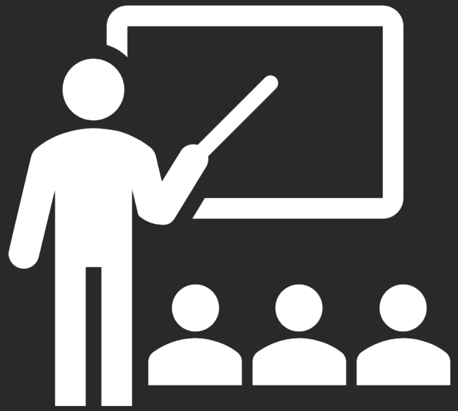
Operators

Game Plan



- operator overloading
- canonical forms
- POLA

operator overloading

Name as many operators as you can!

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

<table>
<tr><td>Arithmetic</td><td>+</td><td>-</td><td>*</td><td>/</td><td>%</td><td></td><td></td></tr>
<tr><td></td><td>+=</td><td>-=</td><td>*=</td><td>/=</td><td>%=</td><td></td><td></td></tr>
<tr><td>Bitwise</td><td></td><td>&</td><td>|</td><td>~</td><td>!</td><td></td><td></td></tr>
<tr><td>Relational</td><td>==</td><td>!=</td><td><</td><td>></td><td><=</td><td>>=</td><td><=></td></tr>
<tr><td>Stream</td><td><<</td><td>>></td><td><<=</td><td>>>=</td><td></td><td></td><td></td></tr>
<tr><td>Logical</td><td>&&</td><td>||</td><td>^</td><td>&=</td><td>|=</td><td>^=</td><td></td></tr>
<tr><td>Increment</td><td>++</td><td>--</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Memory</td><td>-></td><td>->*</td><td>new</td><td>new []</td><td>delete</td><td>delete []</td><td></td></tr>
<tr><td>Misc</td><td>( )</td><td>[ ]</td><td>,</td><td>=</td><td></td><td colspan="2">co_await</td></tr>
</table>

complicated

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

<table>
<tr><td>Arithmetic</td><td>+</td><td>-</td><td>*</td><td>/</td><td>%</td><td></td><td></td></tr>
<tr><td></td><td>+=</td><td>-=</td><td>*=</td><td>/=</td><td>%=</td><td></td><td></td></tr>
<tr><td>Bitwise</td><td></td><td>&</td><td>|</td><td>~</td><td>!</td><td></td><td></td></tr>
<tr><td>Relational</td><td>==</td><td>!=</td><td><</td><td>></td><td><=</td><td>>=</td><td><=></td></tr>
<tr><td>Stream</td><td><<</td><td>>></td><td><<=</td><td>>>=</td><td></td><td></td><td></td></tr>
<tr><td>Logical</td><td>&&</td><td>||</td><td>^</td><td>&=</td><td>|=</td><td>^=</td><td></td></tr>
<tr><td>Increment</td><td>++</td><td>--</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Memory</td><td>-></td><td>->*</td><td>new</td><td>new []</td><td>delete</td><td>delete []</td><td></td></tr>
<tr><td>Misc</td><td>( )</td><td>[ ]</td><td>,</td><td>=</td><td></td><td colspan="2">co_await</td></tr>
</table>

C++ knows how operators work for primitive types.

```cpp
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.

note: there are 2 ways of calling operator in total;
the first one: declare operator as member function: cout.operator(...)
the second one: declare operator as non-member function: operator<<(lhs, rhs)...

```
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?

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

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

return i ieself, a reference to the object ieself

Why are these the function signatures?

corectness: first, to store space, we pass a str as a reference; second, for a const var to be passed as reference, only const referece is allowed; non-const reference leads to compile error

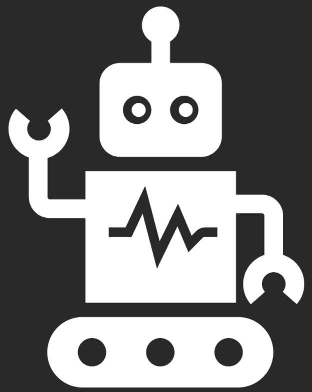
```
vector<string>& vector<string>::operator+=(const string& element) {
      push_back(element);
      return *this;
}

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

this: pointer pointing to the object

cannot return a value(copy): copy disappeared after the execution of the function; weird stuff: (a+=2) += 2

a += 2 returns a reference to variable a with new value return a reference or a value? member or non-member



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?

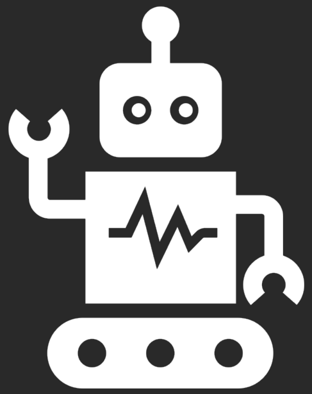
since the +=operator is a member function, the operator is called by an object, the left handside is the object to which this pointer points. v.operator+=(“element”)

unlike normal functions, member functions are called only by member; otherwise it won’t be called

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 AND fraction, +

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 doesn’t change the objects themselves. The compound ones do change the object.

General rule of thumb: member vs. non-member

stream is always on the left of <<

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.

2. cout << vec; lhs: cout(stream); rhs:vec(vector<string>); writing class for rhs(vector<string>); << operator belongs to STD, not class vector<string>: cout.operator << (vec); the operator<< belongs to cout streams.
binary operator: write out the expression involving the operator first. consider lhs and rhs, the operator belongs to the lhs; l and r equality;

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: +=

return value, do we need a reference or a value? when someone calls this function, does he need a value or reference
<< operator: chain together. change the original value or give a new value? e.g cout << ... change the original cout stream

Questions

Example

Operator overloading: vector, []

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];
}
```

变量型vecotr用前者；常量型vector用后者

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; // does not work, v2[1] is const
```

if there is no const version, v2[1] = 0 works, but this is a const vector, nothing can be changed. Unexpected behavior occurs.

What does it mean to << a Fraction into an ostream??

```
Fraction start{3, 4};
Fraction end{9, 14};
cout << start << “ “ << end;
```

Let’s try overloading the stream insertion operator!

```
struct Fraction {
    int num, denom;
}

ostream& operator<<(ostream& out, const Fraction& f) {
    out << f.num << “/”<< f.denom;
    return os;
}
```

Concept check

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

Key Takeaways

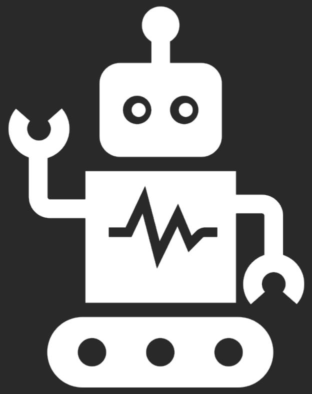
1. Always think about const-ness of parameters. Here, we are modifying the stream, not the Fraction struct.
2. Return reference to support chaining << calls.
3. Here we are overloading << so our class works as the rhs…but we can’t change the class of lhs (stream library).

Questions

Now that we move into designing classes, this might become a problem!

```
class Fraction {
public:
    Fraction(int num, int denom);
    ~Fraction();
    // other methods

private:
    int num, denom; // invariant, fraction is reduced
}
```



Example

Operator overloading: fraction, <<

We can’t access the private members of Fraction!

```
ostream& operator<<(ostream& out, const Fraction& f) {
     out << f.num << “/”<< f.denom;
     return os;
}
```

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

在interface里面说一句就可以了

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

private:
      int hour, minute;
      friend ostream& operator<<(ostream& out, const Fraction& 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, but need access to the private members.

non–member function but needs private access; delare friend inside

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)

C. 160

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)

C. 161

Use nonmember functions for symmetric operators.

Principle of Least Astonishment (POLA)

```
class Fraction {
public:
    Fraction(int num, int denom);
    ~Fraction();
    // other methods
    Fraction& operator+(const Fraction& rhs);
    Fraction& operator+(int rhs);
private:
    int num, denom;
}
```

Principle of Least Astonishment (POLA)

```
Fraction a {3, 8};
Fraction b {11, 8};

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

? try (int lhs, fraction rhs) then fraction + int; both?

Principle of Least Astonishment (POLA)

```
class Fraction {
public:
    Fraction(int num, int denom);
    ~Fraction();
    // other methods
private:
    int num, denom;
    friend Fraction& operator+(const Fraction& rhs,
                                                   int rhs);
}
```

Principle of Least Astonishment (POLA)

Always provide all out of a set of related operators.

配套

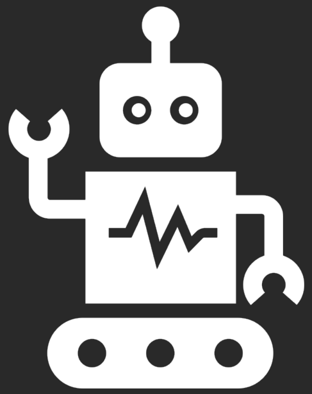
There are 40 (+4) operators you can overload!

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

Principle of Least Astonishment (POLA)

```
Fraction a {3, 8};
Fraction b {9, 16};

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



Example

Operator overloading: fraction, relational operators

Questions

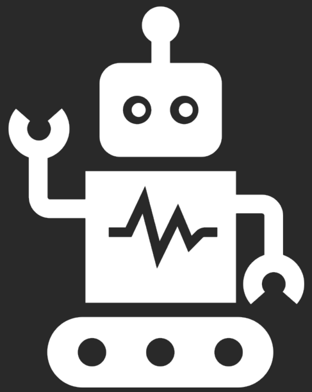
conversion operators (overflow)

Recall: a stream can be converted to a boolean.

```
istringstream ss(“6.9 Ounces”);
int amount;
if (!(ss >> amount)) {
    throw ”Invalid string”;
}
```

Conversion operators allow implicit conversions to another type.

```
Fraction frac{9, 16};
double value = frac;
if (frac) {
    cout << frac << endl;
}
```



Example

Operator overloading: fraction, conversion operator

Conversion operators allow implicit conversions to another type.

```
class Fraction {
public:
        Fraction(int num, int denom);
        ~Fraction();
        operator double() const; // convert to double
        operator bool() const; // is a valid fraction
        // other methods
private:
        int num, denom;
        friend Fraction& operator+(const Fraction& rhs,
                                                   double rhs);
}
```

Implicit conversion are dangerous!

```
Fraction you{9, 16};
Fraction me{1, 2};
double value = you;
if (you) {
    cout << you << endl;
}
cout << you / me << endl;
// why does this compile? We haven’t defined a / operator yet…
```

Require the conversion operator to be explicit.

```
class Fraction {
public:
      Fraction(int num, int denom);
      ~Fraction();
      explicit operator double() const; // convert to double
      explicit operator bool() const; // is a valid fraction
      // other methods
private:
      int num, denom;
      friend Fraction& operator+(const Fraction& rhs,
                                double rhs);
}
```

Prevents unexpected implicit conversions!

```
Fraction you{9, 16};
Fraction me{1, 2};
double value = static_cast<double>(you);
if (you) {
    cout << you << endl;
}
cout << you / me << endl;
// better – can't make accidental mistakes!
```

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 17)

```
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(pi.begin(), pi.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

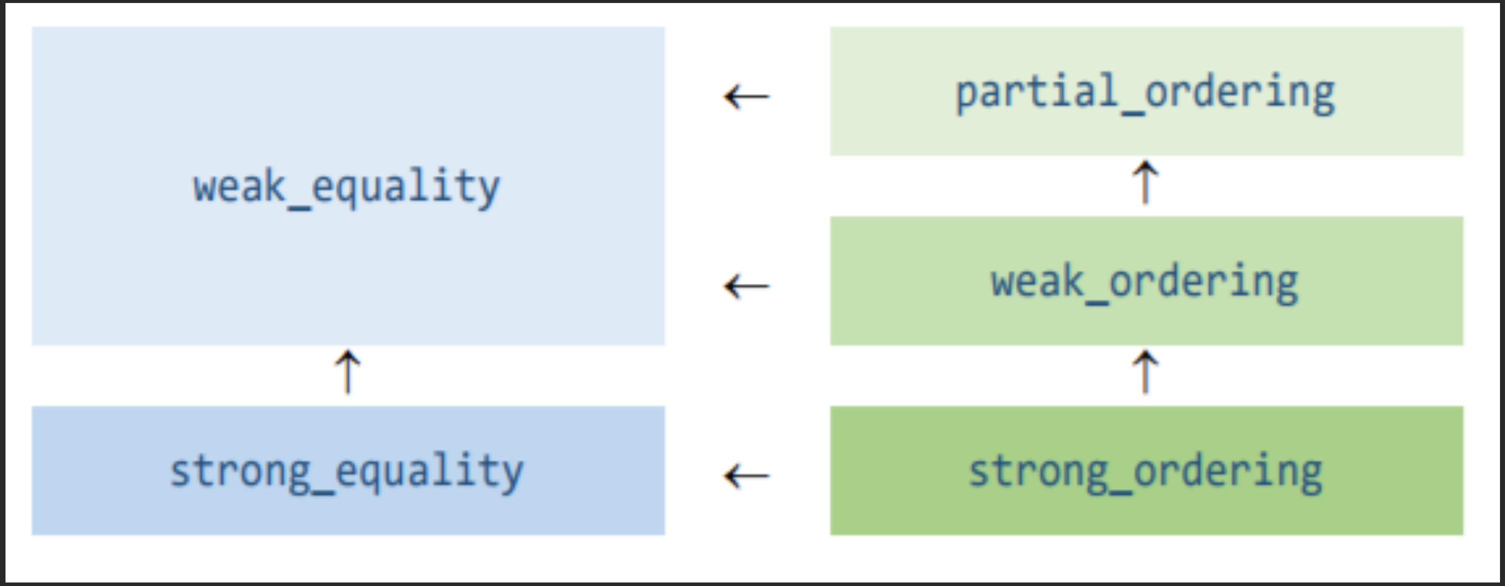
`[[nodiscard]]` (since C++20)

`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 };
}
```

Spaceship operator (C++20)




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

let’s back up one sec

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

```
vector<int> vec{1, 2, 3, 4};
other = vec + 5;
other[0] = 6;
cout << vec[0]; // this should 1
```

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

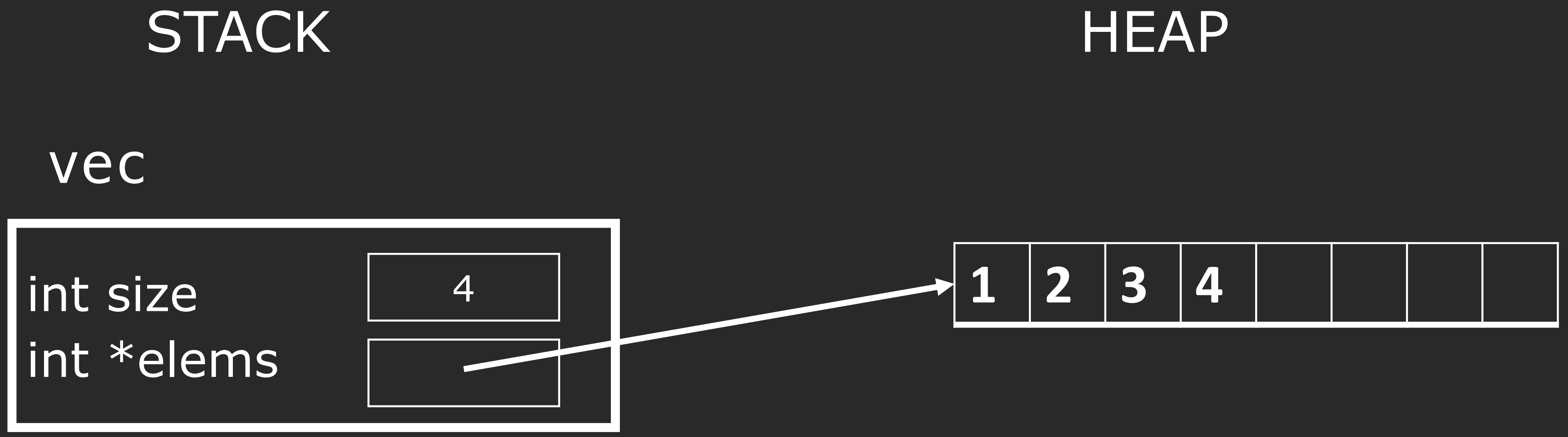
```
vector<int> operator+(const vector<int>& vec, int element) {
      vector<int> copy = vec;
      copy += element;
      return copy;
}

other = vec + 5;
other[0] = 6;
cout << vec[0]; // should be 1
```

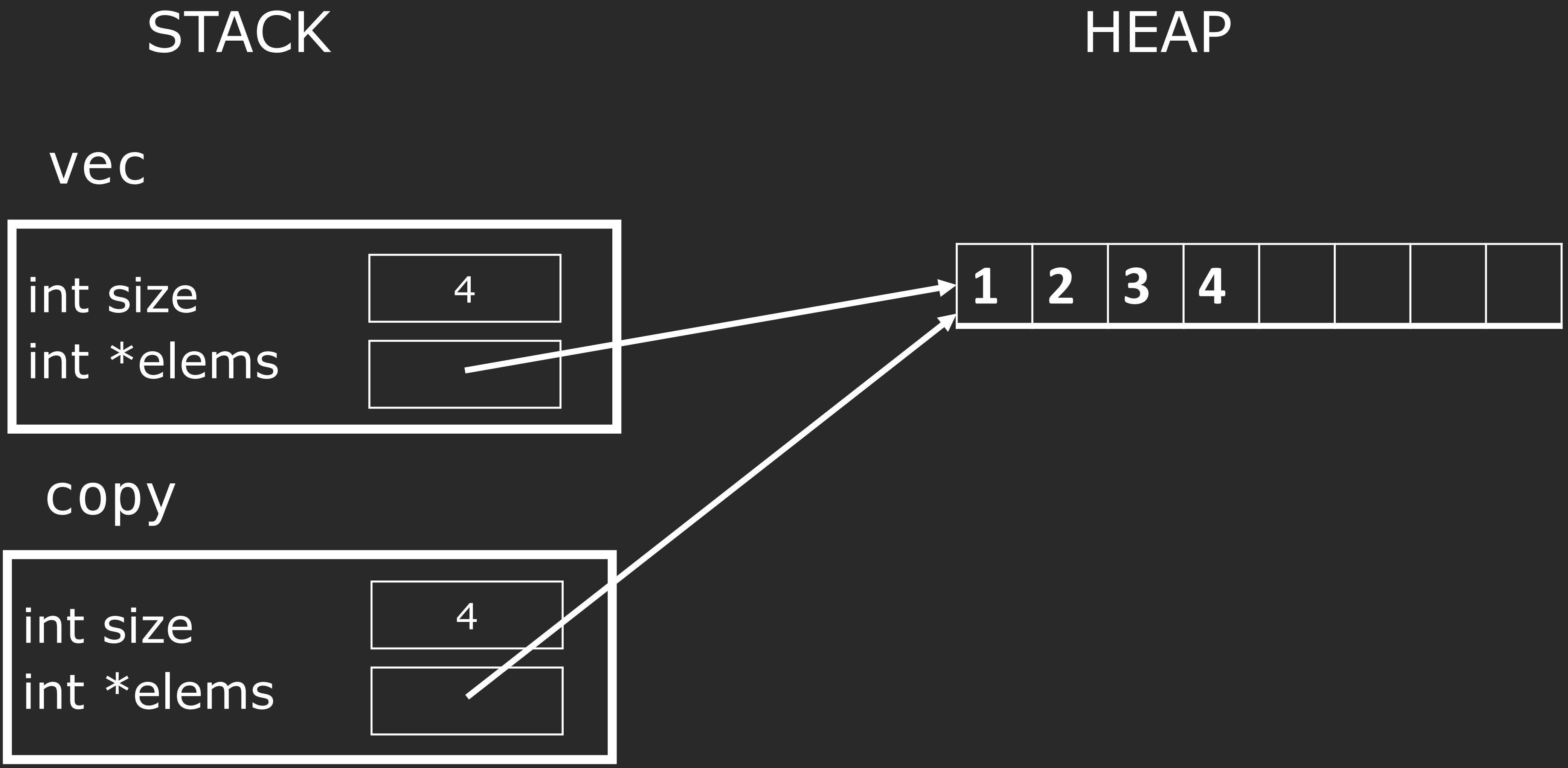
Here we need to create a deep copy of the vector.

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

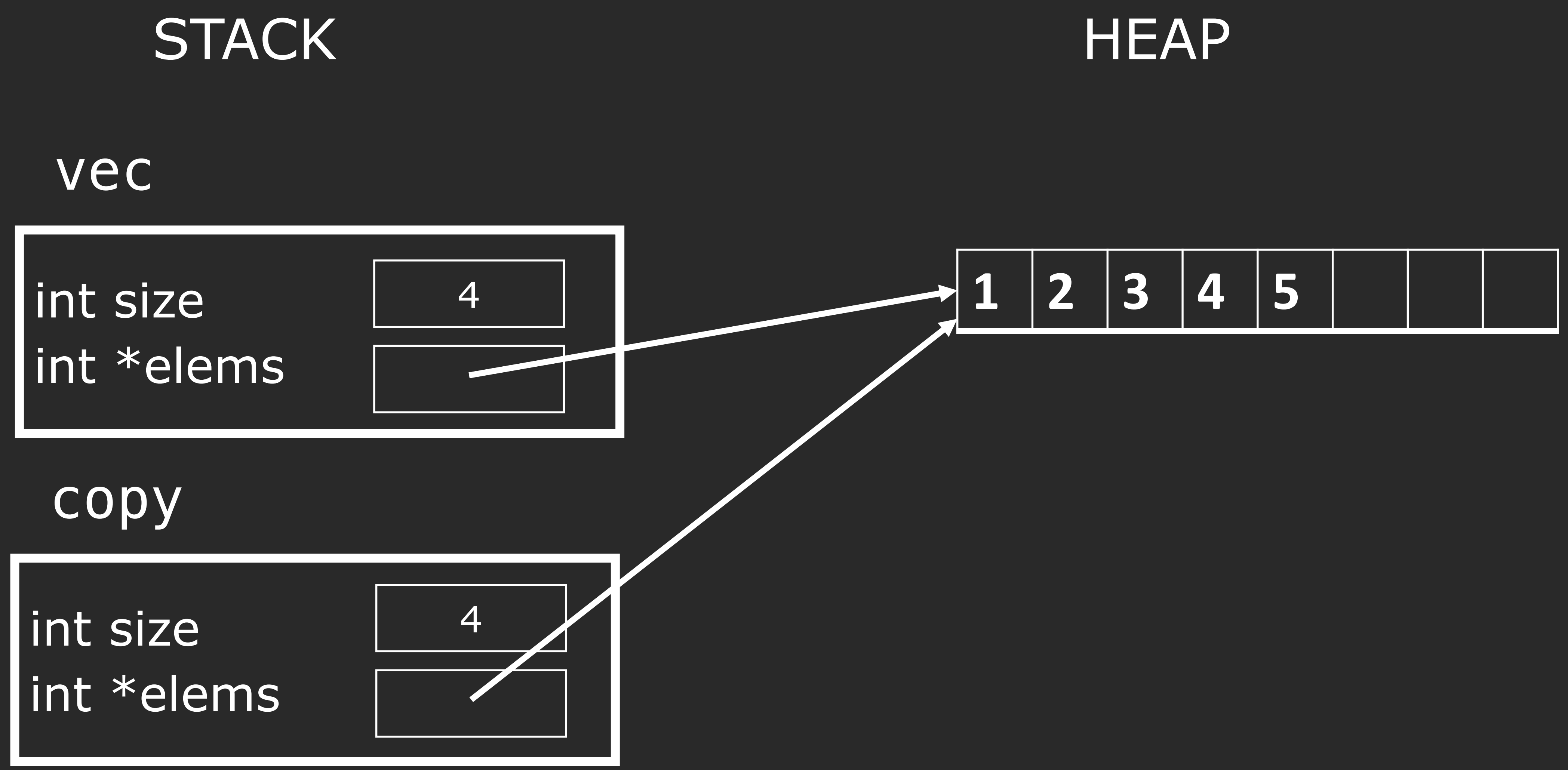
Copy is not as simple as copying each member.



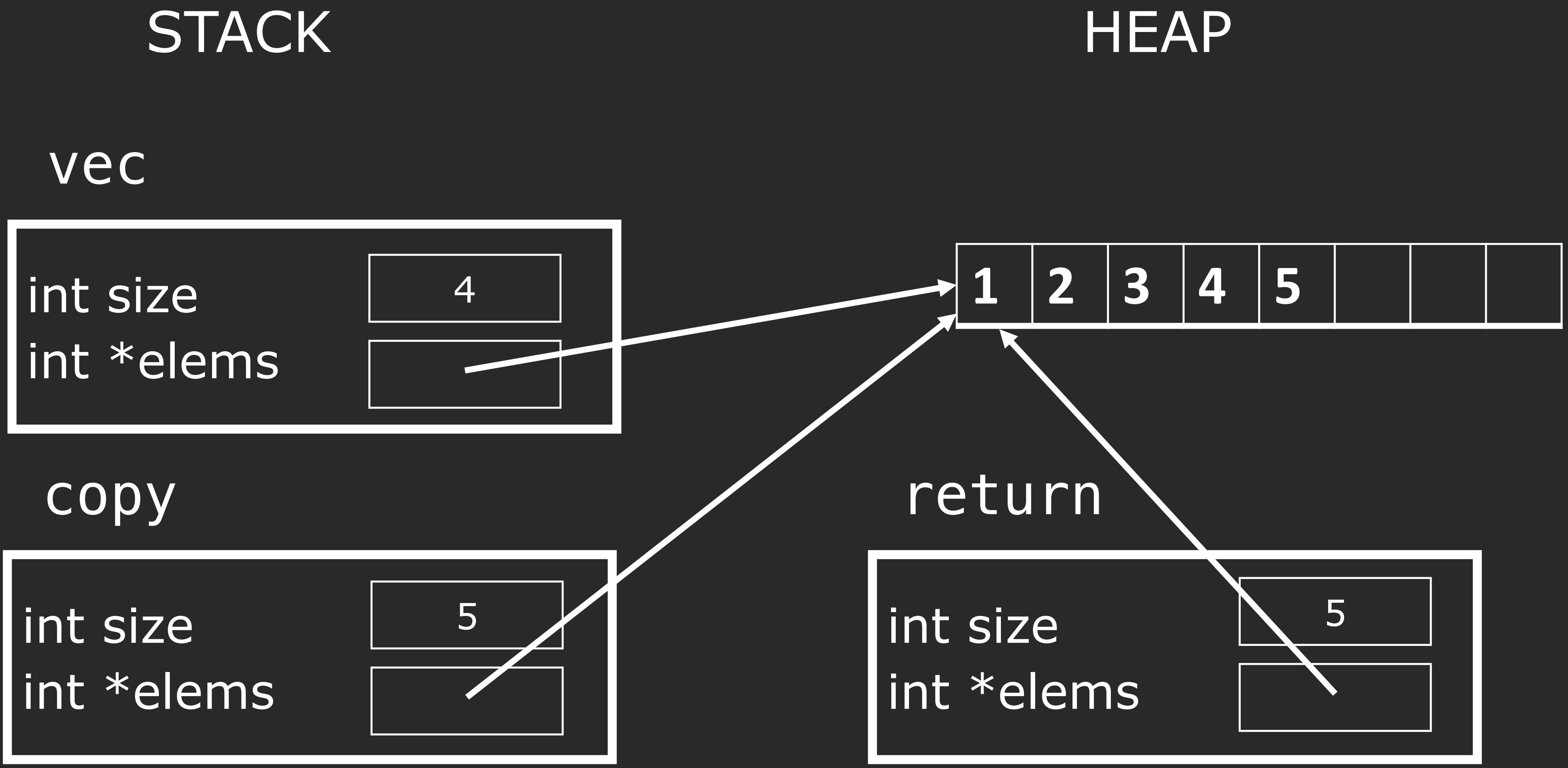
Copy is not as simple as copying each member.



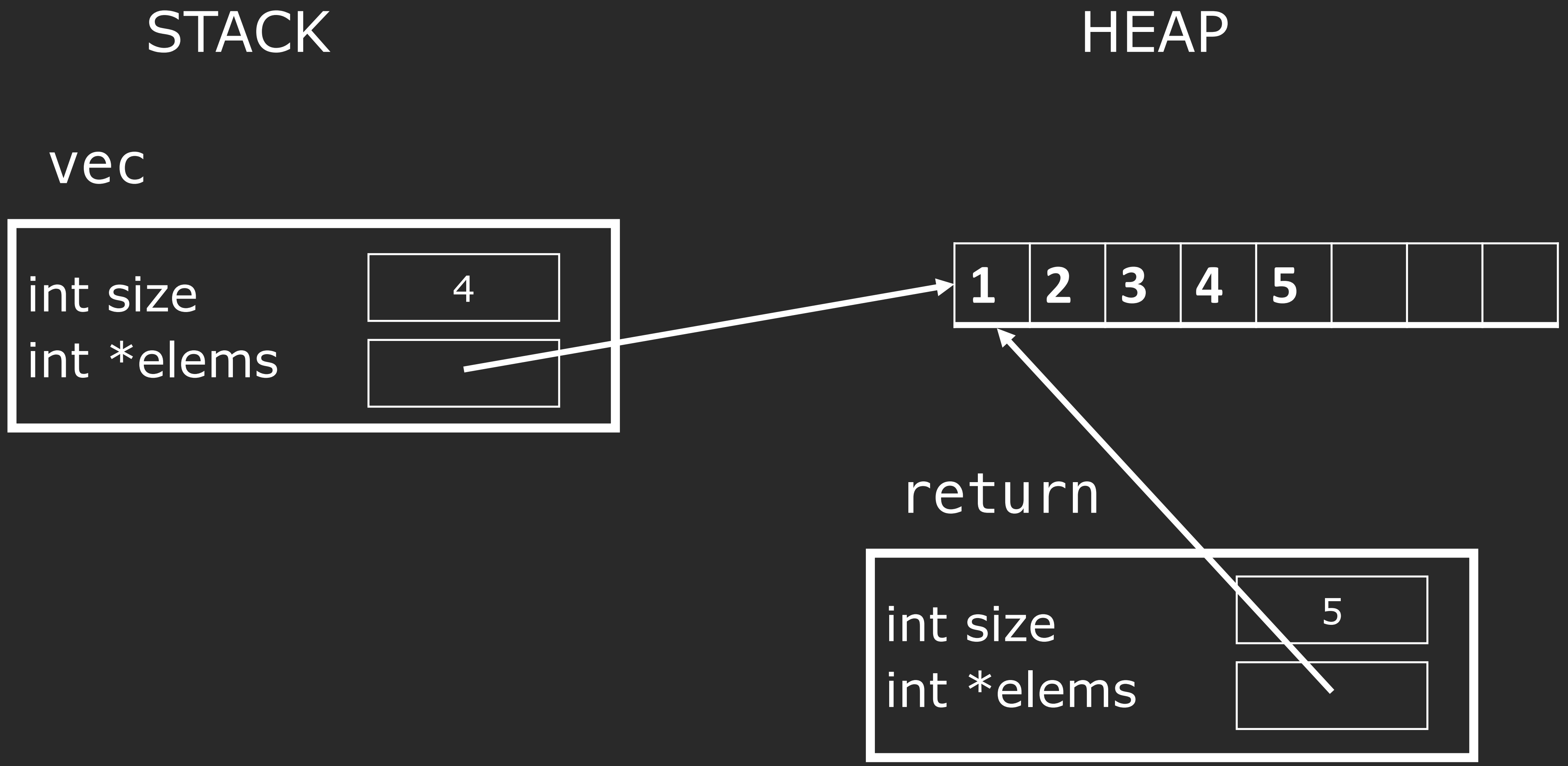
Now we try to add an element



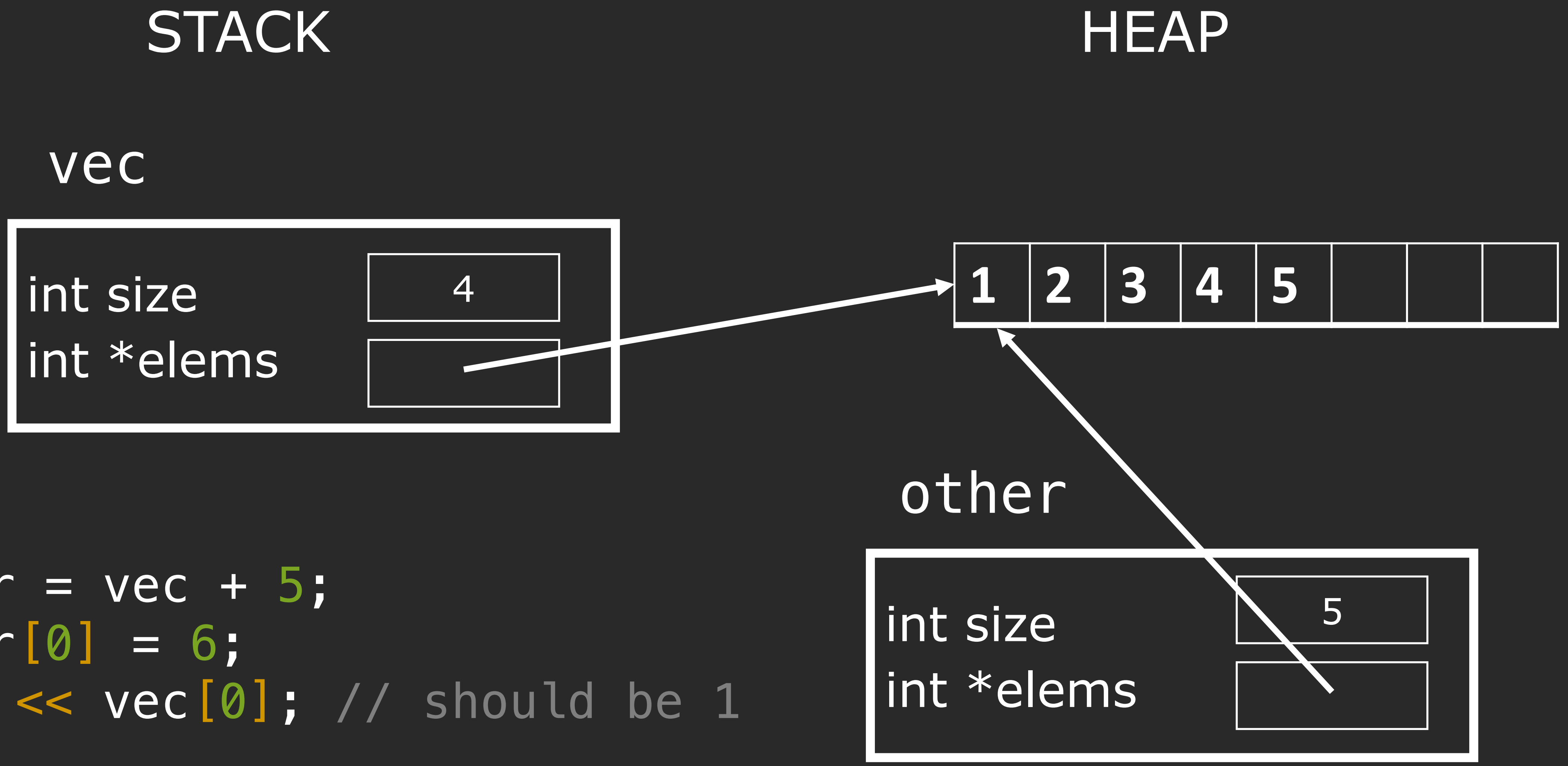
Returning creates another copy.



Local variable copy is gone.

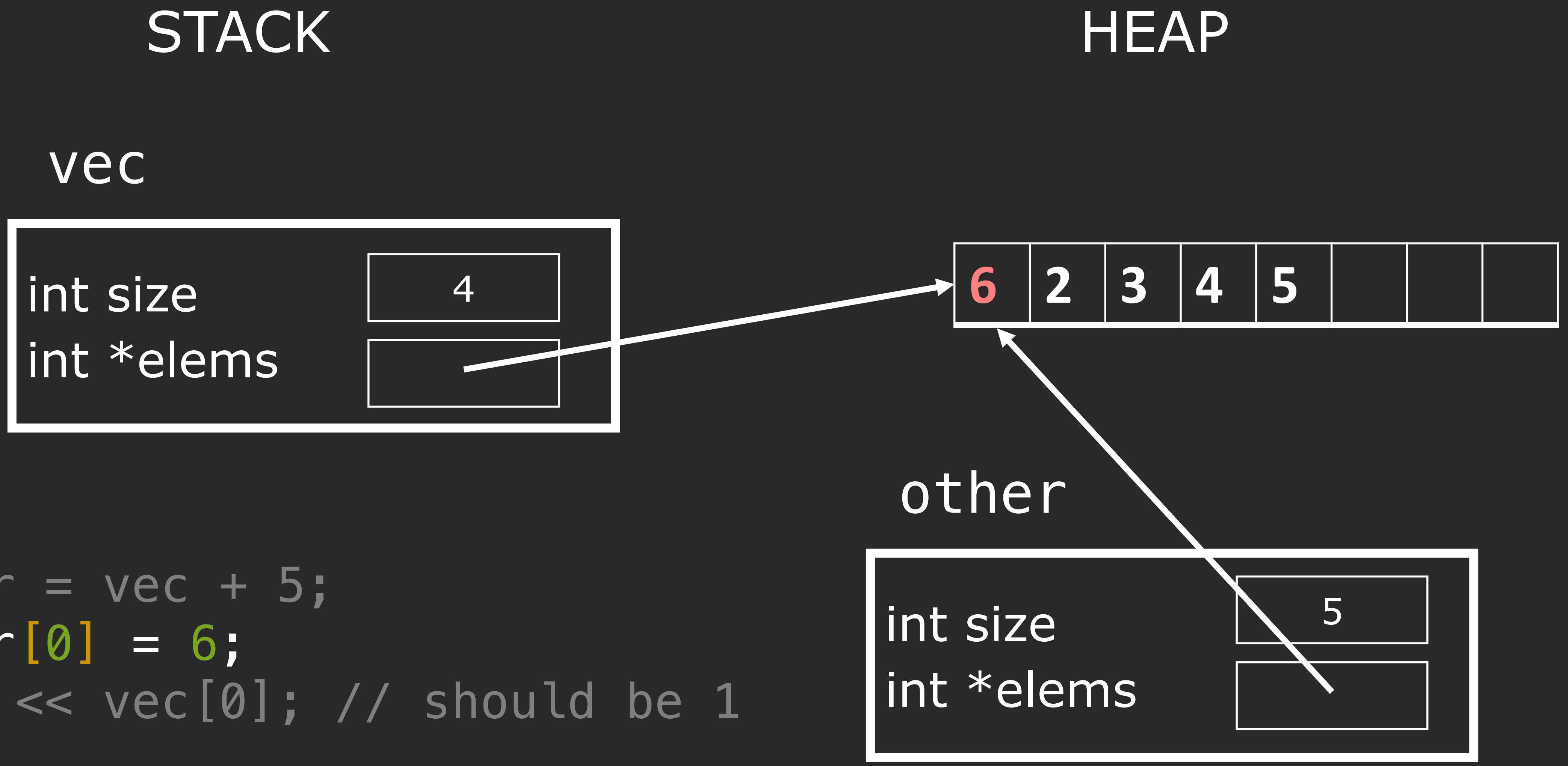


If we continue the code…



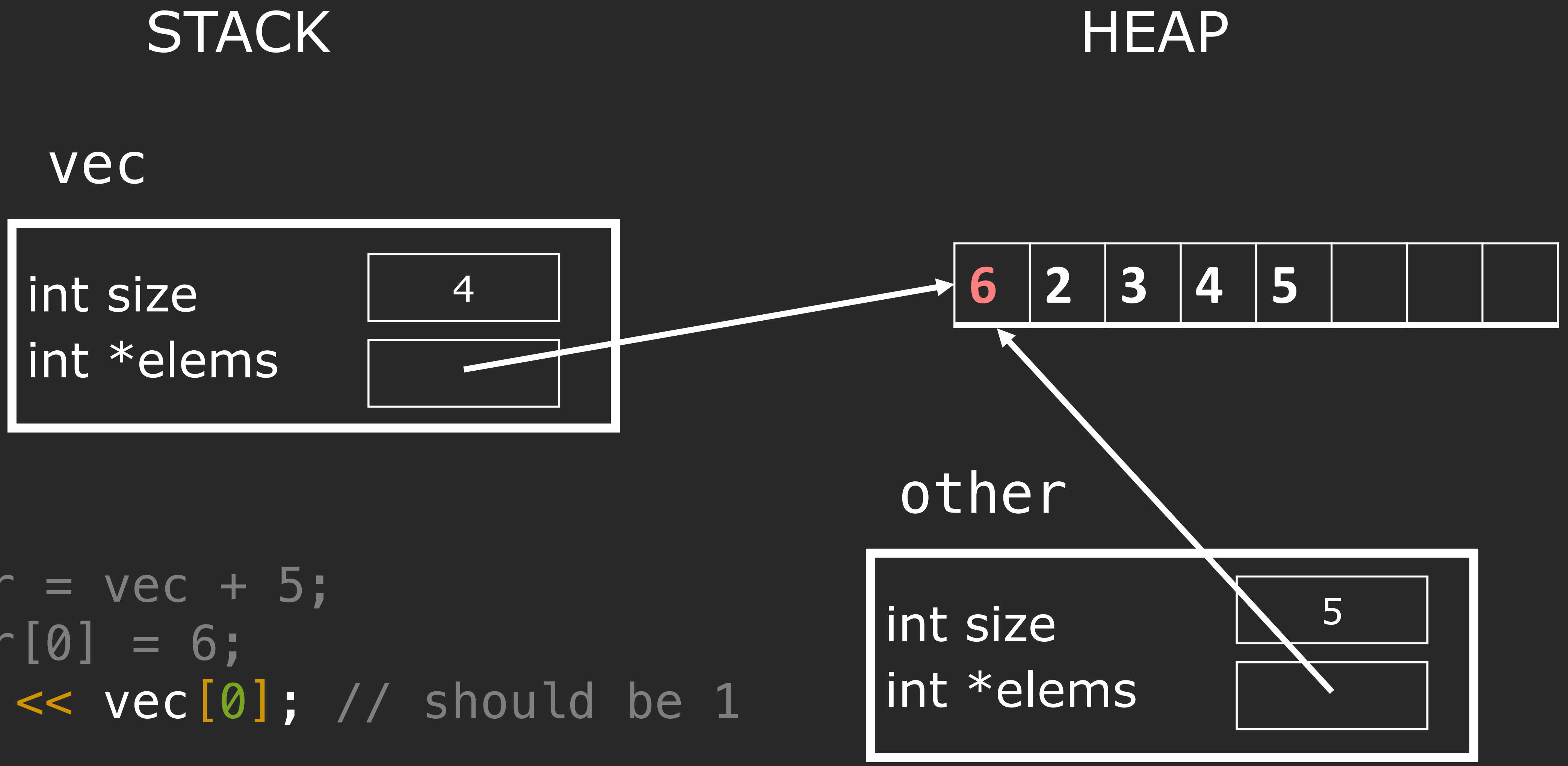

```
other = vec + 5;
other[0] = 6;
cout << vec[0]; // should be 1
```

If we continue the code…




```
other = vec + 5;
other[0] = 6;
cout << vec[0]; // should be 1
```

If we continue the code…




```
other = vec + 5;
other[0] = 6;
cout << vec[0]; // should be 1
```

The culprit of it all?

```
vector<int> vec{1, 2, 3, 4};
other = vec + 5;
other[0] = 6;
cout << vec[0]; // should be 1
```

The culprit of it all?

```
vector<int> vec{1, 2, 3, 4};
other = vec + 5;
other[0] = 6;
cout << vec[0]; // should be 1
```

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 12: All about copying

- Default constructor
- Copy constructor
- Copy assignment
- Destructor

Lecture 13: 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