Operators

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.

Where we're going...

CS 106B has covered the absolute barebones of C++ class design.

We will cover the rest:

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

Key questions we will answer today

- Which operators can be overloaded?
- How can we define operators for our own classes?
- What should we consider when designing those operator functions?

Game Plan

- operator overloading
- canonical forms
- POLA

operator overloading

Name as many operators as you can!

There are 40 (+4) operators you can overload!

Arithmetic	+	-	*	/	%		
	+=	-=	*=	/=	%=		
Bitwise		&	|	~	!		
Relational	==	!=	<	>	<=	>=	<=>
Stream	<<	>>	<<=	>>=			
Logical	&&	||	^	&=	|=	^=	
Increment	++	--					
Memory	->	->*	new	new []	delete	delete []	
Misc	( )	[ ]	,	=		co_await	

There are 40 (+4) operators you can overload!

Arithmetic	+	-	*	/	%		
	+=	-=	*=	/=	%=		
Bitwise		&	|	~	!		
Relational	==	!=	<	>	<=	>=	<=>
Stream	<<	>>	<<=	>>=			
Logical	&&	||	^	&=	|=	^=	
Increment	++	--					
Memory	->	->*	new	new []	delete	delete []	
Misc	( )	[ ]	,	=		co_await	

C++ knows how operators work for primitive types.

```
int i = 0;
double d{2.3};
i++;
d -= 3;
i <<= 2;
a = d > 0 ? 1 : 7;
```

How does C++ know how to apply operators to user-defined classes?

```
vector<string> v{“Hello”, “World”};
cout << v[0];
v[1] += “!”;
```

C++ tries to call these functions.

```
vector<string> v{“Hello”, “World”};
cout.operator<<(v.operator[](0));
v.operator[](1).operator+=(“!”);
```

Or these ones.

```
vector<string> v{“Hello”, “World”};
operator<<(cout, v.operator[](0));
operator+=(operator[](v, 1), “!”);
```

Indeed, the people who wrote the STL wrote these functions.

```
ostream& operator<<(ostream& s, const string& val) {
    ???
}

// must be member, technically it's prob a template
string& vector<string>::operator[](size_t index) const {
    ???
}

string& operator+=(string& lhs, const string& rhs) {
    ???
}
```

examples

Let’s try adding the += operator to our vector<string> class.

```
vector<string> v1;
v1 += “Hello”;
v1 += “World”; // we’re adding an element

vector<string> v2{“Hi”, “Ito”, “En”, “Green”, “Tea”};
v2 += v1; // we’re adding a vector
```

What should the function signature look like?

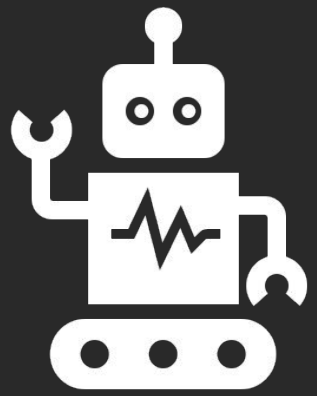
```
// append one element
[some return value] vector<string>::operator+=([some type] element) {
    push_back(element);
    return [something?];
}

// append another vector
[some return value] vector<string>::operator+=([some type] other) {
    for (string val : other) push_back(val);
    return [something?];
}
```

Why are these the function signatures?

```
// append one element
vector<string>& vector<string>::operator+=(const string& element) {
    push_back(element);
    return *this;
}

// append another vector
vector<string>& vector<string>::operator+=(const vector<string>& other) {
    for (string val : other) push_back(val);
    return *this;
}
```



Example

Operator overloading: vector, +=

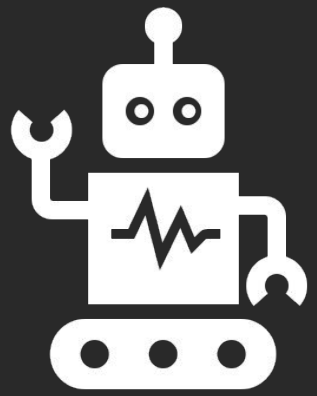
Concept check

1. Why are we returning a reference?
2. Why are we returning *this?
3. The += operator is a binary operator that takes a left and right operand, but the parameter only has the right operand. Where did the left operand go?

Key Takeaways

1. Respect the semantics of the operator. If it normally returns a reference to *this, make sure you do so!
2. When overloading operators as a member function, the left hand argument is the implicit *this.

Questions



Example

Operator overloading: vector, +

Let’s try adding the plus operator to our vector<string> class.

```
vector<string> operator+(const vector<string>& vec,
                         const string& element) {
    vector<string> copy = vec;
    copy += element;
    return copy;
}

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

Concept check

1. Why are we returning by value instead?
2. Why are both parameters const?
3. Why did we declare these as non-member functions?

Key Takeaways

1. The arithmetic operators return copies but don’t change the objects themselves. The compound ones do change the object.
2. Binary arithmetic operators should be implemented as non-member functions to maintain commutativity.
    - Ex.: vec + 3 should work just as well as 3 + vec
    - Ex.: 2 + “Hello” should work just as well as “Hello” + 2

General rule of thumb: member vs. non-member

1. Some operators must be implemented as members (eg. `[]`, `()`, `->`, `=`) due to C++ semantics.
2. Some must be implemented as non-members (eg. `<<`, if you are writing class for rhs, not lhs).
3. If unary operator (eg. `++`), implement as member.

General rule of thumb: member vs. non-member

4. If binary operator and treats both operands equally (eg. both unchanged) implement as non-member (maybe friend). Examples: `+`, `<`.
5. If binary operator and not both equally (changes lhs), implement as member (allows easy access to lhs private members). Examples: `+=`

Member vs. Non-Member:

Arithmetic	+	-	*	/	%		
	+=	-=	*=	/=	%=		
Bitwise		&	|	~	!		
Relational	==	!=	<	>	<=	>=	<=>
Stream	<<	>>	<<=	>>=			
Logical	&&	||	^	&=	|=	^=	
Increment	++	--					
Memory	->	->*	new	new []	delete	delete []	
Misc	( )	[ ]	,	=		co_await	

Questions

Bad Dad Joke of the Day 1:

- Why did the scarecrow earn a Nobel Prize?
- Because he was outstanding in his field!

Creds: Mikey

Bad Dad Joke of the Day 2:

- Q: Why did a Stanford CS student drop course work and go on a worldwide tour?
- A: Because the student believed that would get them the Turing Award.

Creds: Shiva

2-min stretch break!

Bad Dad Joke of the Day 1:

- Why did the scarecrow earn a Nobel Prize?
- Because he was outstanding in his field!

Creds: Mikey

Bad Dad Joke of the Day 2:

- Q: Why did a Stanford CS student drop course work and go on a worldwide tour?
- A: Because the student believed that would get them the Turing Award.

Creds: Shiva

2-min stretch break!

Bad Dad Joke of the Day 3:

- What's an astronaut's favorite part of a computer?
- The space bar.

Creds: Omar R

Bad Dad Joke of the Day 4:

- Did you hear about the new restaurant on the moon???
- They say the food's great, but the atmosphere is lacking.

Creds: Omar M

The subscript operator is one that must be implemented as a member.

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

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

Concept check

1. Why are we returning a reference?
2. Why are there two versions, one that is a const member, and one that is a non-const member?
3. Why are we not performing error-checking?

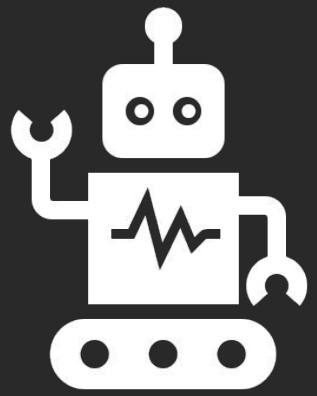
The client could call the subscript for both a const and non-const vector.

```
vector<string> v1{“Green”, “Black”, “Oo-long”};

const vector<string> v2{“16.9”, “fluid”, “ounces”};

v1[1] = 0; // calls non-const version, v1[1] is reference

int a = v2[1]; // calls const version, this works
v2[1] = 0; // ERROR: does not work, v2[1] is const
```



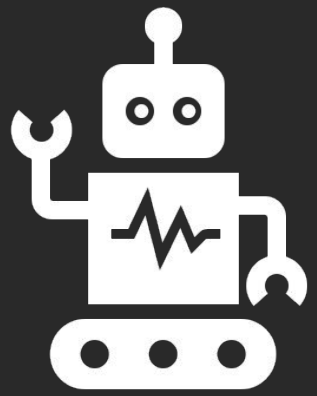
Example

Operator overloading: vector, []
(breakout rooms at the end of class)

What does it mean to << a vector<string> into an ostream?

```
vector<string> v1{“Green”, “Black”, “Oo-long”};
const vector<string> v2{“16.9”, “fluid”, “ounces”};

cout << v1 << “ “ << v2;
```



Example

Operator overloading: vector, <<

Let’s try overloading the stream insertion operator!

```
ostream& operator<<(ostream& out, const vector<T>& rhs) {
    std::copy(rhs.begin(), rhs.end(),
                      std::ostream_iterator<T>(out, " "));
    out << '\n';
    return out;
}
```

Concept check

1. Why is the ostream parameter passed by non-const reference, and the vector passed by const reference?
2. Why are we returning a reference?
3. Why are we implementing this as a non-member function?

Key Takeaways

1. Always think about const-ness of parameters. Here, we are modifying the stream, not the vector.
2. Return reference to support chaining << calls.
3. Here we are overloading << so our class works as the rhs…but we can’t change the class of lhs (stream library).

Questions

Let’s implement the << operator for the Time class...

```
class Time {
public:
    Time(int hour, int minute);
    ~Time();
    // other methods

private:
    int hour, minute;
}

// Non-member functions are defined outside of the class
ostream& operator<<(ostream& out, const Time& t) {
    out << t.hour << “:”<< t.minute;
    return out;
}
```

Problem: We can’t access the private members of Time!

```
class Time {
public:
    Time(int hour, int minute);
    ~Time();
    // other methods

private:
    int hour, minute;
}

// Non-member functions are defined outside of the class
ostream& operator<<(ostream& out, const Time& t) {
    out << t.hour << “:”<< t.minute;
    return out;
}
```

Declare non-member functions as friends of a class to give them access to private members.

```
class Time {
public:
    Time(int hour, int minute);
    ~Time();
    // other methods

private:
    int hour, minute;
    friend ostream& operator<<(ostream& out, const Time& t);
}
```

Concept check

Why do you want friends?

Key Takeaway

Because they help get you through Week 6 😊

If you have to implement an operator as a non-member function, but need access to the private member variables.

Questions

summary of takeaways

Summary of Takeaways

Think about the semantics of the operators (parameter, return value, const-ness, references)

Follow the rule-of-thumb for member vs. non-member/friends.

Principle of Least Astonishment (POLA)

What do you think it means?

Principle of Least Astonishment (POLA)

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

Principle of Least Astonishment (POLA)

Key Idea #1:

Design operators primarily to mimic conventional usage.

Principle of Least Astonishment (POLA)

```
Time start {15, 30};
Time end {16, 20};
if (start < end) {  // obvious
    start += 10;    // is this adding to hour or min?
} else {
    end--;          // again, hour or min?
    end, 3, 4, 5;   // wat is this?
}
```

Principle of Least Astonishment (POLA)

Key Idea #2:

Use nonmember functions for symmetric operators.

Principle of Least Astonishment (POLA)

```
class vector{
public:
   vector(size_t capacity);
   ~vector();
   // other methods
   vector<T>& operator+(T elem);
   vector<T>& operator+(const vector<T>& rhs);
private:
   value_type* _elems;
   // other private members
}
```

Principle of Least Astonishment (POLA)

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

// equivalent to a.operator+(1), compiles
if (a + 1 == b) cout << “I <3 operators!”;
// equivalent to 1.operator+(a), does not compile
if (1 + a == b) cout << “I <3 operators!”;
```

Principle of Least Astonishment (POLA)

```
class vector{
public:
   vector(size_t capacity);
   ~vector();
   // other methods
private:
   value_type* _elems;
   vector<T>& operator+(const vector<T>& lhs, T elem);
   vector<T>& operator+(const vector<T>& lhs,
                        const vector<T>& rhs);
}
```

Principle of Least Astonishment (POLA)

Key Idea #3:

Always provide all out of a set of related operators.

There are 40 (+4) operators you can overload!

Arithmetic	+	-	*	/	%		
	+=	-=	*=	/=	%=		
Bitwise		&	|	~	!		
Relational	==	!=	<	>	<=	>=	<=>
Stream	<<	>>	<<=	>>=			
Logical	&&	||	^	&=	|=	^=	
Increment	++	--					
Memory	->	->*	new	new []	delete	delete []	
Misc	( )	[ ]	,	=		co_await	

Principle of Least Astonishment (POLA)

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

// if the following code works:
if (a < b) cout << “I <3 operators!”;
// then the following better work as well:
if (b > a) cout << “I <3 operators!”;
```

Principle of Least Astonishment (POLA)

```
// Non-Member Functions:
bool operator< (const vector<T>& lhs, const vector<T>& rhs) { /* do actual comparison */ }

bool operator> (const vector<T>& lhs, const vector<T>& rhs) { return rhs < lhs; }

bool operator<=(const vector<T>& lhs, const vector<T>& rhs) { return !(lhs > rhs); }

bool operator>=(const vector<T>& lhs, const vector<T>& rhs) { return !(lhs < rhs); }
```

Questions

looking ahead

There are a few more interesting operators.

Arithmetic	+	-	*	/	%		
	+=	-=	*=	/=	%=		
Bitwise		&	|	~	!		
Relational	==	!=	<	>	<=	>=	<=>
Stream	<<	>>	<<=	>>=			
Logical	&&	||	^	&=	|=	^=	
Increment	++	--					
Memory	->	->*	new	new []	delete	delete []	
Misc	( )	[ ]	,	=		co_await	

Automatic memory management: smart pointers (lecture 15)

```
unique_pointer<Node> ptr{new Node(0)}
ptr->next = nullptr;
```

Functors (lecture 7 – lambdas)

```
class GreaterThan {
  public:
    GreaterThan(int limit) : limit(limit) {}
    bool operator() (int val) {return val >= limit};
  private:
    int limit;
}
int main() {
  int limit = getInteger("Minimum for A?");
  vector<int> grades = readStudentGrades();
  GreaterThan func(limit);
  cout << countOccurences(grades.begin(), grades.end(), func);
}
```

You can define your own memory allocators!

operator new, operator new[]

Defined in header <new>

replaceable allocation functions

[[nodiscard]] (since C++20)

```
void* operator new  ( std::size_t count );                                  (1)
void* operator new[]( std::size_t count );                                  (2)
void* operator new  ( std::size_t count, std::align_val_t al);              (3) (since C++17)
void* operator new[]( std::size_t count, std::align_val_t al);              (4) (since C++17)
```

replaceable non-throwing allocation functions

[[nodiscard]] (since C++20)

```
void* operator new  ( std::size_t count, const std::nothrow_t& tag);        (5)
void* operator new[]( std::size_t count, const std::nothrow_t& tag);        (6)
void* operator new  ( std::size_t count,
                      std::align_val_t al, const std::nothrow_t&);          (7) (since C++17)
void* operator new[]( std::size_t count,
                      std::align_val_t al, const std::nothrow_t&);          (8) (since C++17)
```

non-allocating placement allocation functions

[[nodiscard]] (since C++20)

```
void* operator new  ( std::size_t count, void* ptr );                       (9)
void* operator new[]( std::size_t count, void* ptr );                       (10)
```

Advanced Multithreading Support (C++20)

```
awaiter operator co_await() const noexcept {
      return awaiter{ *this };
}
```

Three-way comparison operator (C++20)

```
std::strong_ordering operator<=> (const Time& rhs) {
    return hour <=> rhs.hour;
}

// The expression lhs <=> rhs returns an object such that:
//      (a <=> b) < 0 if lhs < rhs
//      (a <=> b) > 0 if lhs > rhs
//      (a <=> b) == 0 if lhs and rhs are equal/equivalent.
```

let’s back up one sec

Quick quiz. Based on what we wrote today, what is the result of the following?

```
vector<int> vec{3, 4, 5, 6};
vector<int> other = vec + 7;
other[0] = 1;
cout << vec[0]; // should still be 3
```

We lied...this code doesn't actually work.

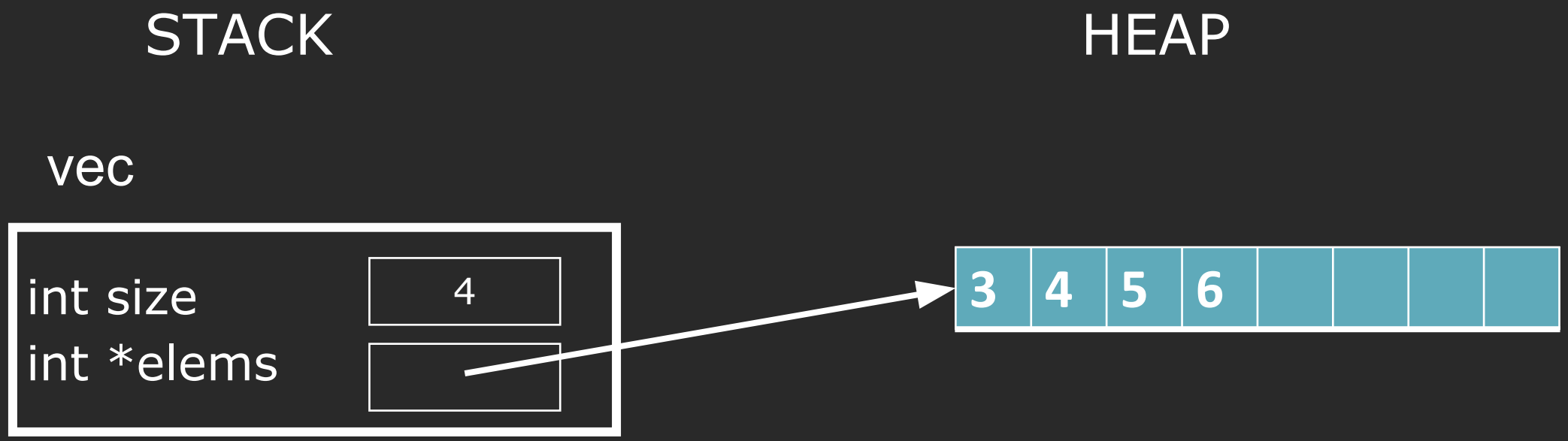
```
vector<T> operator+(const vector<string>& vec, const T& element) {
    vector<T> copy = vec;
    copy += element;
    return copy;
}
```

```
vector<int> vec{3, 4, 5, 6};
vector<int> other = vec + 7;
other[0] = 1;
cout << vec[0]; // should still be 3
```

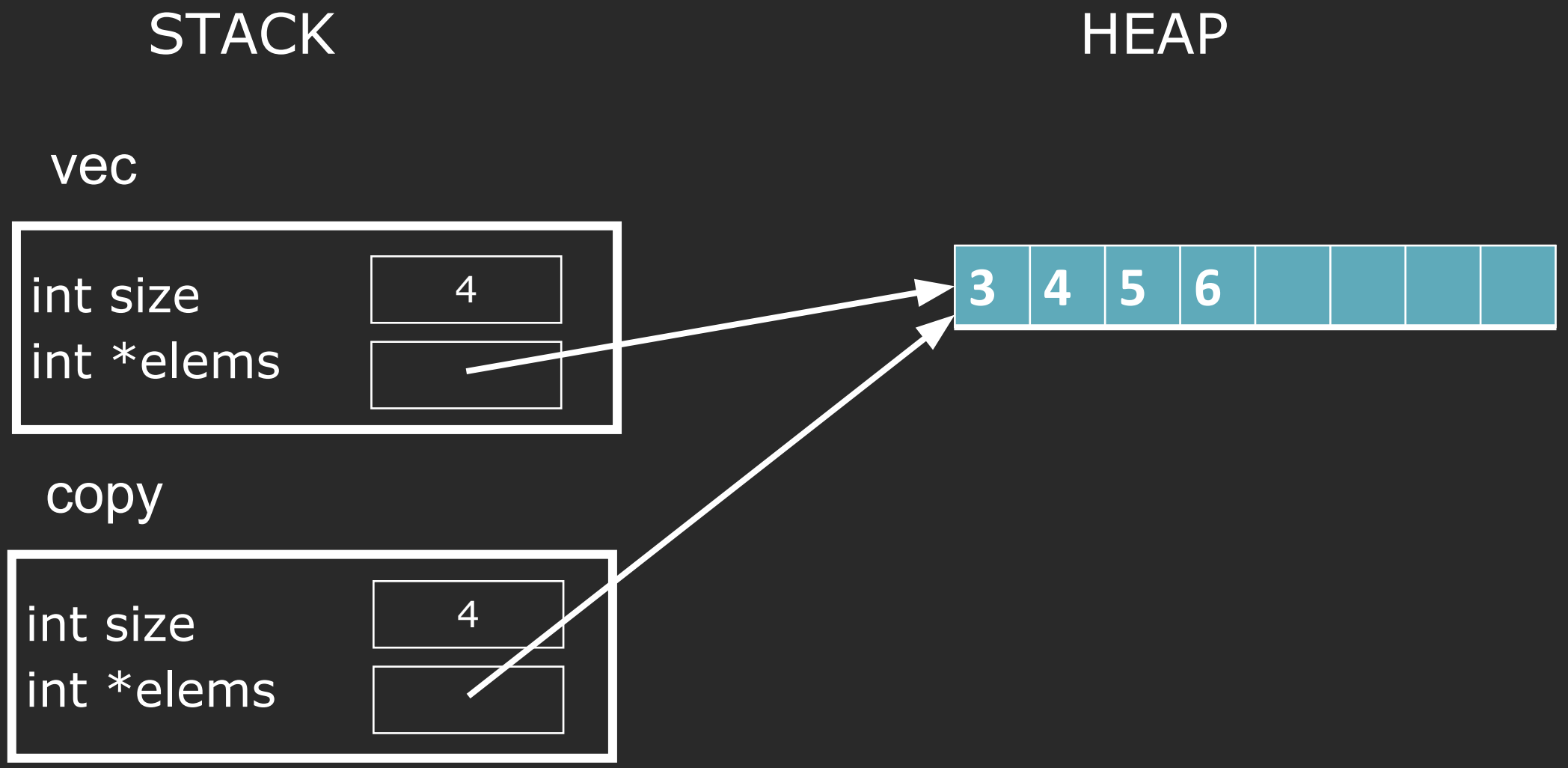
Here we need to create a deep copy of the vector.

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

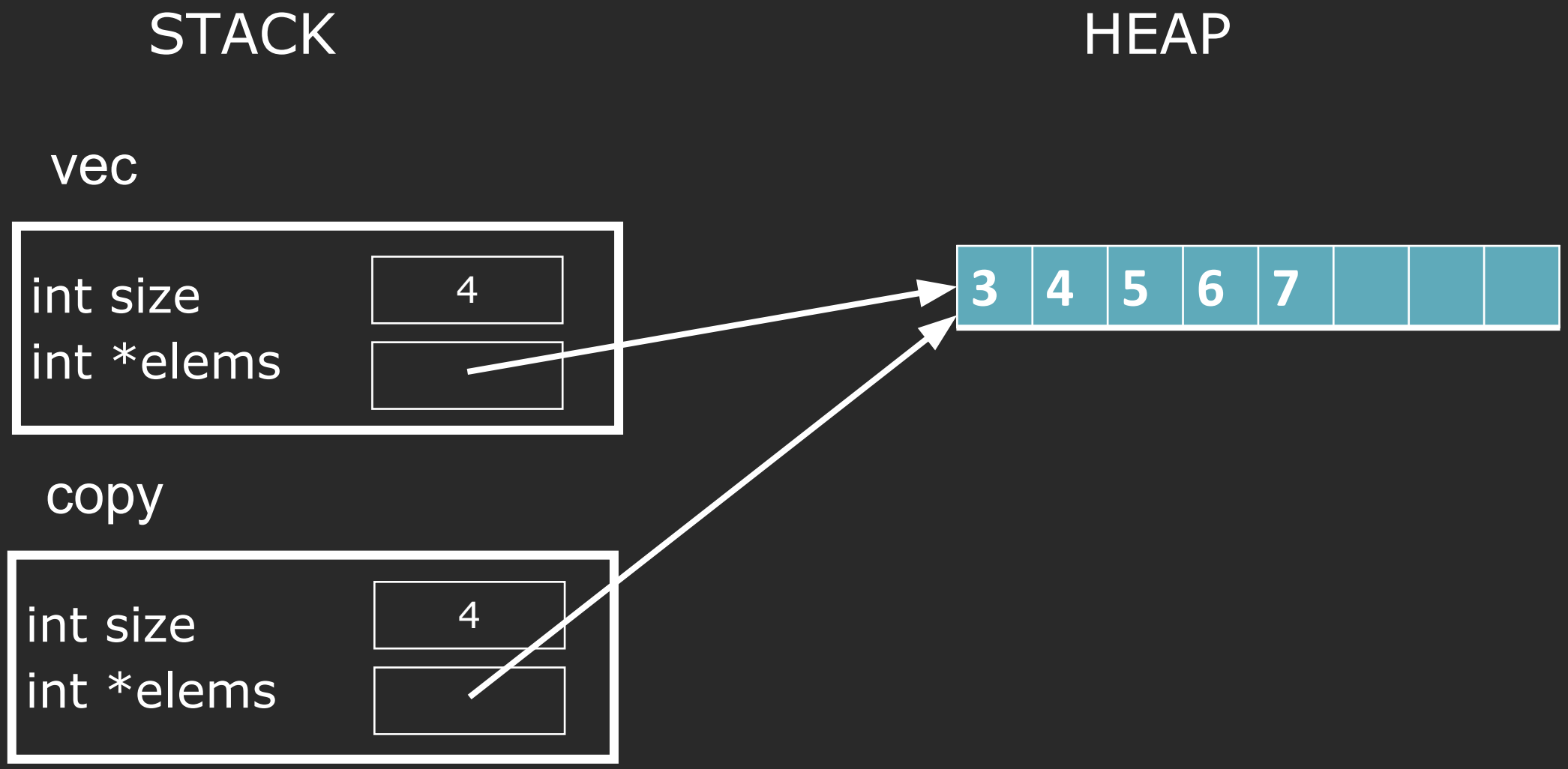
Copy isn’t as simple as copying each private member.



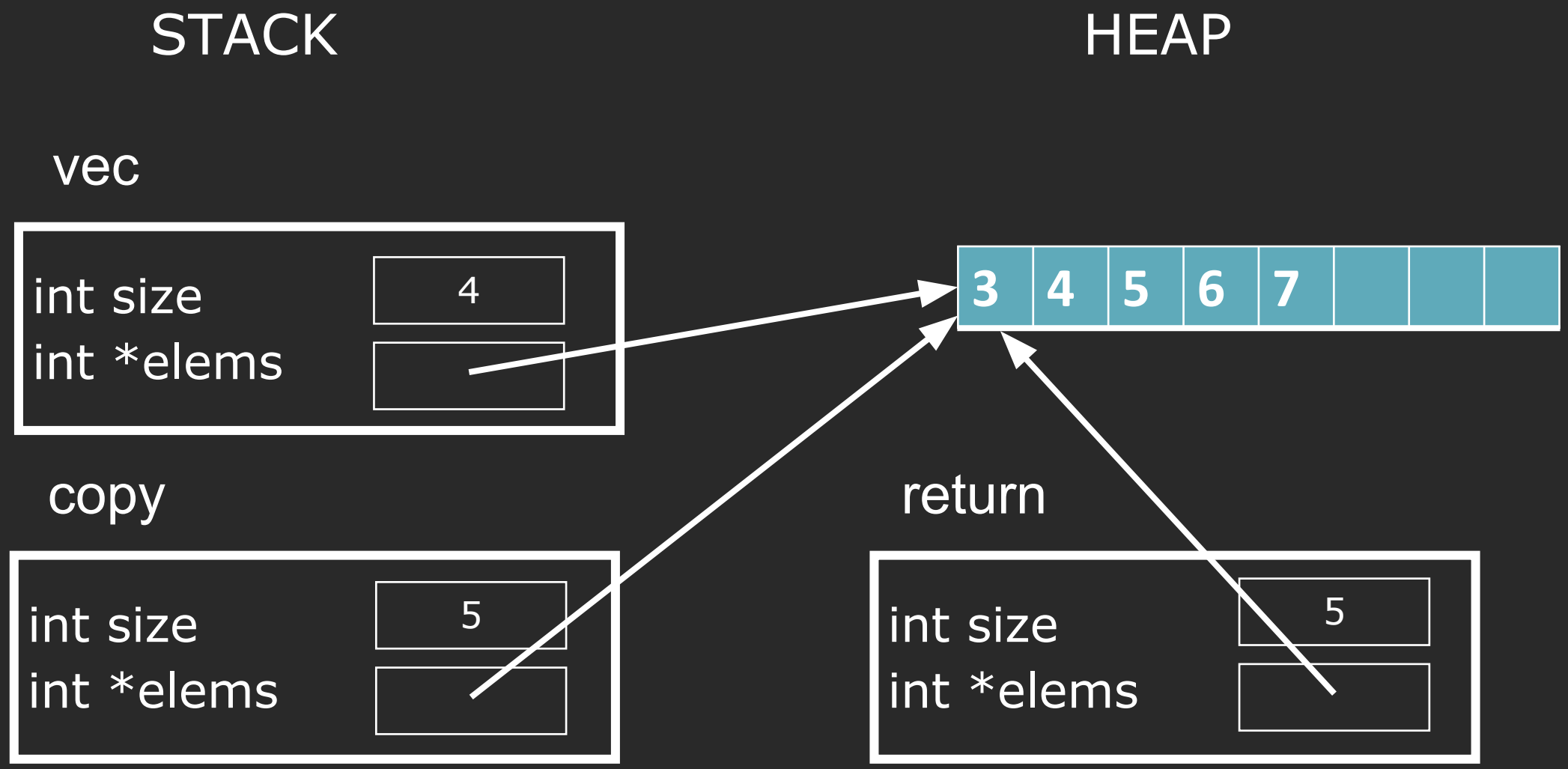
Copy isn’t as simple as copying each private member.



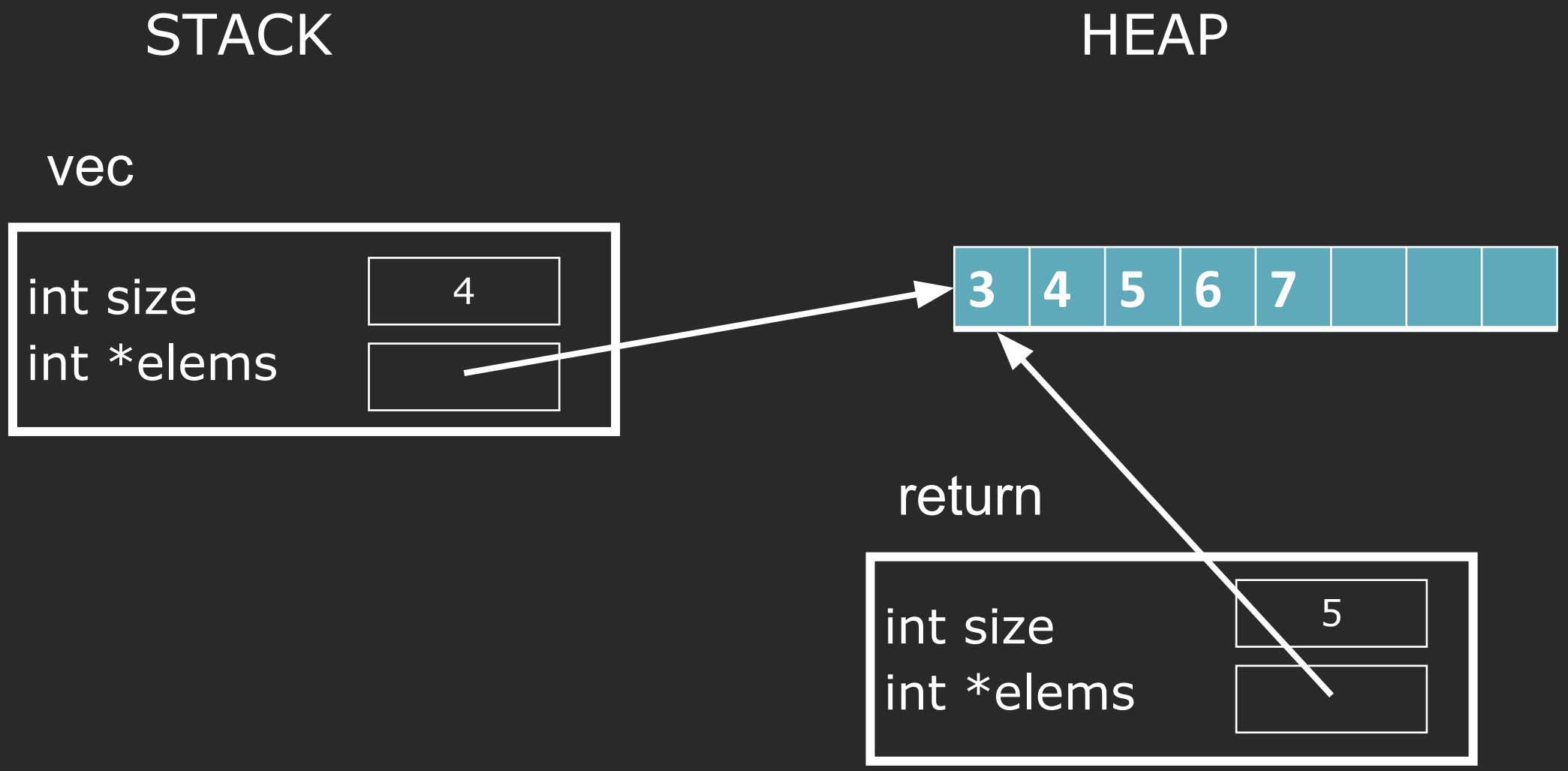
Now we try to add an element



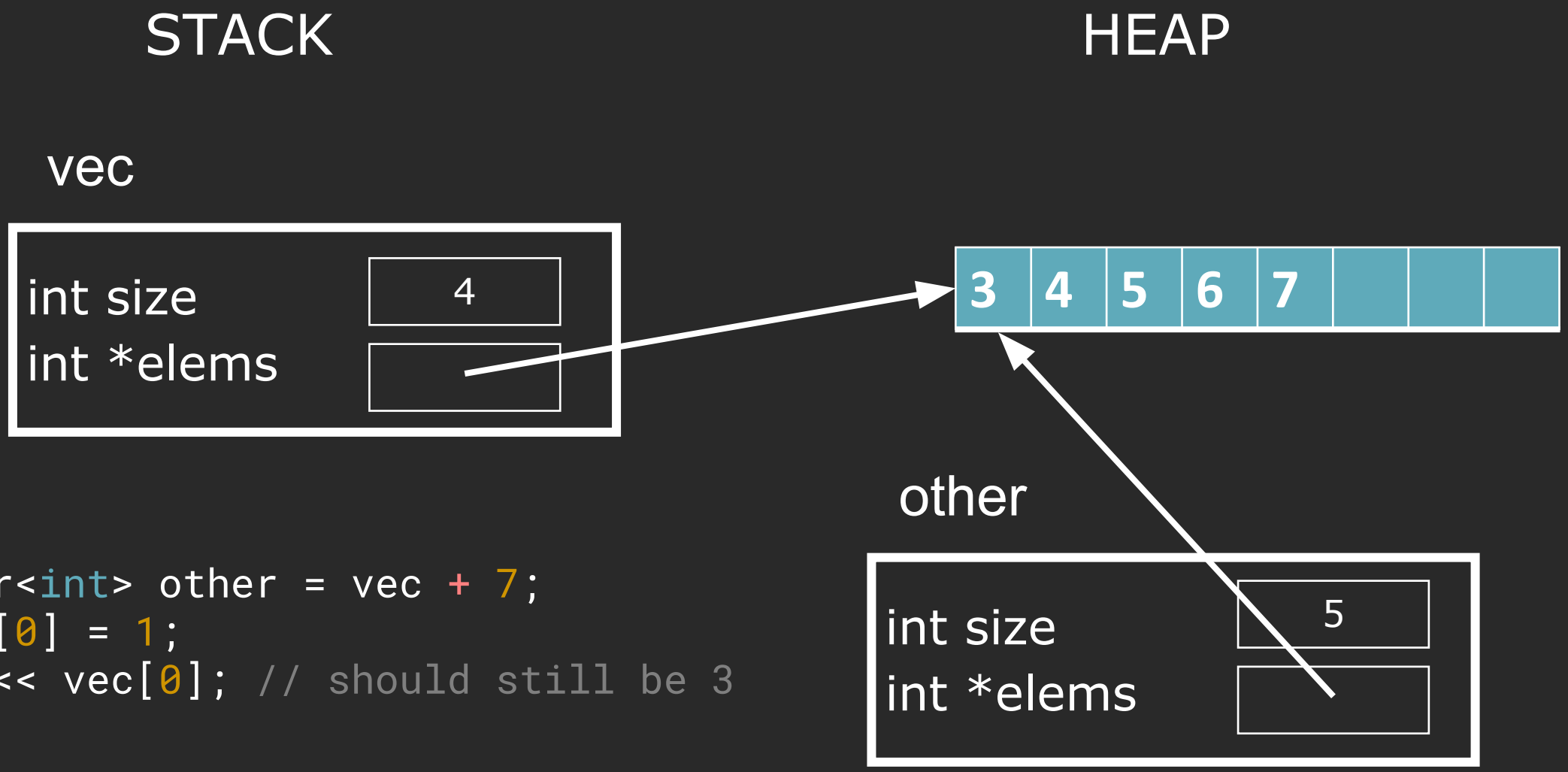
Returning creates another copy.



Local variable copy is gone.

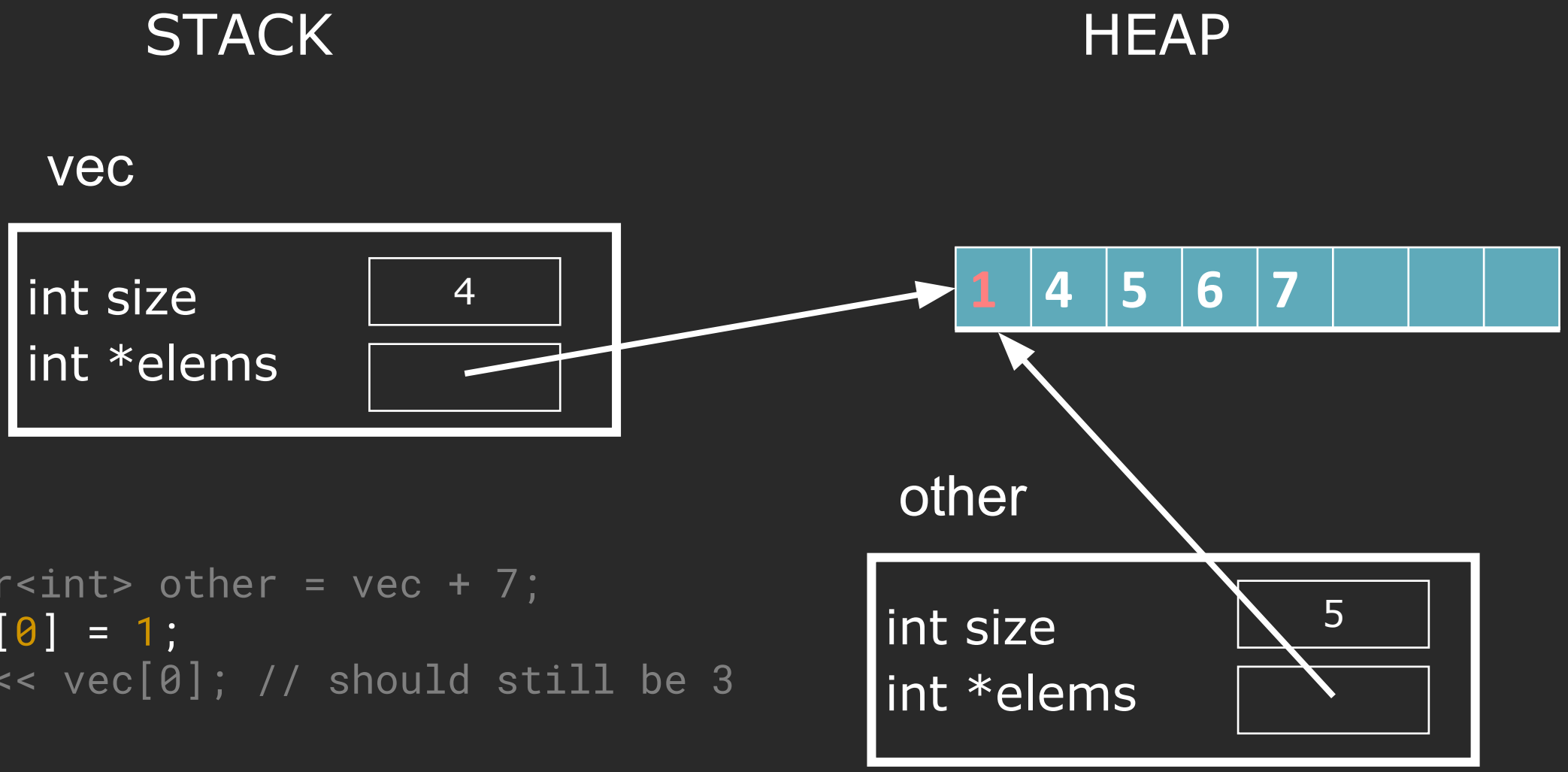


If we continue the code…



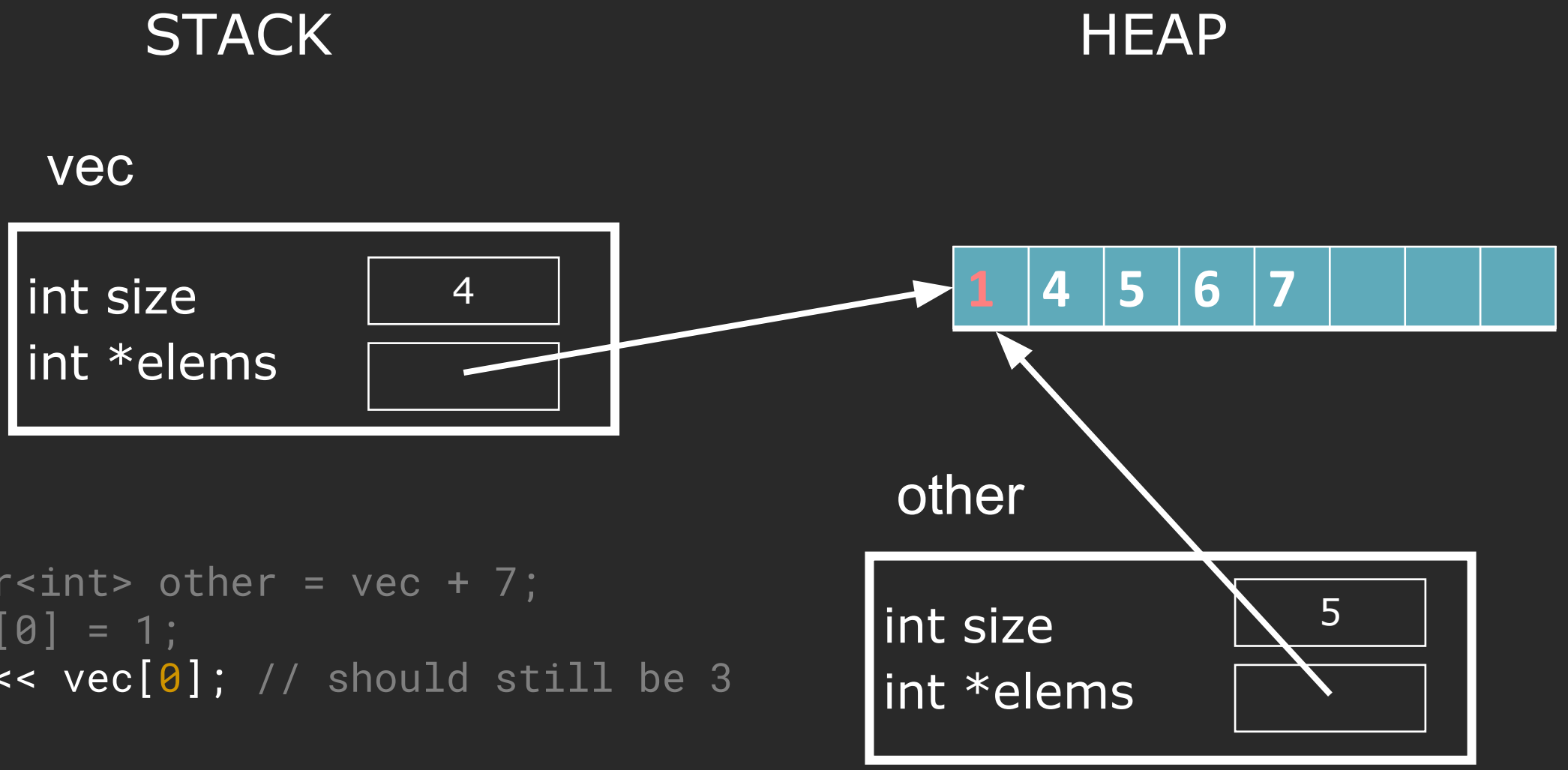

```
vector<int> other = vec + 7;
other[0] = 1;
cout << vec[0]; // should still be 3
```

If we continue the code…




```
vector<int> other = vec + 7;
other[0] = 1;
cout << vec[0]; // should still be 3
```

If we continue the code…




```
vector<int> other = vec + 7;
other[0] = 1;
cout << vec[0]; // should still be 3
```

The culprit of it all?

```
vector<int> vec{3, 4, 5, 6};
vector<int> other = vec + 7;
other[0] = 1;
cout << vec[0]; // should still be 3
```

The culprit of it all?

```
vector<int> vec{3, 4, 5, 6};
vector<int> other = vec + 7;
other[0] = 1;
cout << vec[0]; // should still be 3
```

Lingering Questions

Why does the assignment operator (=) not work as intended?

Lingering Questions

Why does the assignment operator (=) not work as intended?

We only copy pointers to dynamically allocated memory. We need to allocate separate memory for the copy.

Lingering Questions

Why are there so many copies?

Lingering Questions

Why are there so many copies?

After we fix this, every assignment will require a new copy, and this is super slow.

Special Member Functions

The member functions the compiler will *sometimes* generate for you that may or may not be correct.

Lecture 13: All about copying

- Default constructor
- Copy constructor
- Copy assignment
- Destructor

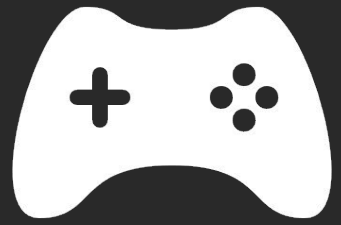
Lecture 14: Move semantics

- Move constructor
- Move assignment

Deep C++ Questions

Why are some things just not copyable?

How do you get around that? (well, you move it!)



Next time

Special Member Functions