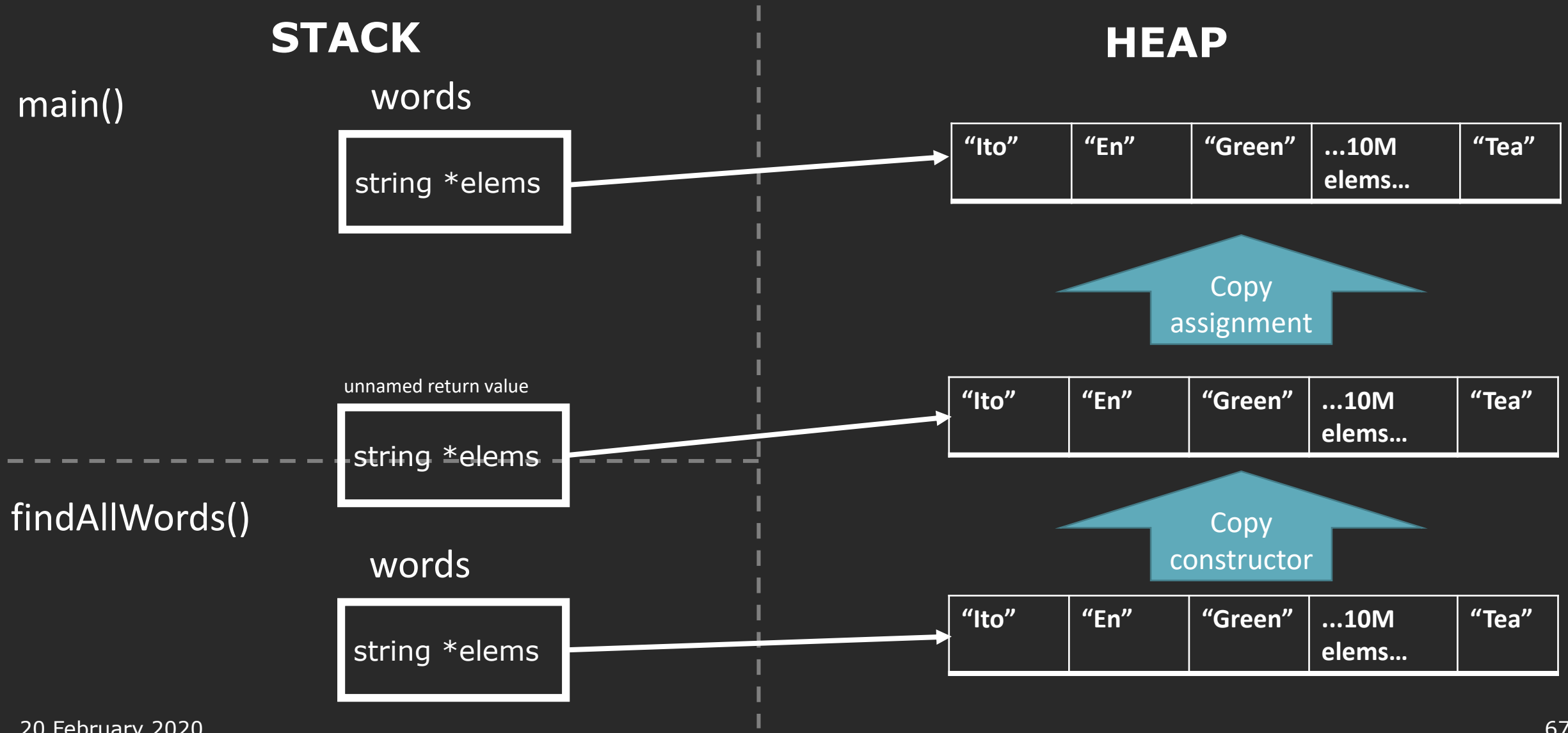
How many vectors are created?



How many vectors are created?



That is a lot of copies.



copy elision and return value optimization (RVO)

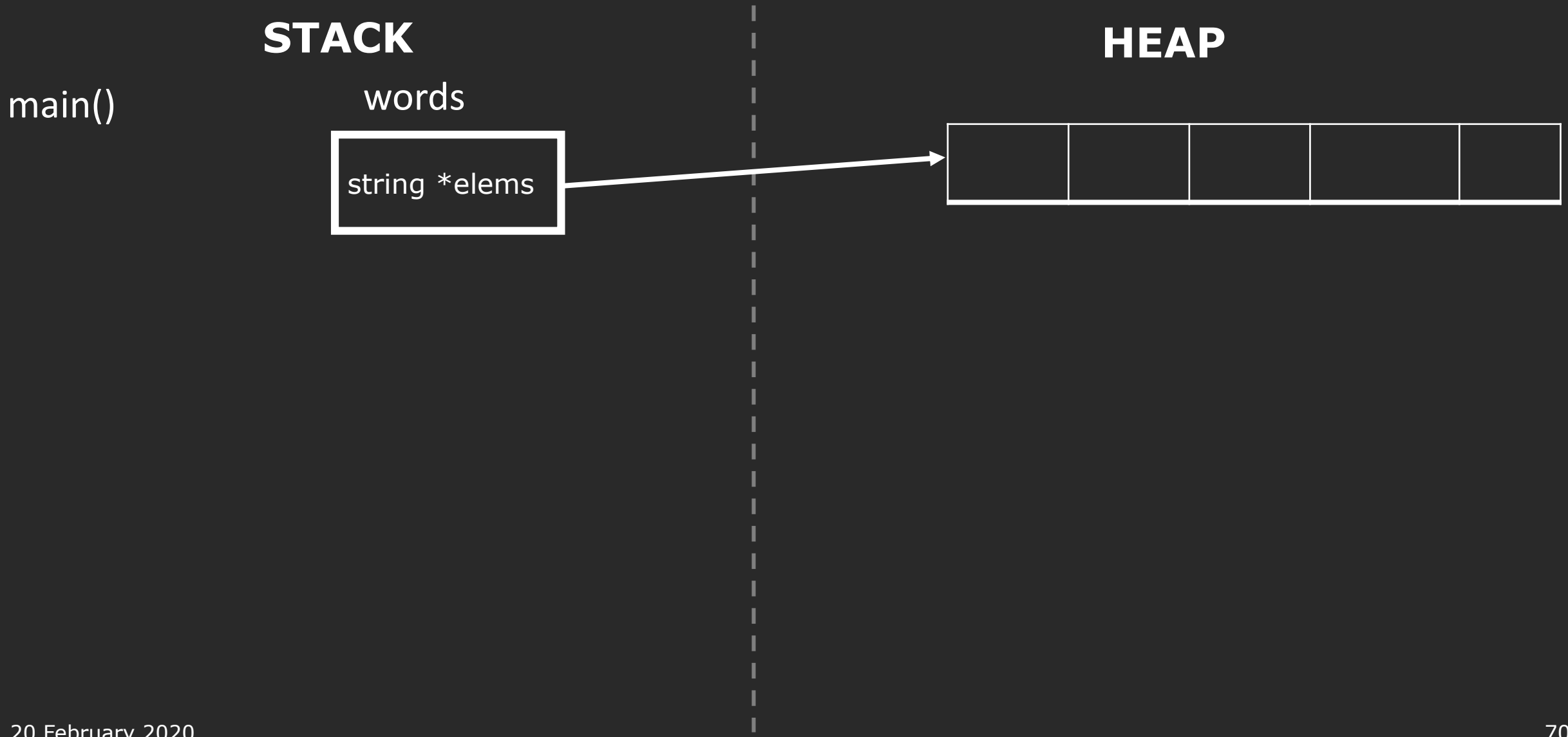
In practice: copy elision.

STACK

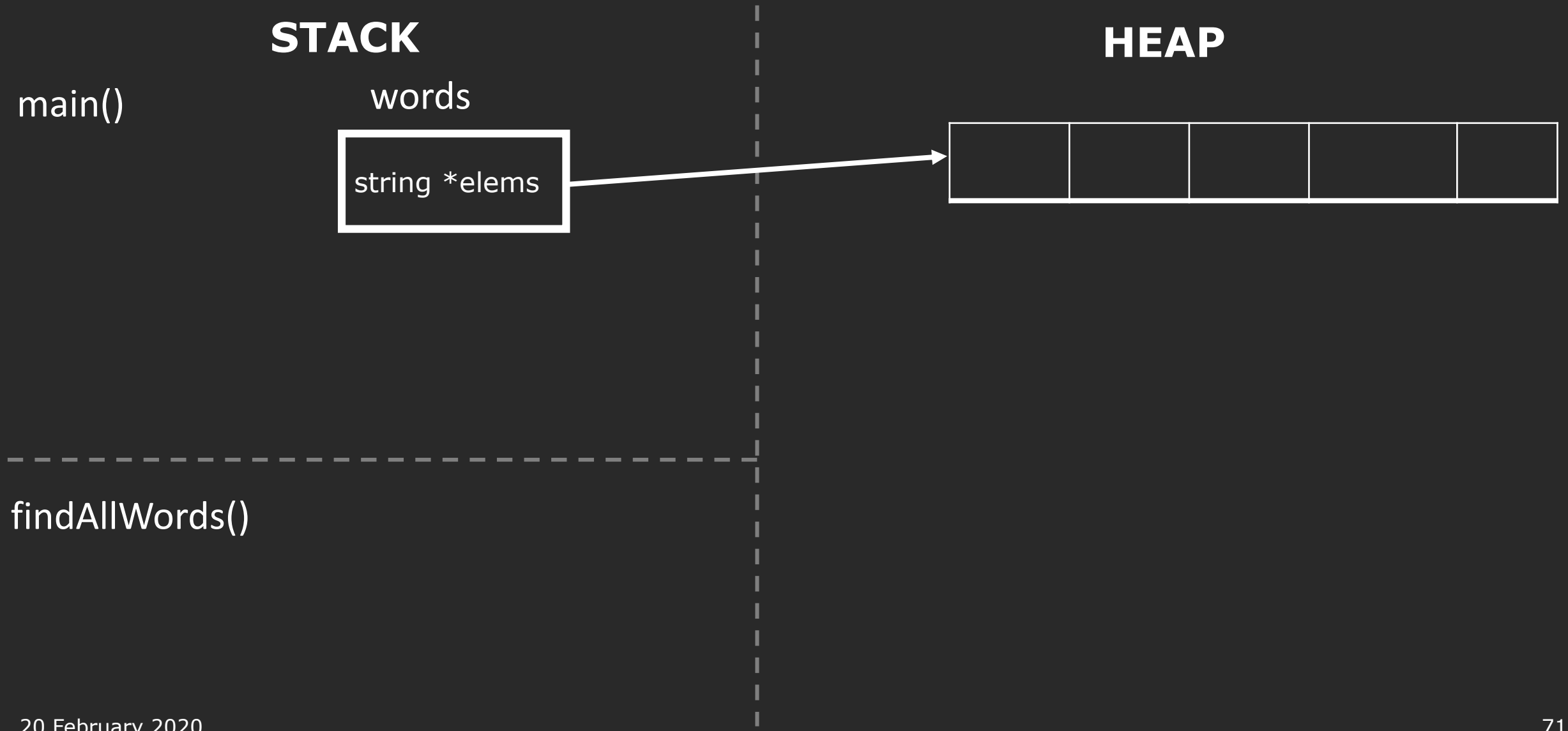
HEAP

main()

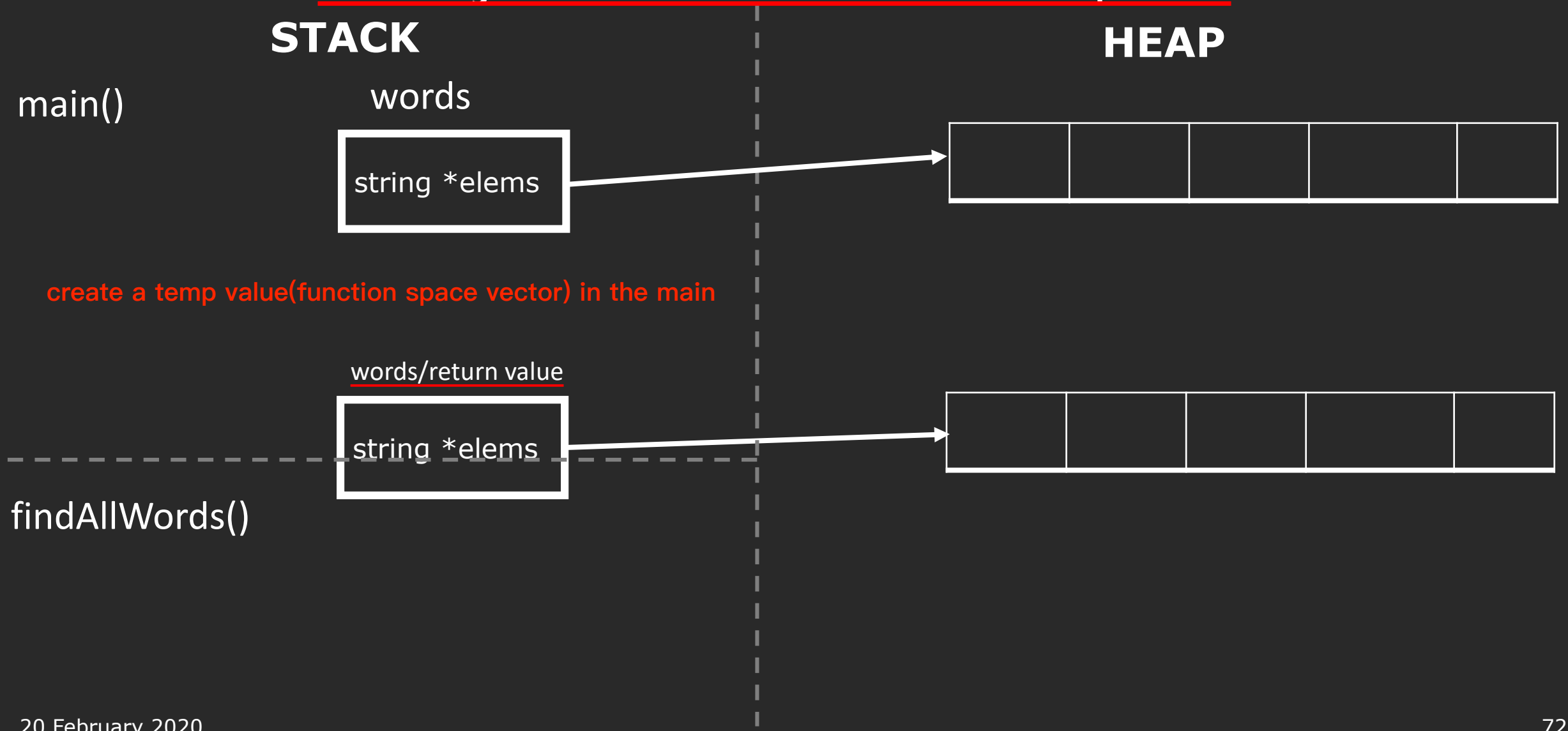
In practice: copy elision.



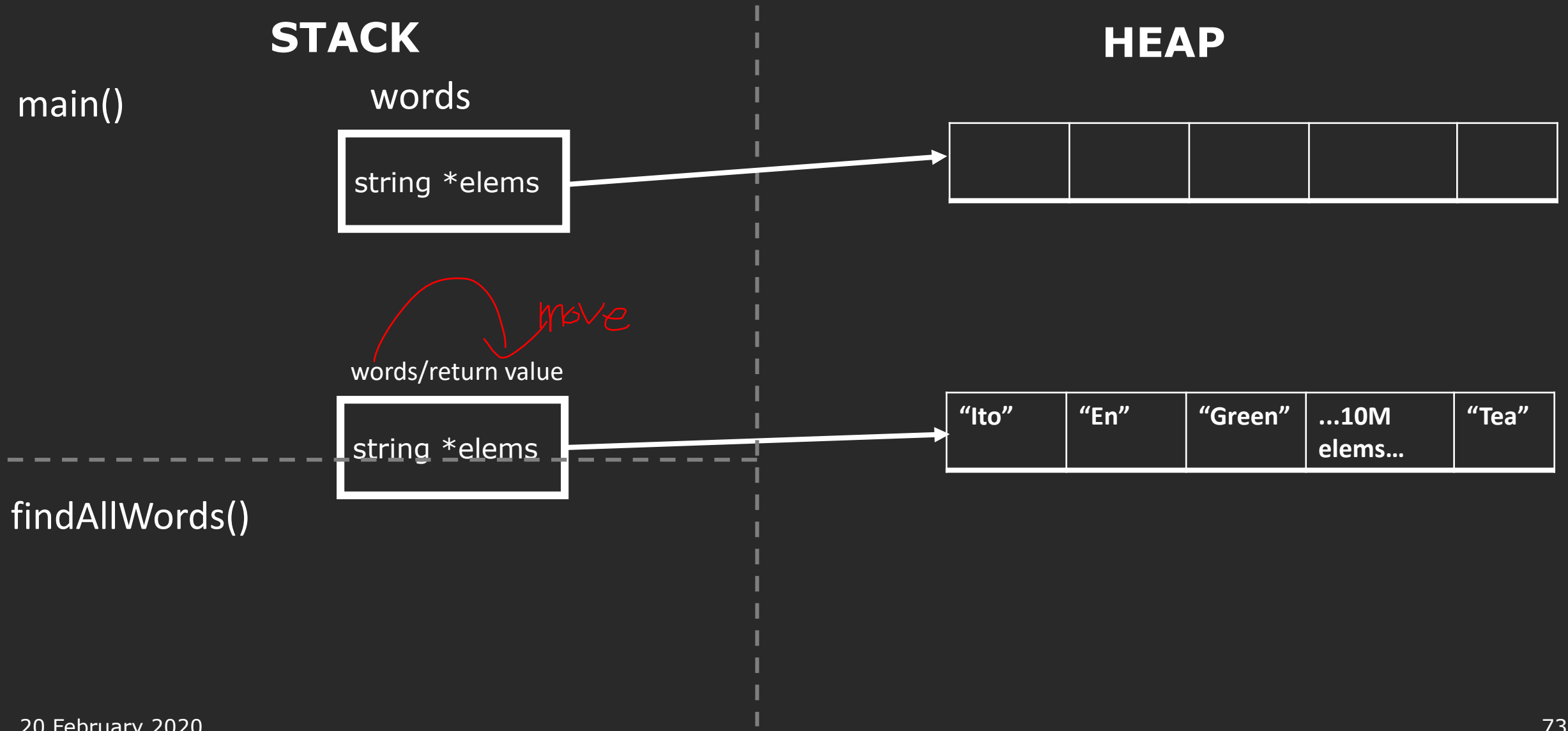
In practice: copy elision.



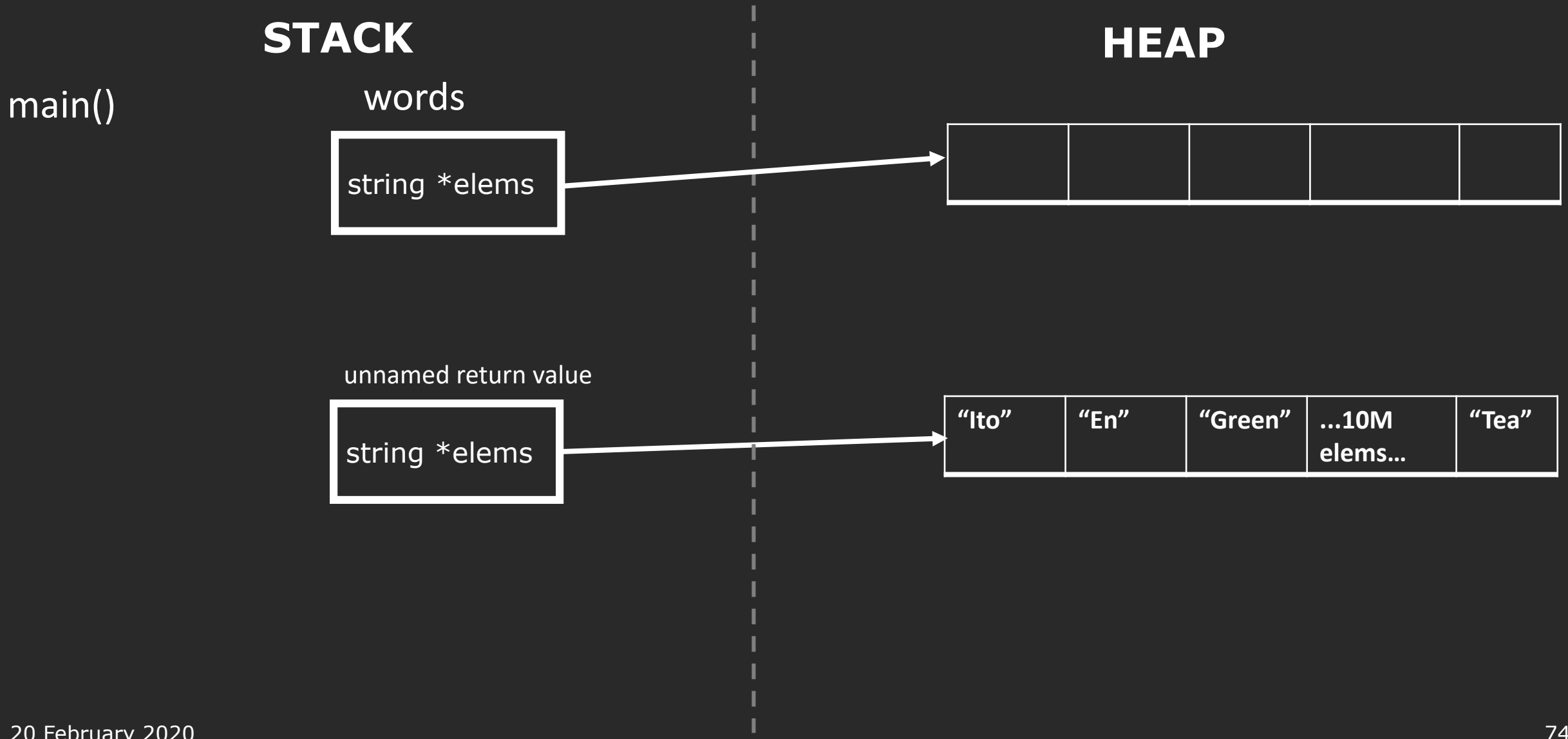
Compiler: "I know this vector is going to be returned, so I'll just create it in main's space.



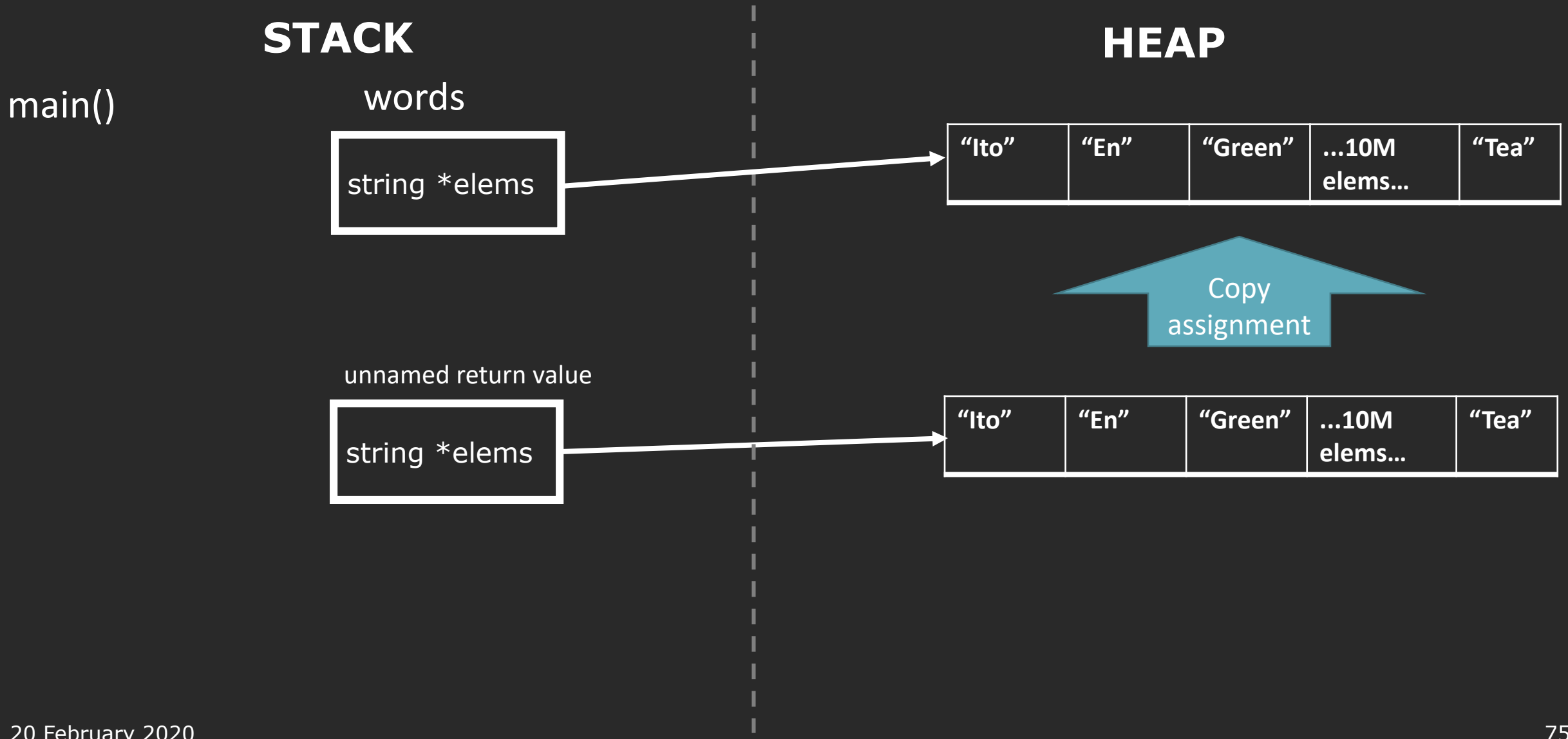
The words are still added as normal.



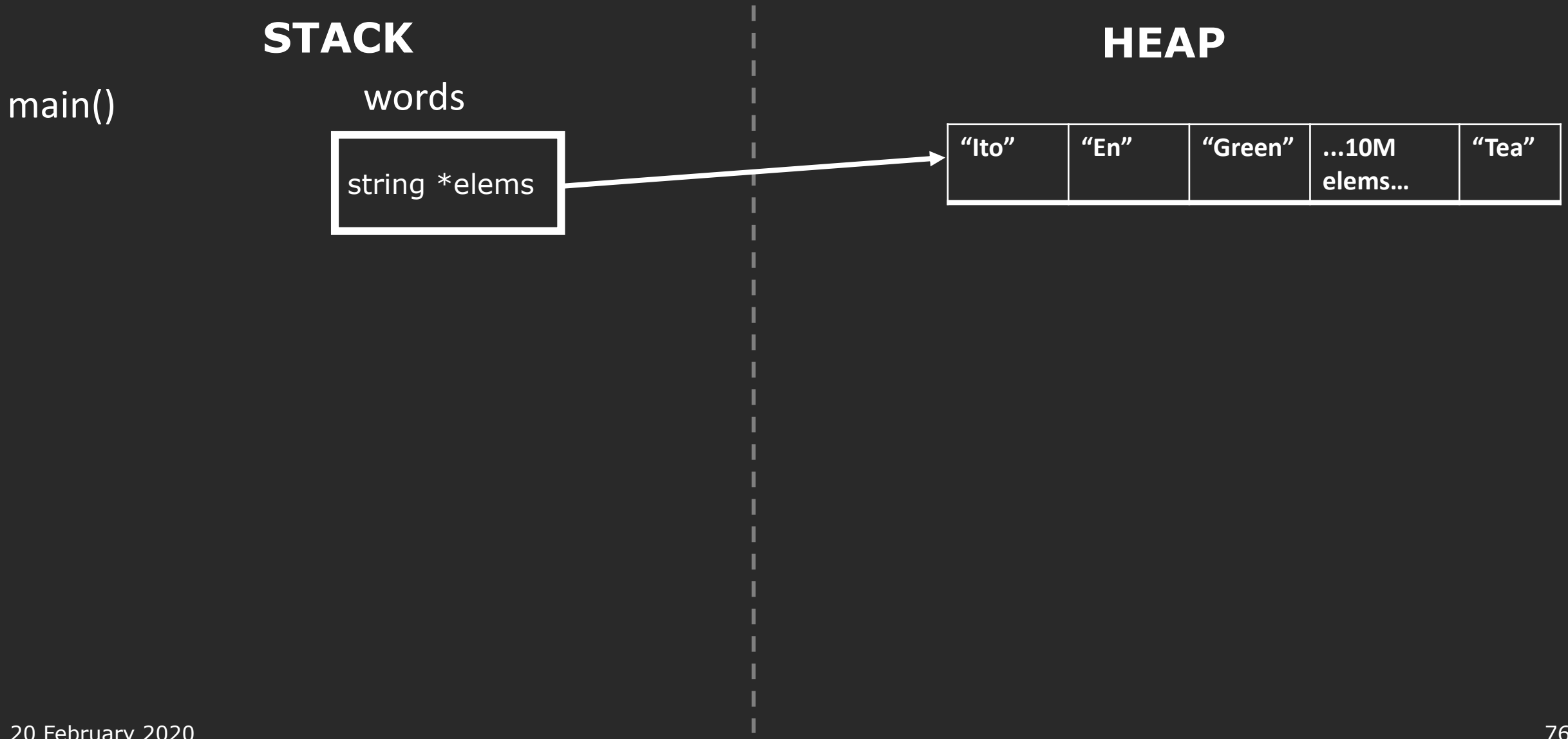
We get to skip the copy constructor call for the return.



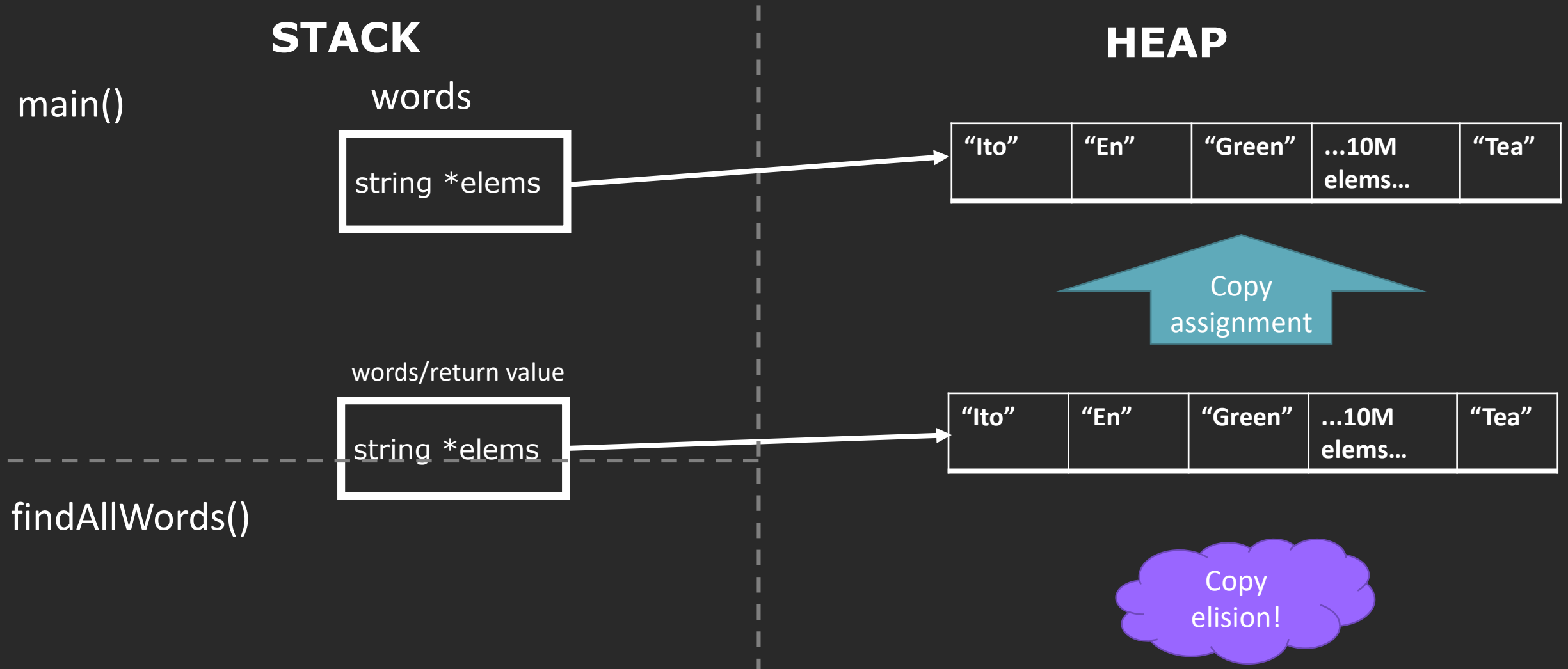
We get to skip the copy constructor call for the return.



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



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



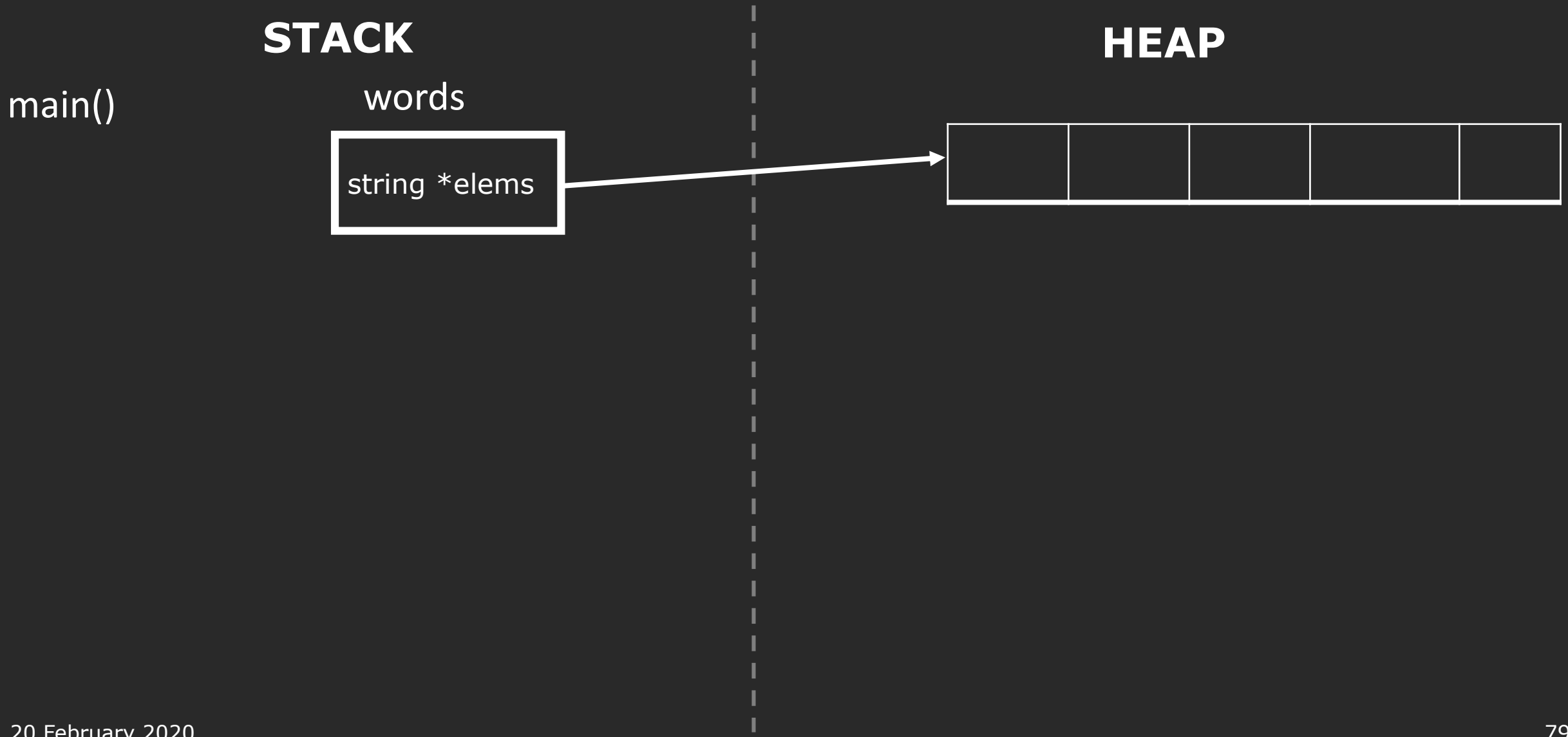
Here’s a more clever idea?

STACK

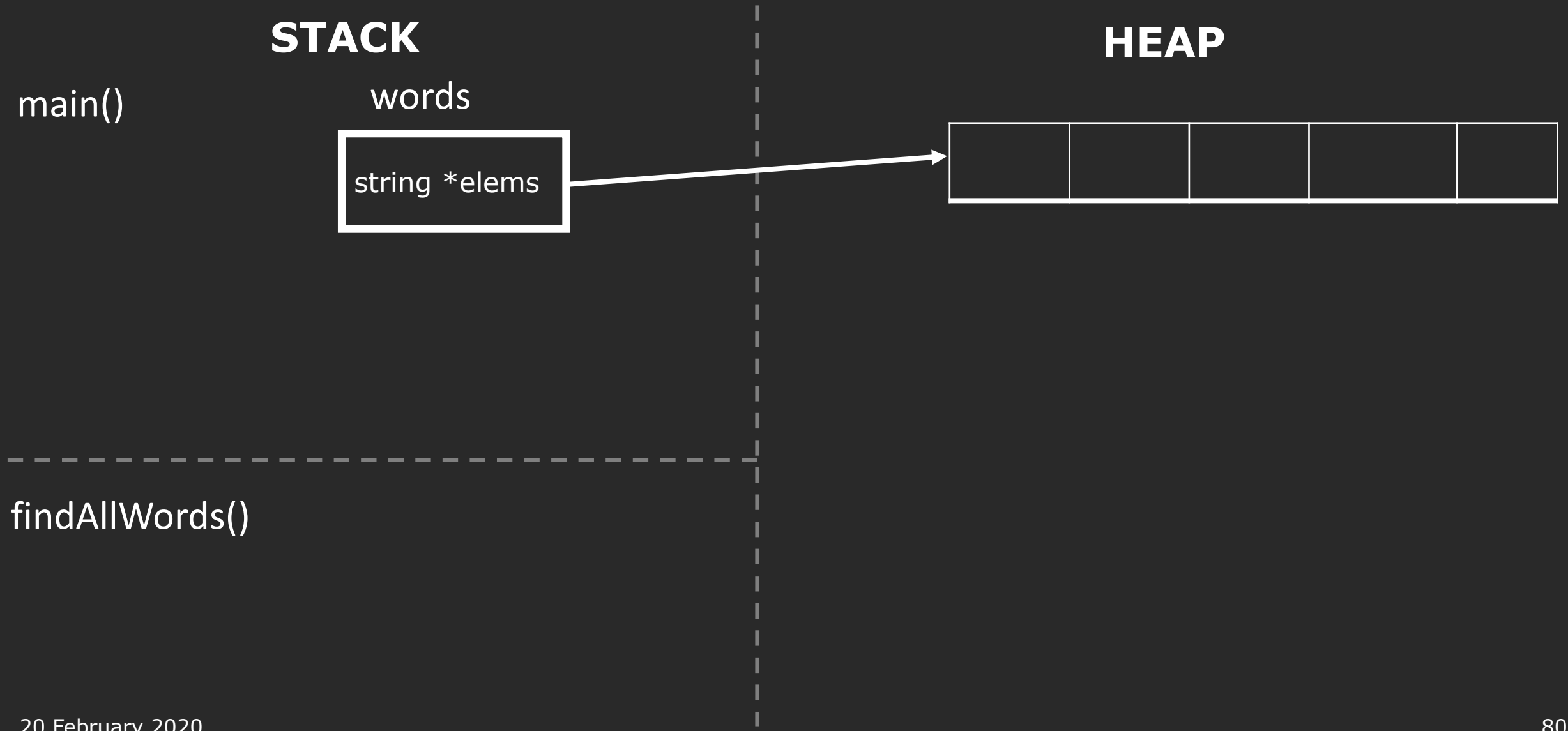
main()

HEAP

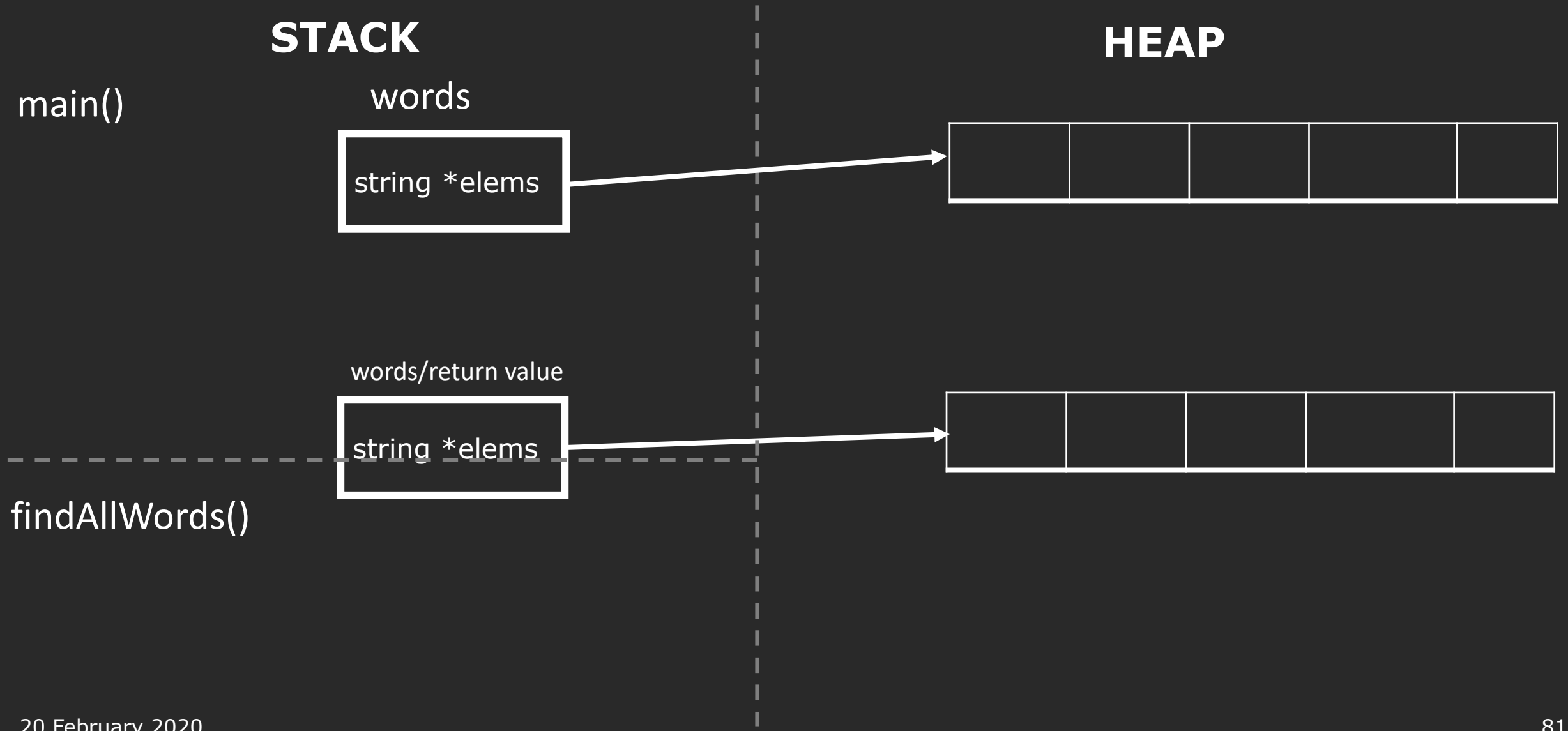
Here’s a more clever idea?



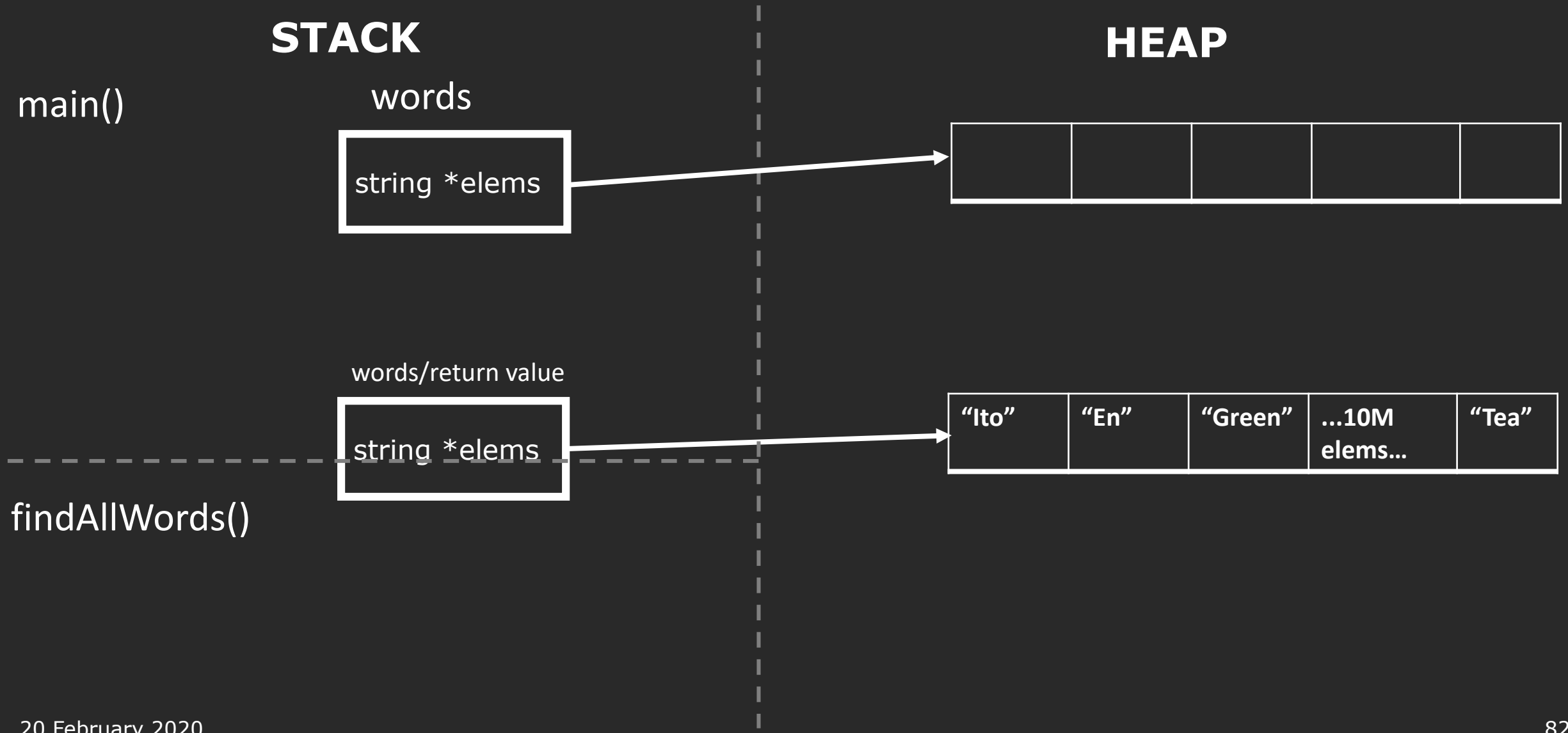
Here’s a more clever idea?



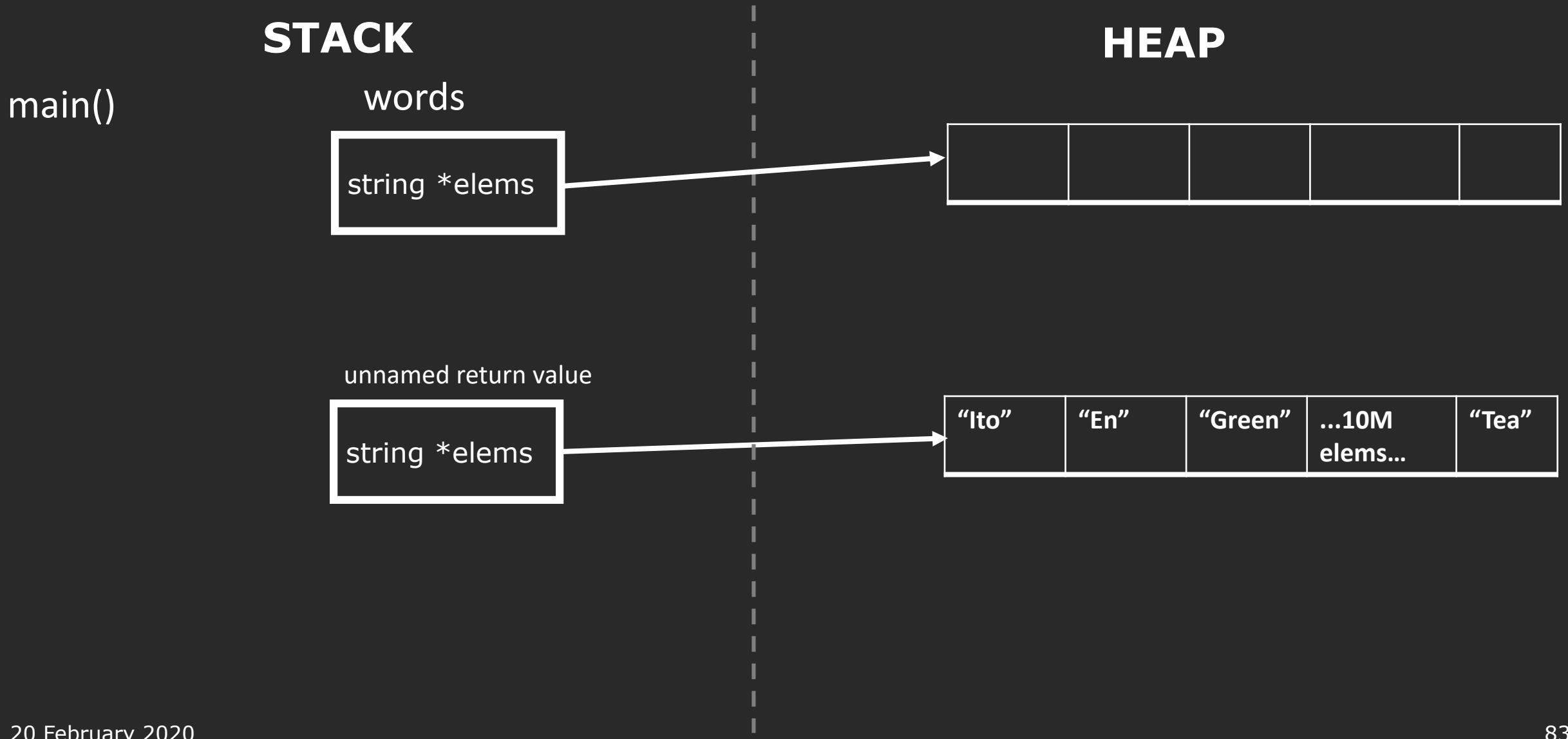
Here’s a more clever idea?



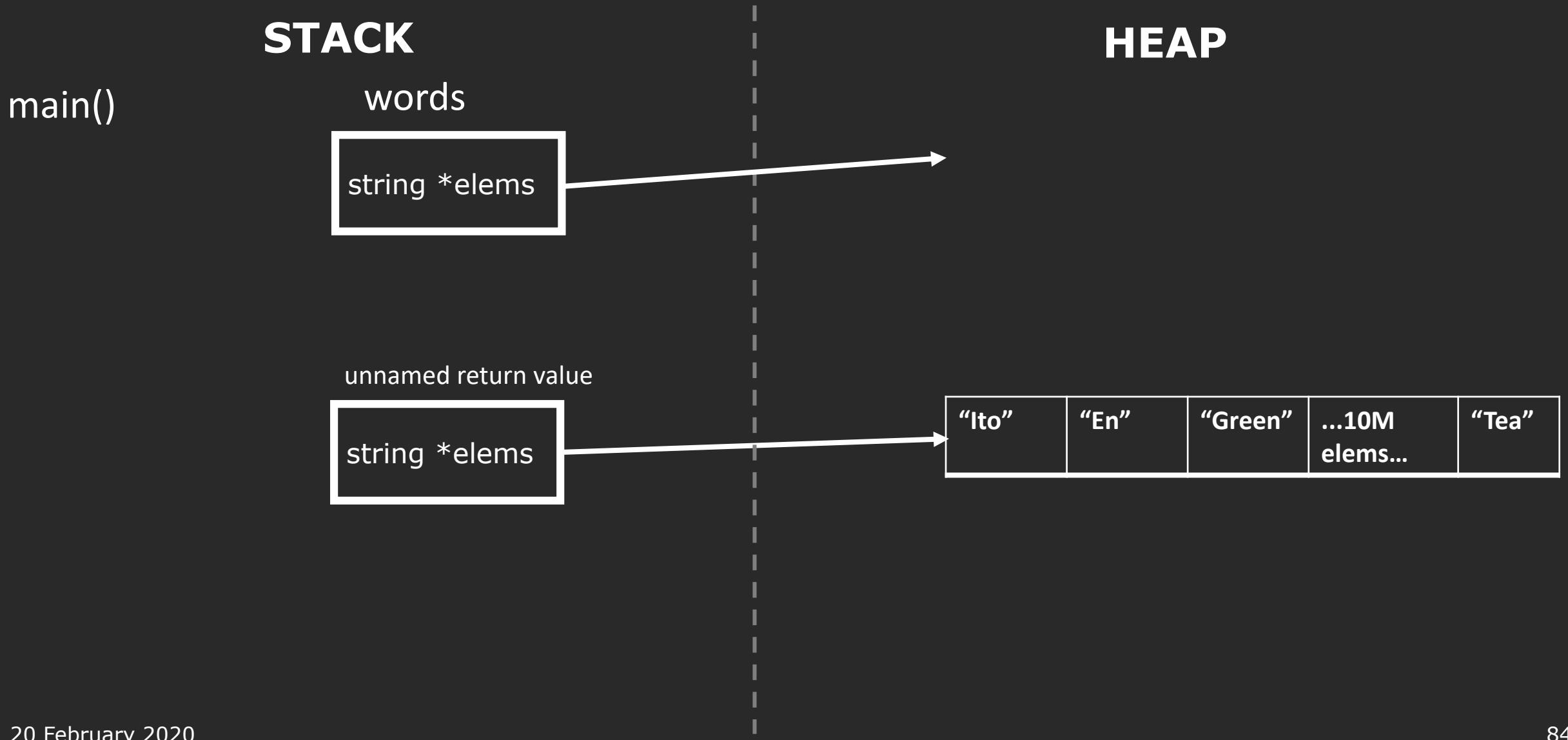
Here’s a more clever idea?



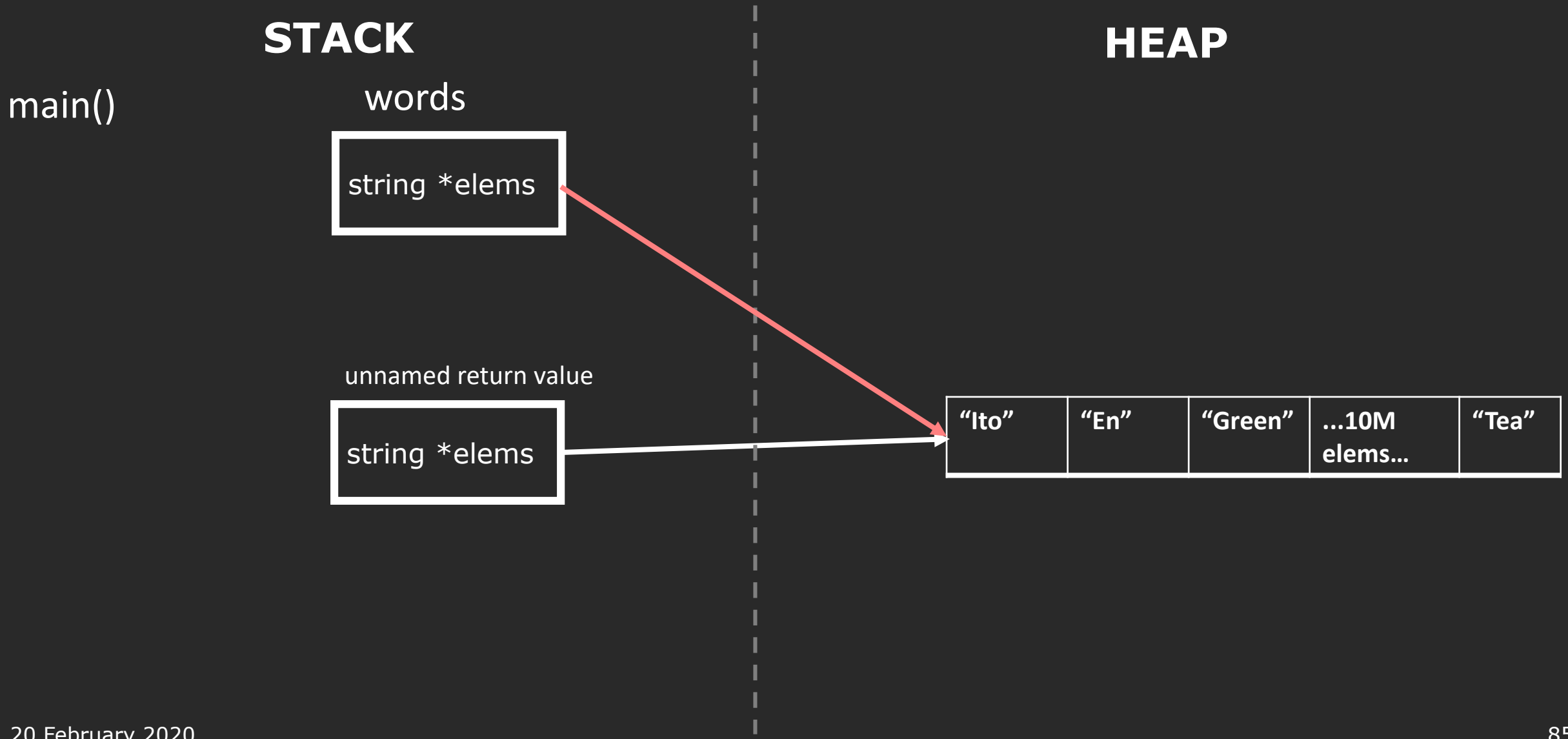
Here's a more clever idea?



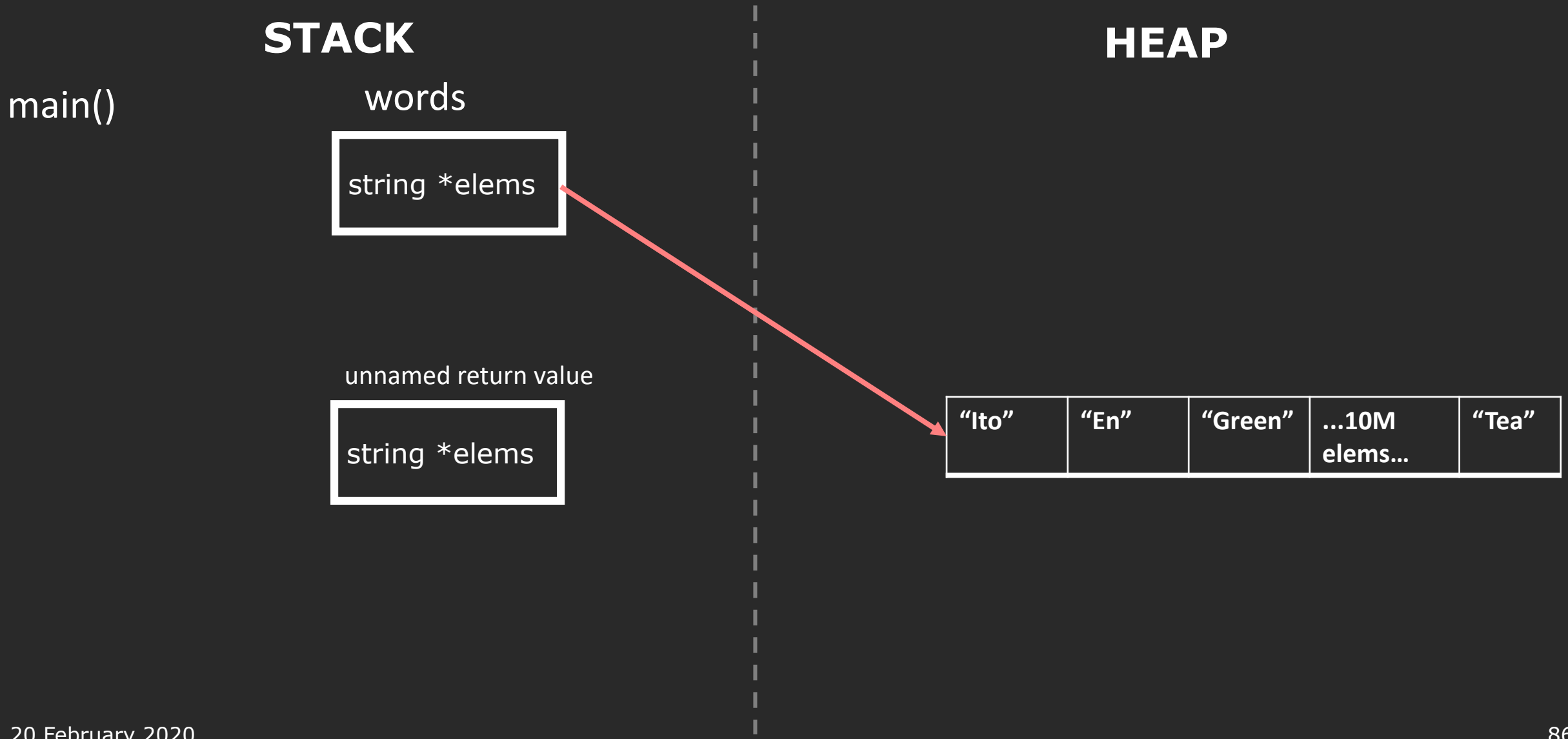
Let’s get rid of that empty array.



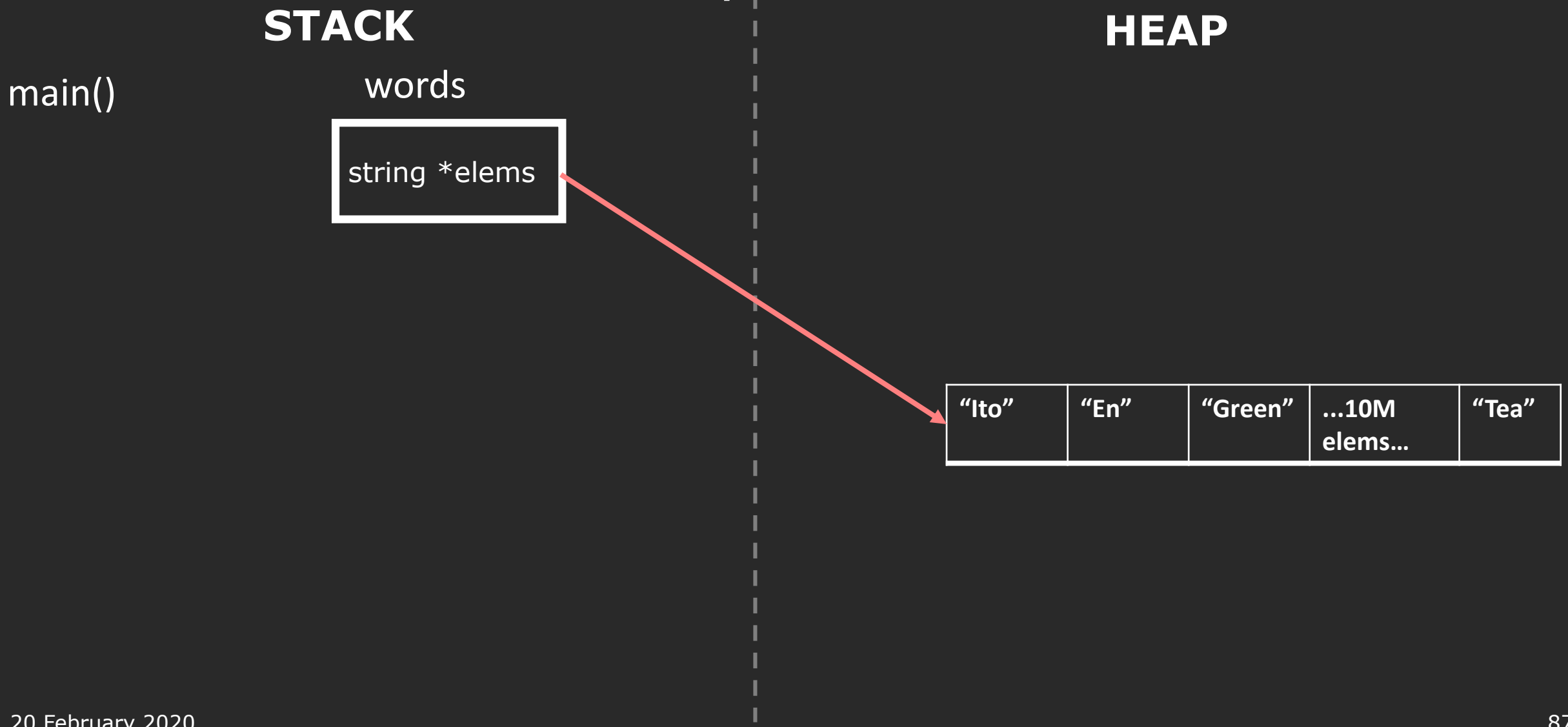
Steal the array of the unnamed return value.



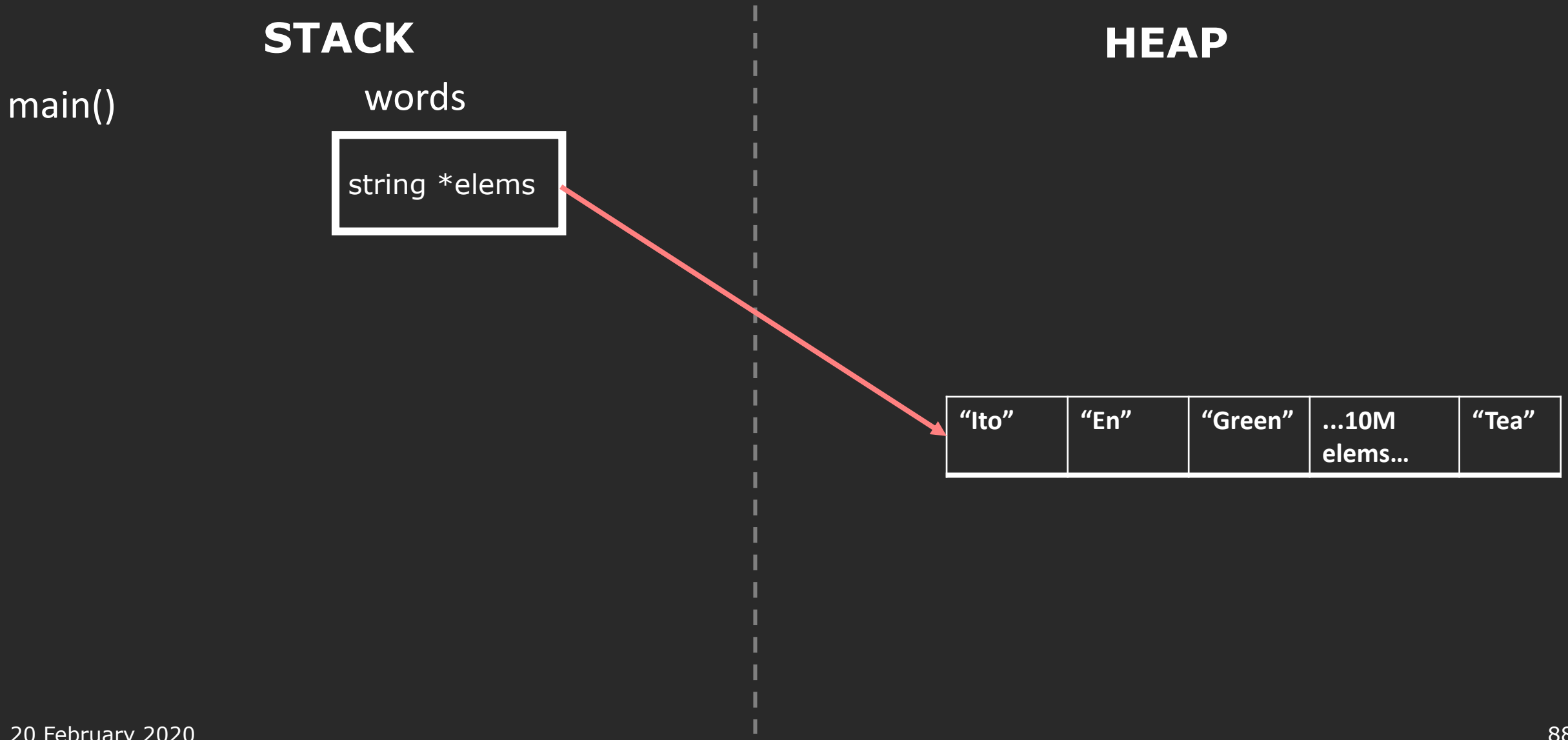
Evict the unnamed return value's claim over the array.



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

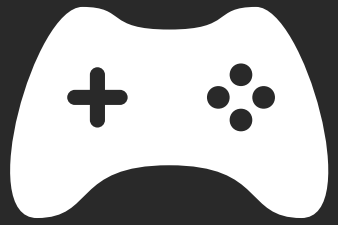


Zero unnecessary copies!!!



Questions to ponder when procrastinating in Week 7.

- Are there instances where we can only copy, but NOT move?
- Are there instances where we can move, but NOT copy?
- Are there instances where we can either copy or move, and we PREFER move?
- How can a compiler tell whether we are ALLOWED to move?
- How does move impact compiler optimizations like RVO?



Next time

Move Semantics