Collections, Part One

Outline for Today

- ***Parameter Passing in C++***
  - On xeroxes and master copies.
- ***Container Types***
  - Holding lots of pieces of data.
- ***The `Vector` type***
  - Storing sequences.
- ***Recursion on `Vectors`***
  - More practice with sequences.

Parameter Passing in C++

Make a Prediction!

- Look over this piece of C++ code:

```
void becomeWealthy(int netWorth) {
    netWorth = 1000000000;
}
int main() {
    int value = 137;
    becomeWealthy(value);
    cout << value << endl; // <-- Here
    return 0;
}
```

- What do you think will get printed at the indicated point? Why?

Formulate a hypothesis!

How it Works

```
int main() {
    int value = 137;
    becomeWealthy(value);
    cout << value << endl;
    return 0;
}
```

How it Works

```
int main() {
    int value = 137;
    becomeWealthy(value);
    cout << value << endl;
    return 0;
}
```

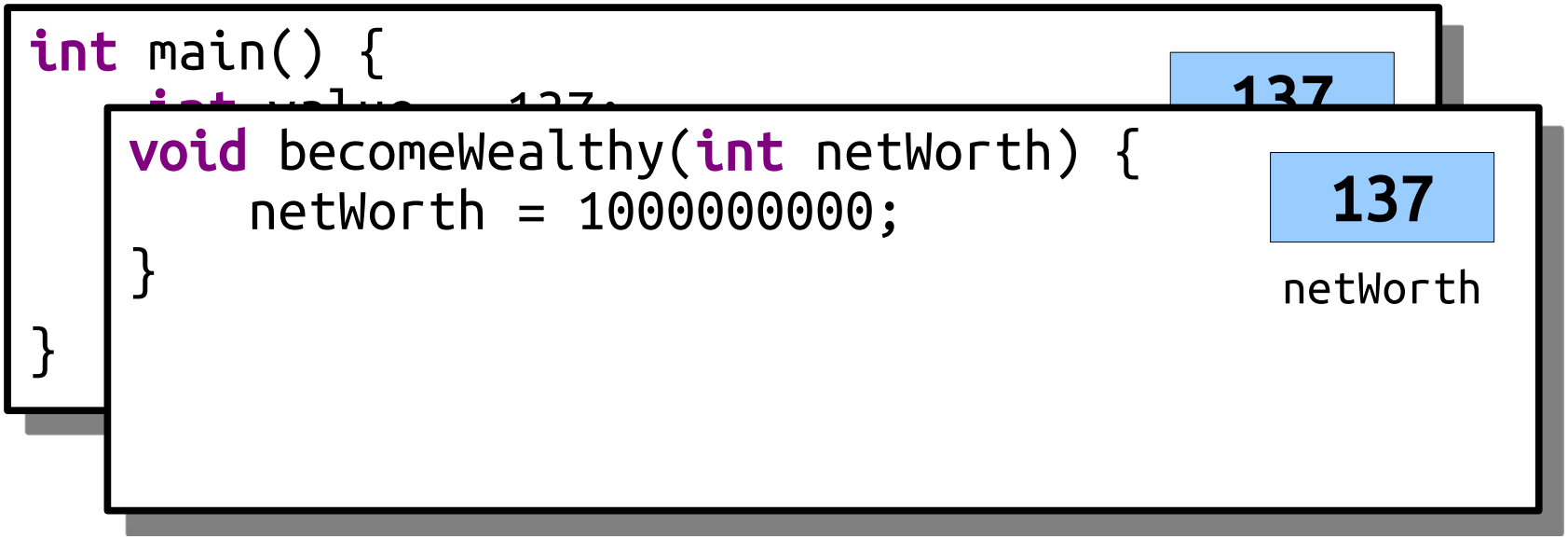
How it Works

```
int main() {
    int value = 137;
    becomeWealthy(value);
    cout << value << endl;
    return 0;
}
```

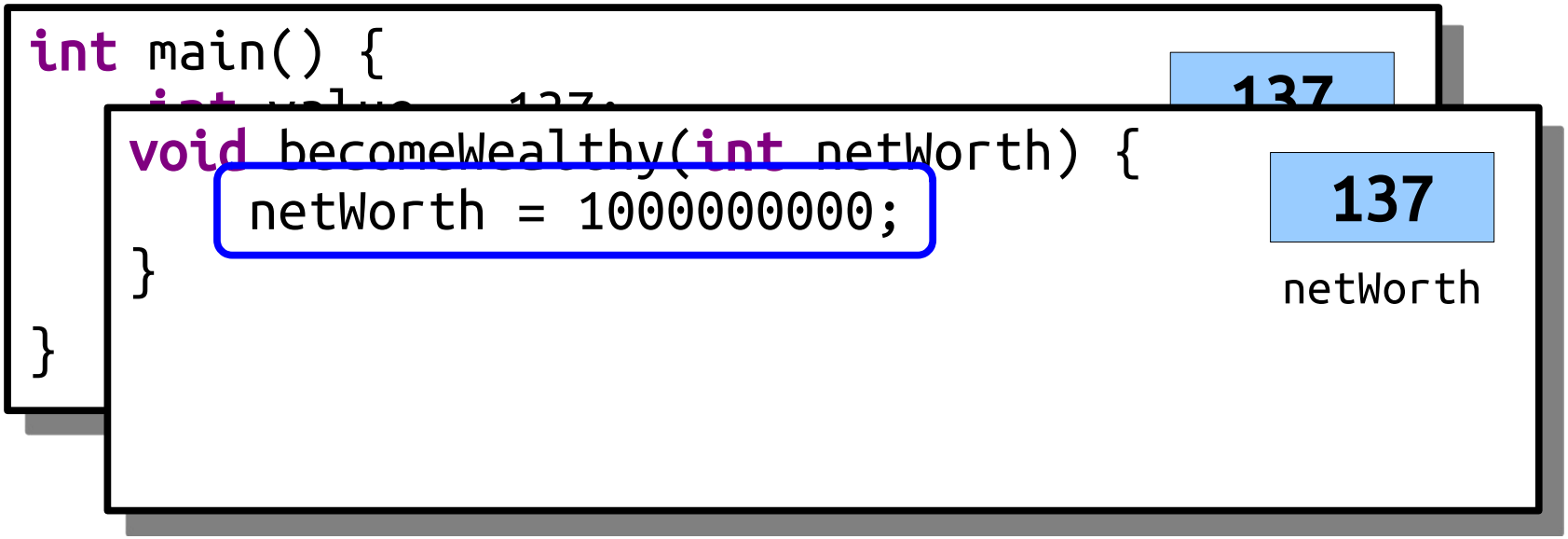
How it Works

```
int main() {
    int value = 137;
    becomeWealthy(value);
    cout << value << endl;
    return 0;
}
```

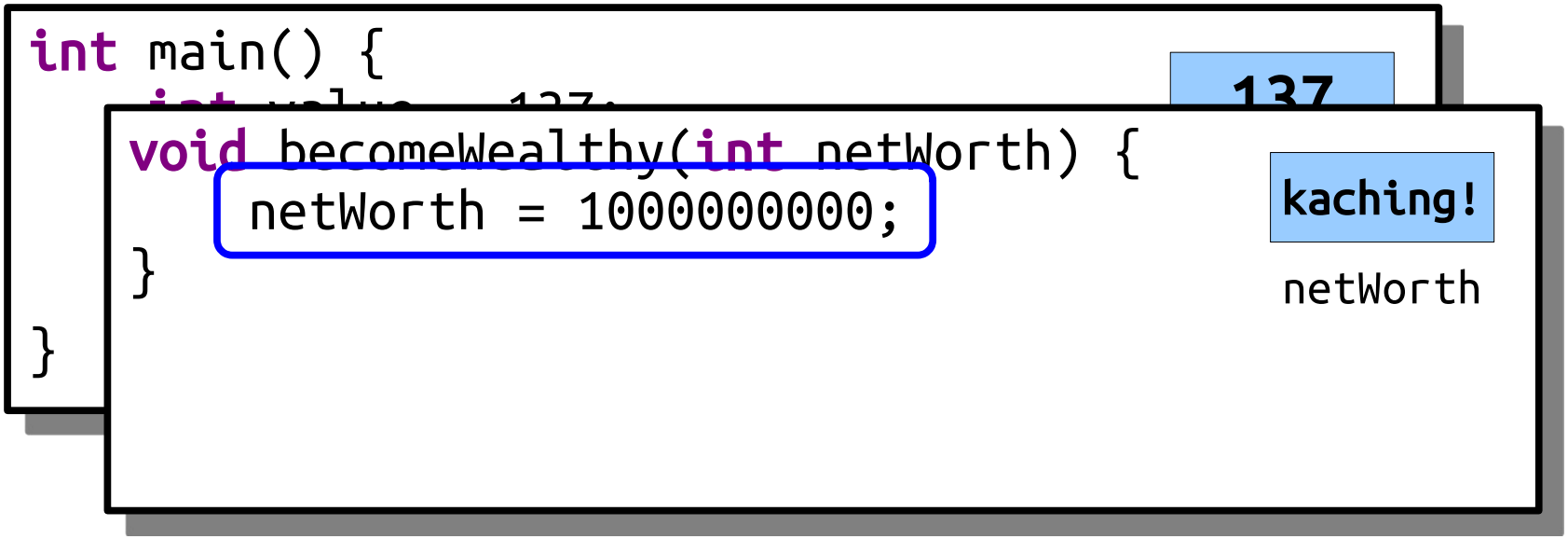
How it Works


int main() {
}
void becomeWealthy(int netWorth) {
netWorth = 1000000000;
}
137
137
netWorth

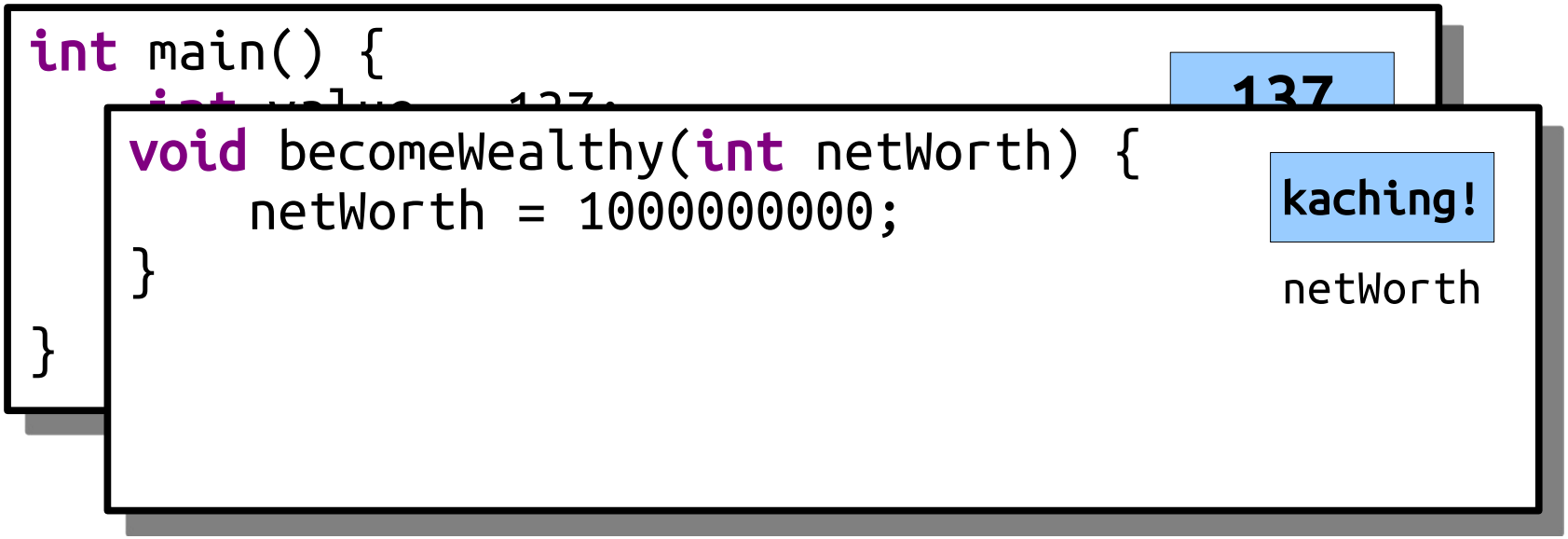
How it Works


int main() {
}
void becomeWealthy(int netWorth) {
netWorth = 1000000000;
}
137
137
netWorth

How it Works


int main() {
137
void becomeWealthy(int netWorth) {
netWorth = 1000000000;
}
}
kaching!
netWorth

How it Works


int main() {
}
137
void becomeWealthy(int netWorth) {
netWorth = 1000000000;
}
kaching!
netWorth

How it Works

```
int main() {
    int value = 137;
    becomeWealthy(value);
    cout << value << endl;
    return 0;
}
```

Parameter Passing in C++

- By default, in C++, parameters are passed by value.

```
/* This function gets a copy of the integer passed
 * into it, so we only change our local copy. The
 * caller won't see any changes.
 */
void byValue(int number) {
    number = 137;
}
```

- You can place an ampersand after the type name to take the parameter by reference.

```
/* This function takes its argument by reference, so
 * when the function returns, the int passed in will have
 * been permanently changed.
 */
void byReference(int& number) {
    number = 137;
}
```

How it Works Now

```
int main() {
    int value = 137;
    becomeWealthy(value);
    cout << value << endl;
    return 0;
}
```

How it Works Now

```
int main() {
    int value = 137;
    becomeWealthy(value);
    cout << value << endl;
    return 0;
}
```

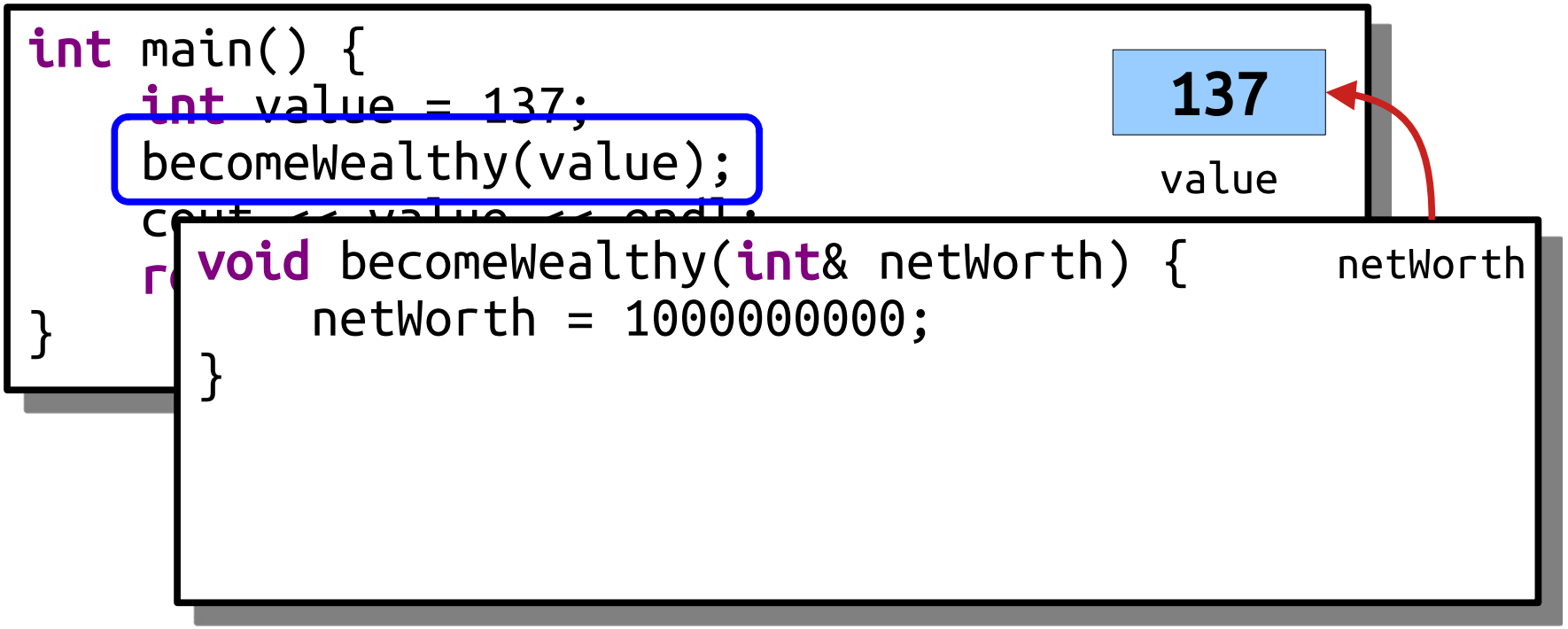
How it Works Now

```
int main() {
    int value = 137;
    becomeWealthy(value);
    cout << value << endl;
    return 0;
}
```

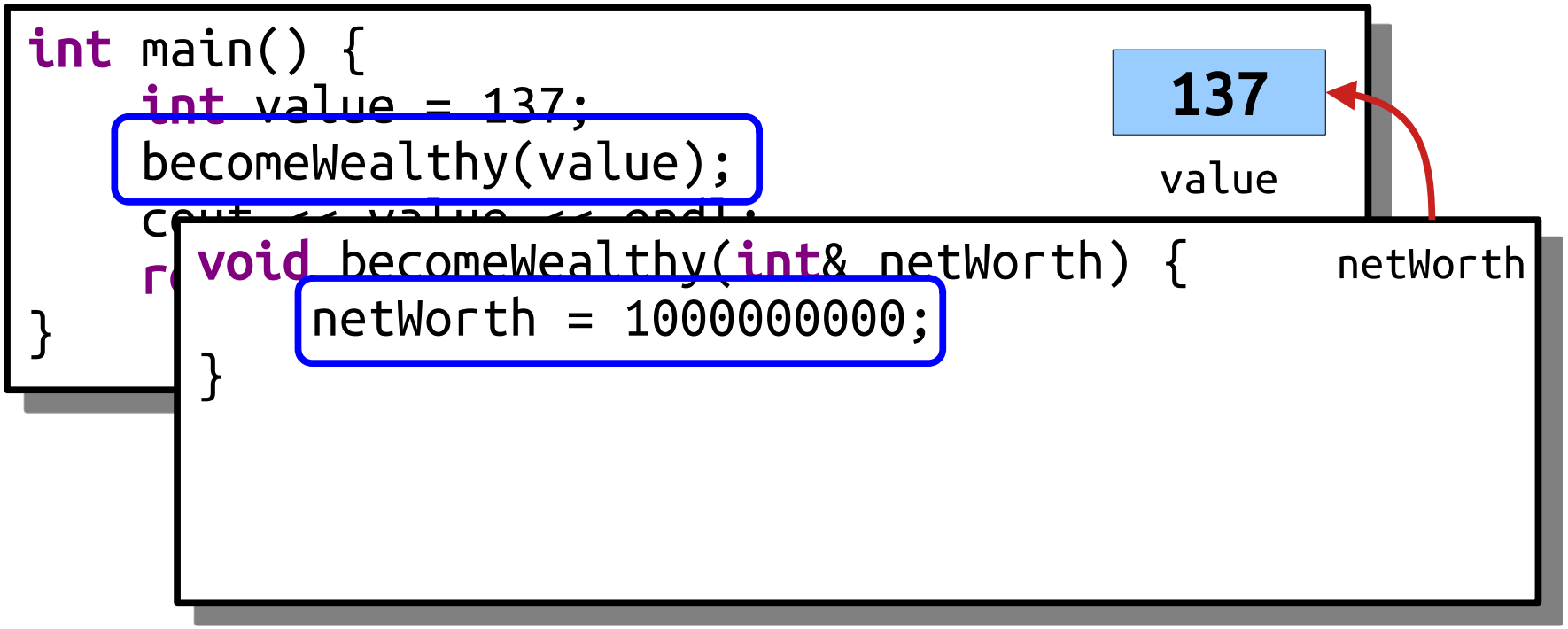
How it Works Now

```
int main() {
    int value = 137;
    becomeWealthy(value);
    cout << value << endl;
    return 0;
}
```

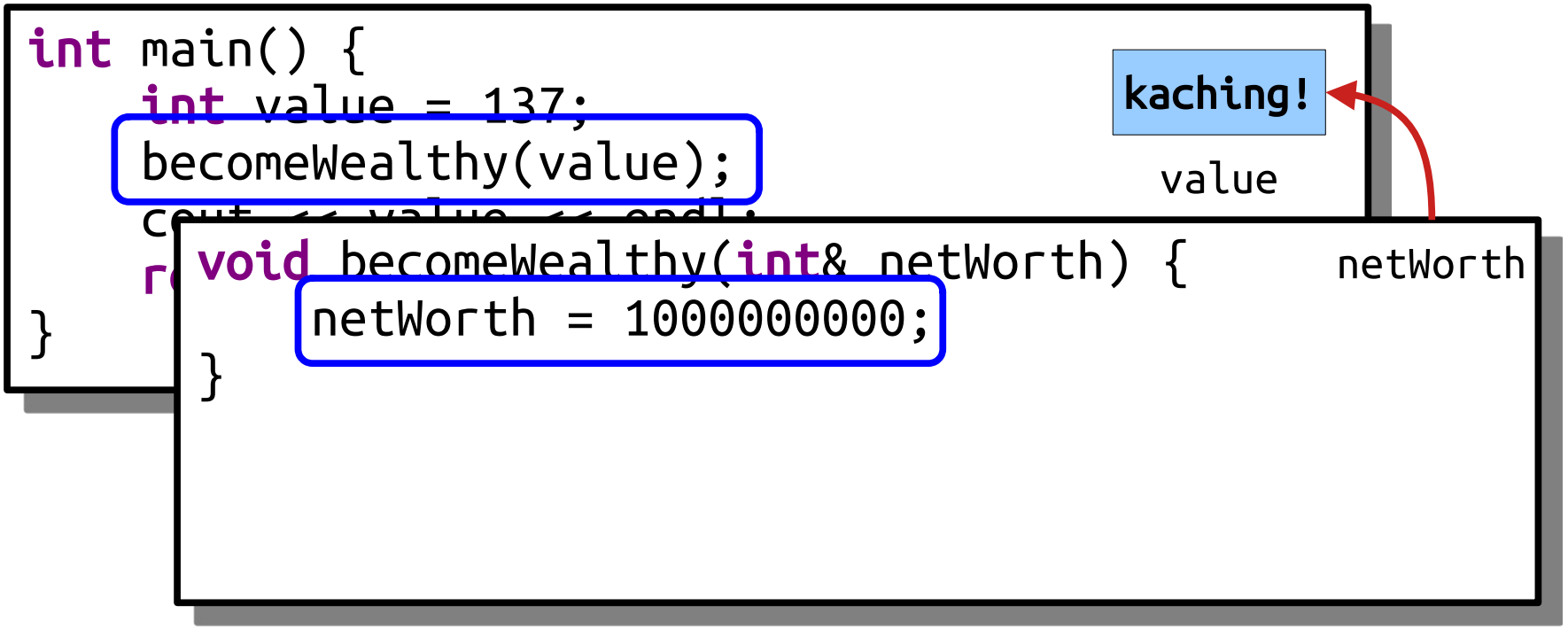
How it Works Now



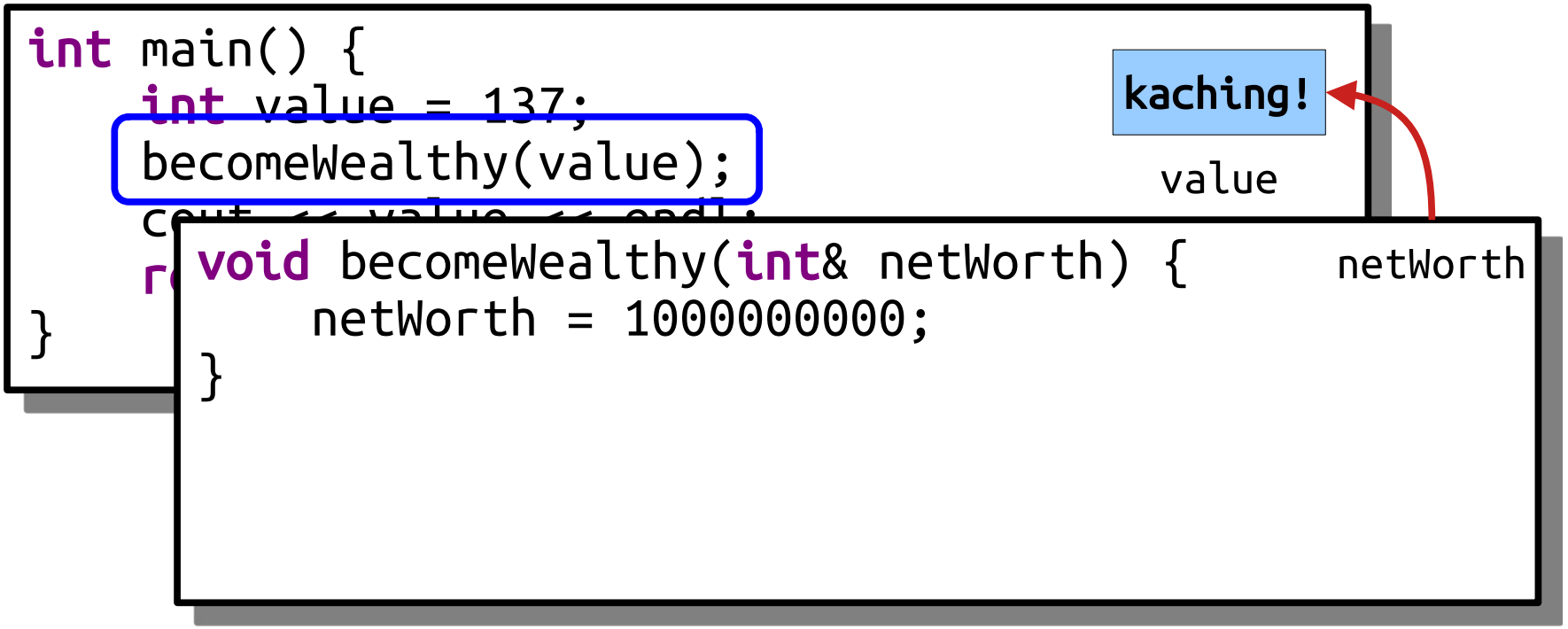
How it Works Now



How it Works Now

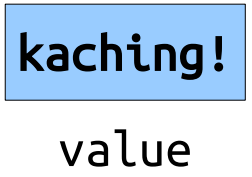


How it Works Now



How it Works Now

```
int main() {
    int value = 137;
    becomeWealthy(value);
    cout << value << endl;
    return 0;
}
```



Make a Prediction!

- Look over this piece of C++ code:

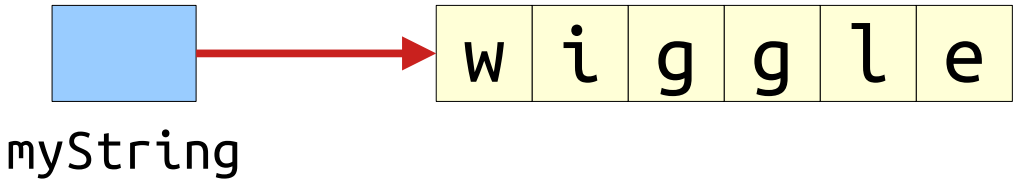
```
void gollyGee(string text) {
    text[0] = 'g';
}
int main() {
    string message = "wiggle";
    gollyGee(message);
    cout << message << endl; // <-- Here
    return 0;
}
```

- What do you think will get printed at the indicated point? Why?

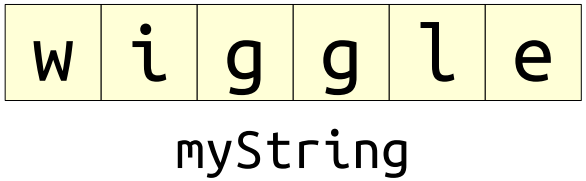
Formulate a hypothesis!

Strings in C++

- In Python, Java, and JavaScript, string variables are not the strings themselves. They're pointers to those strings.



- In C++, a variable of type `string` is an actual, concrete, honest-to-goodness string.



How it Works

```
int main() {
    string message = "wiggle";
    gollyGee(message);
    cout << message << endl;
    return 0;
}
```

How it Works

```
int main() {
    string message = "wiggle";
    gollyGee(message);
    cout << message << endl;
    return 0;
}
```

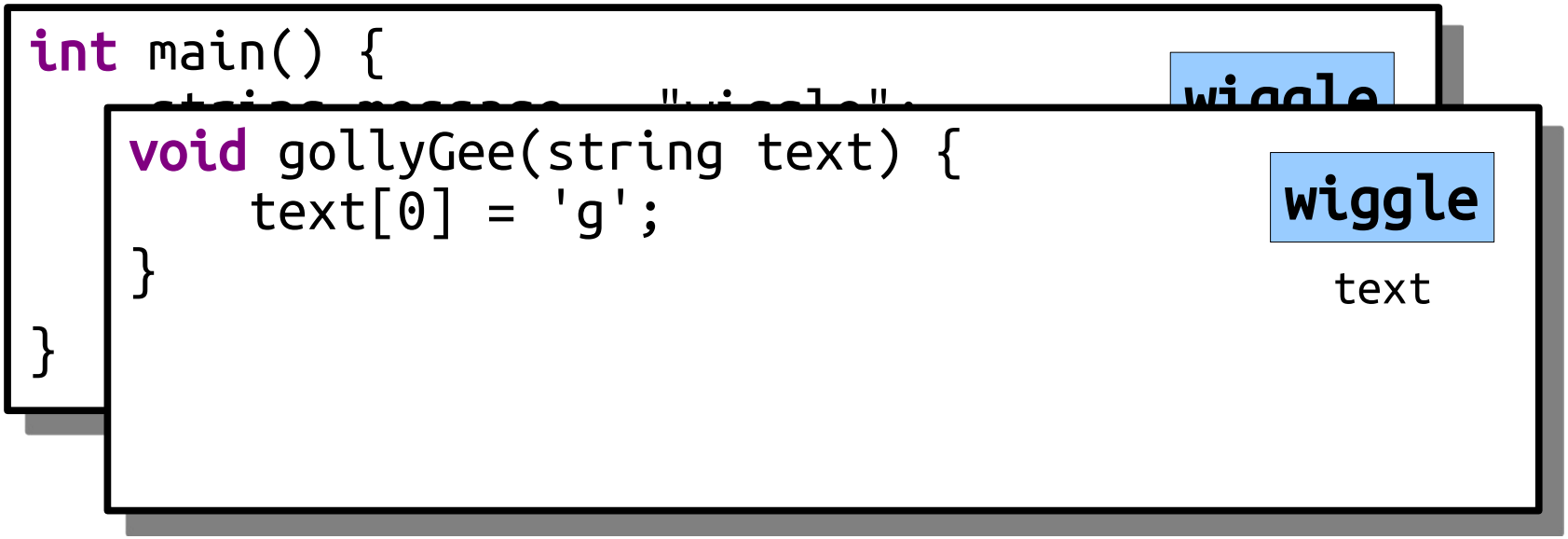
How it Works

```
int main() {
    string message = "wiggle";
    gollyGee(message);
    cout << message << endl;
    return 0;
}
```

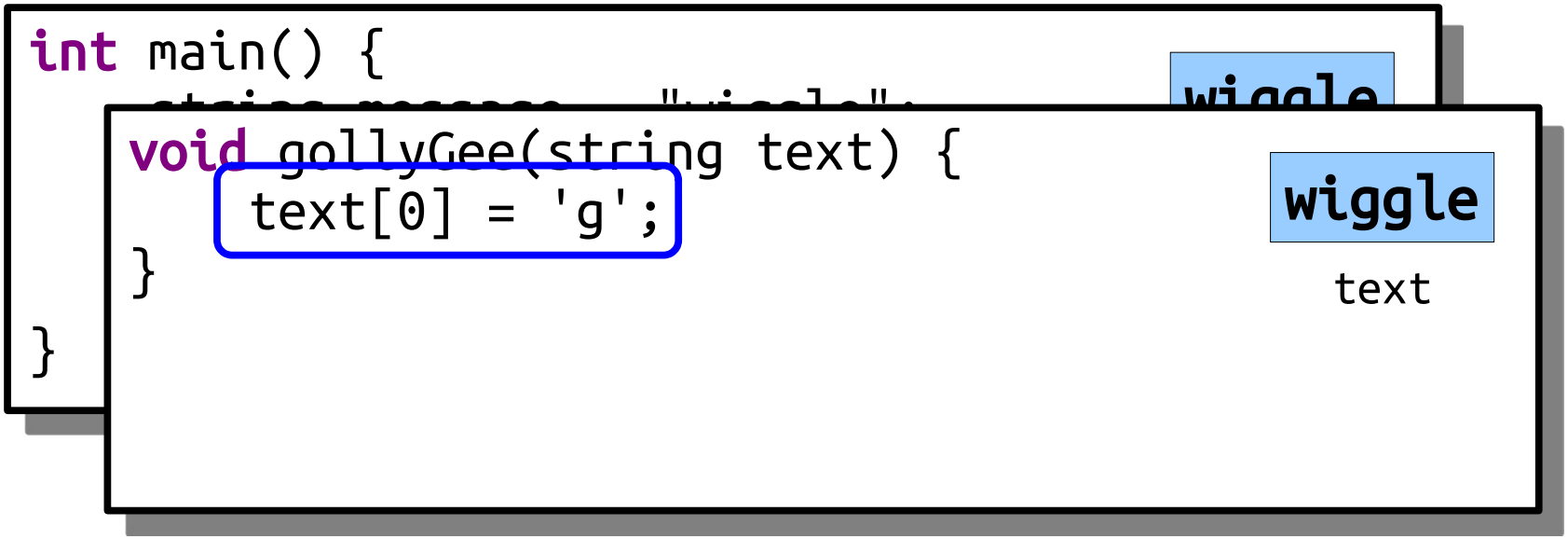
How it Works

```
int main() {
    string message = "wiggle";
    gollyGee(message);
    cout << message << endl;
    return 0;
}
```

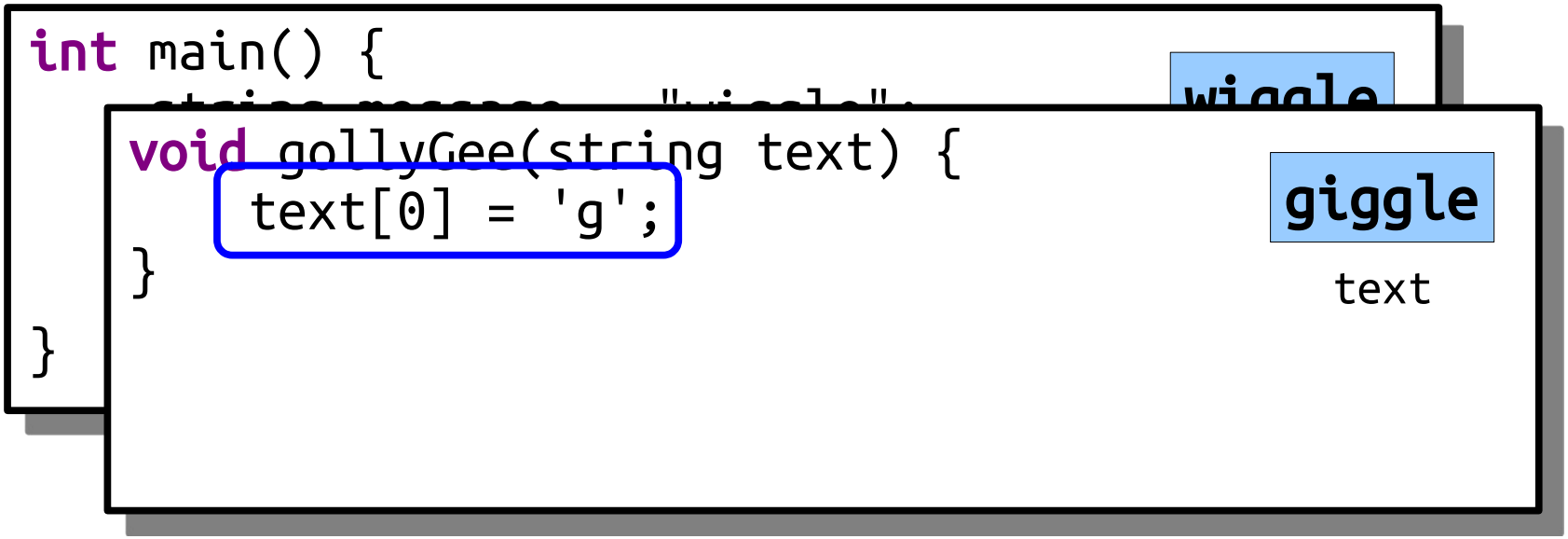
How it Works


int main() {
}
void gollyGee(string text) {
text[0] = 'g';
}
wiggle
wiggle
text

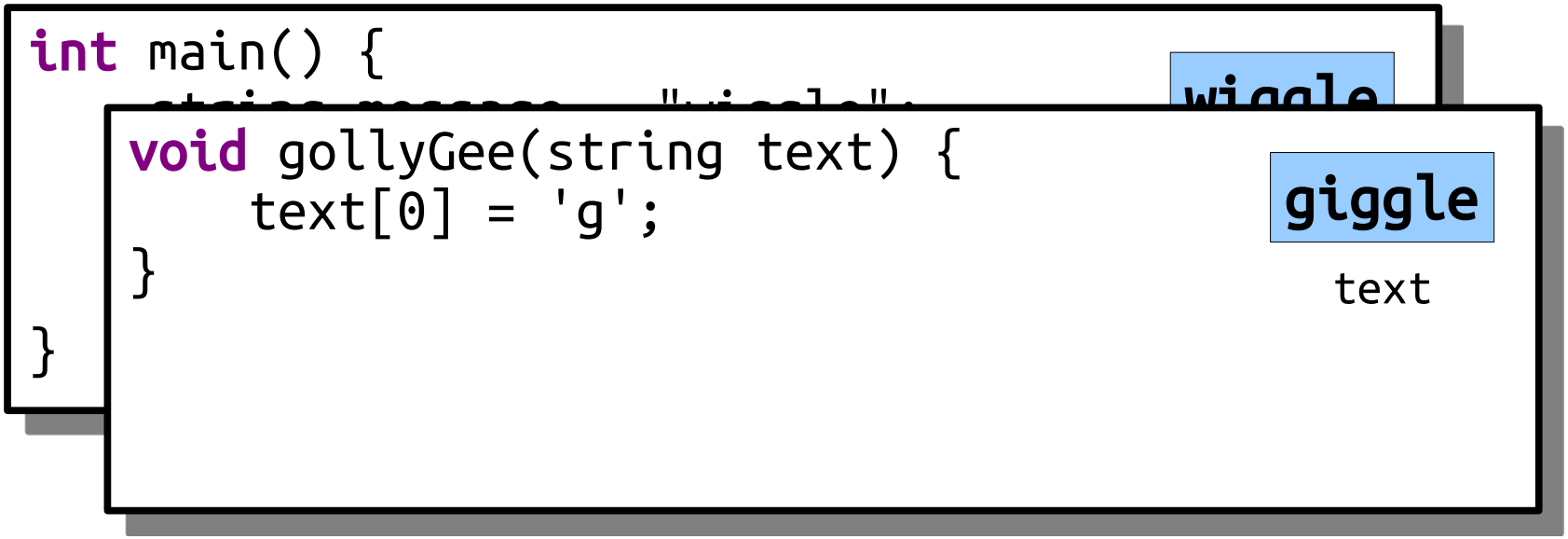
How it Works


int main() {
}
void gollyGee(string text) {
text[0] = 'g';
}
wiggle
wiggle
text

How it Works


int main() {
}
void gollyGee(string text) {
text[0] = 'g';
}
wiggle
giggle
text

How it Works


int main() {
}
void gollyGee(string text) {
text[0] = 'g';
}
wiggle
giggle
text

How it Works

```
int main() {
    string message = "wiggle";
    gollyGee(message);
    cout << message << endl;
    return 0;
}
```

Adding An Ampersand

How it Works Now

```
int main() {
    string message = "wiggle";
    gollyGee(message);
    cout << message << endl;
    return 0;
}
```

How it Works Now

```
int main() {
    string message = "wiggle";
    gollyGee(message);
    cout << message << endl;
    return 0;
}
```

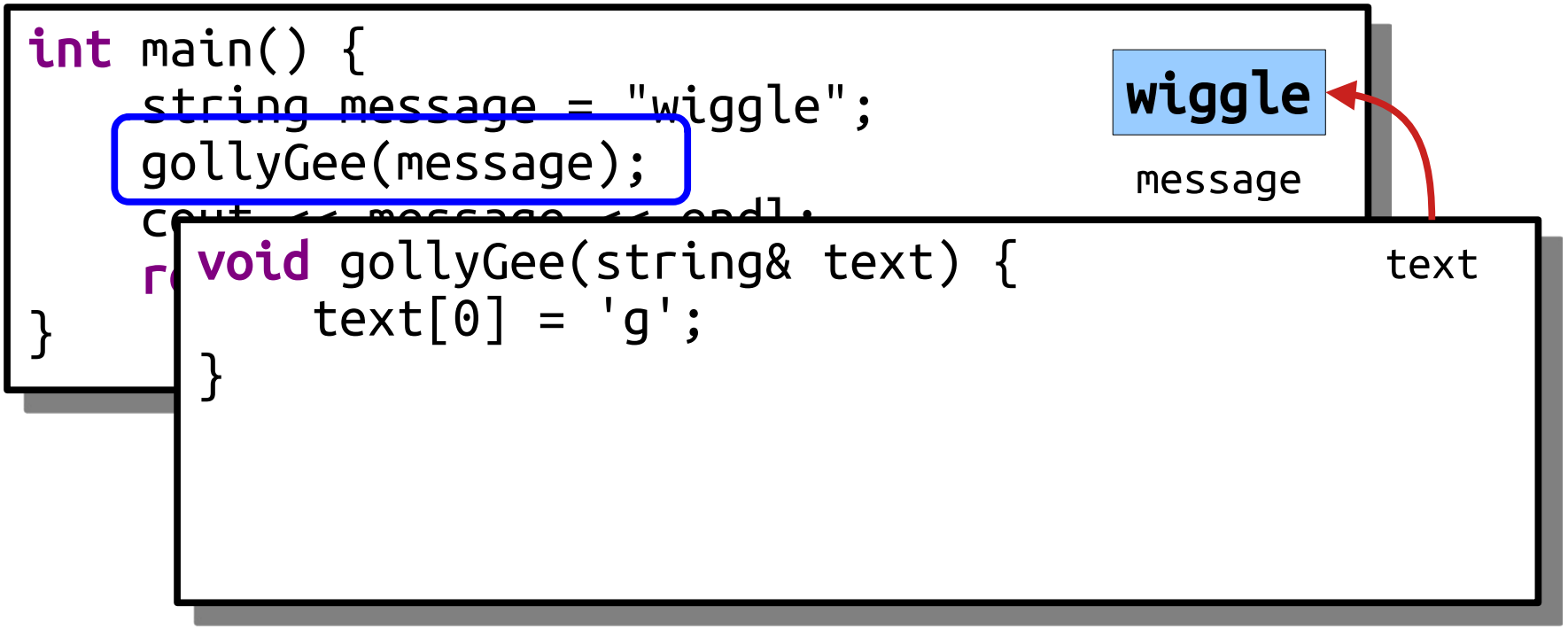
How it Works Now

```
int main() {
    string message = "wiggle";
    gollyGee(message);
    cout << message << endl;
    return 0;
}
```

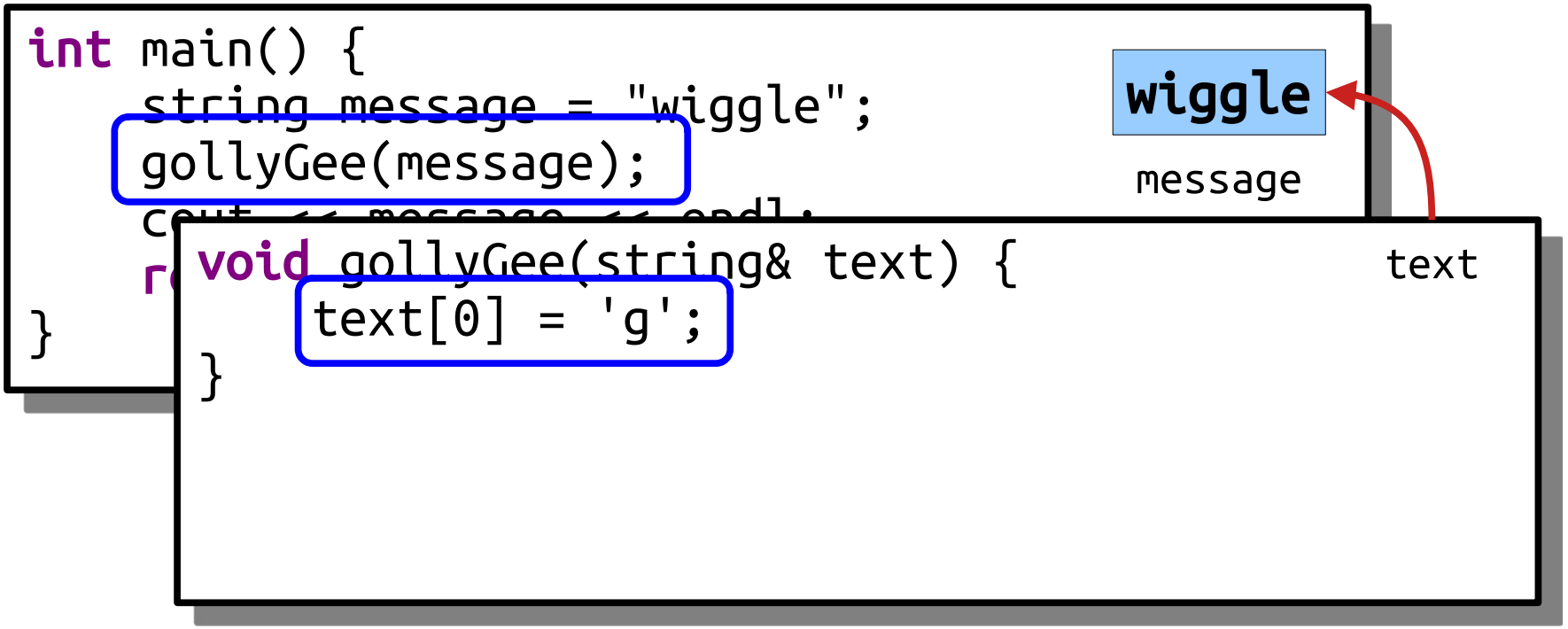
How it Works Now

```
int main() {
    string message = "wiggle";
    gollyGee(message);
    cout << message << endl;
    return 0;
}
```

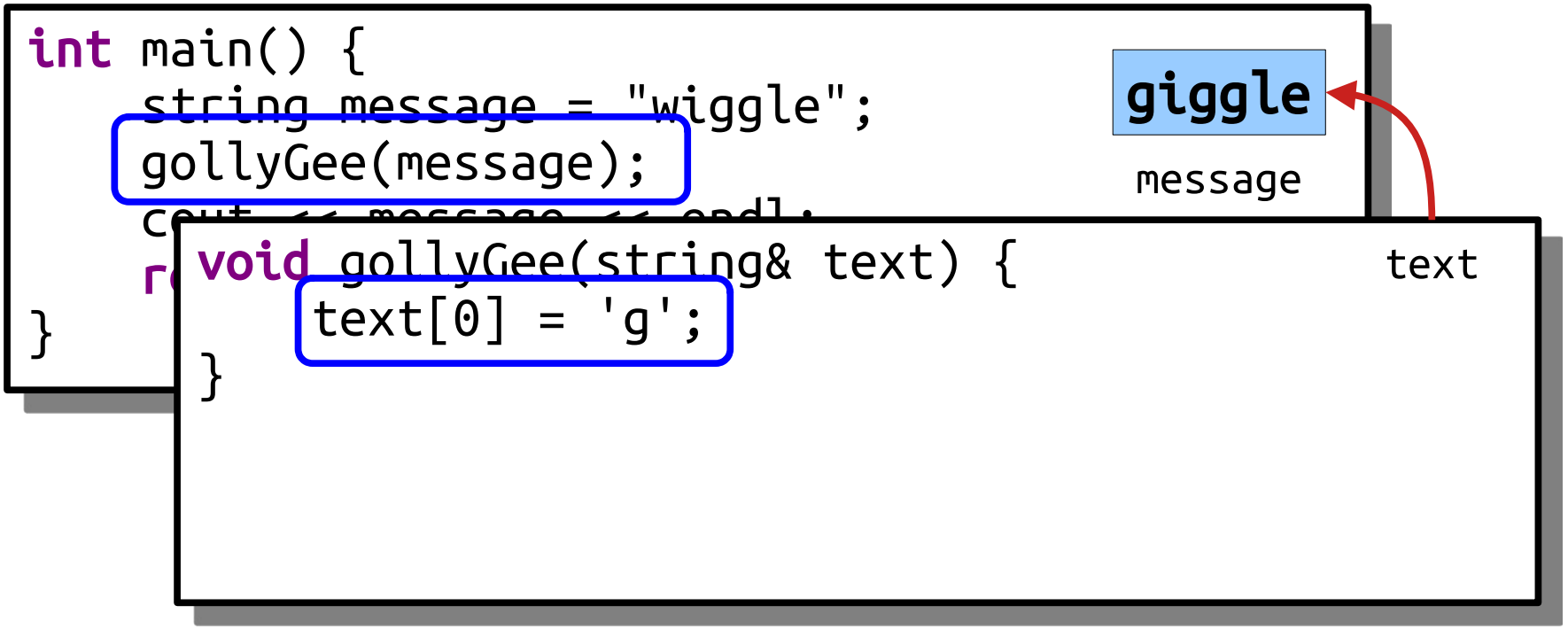
How it Works Now



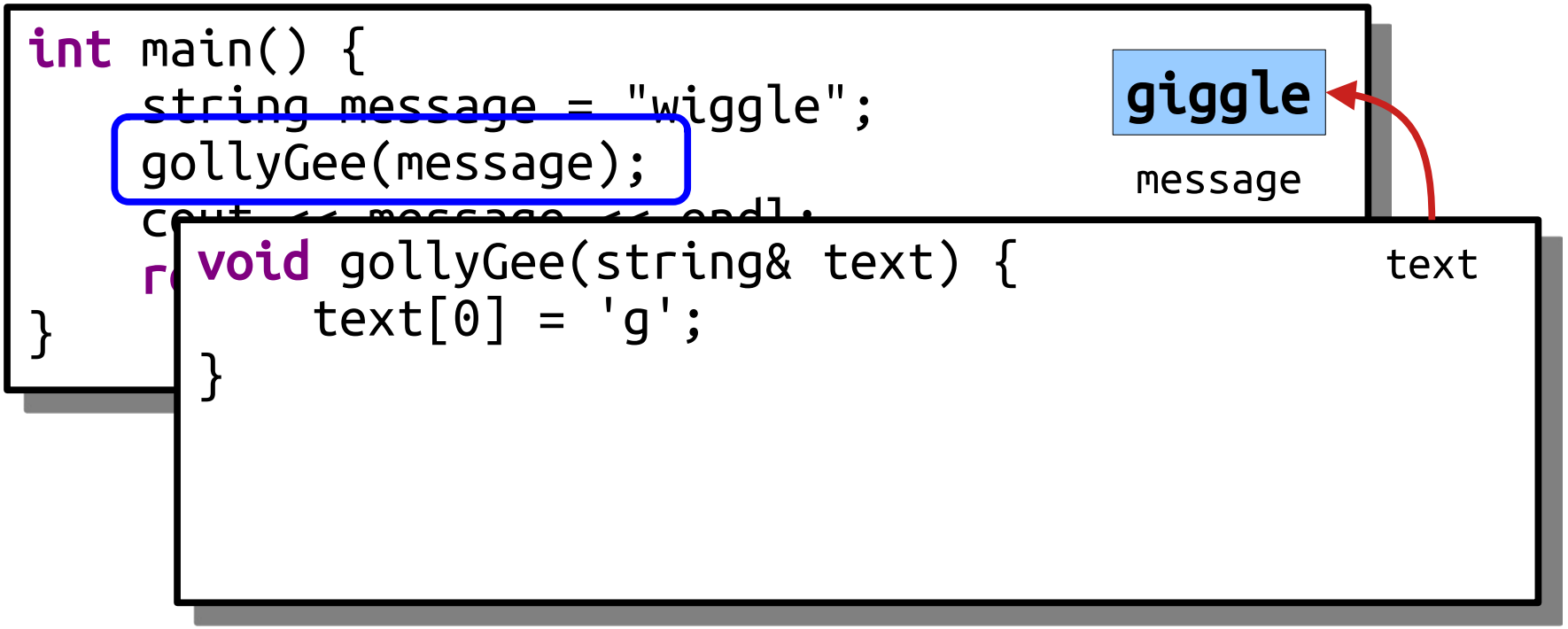
How it Works Now



How it Works Now



How it Works Now

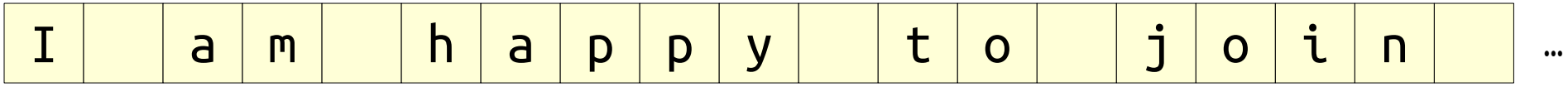


How it Works Now

```
int main() {
    string message = "wiggle";
    gollyGee(message);
    cout << message << endl;
    return 0;
}
```

A Question of Speed

- When working with strings, pass-by-value is slower than pass-by-reference because of the cost of copying the string.



- ***General principle:*** When passing a string into a function, use pass-by-reference unless you actually want a copy of the string.

Do You Trust Me?

- Suppose you've written the next Great American Novel and the single, sole copy is stored in the variable

```
string myMasterpiece;
```

- You see a function with this signature:

```
void totallyNotSketchy(string& text);
```

- Would you make this call?

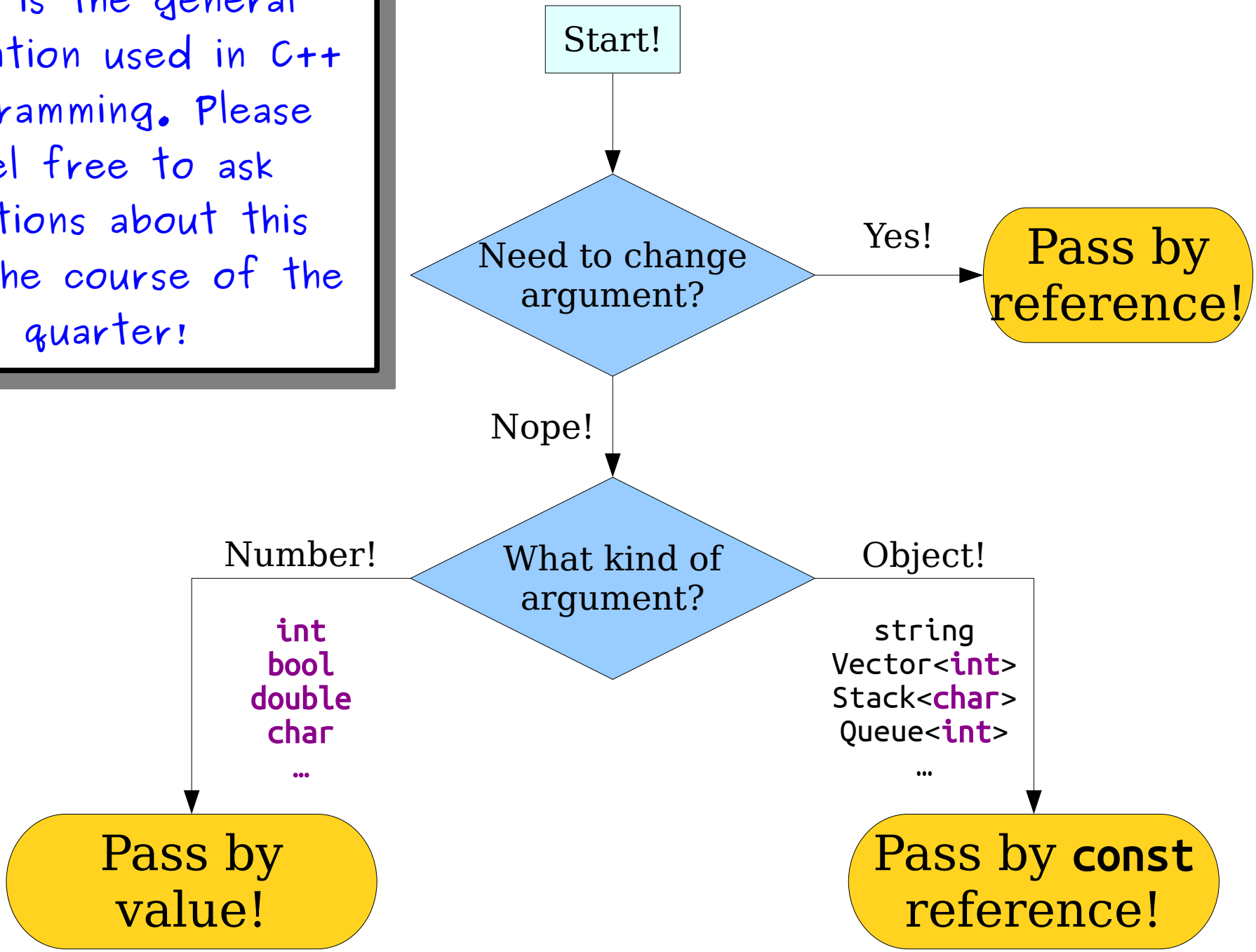
```
totallyNotSketchy(myMasterpiece);
```

Pass-by-const-Reference

- If you want to look at, but not modify, a function parameter, pass it by ***const reference***:
  - The “by reference” part avoids a copy.
  - The “**const**” (constant) part means that the function can’t change that argument.
- For example:

```
void proofreadLongEssay(const string& essay) {
   /* can read, but not change, the essay.  */
}
```

This is the general convention used in C++ programming. Please feel free to ask questions about this over the course of the quarter!



Container Types

Container Types

- A ***collection class*** (also called an ***abstract data type*** or ***container class***) is a data type used to store and organize data in some form.
  - These are things like arrays, lists, maps, dictionaries, etc.
- Our next three lectures exploring collections and how to use them appropriately.
- Later, we'll analyze their efficiencies. For now, let's just focus on how to use them.

Vector

Vector

- A **`Vector`** is a collection class representing a list of things.
- It's similar to Java's `ArrayList`, JavaScript's arrays, and Python's lists.
- To make a `Vector`, use this syntax:

```
Vector<type> name;
```

- All elements of a `Vector` have to have the same type. You specify that type by placing it in <angle brackets> after the word `Vector`.

`Vector` in Action

```
/*      Stanford C++ Version      */
Vector<int> v = { 1, 3, 7 };


v += 271;

cout << v[0] << endl;
cout << v[v.size() - 1] << endl;

Vector<int> first = v.subList(0, 2);
Vector<int> last  = v.subList(2);

v.remove(0);
```

```
"""   Python Version   """
v = [1, 3, 7]


v.append(271)

print(v[0])
print(v[-1])

first = v[0:2]
last  = v[2:]

del v[0]
```

```
/*          Java Version          */
List<> v = new ArrayList<Integer>();
v.add(1); v.add(3); v.add(7);

v.add(271);

System.out.println(v.get(0));
System.out.println(v.get(v.size()-1));

List<Integer> first = v.subList(0, 2);
List<Integer> last  = v.subList(2);

v.remove(0);
```

```
//    JavaScript Version
let v = [1, 3, 7];


v.push(271);

console.log(v[0]);
console.log(v[v.length - 1]);

let first = v.slice(0, 2);
let last  = v.slice(2);

v.splice(0, 0);
```

```
/*      Stanford C++ Version      */
Vector<int> v = { 1, 3, 7 };


v += 271;

cout << v[0] << endl;
cout << v[v.size() - 1] << endl;

Vector<int> first = v.subList(0, 2);
Vector<int> last  = v.subList(2);

v.remove(0);
```

```
"""   Python Version   """
v = [1, 3, 7]


v.append(271)

print(v[0])
print(v[-1])

first = v[0:2]
last  = v[2:]

del v[0]
```

```
/*           Java Version           */
List<> v = new ArrayList<Integer>();
v.add(1); v.add(3); v.add(7);

v.add(271);

System.out.println(v.get(0));
System.out.println(v.get(v.size()-1));

List<Integer> first = v.subList(0, 2);
List<Integer> last  = v.subList(2);

v.remove(0);
```

```
//    JavaScript Version
let v = [1, 3, 7];


v.push(271);

console.log(v[0]);
console.log(v[v.length - 1]);

let first = v.slice(0, 2);
let last  = v.slice(2);

v.splice(0, 0);
```

```
/*      Stanford C++ Version      */
Vector<int> v = { 1, 3, 7 };

v += 271;

cout << v[0] << endl;
cout << v[v.size() - 1] << endl;

Vector<int> first = v.subList(0, 2);
Vector<int> last  = v.subList(2);

v.remove(0);
```

```
"""    Python Version    """
v = [1, 3, 7]

v.append(271)

print(v[0])
print(v[-1])

first = v[0:2]
last  = v[2:]
```

Note the use of curly braces rather than square brackets here.

```
/*           Java Version           */
List<> v = new ArrayList<Integer>();
v.add(1); v.add(3); v.add(7);

v.add(271);

System.out.println(v.get(0));
System.out.println(v.get(v.size()-1));

List<Integer> first = v.subList(0, 2);
List<Integer> last  = v.subList(2);

v.remove(0);
```

```
let v = [1, 3, 7];


v.push(271);

console.log(v[0]);
console.log(v[v.length - 1]);

let first = v.slice(0, 2);
let last  = v.slice(2);

v.splice(0, 0);
```

```
/*      Stanford C++ Version      */
Vector<int> v = { 1, 3, 7 };

v += 271;

cout << v[0] << endl;
cout << v[v.size() - 1] << endl;

Vector<int> first = v.subList(0, 2);
Vector<int> last  = v.subList(2);

v.remove(0);
```

```
"""    Python Version    """
v = [1, 3, 7]

v.append(271)

print(v[0])
print(v[-1])

first = v[0:2]
last  = v[2:]
```

We append elements using the += operator.

```
/*           Java Version           */
List<> v = new ArrayList<Integer>();
v.add(1); v.add(3); v.add(7);

v.add(271);

System.out.println(v.get(0));
System.out.println(v.get(v.size()-1));

List<Integer> first = v.subList(0, 2);
List<Integer> last  = v.subList(2);

v.remove(0);
```

```
let v = [1, 3, 7];

v.push(271);

console.log(v[0]);
console.log(v[v.length - 1]);

let first = v.slice(0, 2);
let last  = v.slice(2);

v.splice(0, 0);
```

```
/*      Stanford C++ Version      */
Vector<int> v = { 1, 3, 7 };

v += 271;

cout << v[0] << endl;
cout << v[v.size() - 1] << endl;

Vector<int> first = v.subList(0, 2);
Vector<int> last  = v.subList(2);

v.remove(0);
```

```
"""    Python Version    """
v = [1, 3, 7]

v.append(271)

print(v[0])
print(v[-1])

first = v[0:2]
last  = v[2:]
```

We select individual elements out of a Vector using square brackets. Everything is zero-indexed.

```
/*           Java Version           */
List<> v = new ArrayList<Integer>();
v.add(1); v.add(3); v.add(7);

v.add(271);

System.out.println(v.get(0));
System.out.println(v.get(v.size()-1));

List<Integer> first = v.subList(0, 2);
List<Integer> last  = v.subList(2);

v.remove(0);
```

```
v.push(271);

console.log(v[0]);
console.log(v[v.length - 1]);

let first = v.slice(0, 2);
let last  = v.slice(2);

v.splice(0, 0);
```

```
/*      Stanford C++ Version      */
Vector<int> v = { 1, 3, 7 };


v += 271;

cout << v[0] << endl;
cout << v[v.size() - 1] << endl;

Vector<int> first = v.subList(0, 2);
Vector<int> last  = v.subList(2);

v.remove(0);
```

```
"""    Python Version    """
v = [1, 3, 7]


v.append(271)

print(v[0])
print(v[-1])

first = v[0:2]
last  = v[2:]

v.splice(0, 0);
```

```
/*           Java Version           */
List<> v = new ArrayList<Integer>();
v.add(1); v.add(3); v.add(7);

v.add(271);

System.out.println(v.get(0));
System.out.println(v.get(v.size()-1));

List<Integer> first = v.subList(0, 2);
List<Integer> last  = v.subList(2);

v.remove(0);
```

C++ doesn't support negative array indices to mean "count from the back." We have to do some math to find the index of the last element.

We use the syntax v.size() to get the length of a Vector.

```
/*      Stanford C++ Version      */
Vector<int> v = { 1, 3, 7 };


v += 271;

cout << v[0] << endl;
cout << v[v.size() - 1] << endl;

Vector<int> first = v.subList(0, 2);
Vector<int> last  = v.subList(2);

v.remove(0);
```

```
"""    Python Version    """
v = [1, 3, 7]


v.append(271)

print(v[0])
print(v[-1])

first = v[0:2]
last  = v[2:]

del v[0]
```

```
/*           Java Version           */
List<> v = new ArrayList<Integer>();
v.add(1); v.add(3); v.add(7);

v.add(271);

System.out.println(v.get(0));
System.out.println(v.get(v.size()-1));

List<Integer> first = v.subList(0, 2);
List<Integer> last  = v.subList(2);

v.remove(0);
```

The subList member function is used to get a subrange of the subList. Here, first will be the first two elements of the Vector, and last will be the list starting at position 2.

```
/*      Stanford C++ Version      */
Vector<int> v = { 1, 3, 7 };


v += 271;

cout << v[0] << endl;
cout << v[v.size() - 1] << endl;

Vector<int> first = v.subList(0, 2);
Vector<int> last  = v.subList(2);

v.remove(0);
```

```
"""    Python Version    """
v = [1, 3, 7]


v.append(271)

print(v[0])
print(v[-1])

first = v[0:2]
last  = v[2:]

del v[0]
```

```
/*           Java Version           */
List<> v = new ArrayList<Integer>();
v.add(1); v.add(3); v.add(7);

v.add(271);

System.out.println(v.get(0));
System.out.println(v.get(v.size()-1));

List<Integer> first = v.subList(0, 2);
List<Integer> last  = v.subList(2);

v.remove(0);
```

```
let first = v.slice(0, 2);
let last  = v.slice(2);

v.splice(0, 0);
```

We can use the remove member function to remove the element at a given index.

```
/*      Stanford C++ Version      */
Vector<int> v = { 1, 3, 7 };


v += 271;

cout << v[0] << endl;
cout << v[v.size() - 1] << endl;

Vector<int> first = v.subList(0, 2);
Vector<int> last  = v.subList(2);

v.remove(0);
```

```
"""   Python Version   """
v = [1, 3, 7]


v.append(271)

print(v[0])
print(v[-1])

first = v[0:2]
last  = v[2:]

del v[0]
```

```
/*          Java Version          */
List<> v = new ArrayList<Integer>();
v.add(1); v.add(3); v.add(7);

v.add(271);

System.out.println(v.get(0));
System.out.println(v.get(v.size()-1));

List<Integer> first = v.subList(0, 2);
List<Integer> last  = v.subList(2);

v.remove(0);
```

```
//    JavaScript Version
let v = [1, 3, 7];


v.push(271);

console.log(v[0]);
console.log(v[v.length - 1]);

let first = v.slice(0, 2);
let last  = v.slice(2);

v.splice(0, 0);
```

```
/*      Stanford C++ Version        */
Vector<string> v = { "A", "B", "C" };

/* Counting for loop. */
for (int i = 0; i < v.size(); i++) {
    cout << v[i] << endl;
}

/* Range-based for loop. */
for (string elem: v) {
    cout << elem << endl;
}
```

```
"""     Python Version      """
v = ["A", "B", "C"]

# Counting for loop.
for i in range(len(v)):
    print(v[i])

# Range-based for loop.
for elem in v:
    print(elem)
```

```
/*          Java Version            */
List<> v = new ArrayList<String>();
v.add("A"); v.add("B"); v.add("C");

/* Counting for loop. */
for (int i = 0; i < v.size(); i++) {
    System.out.println(v[i]);
}

/* Range-based for loop. */
for (String elem: v) {
    System.out.println(elem);
}
```

```
//    JavaScript Version
let v = ["A", "B", "C"];

// Counting for loop.
for (let i in v) {
    console.log(v[i]);
}

// Range-based for loop.
for (let elem of v) {
    console.log(elem);
}
```

```cpp
/*      Stanford C++ Version        */
Vector<string> v = { "A", "B", "C" };

/* Counting for loop. */
for (int i = 0; i < v.size(); i++) {
    cout << v[i] << endl;
}

/* Range-based for loop. */
for (string elem: v) {
    cout << elem << endl;
}
```

```python
"""    Python Version     """
v = ["A", "B", "C"]

# Counting for loop.
for i in range(len(v)):
    print(v[i])

# Range-based for loop.
for elem in v:
    print(elem)
```

```java
/*           Java Version             */
List<> v = new ArrayList<String>();
v.add("A"); v.add("B"); v.add("C");

/* Counting for loop. */
for (int i = 0; i < v.size(); i++) {
    System.out.println(v[i]);
}

/* Range-based for loop. */
for (String elem: v) {
    System.out.println(elem);
}
```

```javascript
//    JavaScript Version
let v = ["A", "B", "C"];

// Counting for loop.
for (let i in v) {
    console.log(v[i]);
}

// Range-based for loop.
for (let elem of v) {
    console.log(elem);
}
```

```cpp
/*      Stanford C++ Version        */
Vector<string> v = { "A", "B", "C" };

/* Counting for loop. */
for (int i = 0; i < v.size(); i++) {
    cout << v[i] << endl;
}

/* Range-based for loop. */
for (string elem: v) {
    cout << elem << endl;
}
```

```python
"""     Python Version      """
v = ["A", "B", "C"]

# Counting for loop.
for i in range(len(v)):
    print(v[i])

# Range-based for loop.
for elem in v:
    print(elem)
```

```java
/*          Java Version            */
List<> v = new ArrayList<String>();
v.add("A"); v.add("B"); v.add("C");

/* Counting for loop. */
for (int i = 0; i < v.size(); i++) {
    System.out.println(v[i]);
}

/* Range-based for loop. */
for (String elem: v) {
    System.out.println(elem);
}
```

```javascript
    console.log(elem);
}
```

We can iterate over the elements of a Vector by counting upward from 0 (inclusive) to its size (exclusive) and accessing each element.

```cpp
/*      Stanford C++ Version        */
Vector<string> v = { "A", "B", "C" };

/* Counting for loop. */
for (int i = 0; i < v.size(); i++) {
    cout << v[i] << endl;
}

/* Range-based for loop. */
for (string elem: v) {
    cout << elem << endl;
}
```

```python
"""     Python Version      """
v = ["A", "B", "C"]

# Counting for loop.
for i in range(len(v)):
    print(v[i])

# Range-based for loop.
for elem in v:
    print(elem)
```

```java
/*          Java Version            */
List<> v = new ArrayList<String>();
v.add("A"); v.add("B"); v.add("C");

/* Counting for loop. */
for (int i = 0; i < v.size(); i++) {
    System.out.println(v[i]);
}

/* Range-based for loop. */
for (String elem: v) {
    System.out.println(elem);
}
```

```javascript
// Range-based for loop.
for (let elem of v) {
    console.log(elem);
}
```

We can also use this loop structure, which visits each element of the Vector in the order in which they appear.

```
/*      Stanford C++ Version        */
Vector<string> v = { "A", "B", "C" };

/* Counting for loop. */
for (int i = 0; i < v.size(); i++) {
    cout << v[i] << endl;
}

/* Range-based for loop. */
for (string elem: v) {
    cout << elem << endl;
}
```

```
"""     Python Version      """
v = ["A", "B", "C"]

# Counting for loop.
for i in range(len(v)):
    print(v[i])

# Range-based for loop.
for elem in v:
    print(elem)
```

```
/*           Java Version           */
List<> v = new ArrayList<String>();
v.add("A"); v.add("B"); v.add("C");

/* Counting for loop. */
for (int i = 0; i < v.size(); i++) {
    System.out.println(v[i]);
}

/* Range-based for loop. */
for (String elem: v) {
    System.out.println(elem);
}
```

```
//     JavaScript Version
let v = ["A", "B", "C"];

// Counting for loop.
for (let i in v) {
    console.log(v[i]);
}

// Range-based for loop.
for (let elem of v) {
    console.log(elem);
}
```

To read more about the `Vector` and how to use it, check out the

Stanford C++ Library Documentation

up on the course website.

Make a Prediction!

- Look over this piece of C++ code:

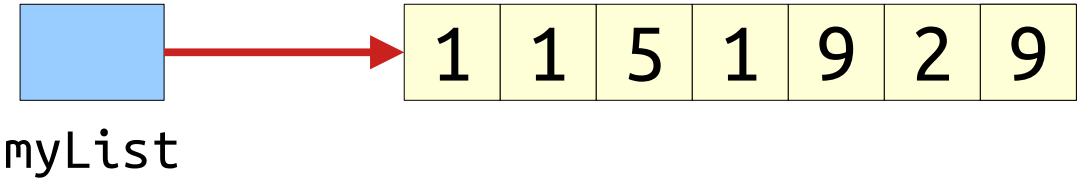
```
void dream(Vector<int> numbers) {
    numbers[1] = 1963;
}
int main() {
    Vector<int> values = { 1929, 1955, 1964 };
    dream(values);
    cout << values << endl; // <-- Here
    return 0;
}
```

- What do you think will get printed at the indicated point? Why?

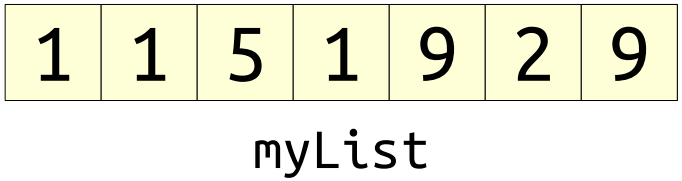
Formulate a hypothesis!

Objects in C++

- In Python, Java, and JavaScript, object variables are not the objects themselves. They're pointers to those objects:



- In C++, a variable of object type is an actual, concrete, honest-to-goodness object.



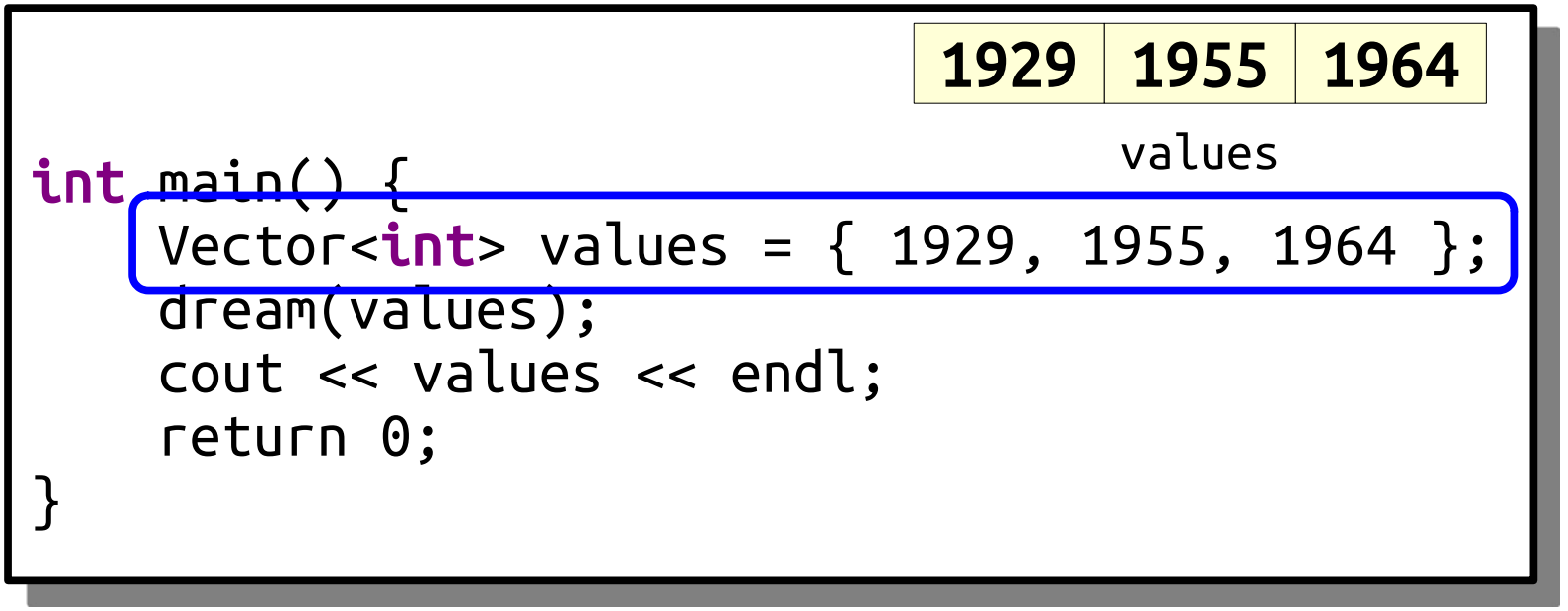
How it Works

```
int main() {
    Vector<int> values = { 1929, 1955, 1964 };
    dream(values);
    cout << values << endl;
    return 0;
}
```

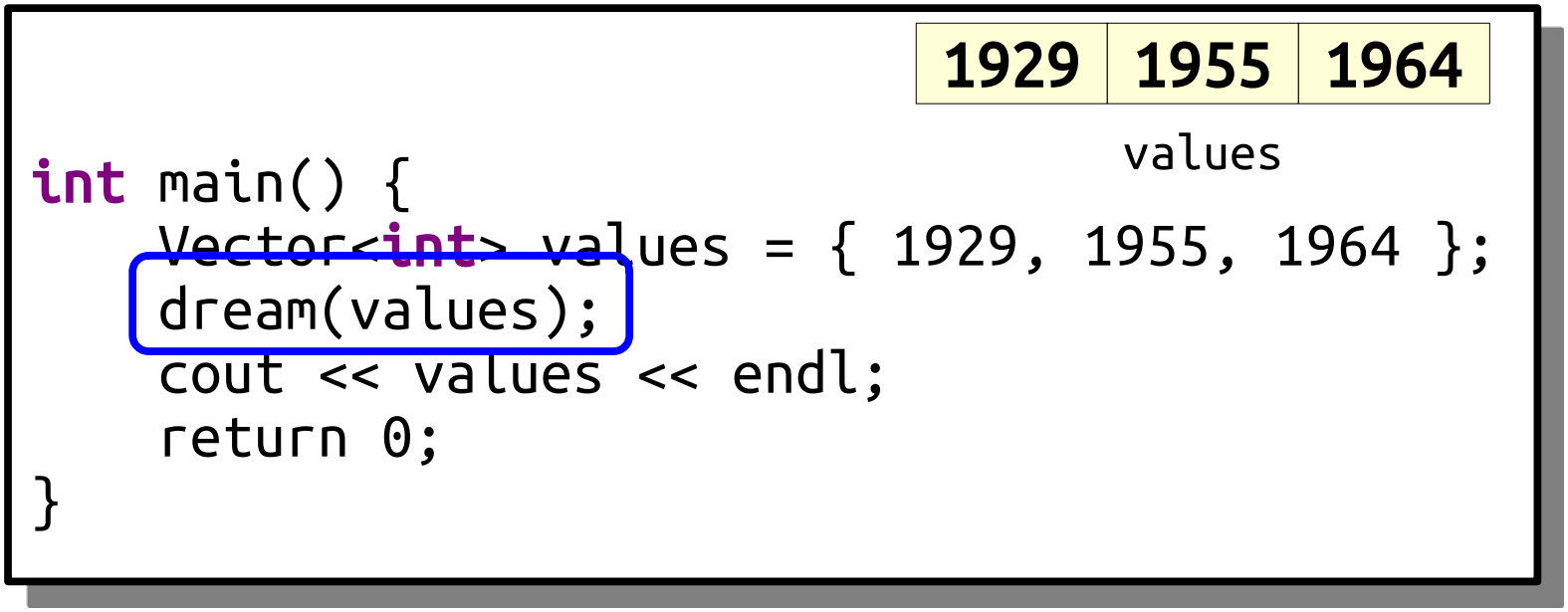
How it Works

```
int main() {
    Vector<int> values = { 1929, 1955, 1964 };
    dream(values);
    cout << values << endl;
    return 0;
}
```

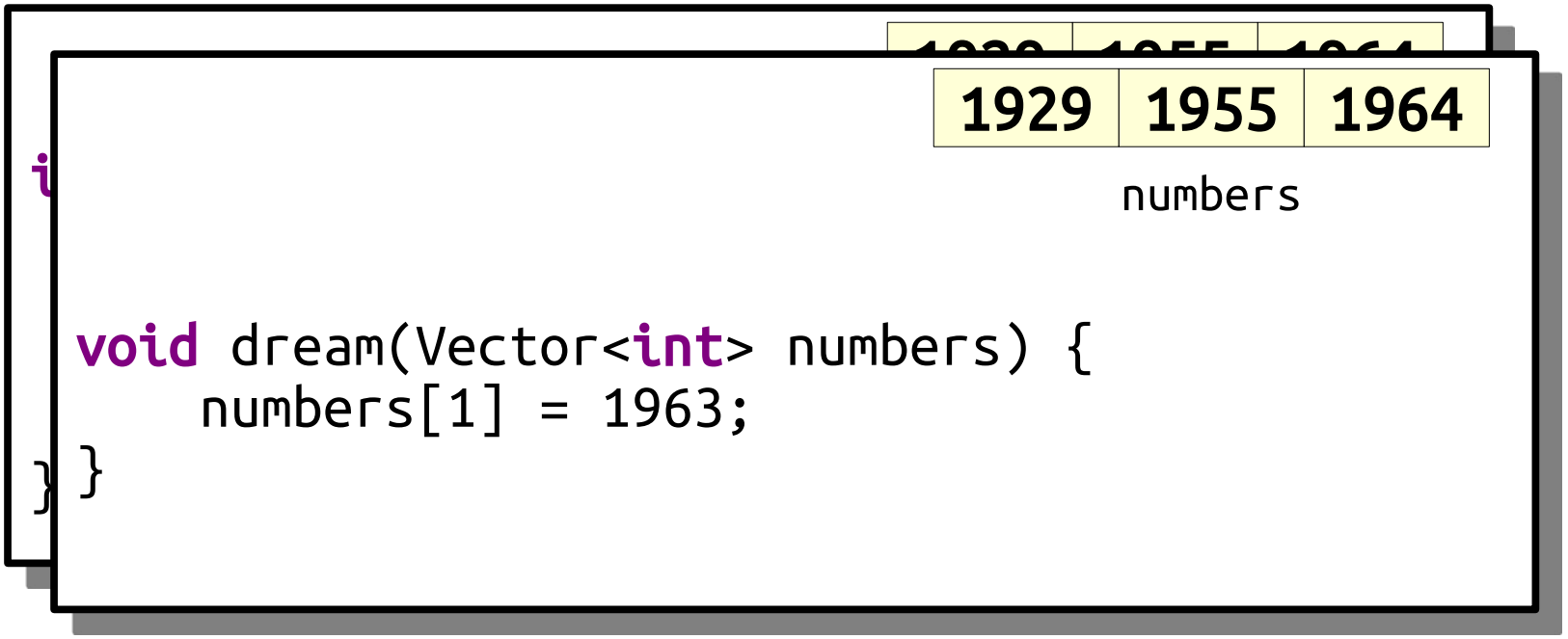
How it Works


1929
1955
1964
values
int main() {
Vector<int> values = { 1929, 1955, 1964 };
dream(values);
cout << values << endl;
return 0;
}

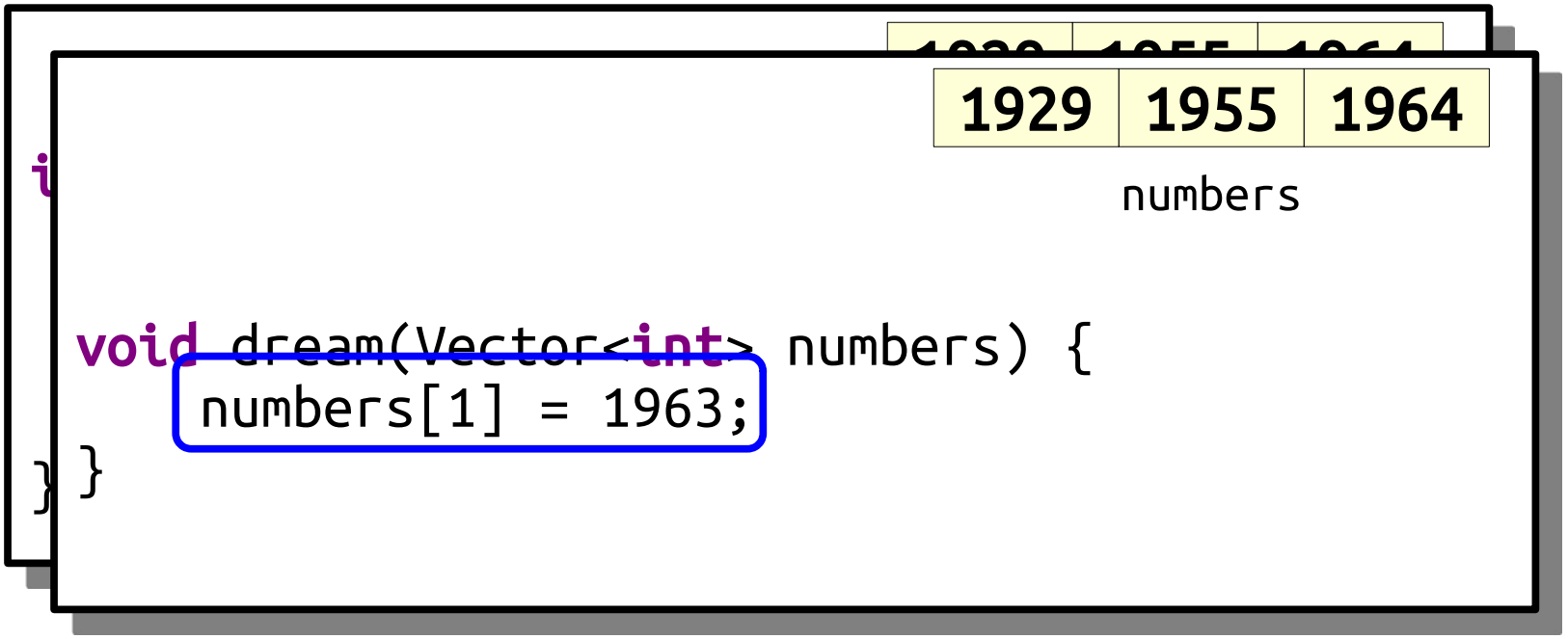
How it Works


1929
1955
1964
values
int main() {
Vector<int> values = { 1929, 1955, 1964 };
dream(values);
cout << values << endl;
return 0;
}

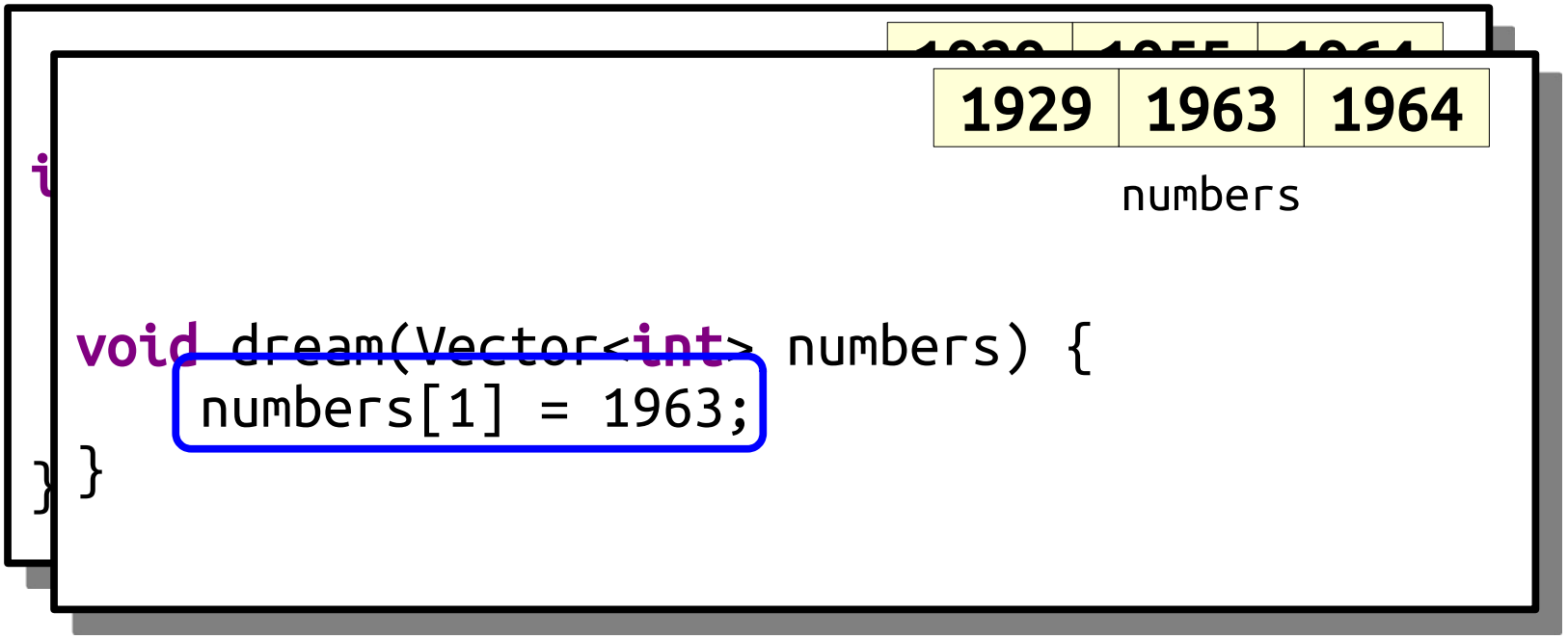
How it Works


1929
1955
1964
numbers
void dream(Vector<int> numbers) {
numbers[1] = 1963;
}

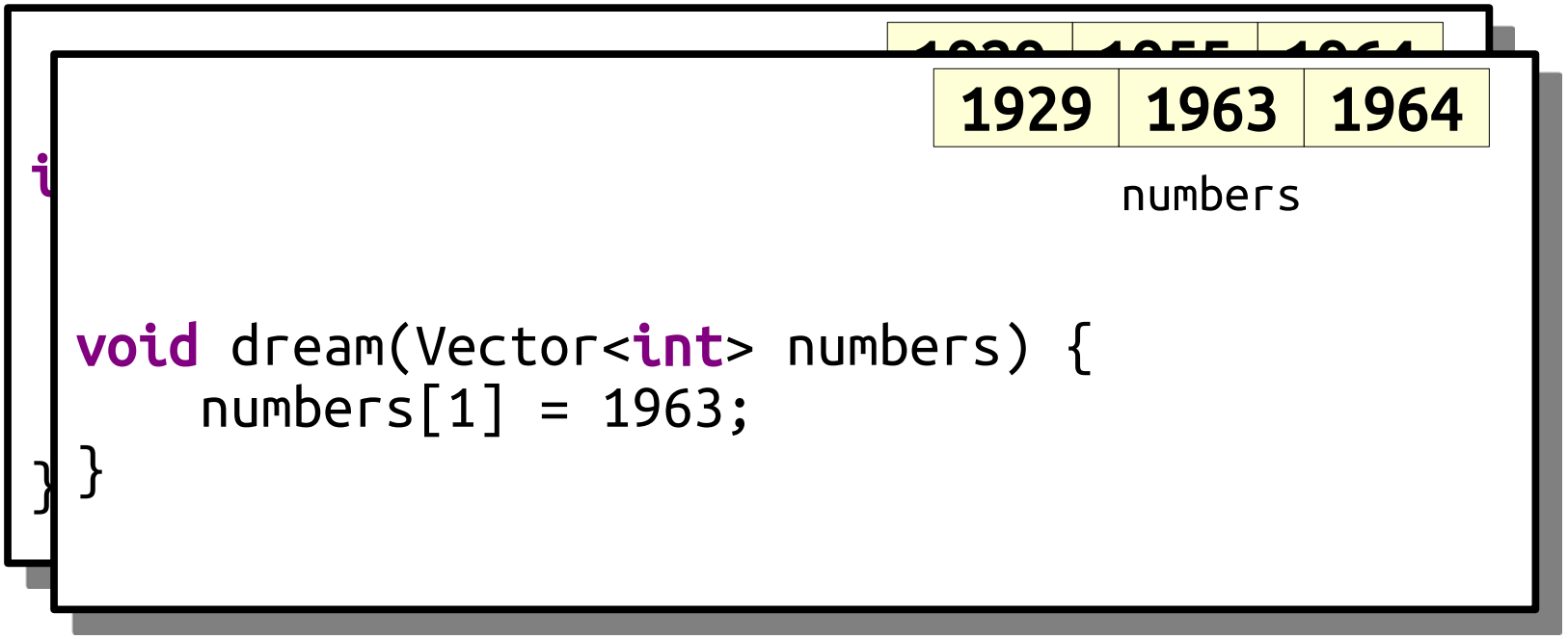
How it Works


1929
1955
1964
numbers
void dream(Vector<int> numbers) {
numbers[1] = 1963;
}

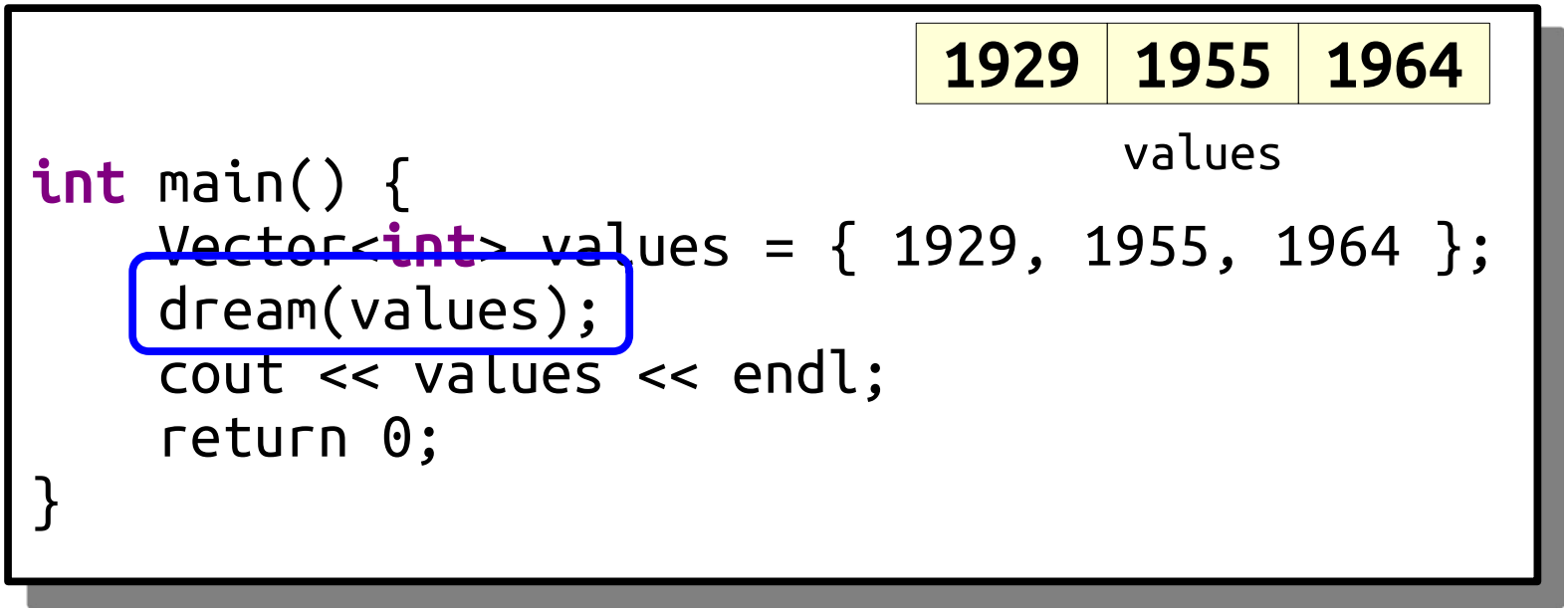
How it Works


1929
1963
1964
numbers
void dream(Vector<int> numbers) {
numbers[1] = 1963;
}

How it Works


1929
1963
1964
numbers
void dream(Vector<int> numbers) {
numbers[1] = 1963;
}

How it Works


1929
1955
1964
values
int main() {
Vector<int> values = { 1929, 1955, 1964 };
dream(values);
cout << values << endl;
return 0;
}

How it Works Now

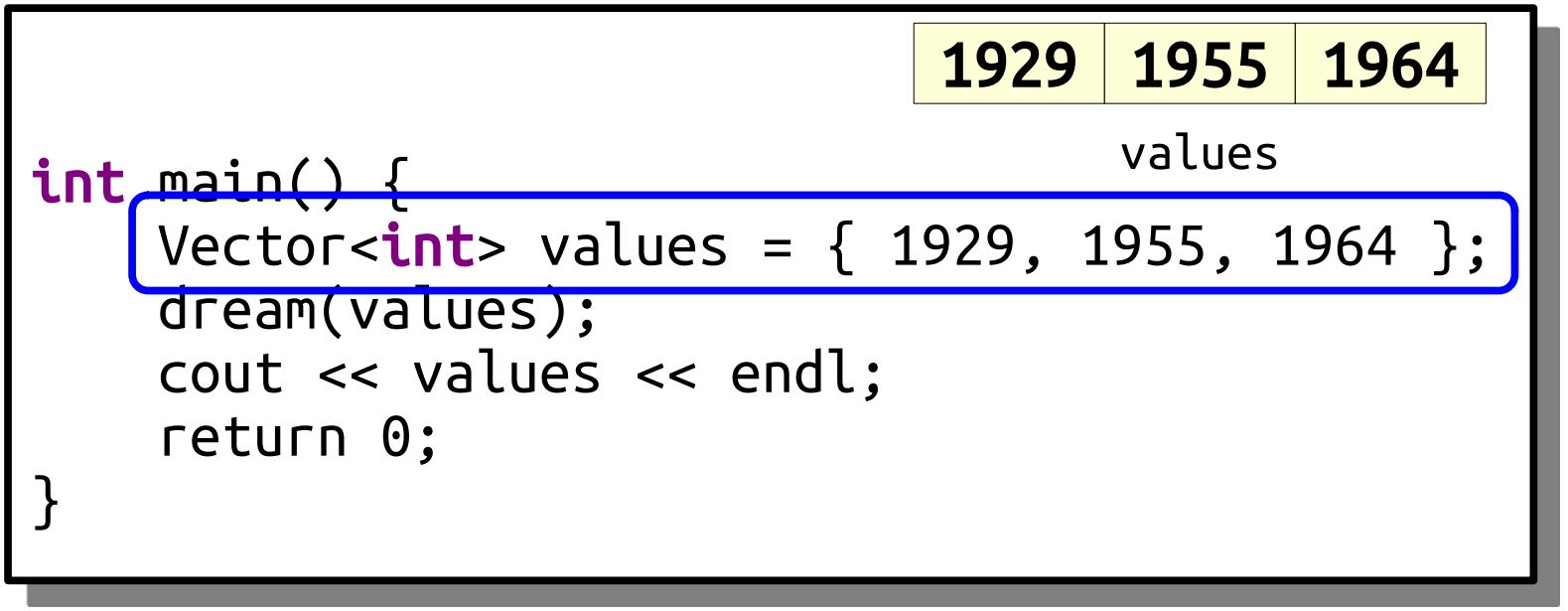
How it Works Now

```
int main() {
    Vector<int> values = { 1929, 1955, 1964 };
    dream(values);
    cout << values << endl;
    return 0;
}
```

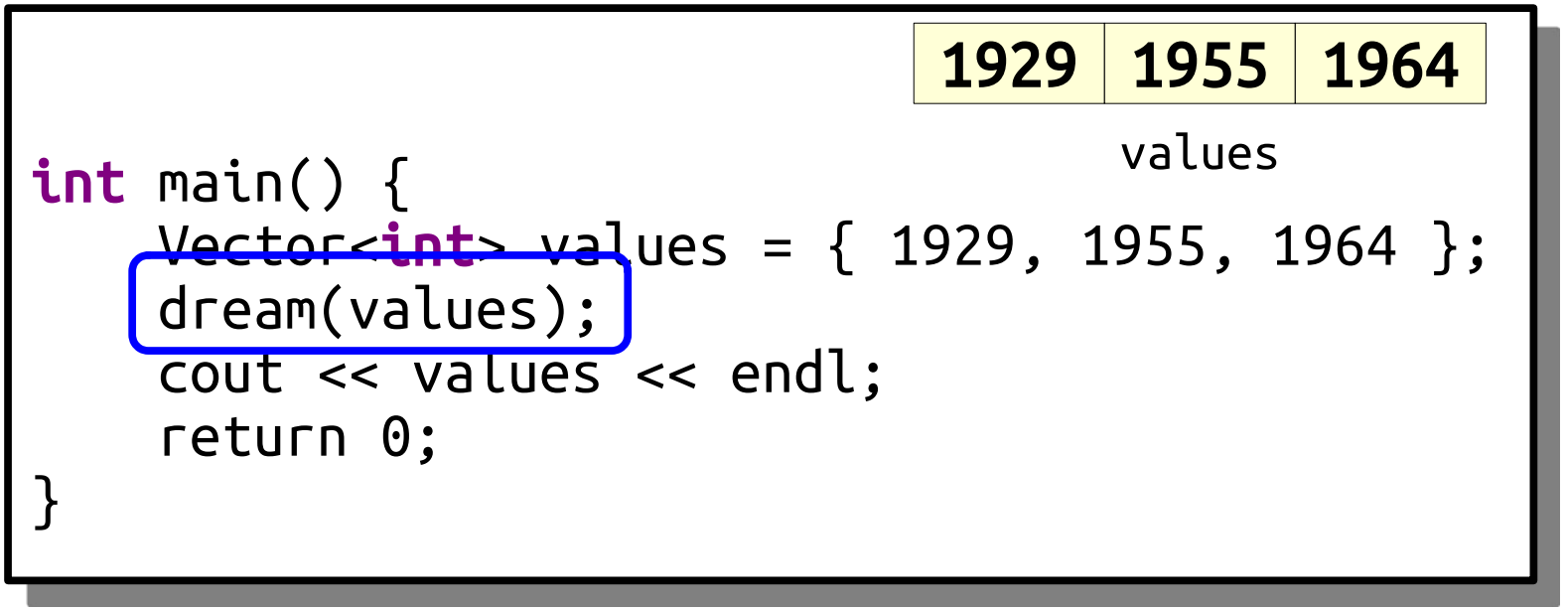
How it Works Now

```
int main() {
    Vector<int> values = { 1929, 1955, 1964 };
    dream(values);
    cout << values << endl;
    return 0;
}
```

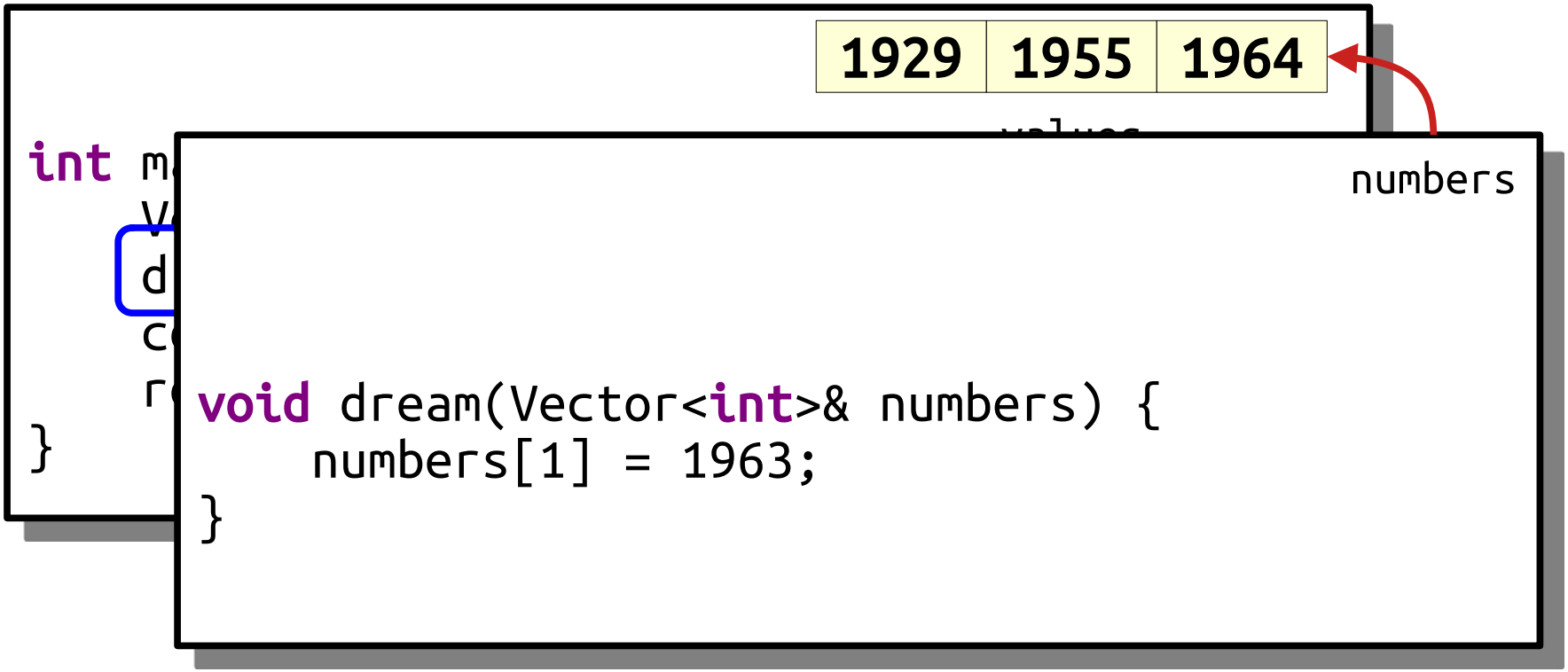
How it Works Now



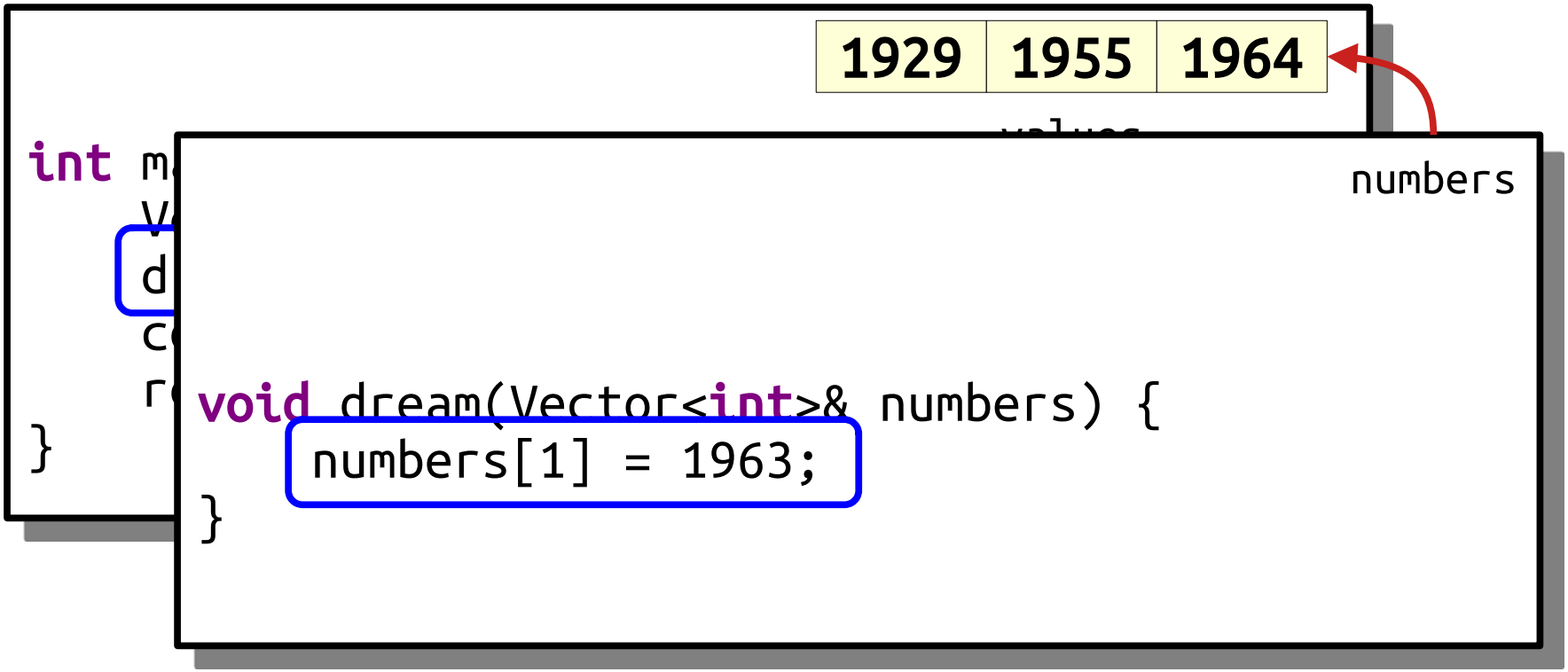
How it Works Now



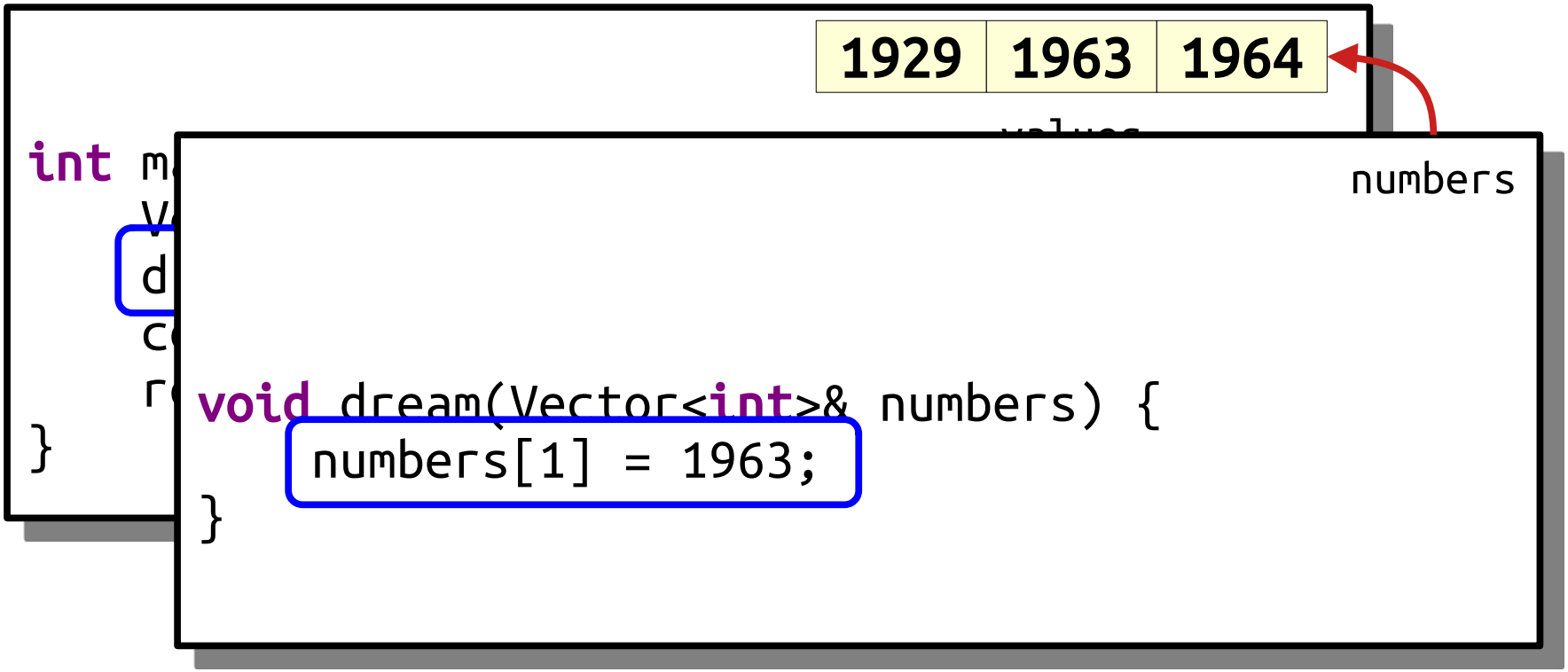
How it Works Now



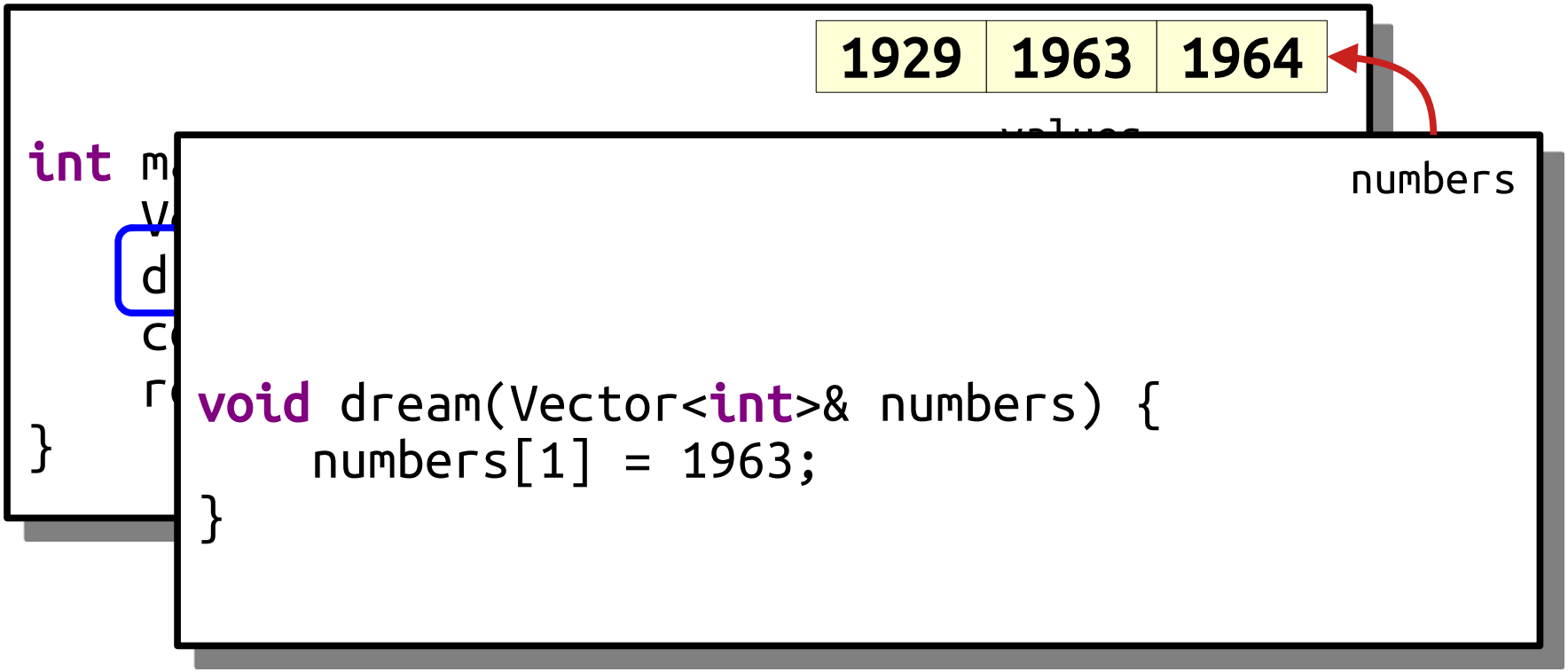
How it Works Now



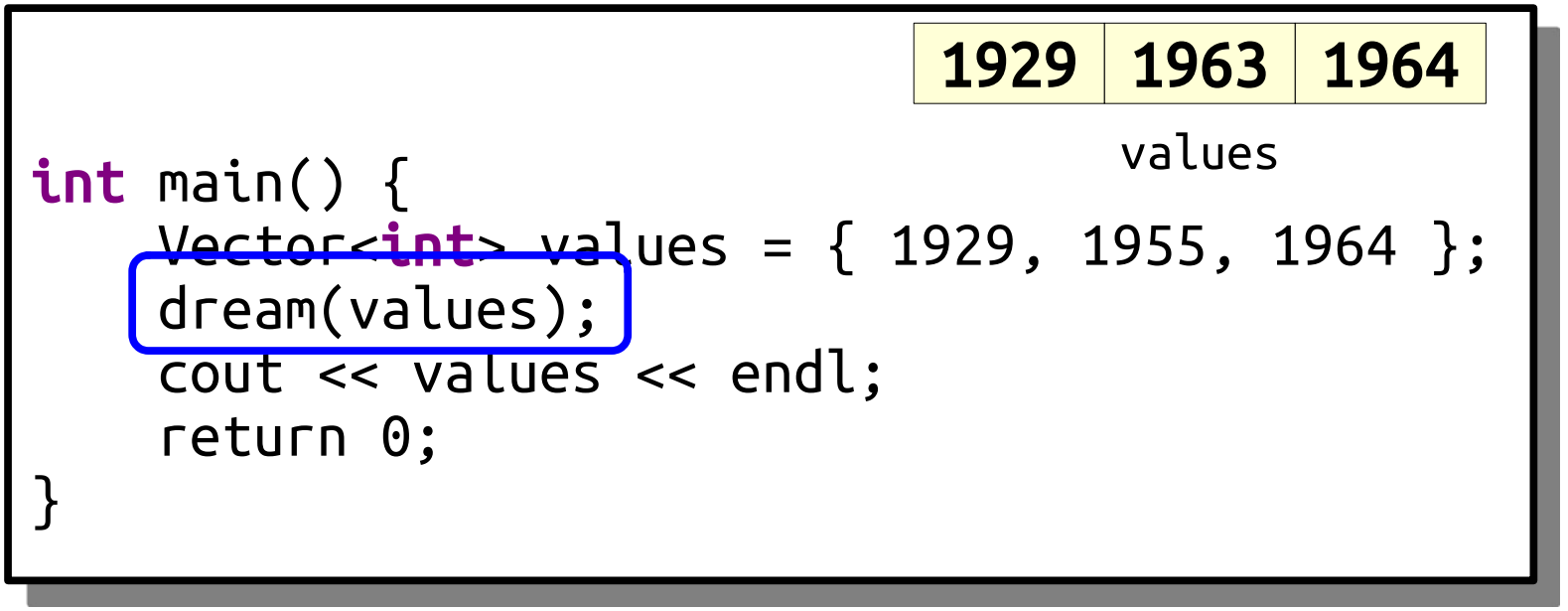
How it Works Now



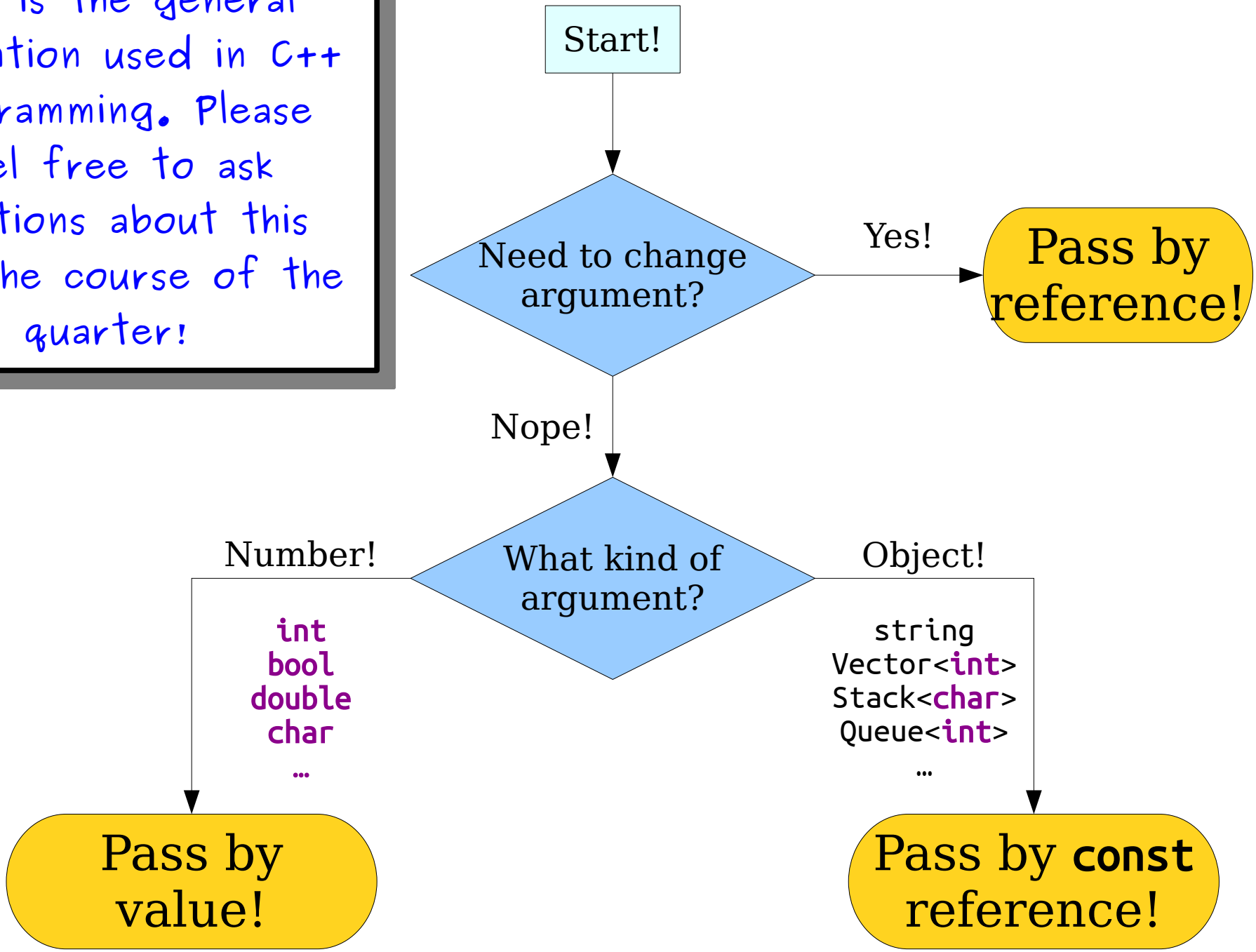
How it Works Now



How it Works Now



This is the general convention used in C++ programming. Please feel free to ask questions about this over the course of the quarter!



Recursion on `Vector`s

Finding the Largest Number

Finding the Largest Number

- Our goal is to write a function

```
int maxOf(const Vector<int>& numbers);
```

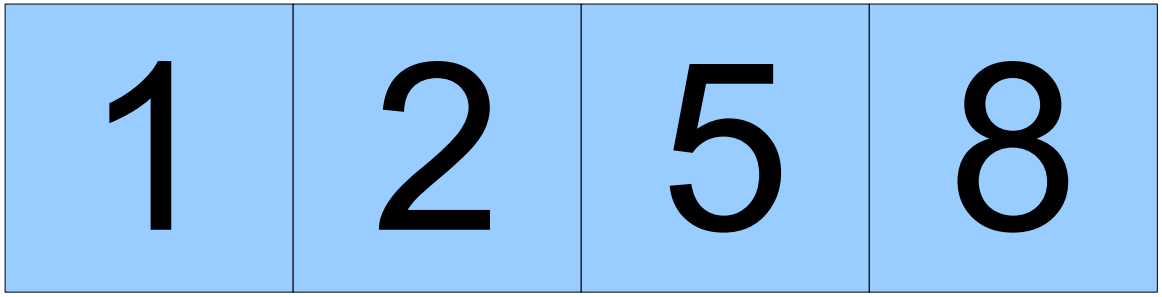
  that takes as input a `Vector<int>`, then returns the largest number in the `Vector`.
- We're going to assume the `Vector` has at least one element in it; otherwise, it's not possible to return the largest value!
- Let's see how to do this.

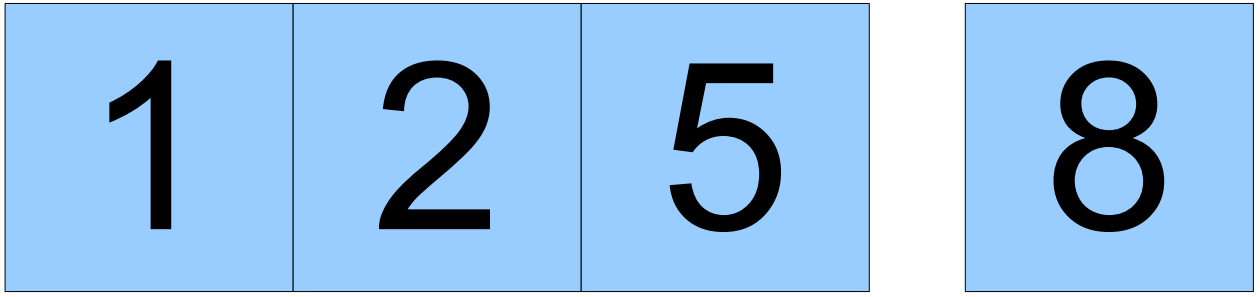
Thinking Recursively

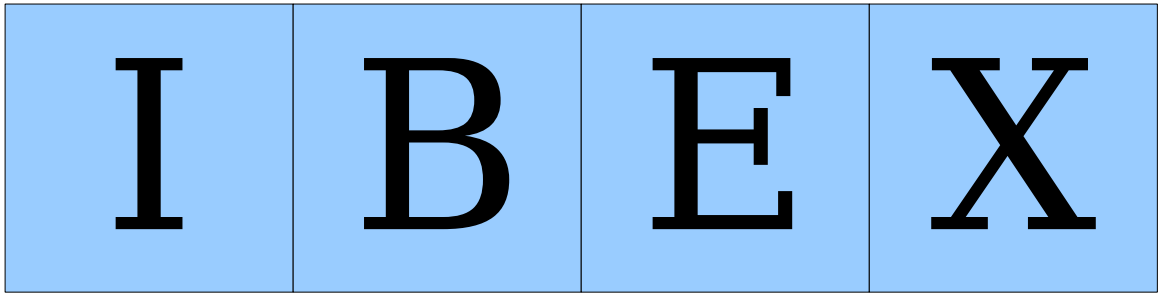
```
if (The problem is very simple) {
   Directly solve the problem.
   Return the solution.
} else {
   Split the problem into one or more
   smaller problems with the same
   structure as the original.
   Solve each of those smaller problems.
   Combine the results to get the overall
   solution.
   Return the overall solution.
}
```

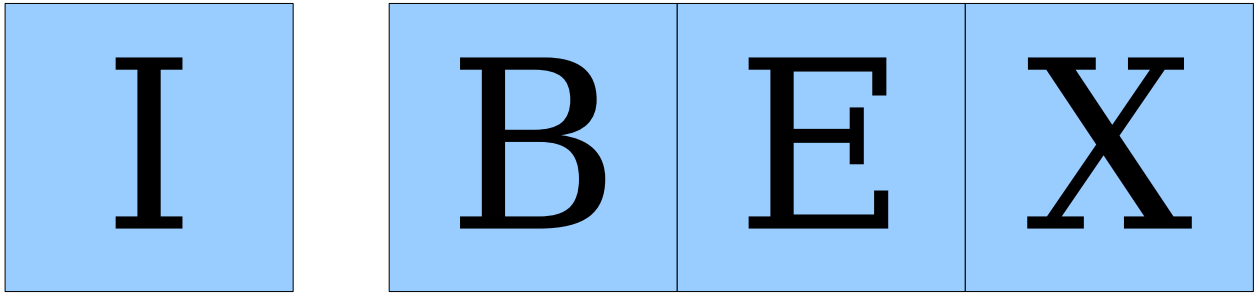
These simple cases are called *base cases*.

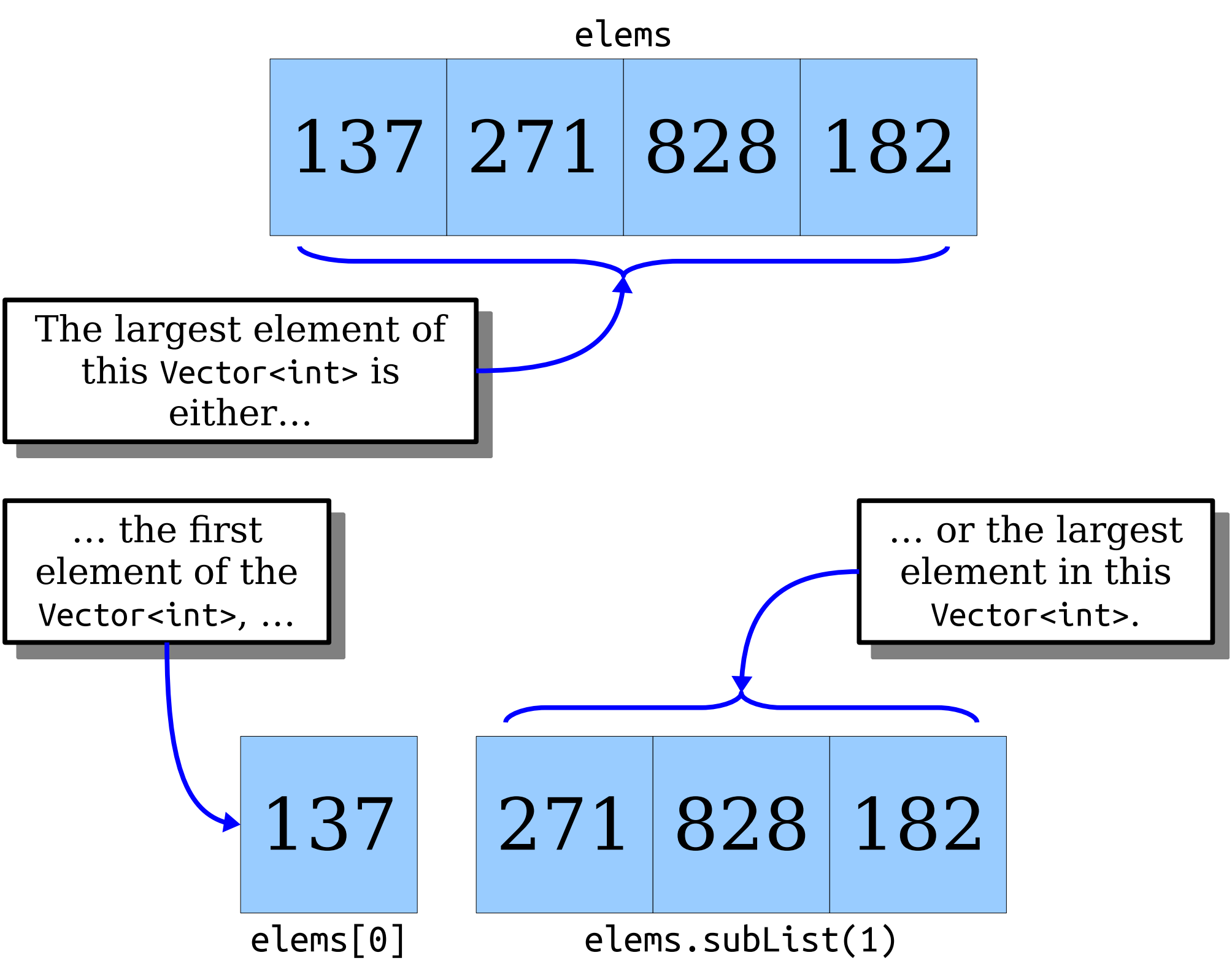
These are the *recursive cases*.


1
2
5
8


1
2
5
8


I B E X


I B E X


elems
137
271
828
182
The largest element of this Vector<int> is either...
... the first element of the Vector<int>, ...
... or the largest element in this Vector<int>.
137
271
828
182
elems[0]
elems.subList(1)

Tracing the Recursion

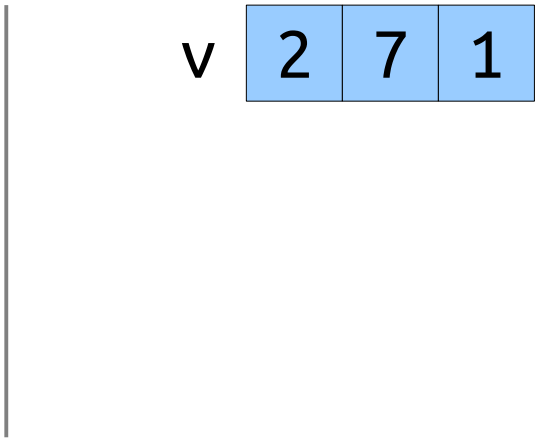
```
int main() {
    Vector<int> v = { 2, 7, 1 };
    cout << maxOf(v) << endl;
    return 0;
}
```

Tracing the Recursion

```
int main() {
    Vector<int> v = { 2, 7, 1 };
    cout << maxOf(v) << endl;
    return 0;
}
```

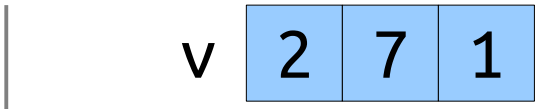
Tracing the Recursion

```
int main() {
    Vector<int> v = { 2, 7, 1 };
    cout << maxOf(v) << endl;
    return 0;
}
```



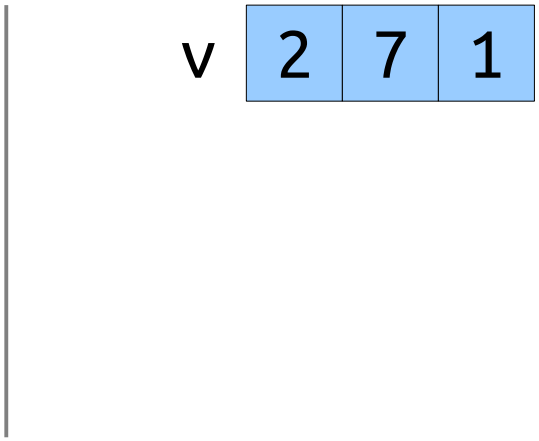
Tracing the Recursion

```
int main() {
    Vector<int> v = { 2, 7, 1 };
    cout << maxOf(v) << endl;
    return 0;
}
```



Tracing the Recursion

```
int main() {
    Vector<int> v = { 2, 7, 1 };
    cout << maxOf(v) << endl;
    return 0;
}
```



Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int first = elems[0];
    Vector<int> rest = elems.subList(1);
    return max(first, maxOf(rest));
  }
}
```

Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int first = elems[0];
    Vector<int> rest = elems.subList(1);
    return max(first, maxOf(rest));
  }
}
```

Tracing the Recursion

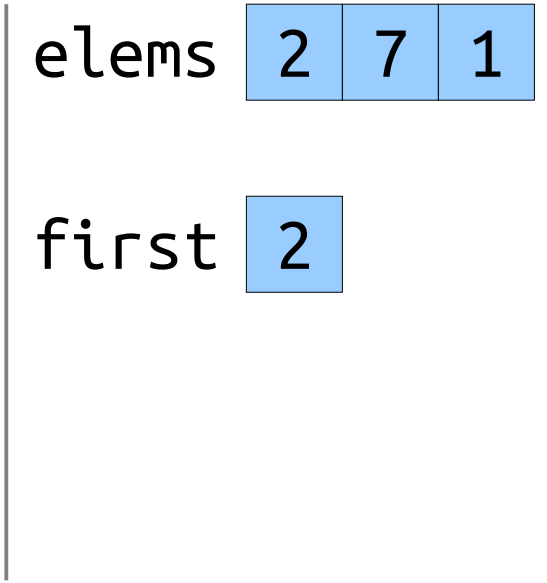
```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int first = elems[0];
    Vector<int> rest = elems.subList(1);
    return max(first, maxOf(rest));
  }
}
```

Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int first = elems[0];
    Vector<int> rest = elems.subList(1);
    return max(first, maxOf(rest));
  }
}
```

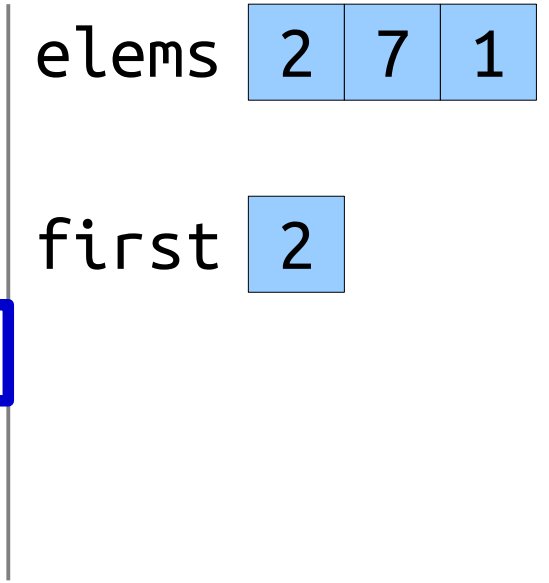
Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int first = elems[0];
    Vector<int> rest = elems.subList(1);
    return max(first, maxOf(rest));
  }
}
```



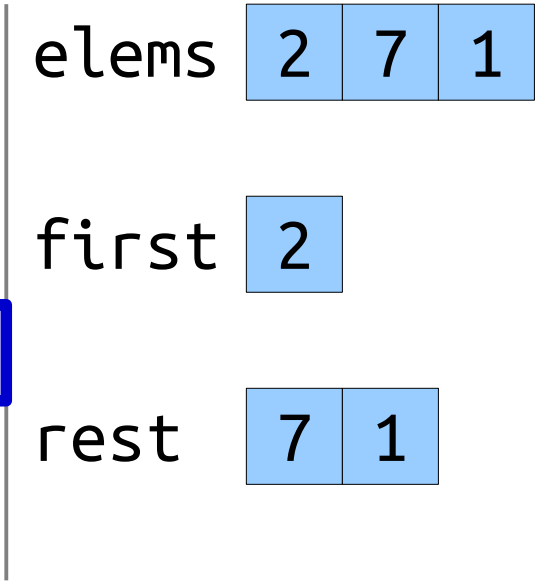
Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int first = elems[0];
    Vector<int> rest = elems.subList(1);
    return max(first, maxOf(rest));
  }
}
```



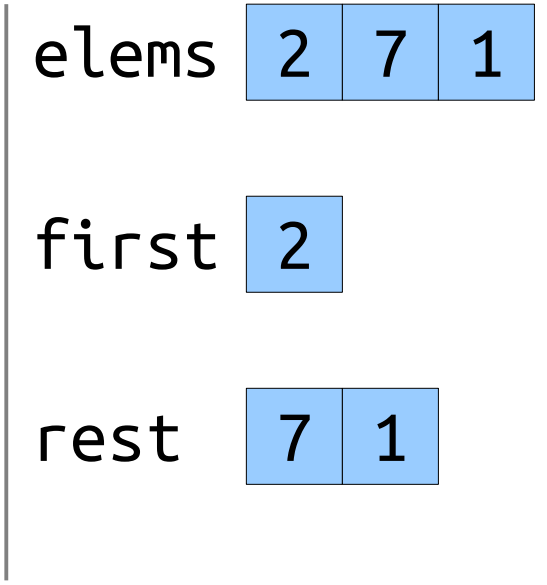
Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int first = elems[0];
    Vector<int> rest = elems.subList(1);
    return max(first, maxOf(rest));
  }
}
```



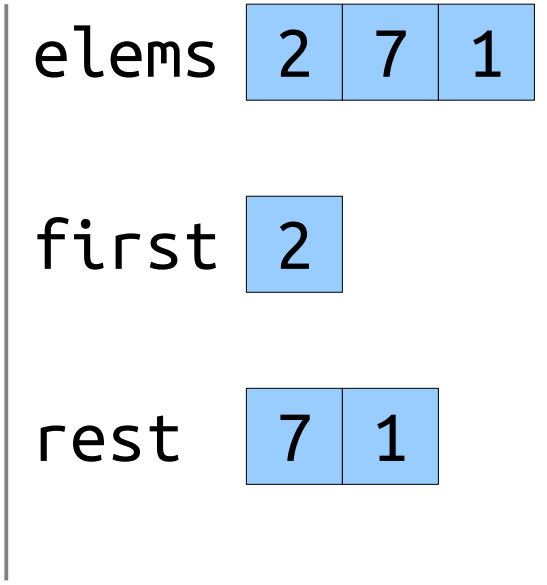
Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int first = elems[0];
    Vector<int> rest = elems.subList(1);
    return max(first, maxOf(rest));
  }
}
```



Tracing the Recursion

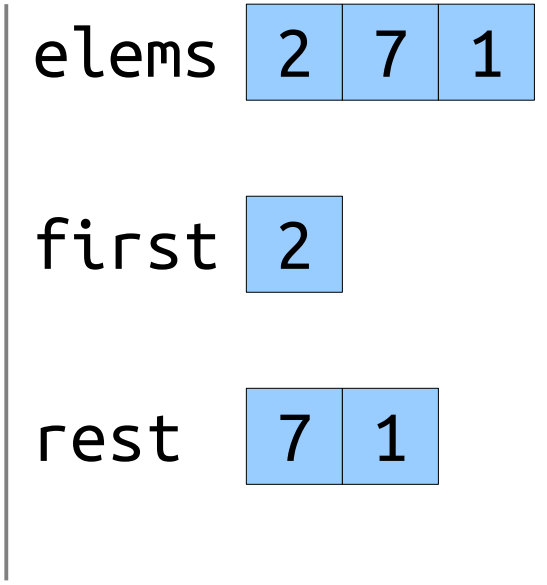
```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int first = elems[0];
    Vector<int> rest = elems.subList(1);
    return max(first, maxOf(rest));
  }
}
```



Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int first = elems[0];
    Vector<int> rest = elems.subList(1);
    return max(first, maxOf(rest));
  }
}
```

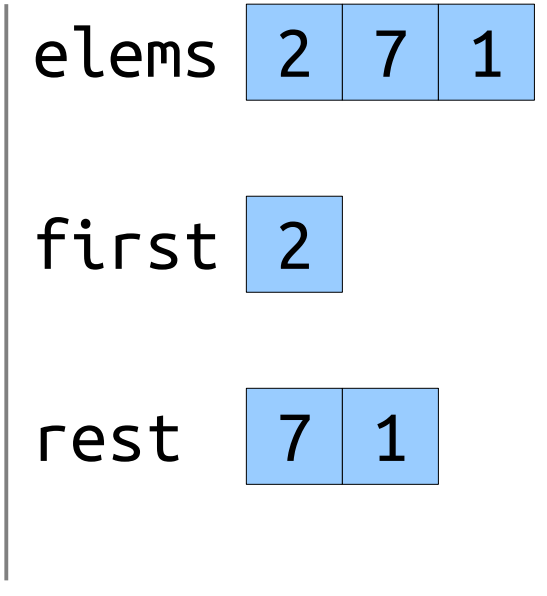
2



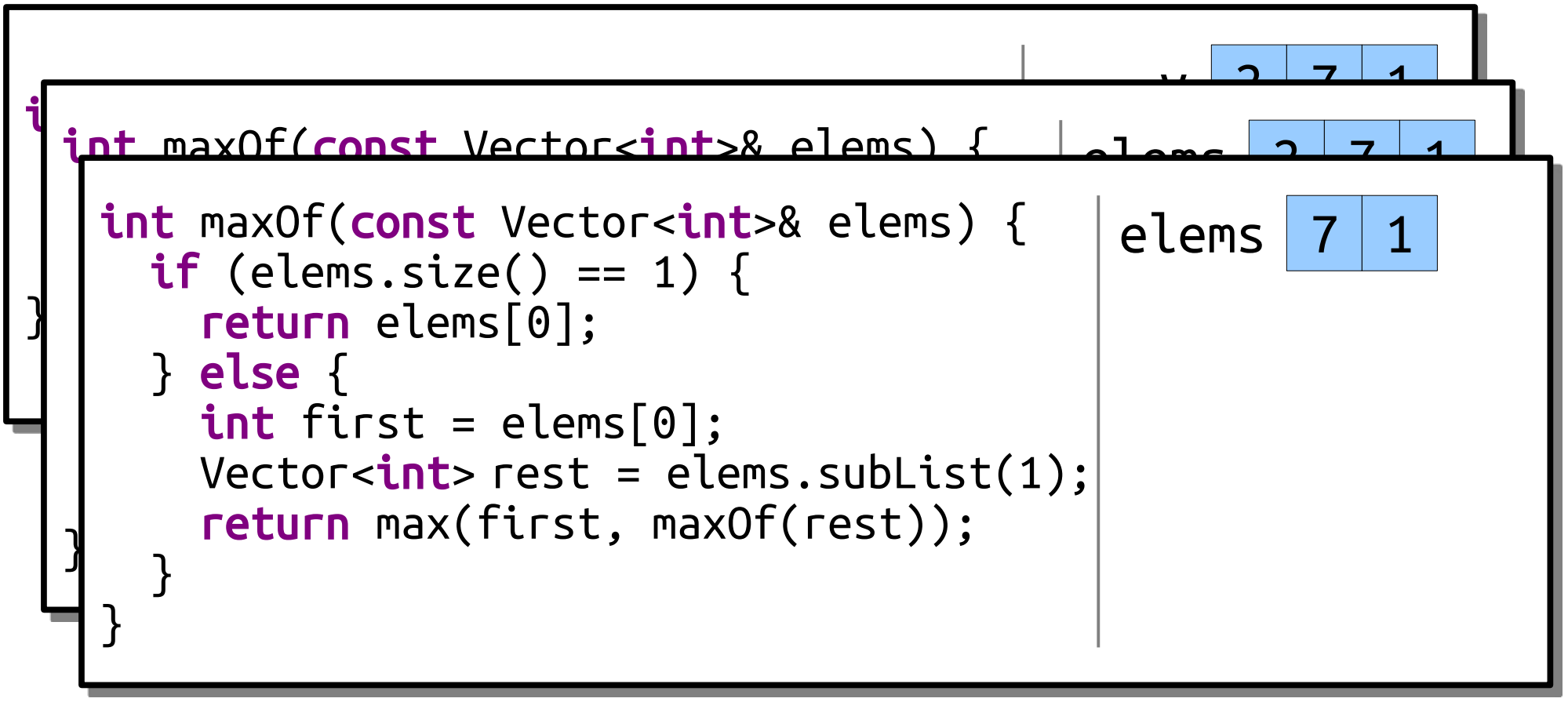
Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int first = elems[0];
    Vector<int> rest = elems.subList(1);
    return max(first, maxOf(rest));
  }
}
```

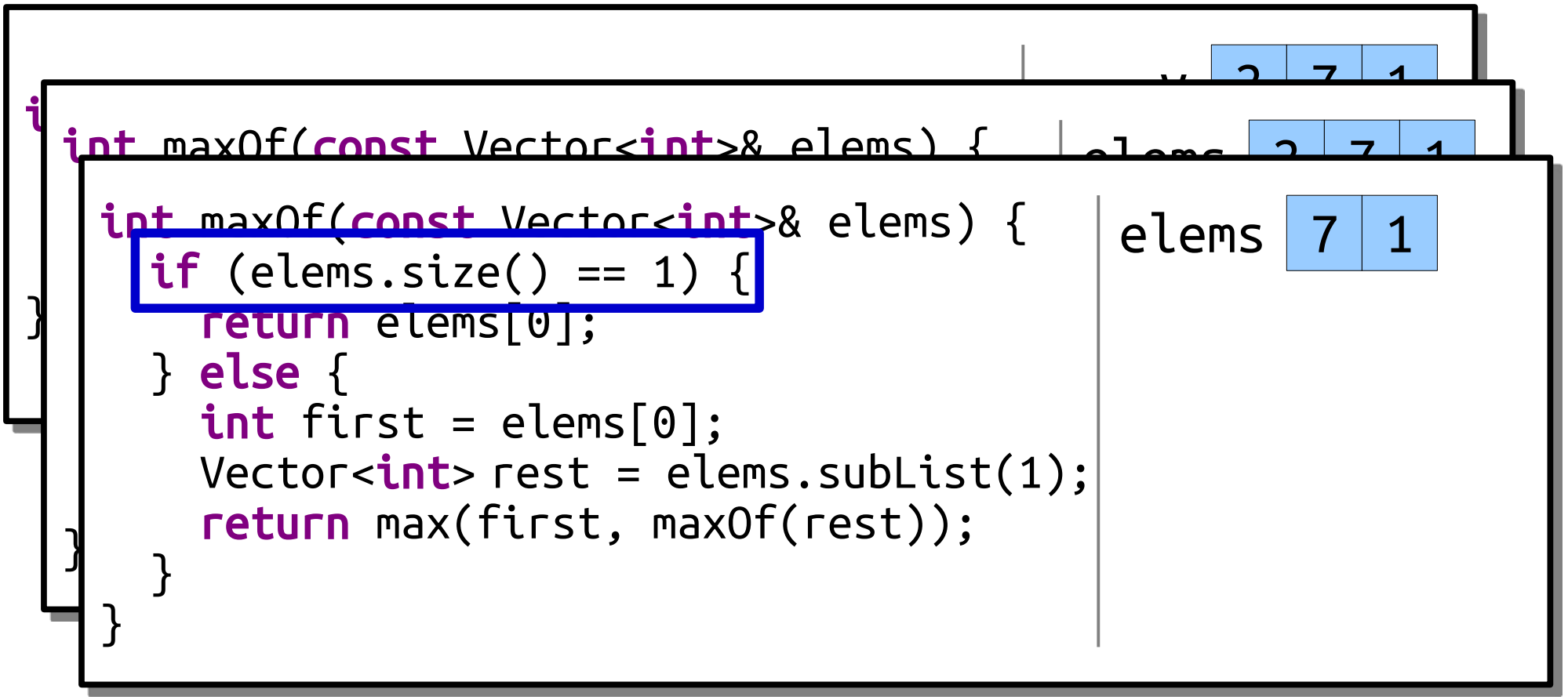
2



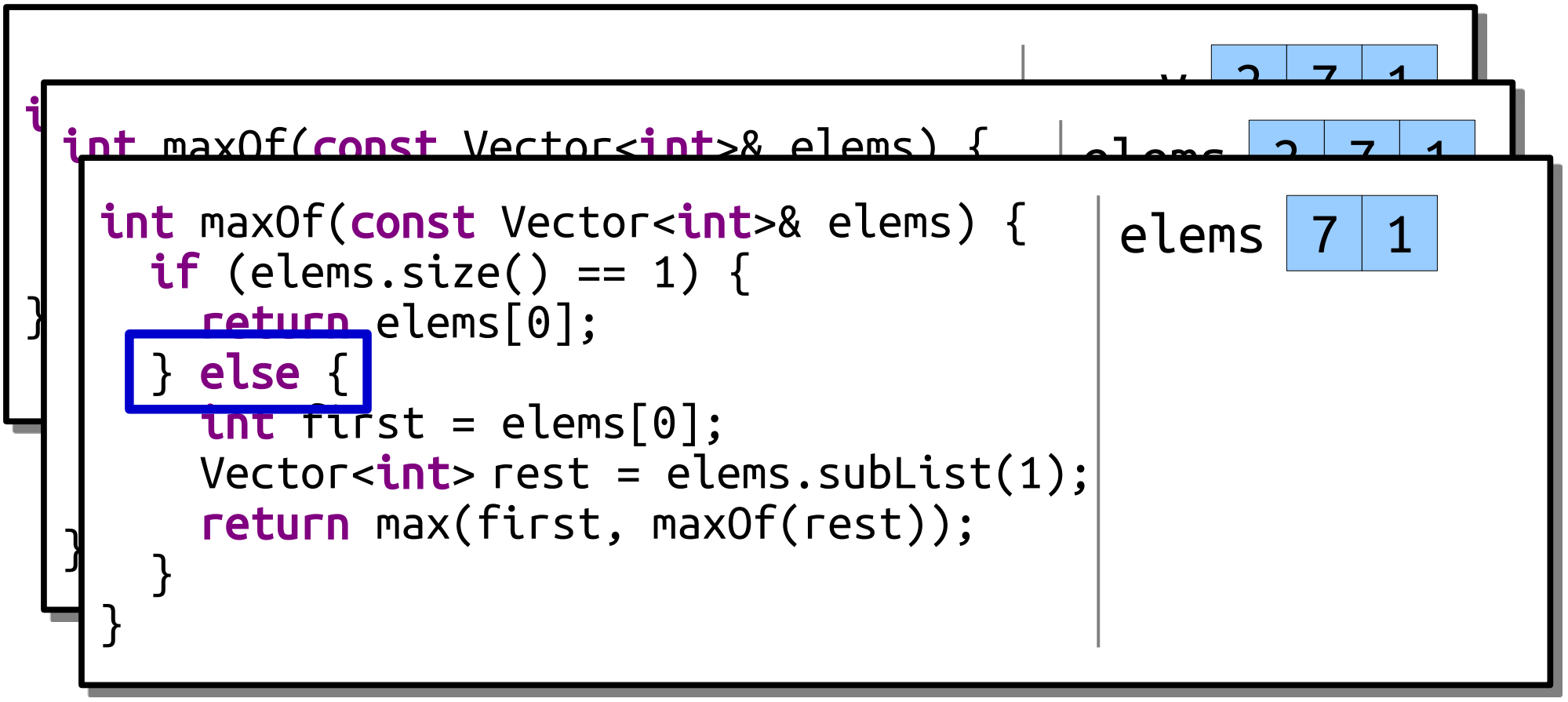
Tracing the Recursion



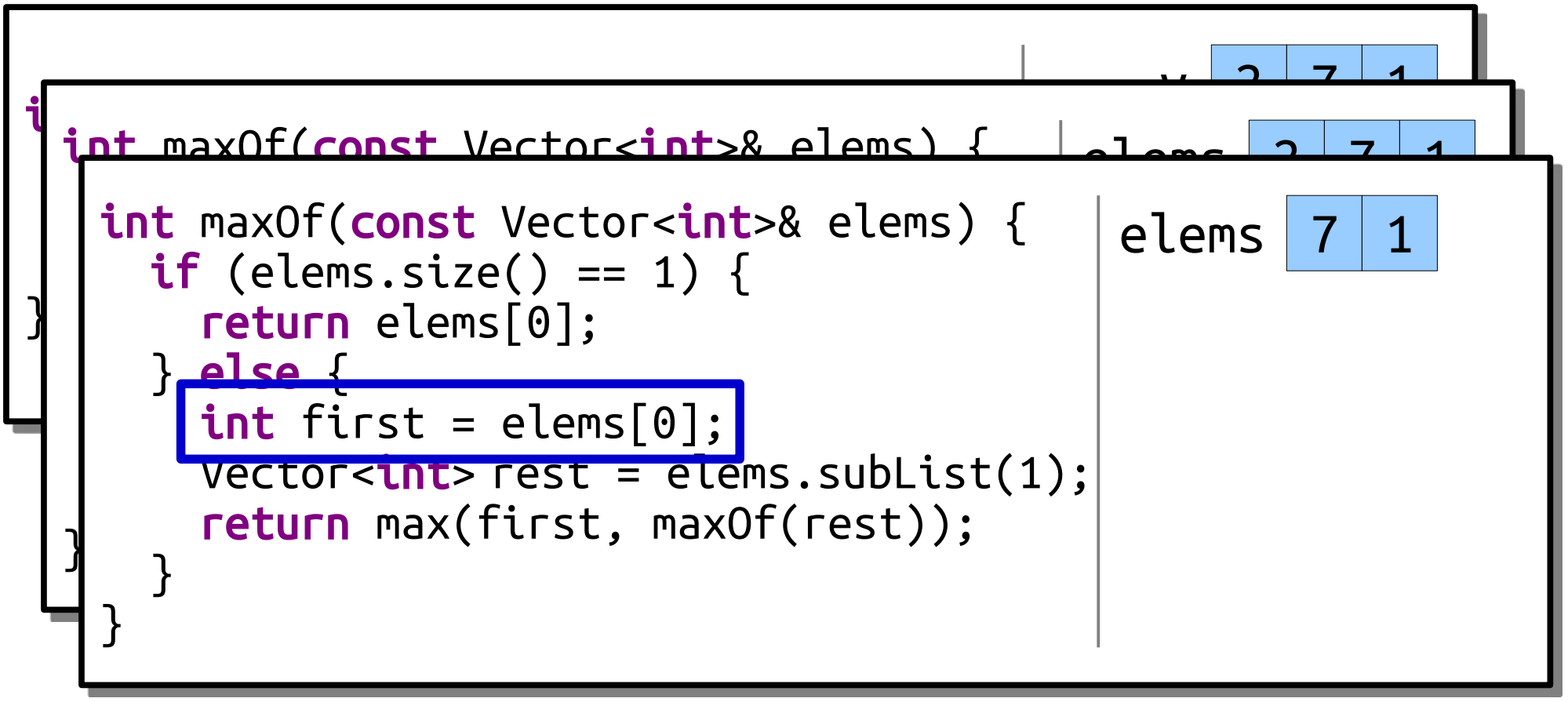
Tracing the Recursion



Tracing the Recursion



Tracing the Recursion



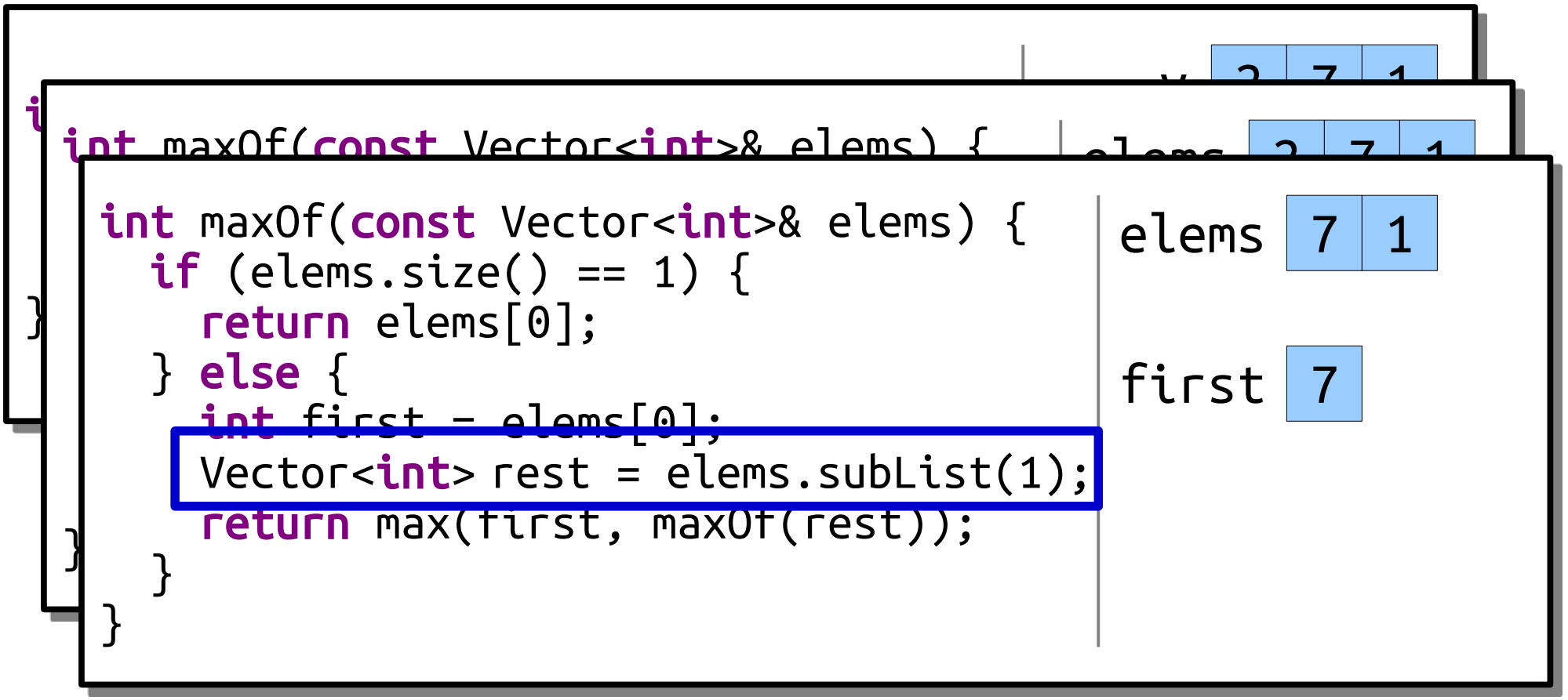
Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int first = elems[0];
    Vector<int> rest = elems.subList(1);
    return max(first, maxOf(rest));
  }
}
```

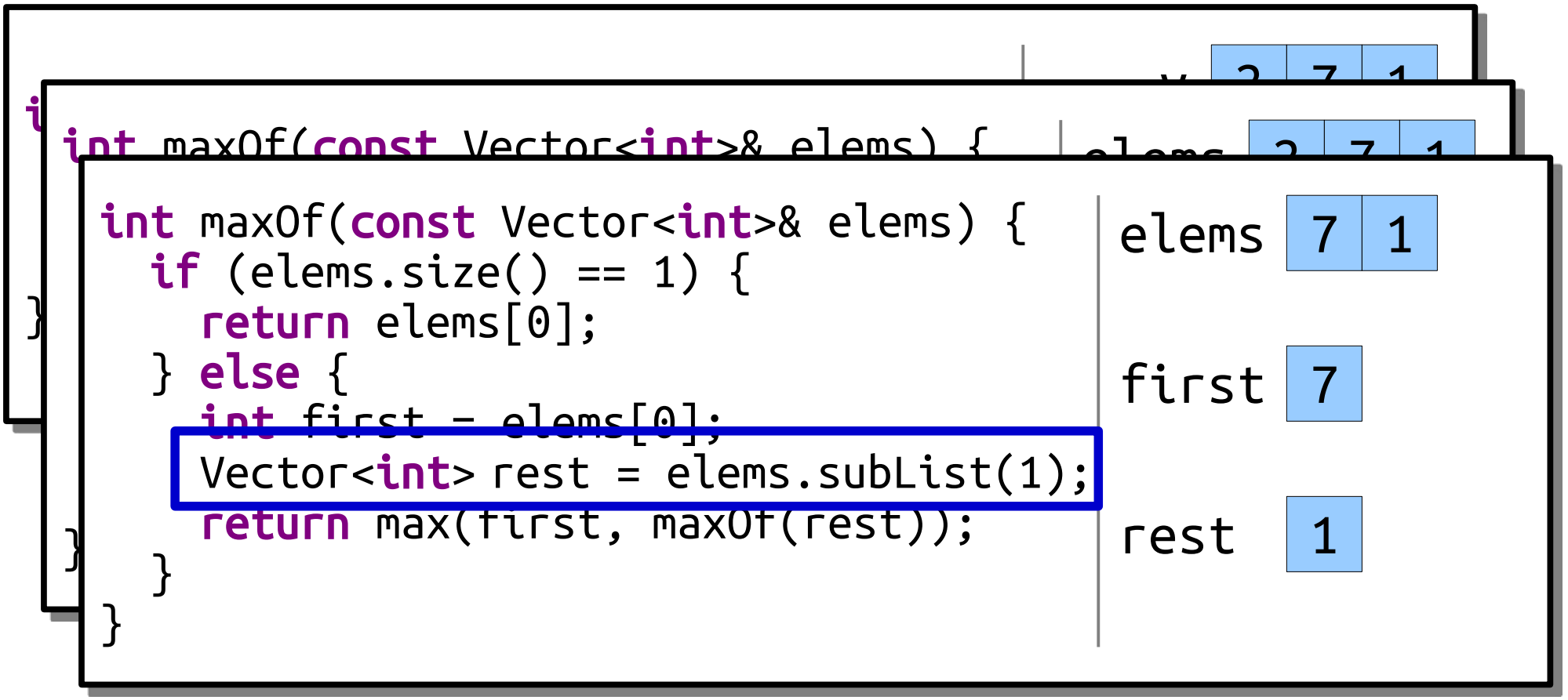
elems	7	1

first	7

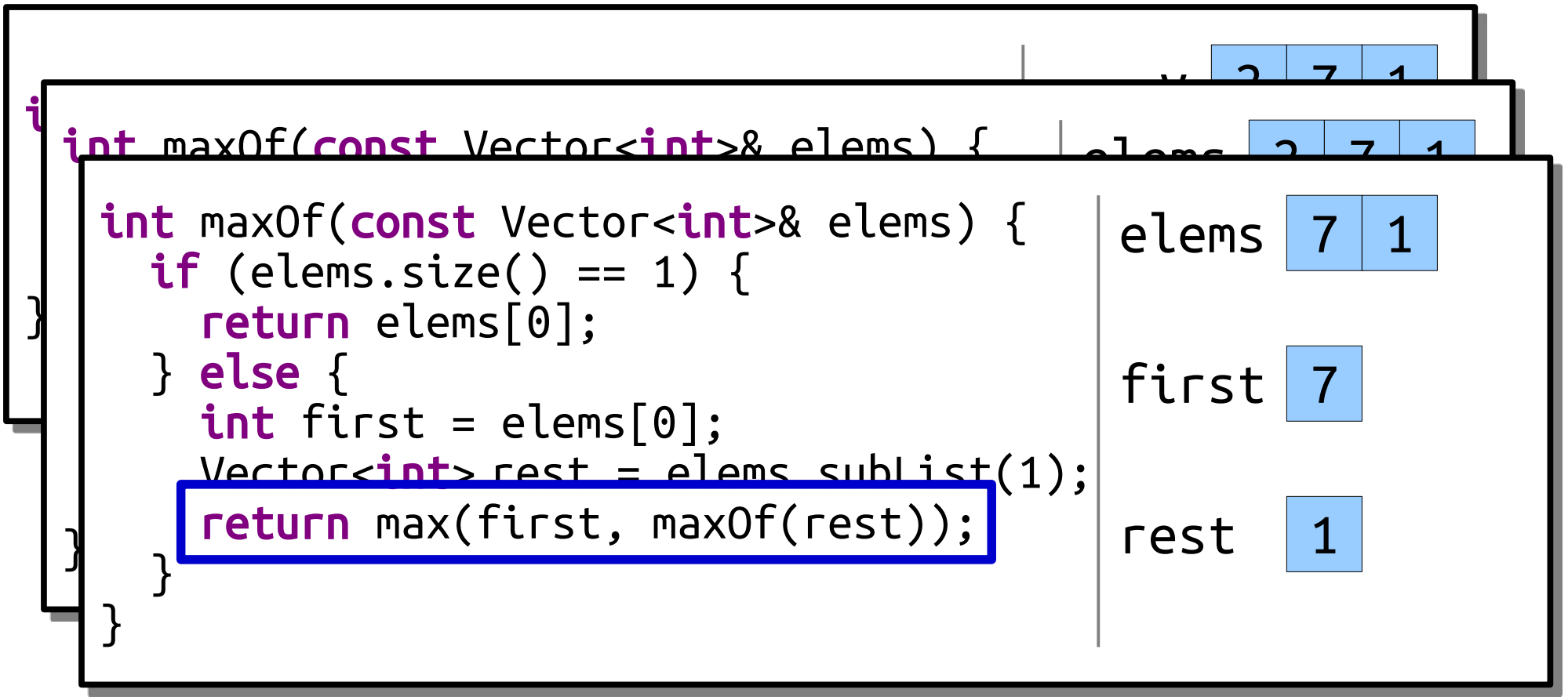
Tracing the Recursion



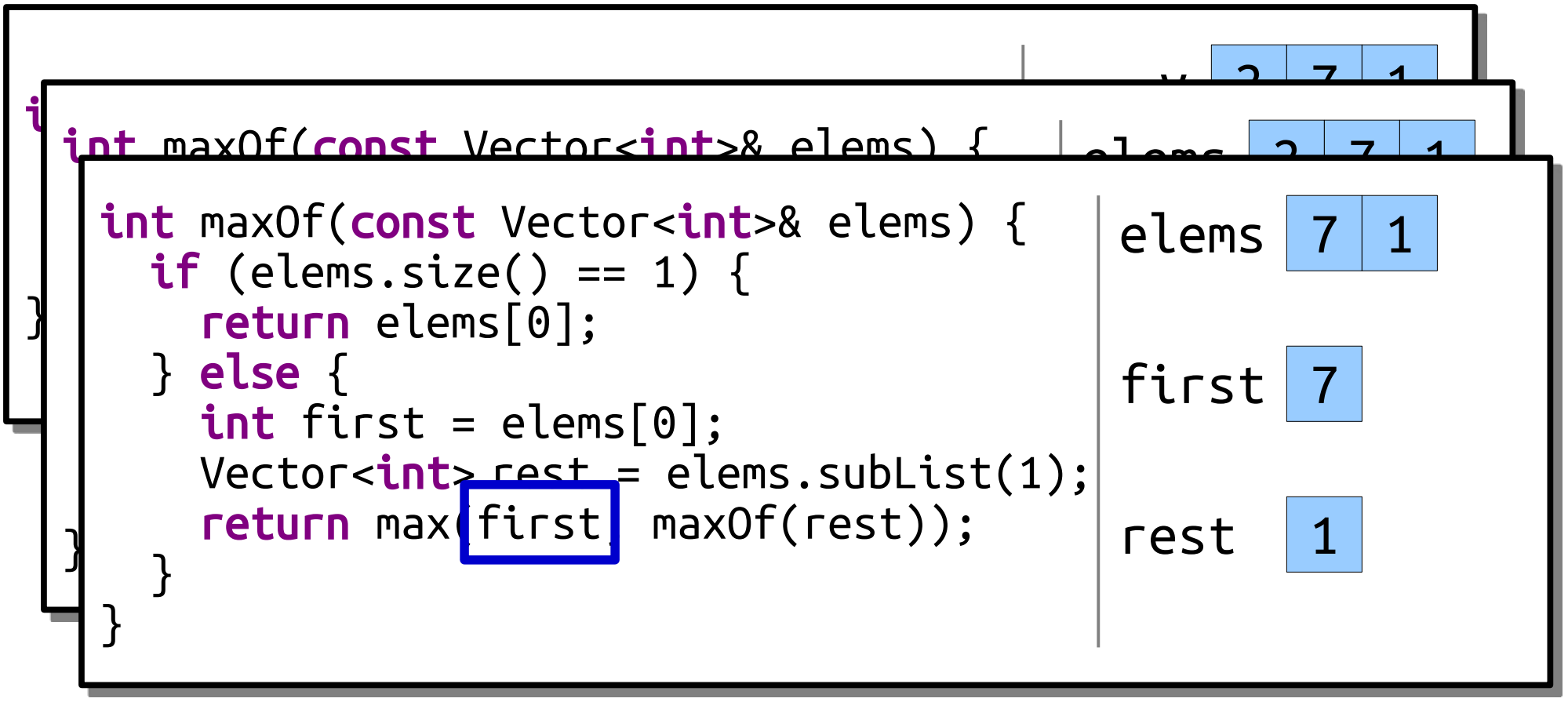
Tracing the Recursion



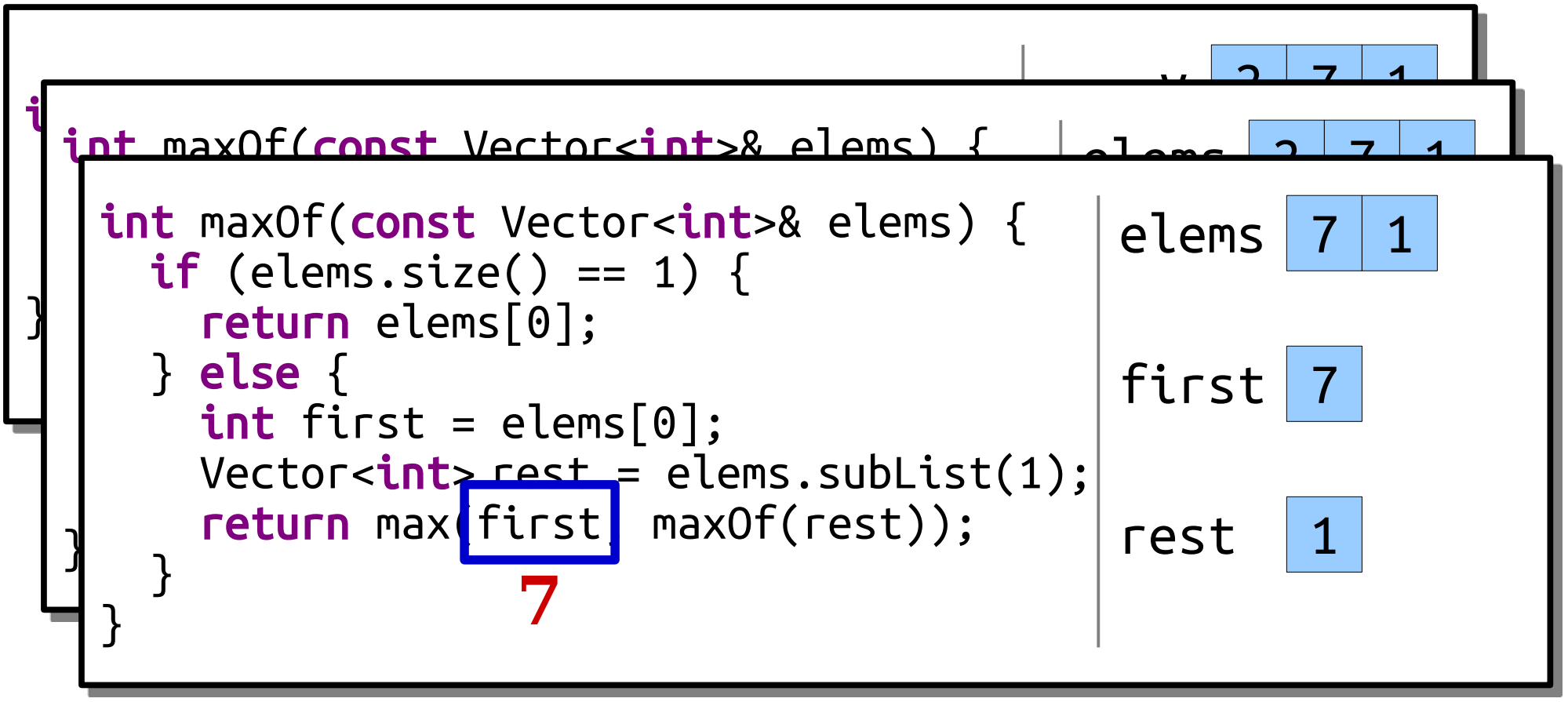
Tracing the Recursion



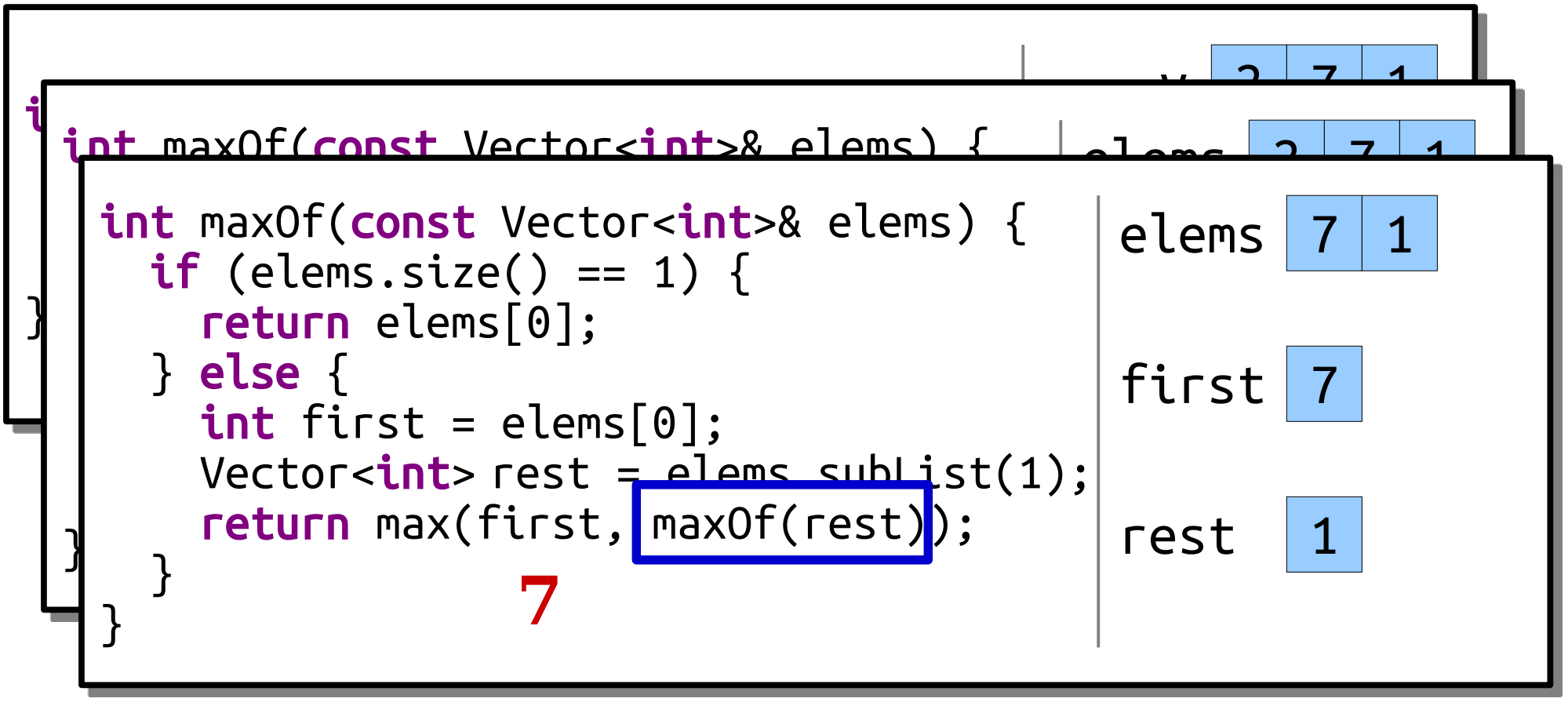
Tracing the Recursion



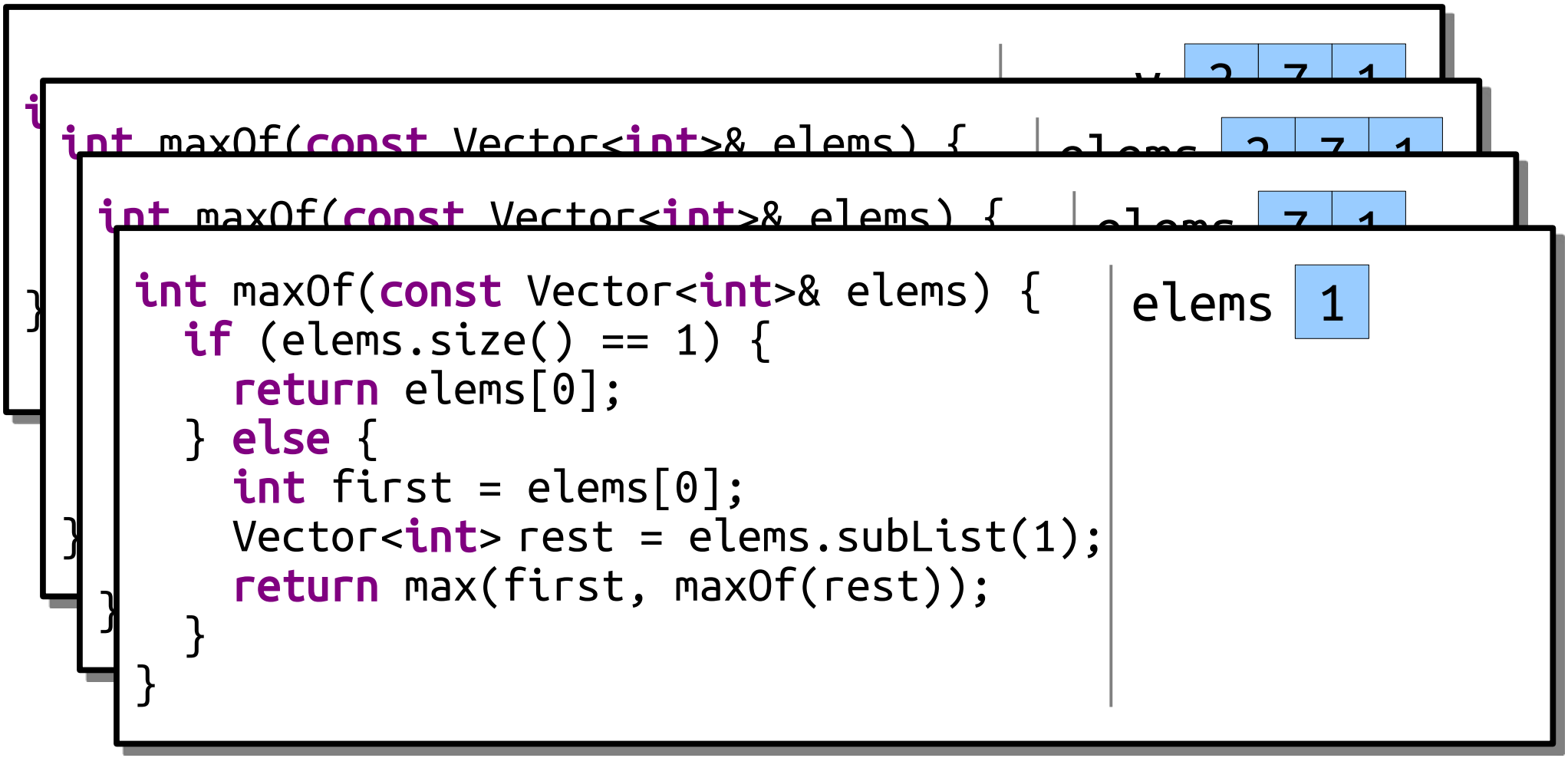
Tracing the Recursion



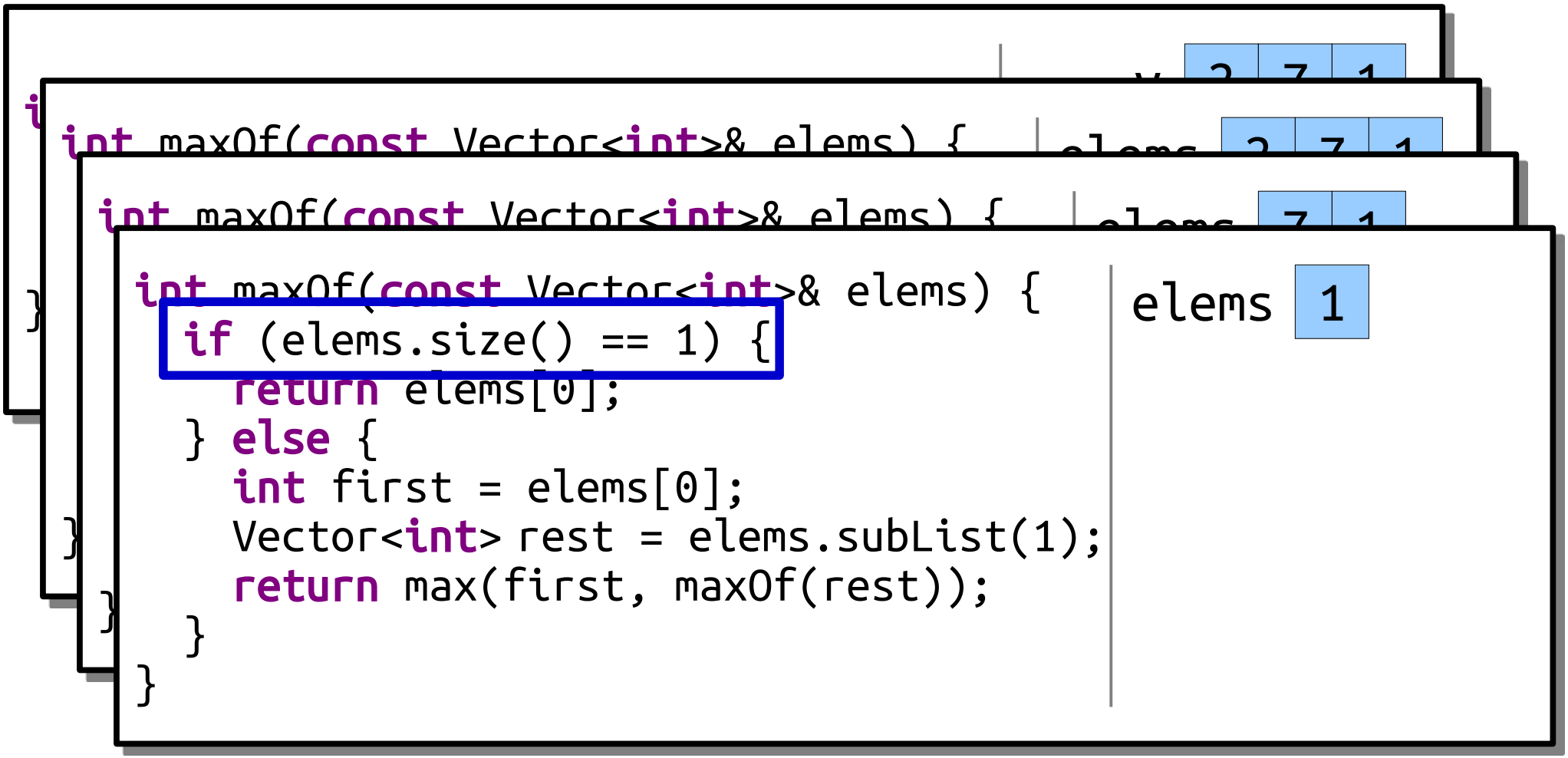
Tracing the Recursion



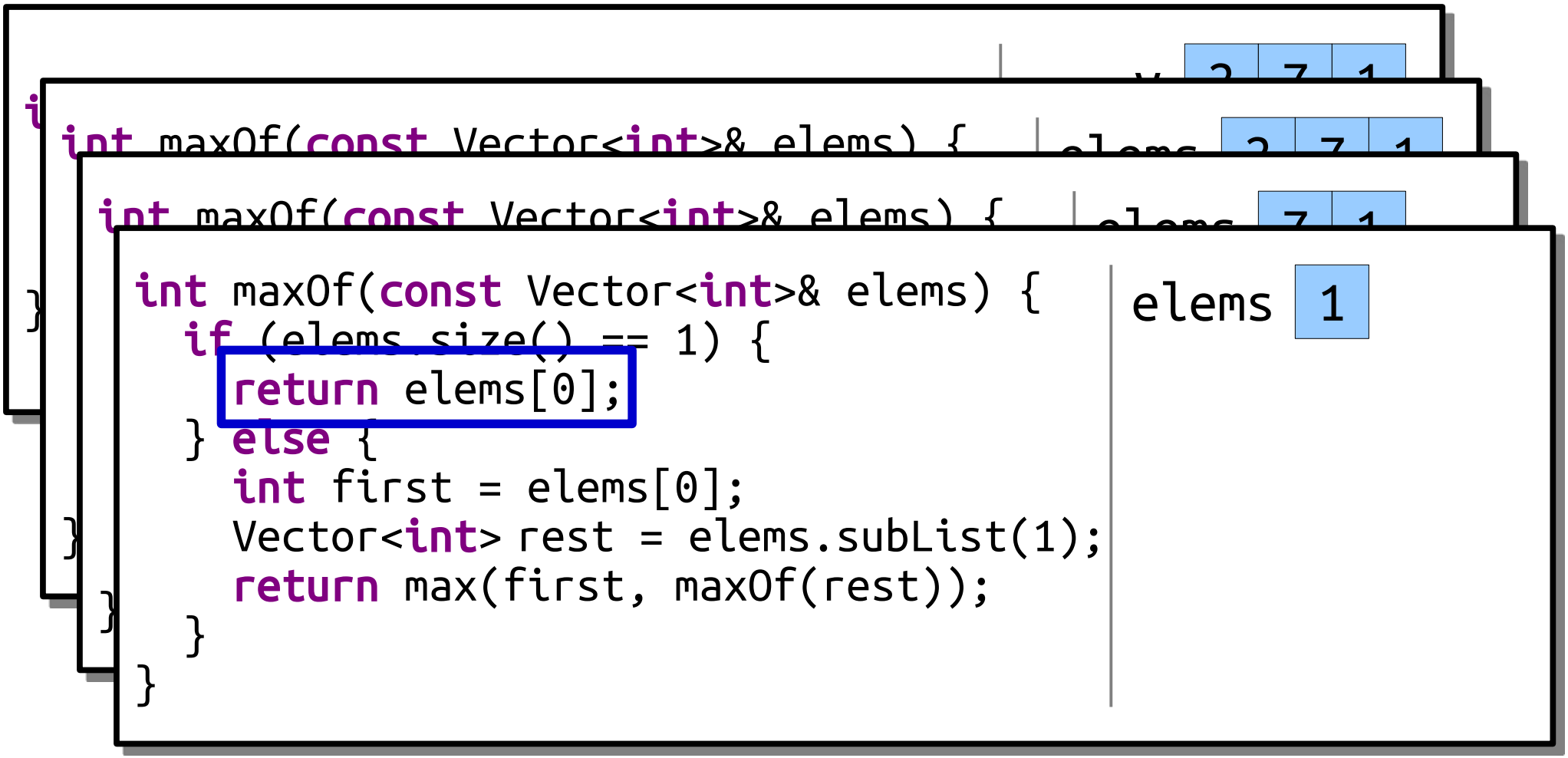
Tracing the Recursion



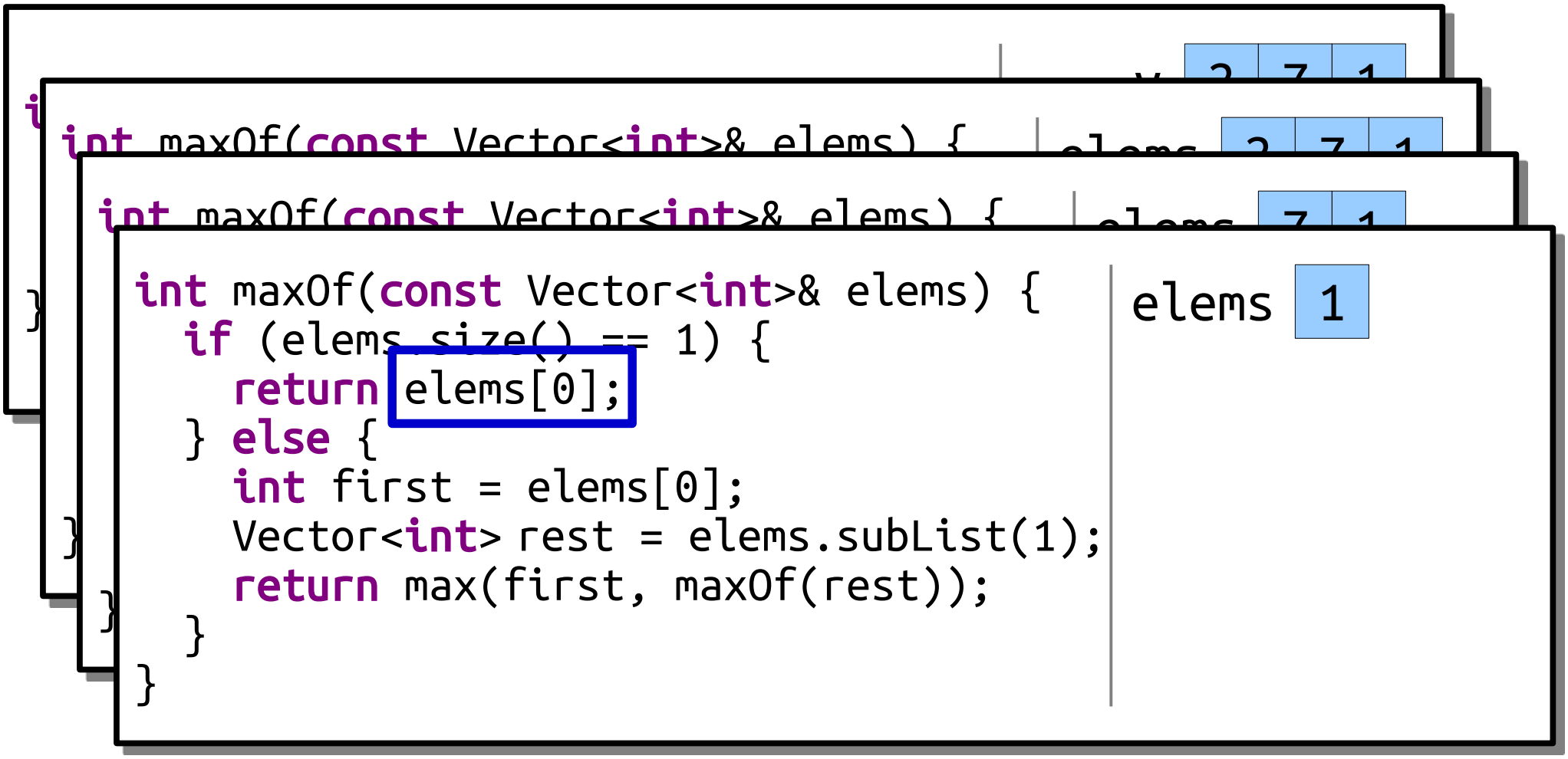
Tracing the Recursion



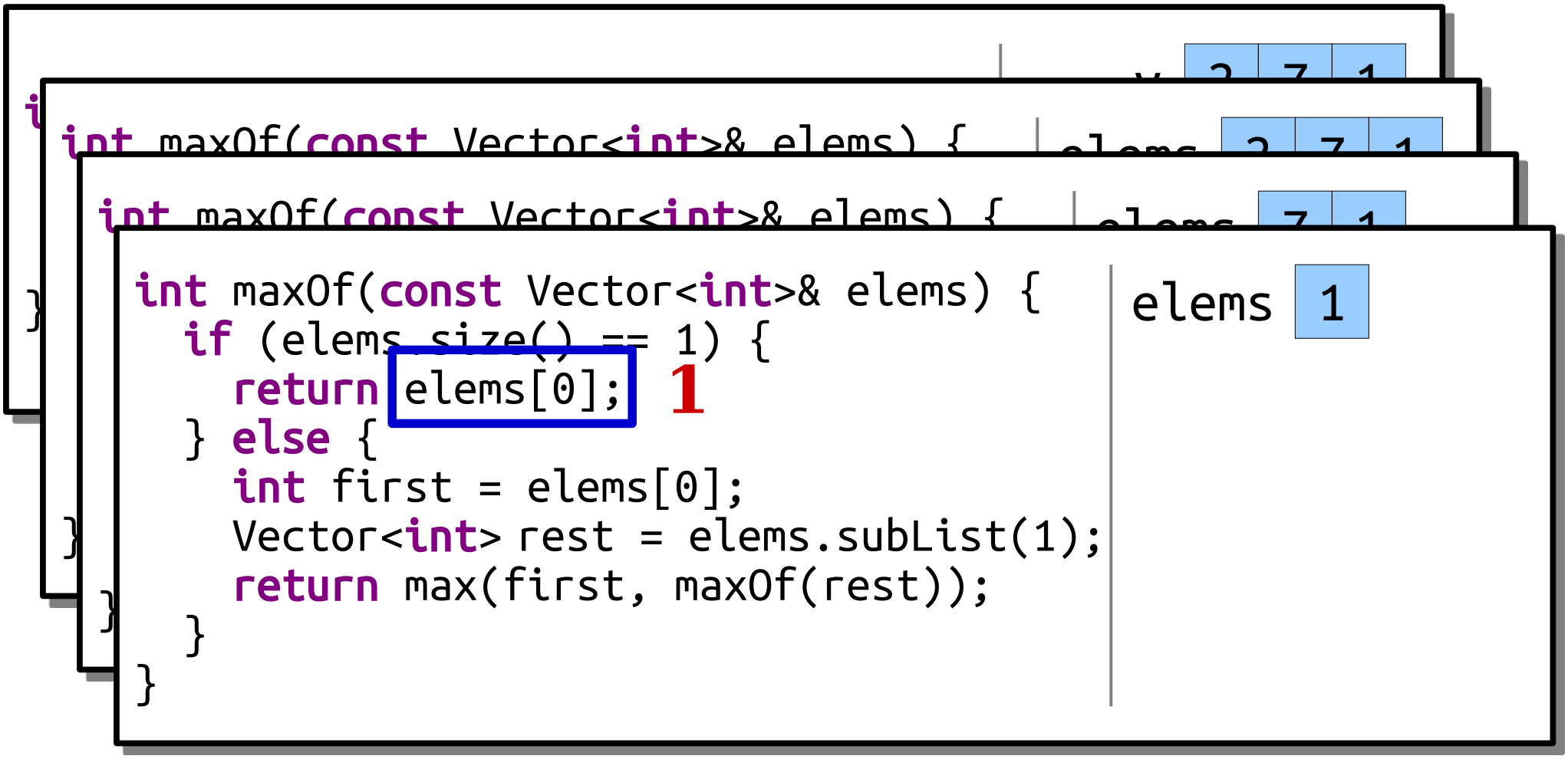
Tracing the Recursion



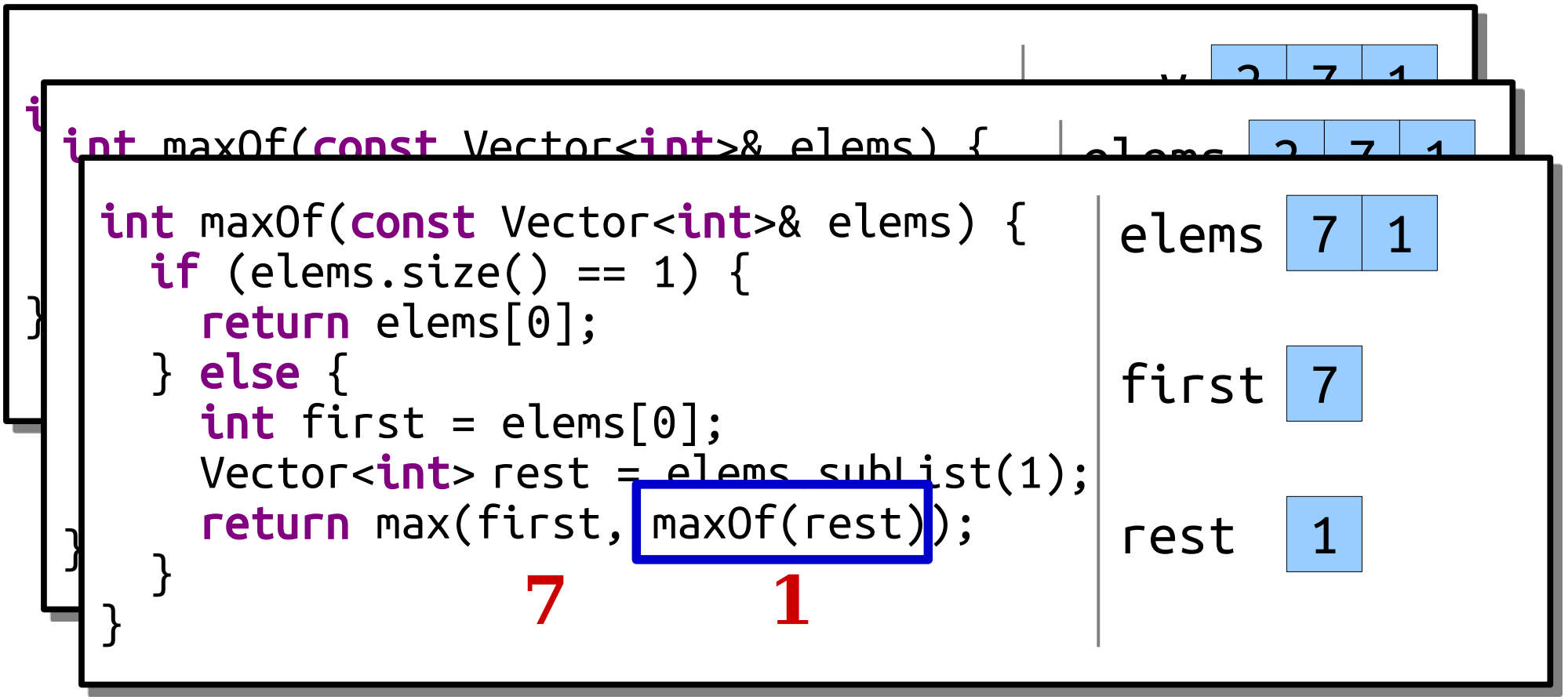
Tracing the Recursion



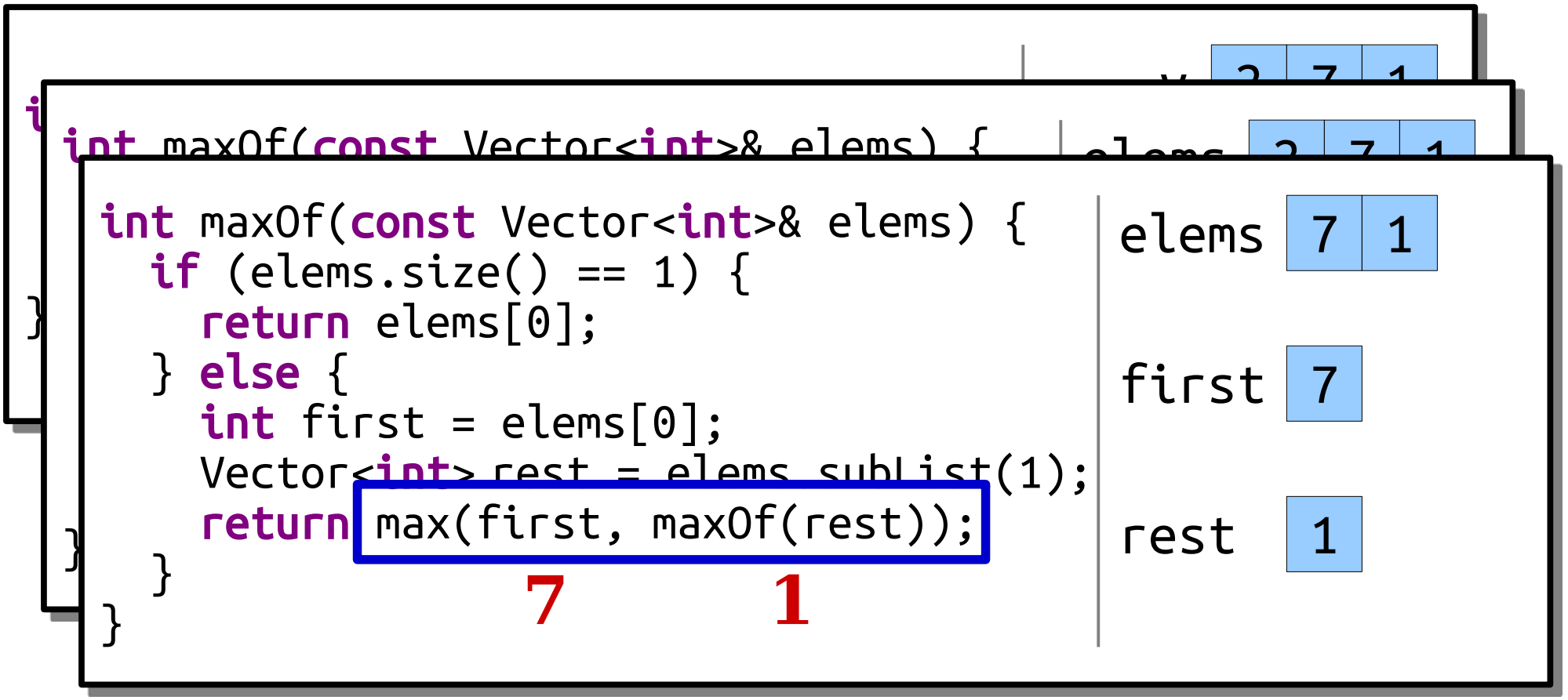
Tracing the Recursion



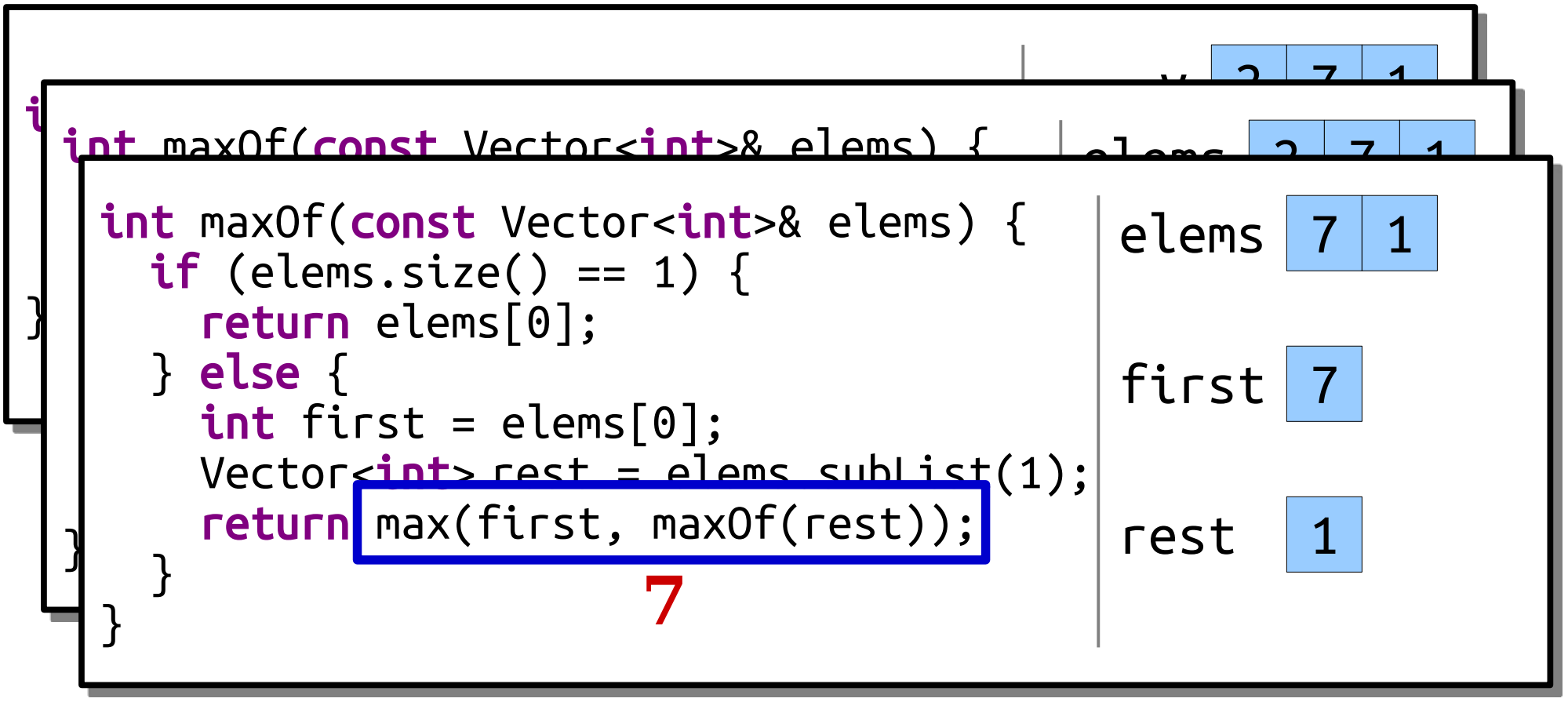
Tracing the Recursion



Tracing the Recursion



Tracing the Recursion



Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int first = elems[0];
    Vector<int> rest = elems.subList(1);
    return max(first, maxOf(rest));
  }
}
```

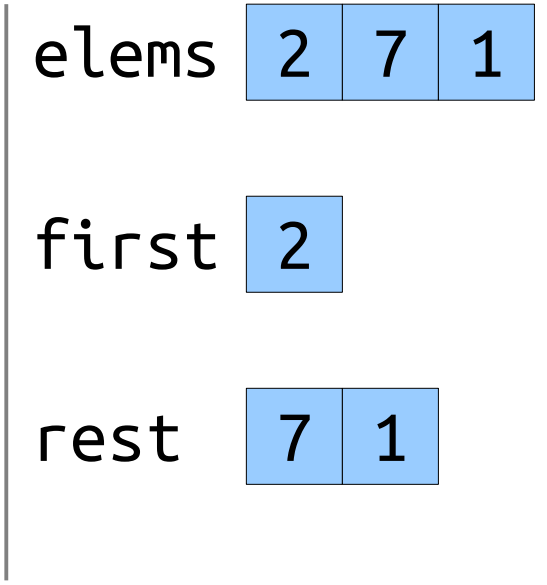
7

elems	7	1
first	7	
rest	1	

Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int first = elems[0];
    Vector<int> rest = elems.subList(1);
    return max(first, maxOf(rest));
  }
}
```

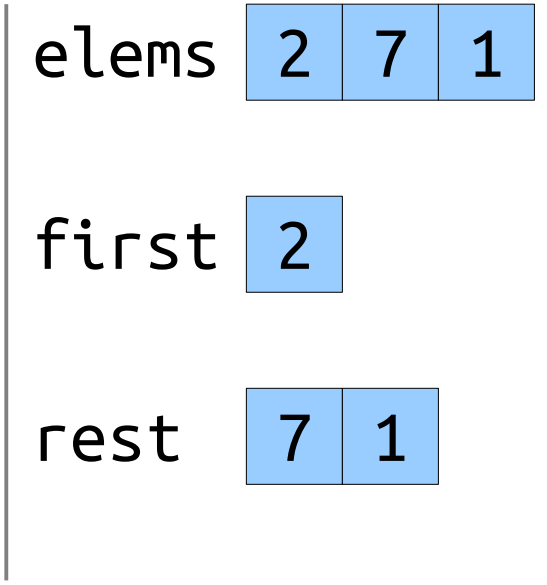
2 7



Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int first = elems[0];
    Vector<int> rest = elems.subList(1);
    return max(first, maxOf(rest));
  }
}
```

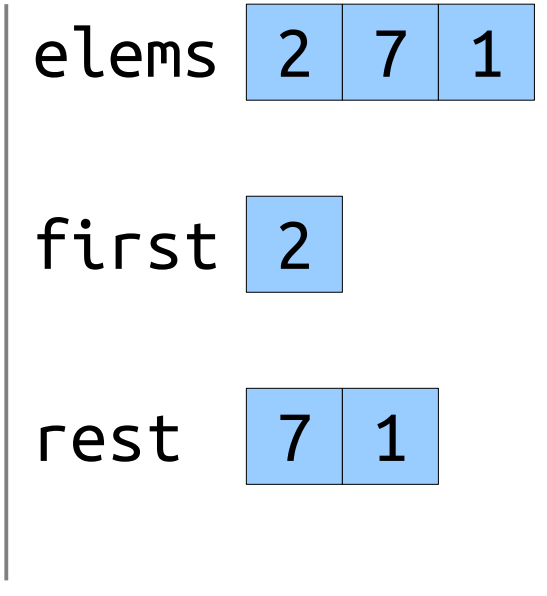
2 7



Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int first = elems[0];
    Vector<int> rest = elems.subList(1);
    return max(first, maxOf(rest));
  }
}
```

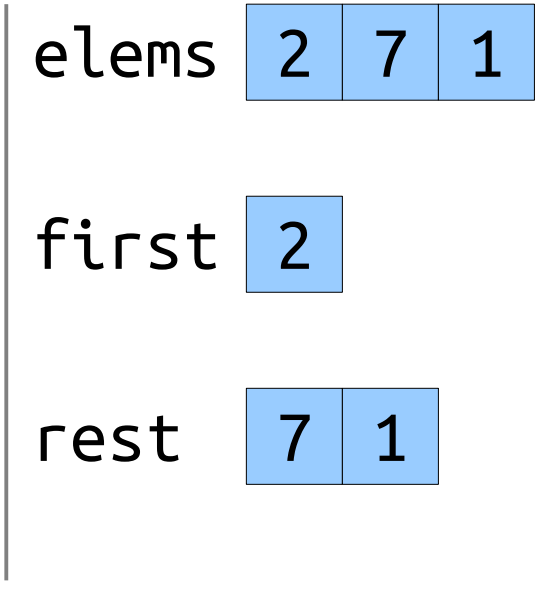
7



Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int first = elems[0];
    Vector<int> rest = elems.subList(1);
    return max(first, maxOf(rest));
  }
}
```

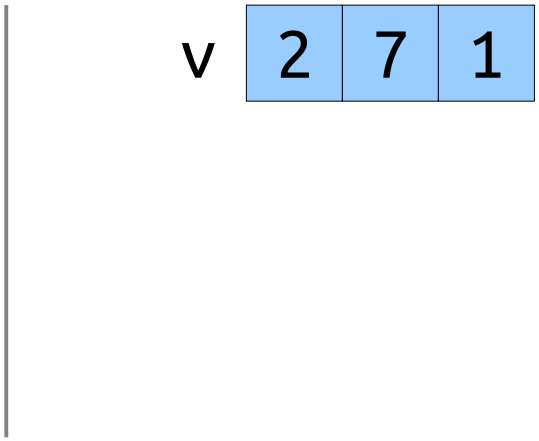
7



Tracing the Recursion

```
int main() {
    Vector<int> v = { 2, 7, 1 };
    cout << maxOf(v) << endl;
    return 0;
}
```

7



Summary from Today

- The `Vector<T>` type in C++ represents a sequence of elements.
- Parameters in C++ are passed by *value* by default. You can change that to use pass by *reference* if you'd like.
- Use pass-by-`const`-reference for objects you don't intend to change.
- Each stack frame from a recursive function gets its own copies of all the local variables.

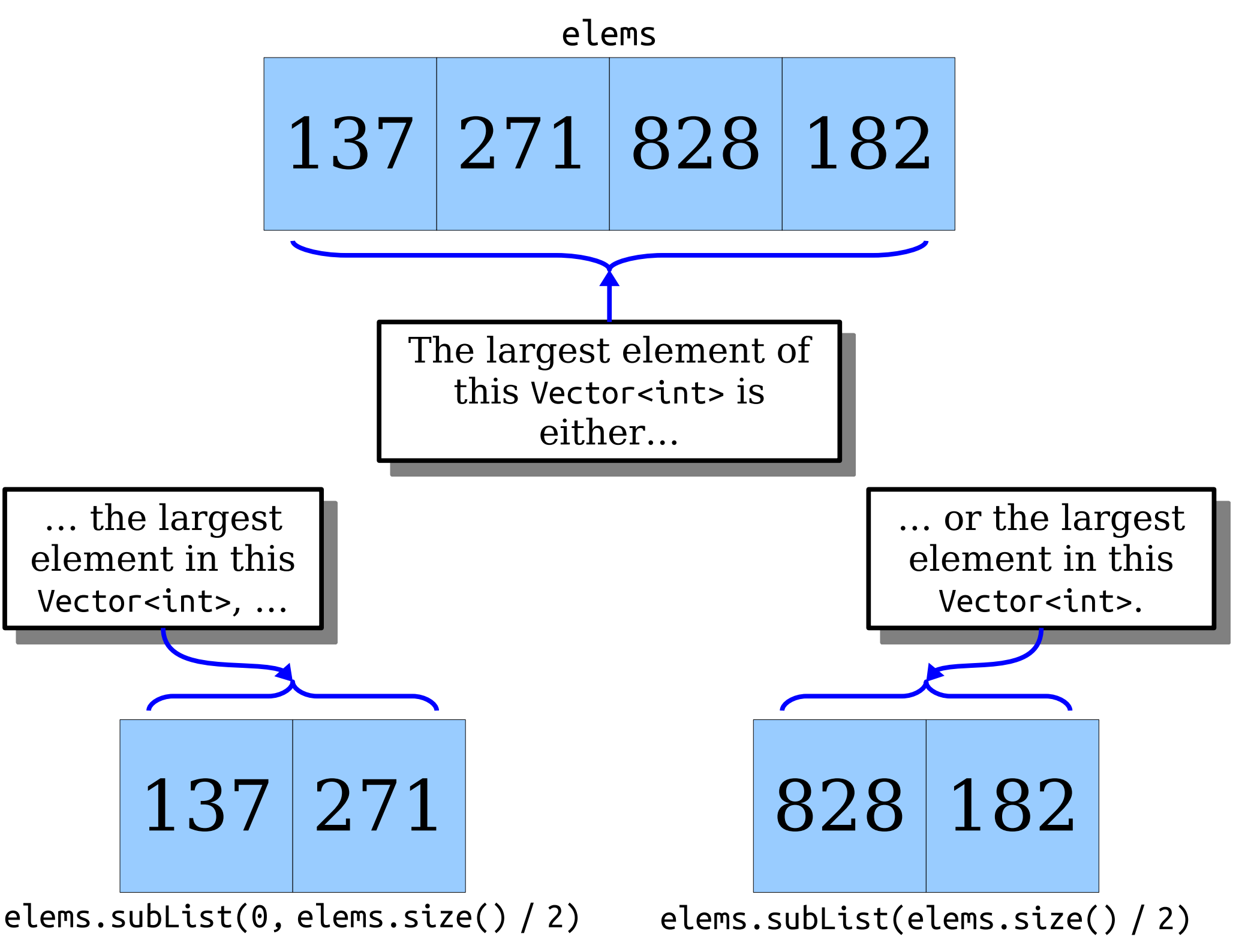
Your Action Items

- ***Read Chapter 5.1 and Chapter 5.2 of the textbook.***
  - It's all about `Vector` and `Grid`! There are some goodies there.
- ***Work on Assignment 1.***
  - Aim to complete the Debugger Warmups and Fire by Monday and to start Only Connect on Monday.
- ***Explore the `maxOf` example.***
  - Tinker and play around with this one. See if you can get very comfortable with how it works.

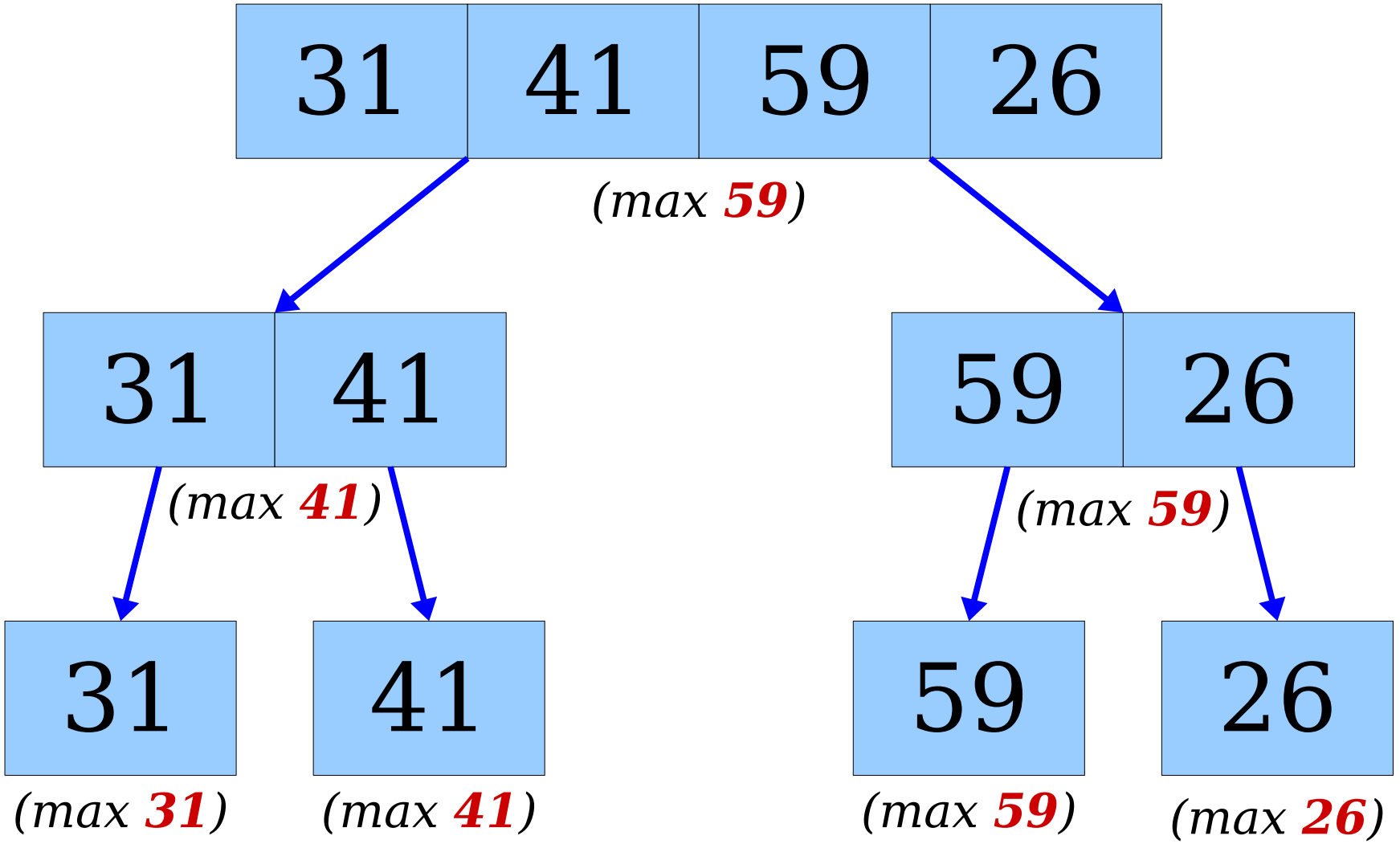
Next Time

- ***Lexicons, Sets, and Maps.***
  - Storing words.
  - Storing items in No Particular Order.
  - Associating items with one another.

Appendix: Finding the max, another way.


elems
137
271
828
182
The largest element of this Vector<int> is either...
... the largest element in this Vector<int>, ...
... or the largest element in this Vector<int>.
137
271
828
182
elems.subList(0, elems.size() / 2)
elems.subList(elems.size() / 2)

`maxOf` as a Tournament



maxOf as a Tournament

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int half = elems.size() / 2;

    /* First half of the elements. */
    Vector<int> left  = elems.subList(0, half);

    /* Second half of the elements. */
    Vector<int> right = elems.subList(half);

    /* Biggest value in the overall list is either
     * the largest element in the first half or
     * the largest element in the second half.
     */
    return max(maxOf(left), maxOf(right));
  }
}
```

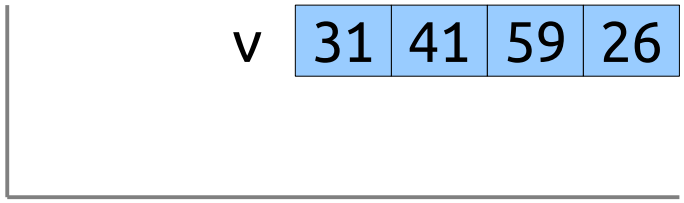
Tracing the Recursion

```
int main() {
    Vector<int> v = { 31, 41, 59, 26 };
    cout << maxOf(v) << endl;
    return 0;
}
```

Tracing the Recursion

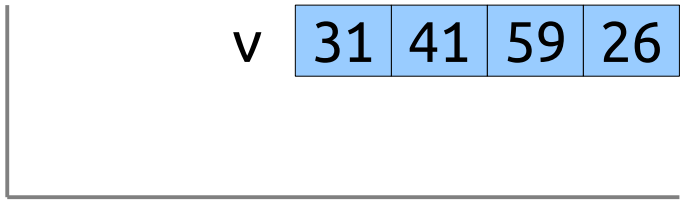
```
int main() {
    Vector<int> v = { 31, 41, 59, 26 };
    cout << maxOf(v) << endl;
    return 0;
}
```

Tracing the Recursion



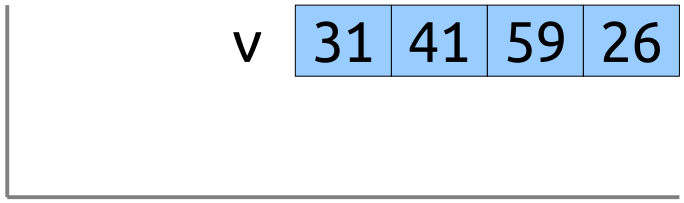

```
int main() {
    Vector<int> v = { 31, 41, 59, 26 };
    cout << maxOf(v) << endl;
    return 0;
}
```

Tracing the Recursion




```
int main() {
   Vector<int> v = { 31, 41, 59, 26 };
   cout << maxOf(v) << endl;
   return 0;
}
```

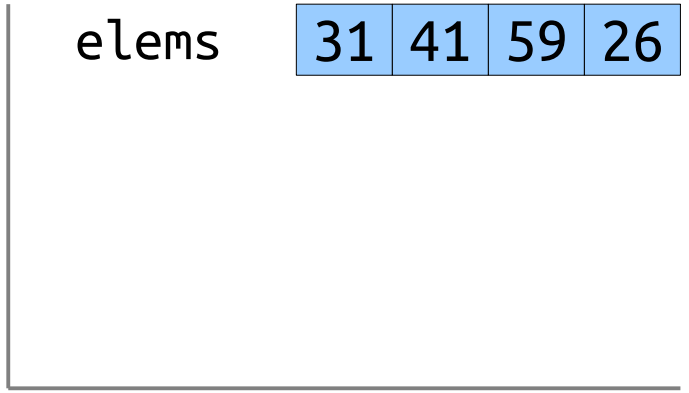
Tracing the Recursion



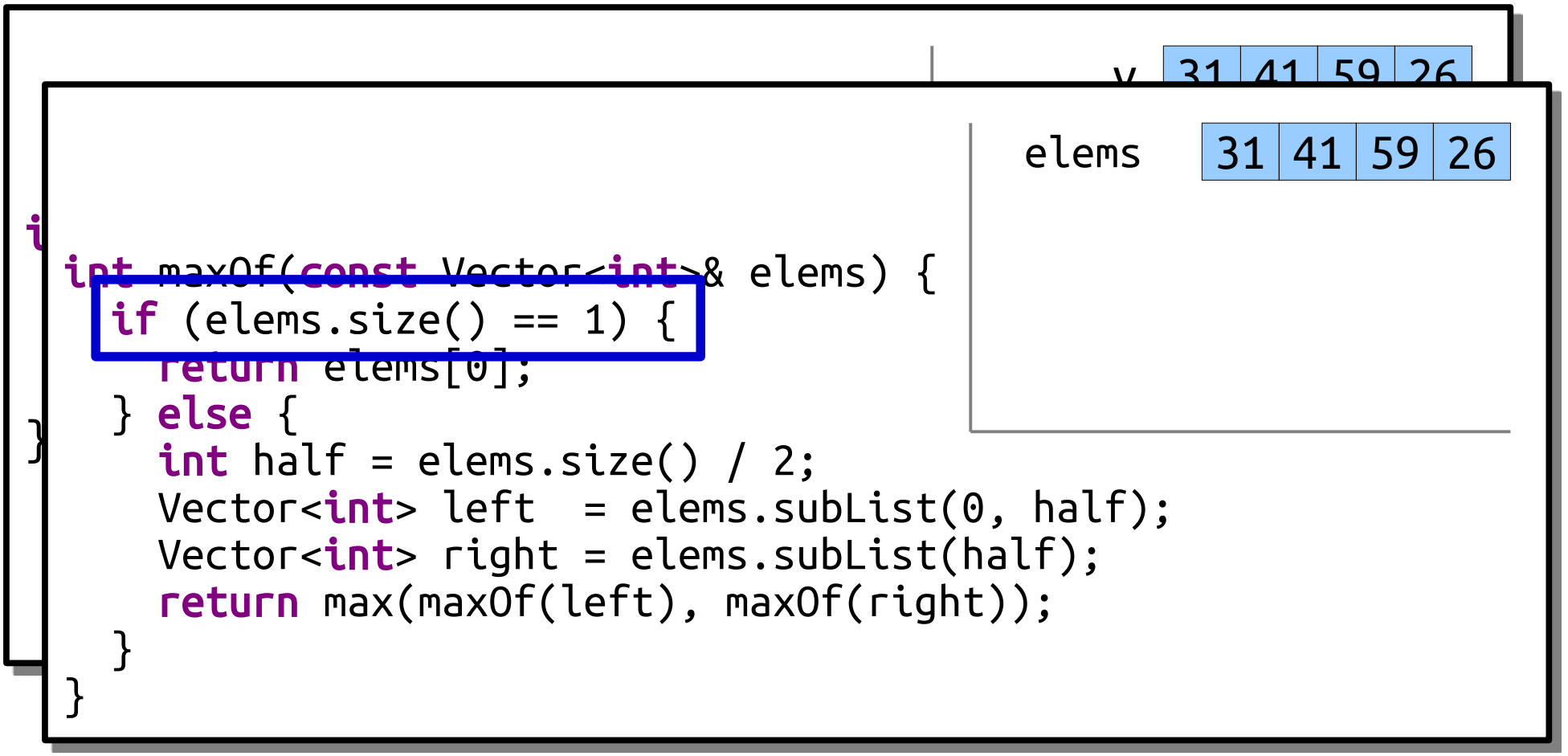

```
int main() {
    Vector<int> v = { 31, 41, 59, 26 };
    cout << maxOf(v) << endl;
    return 0;
}
```

Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int half = elems.size() / 2;
    Vector<int> left  = elems.subList(0, half);
    Vector<int> right = elems.subList(half);
    return max(maxOf(left), maxOf(right));
  }
}
```

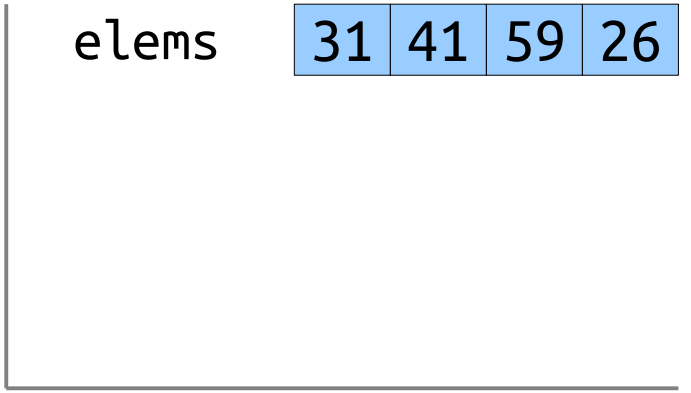


Tracing the Recursion



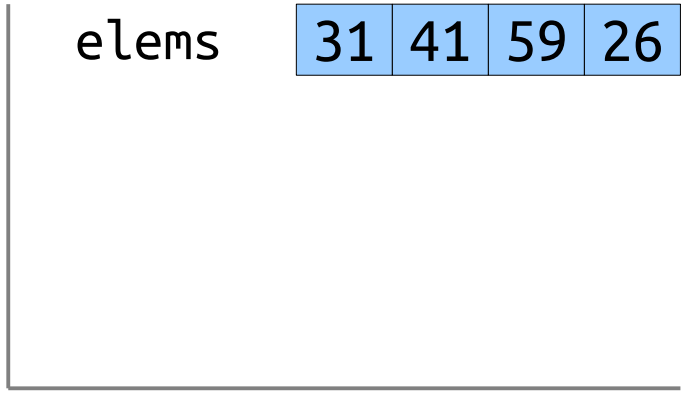
Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int half = elems.size() / 2;
    Vector<int> left  = elems.subList(0, half);
    Vector<int> right = elems.subList(half);
    return max(maxOf(left), maxOf(right));
  }
}
```



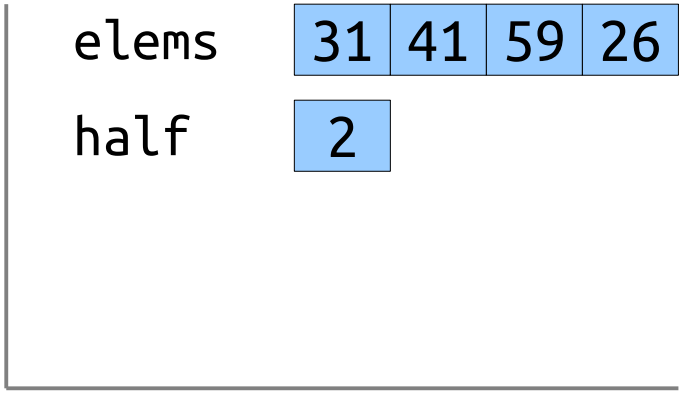
Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int half = elems.size() / 2;
    Vector<int> left  = elems.subList(0, half);
    Vector<int> right = elems.subList(half);
    return max(maxOf(left), maxOf(right));
  }
}
```



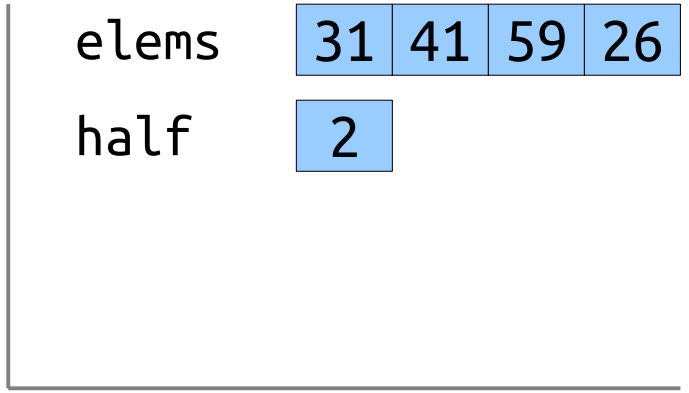
Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int half = elems.size() / 2;
    Vector<int> left  = elems.subList(0, half);
    Vector<int> right = elems.subList(half);
    return max(maxOf(left), maxOf(right));
  }
}
```



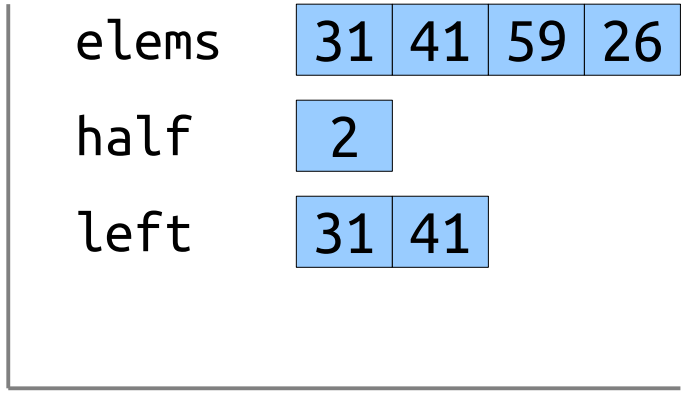
Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int half = elems.size() / 2;
    Vector<int> left  = elems.subList(0, half);
    Vector<int> right = elems.subList(half);
    return max(maxOf(left), maxOf(right));
  }
}
```



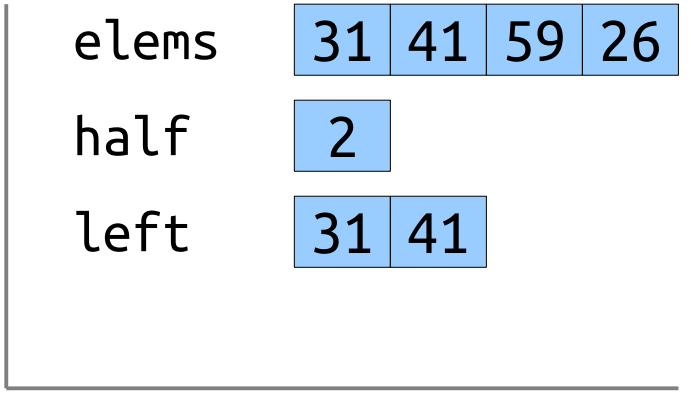
Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int half = elems.size() / 2;
    Vector<int> left  = elems.subList(0, half);
    Vector<int> right = elems.subList(half);
    return max(maxOf(left), maxOf(right));
  }
}
```



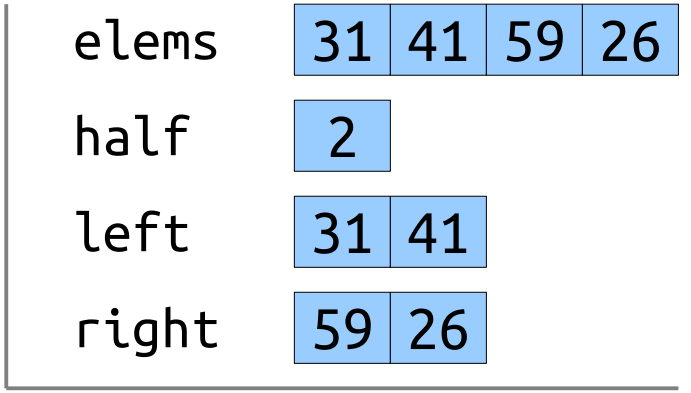
Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int half = elems.size() / 2;
    Vector<int> left  = elems.subList(0, half);
    Vector<int> right = elems.subList(half);
    return max(maxOf(left), maxOf(right));
  }
}
```



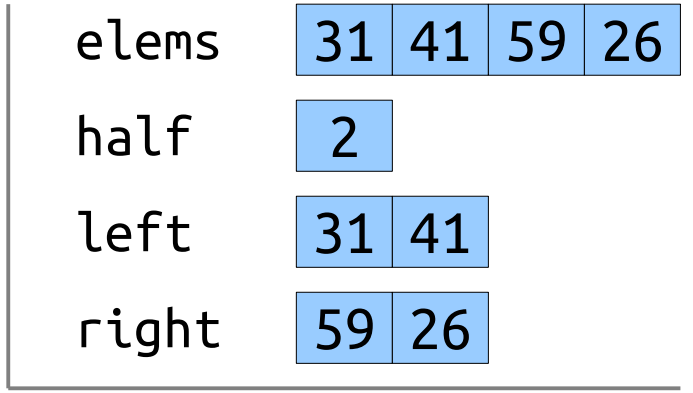
Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int half = elems.size() / 2;
    Vector<int> left  = elems.subList(0, half);
    Vector<int> right = elems.subList(half);
    return max(maxOf(left), maxOf(right));
  }
}
```



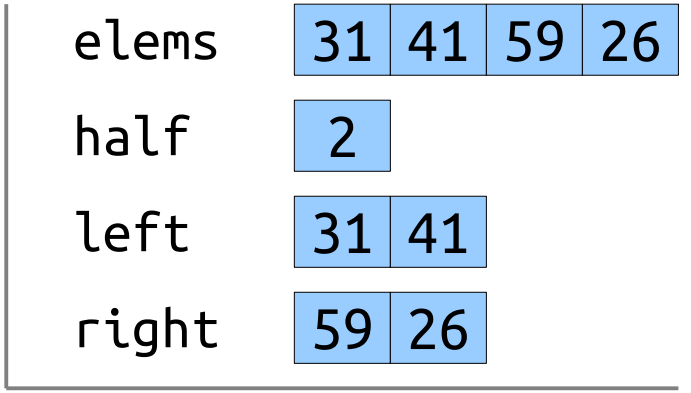
Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int half = elems.size() / 2;
    Vector<int> left  = elems.subList(0, half);
    Vector<int> right = elems.subList(half);
    return max(maxOf(left), maxOf(right));
  }
}
```

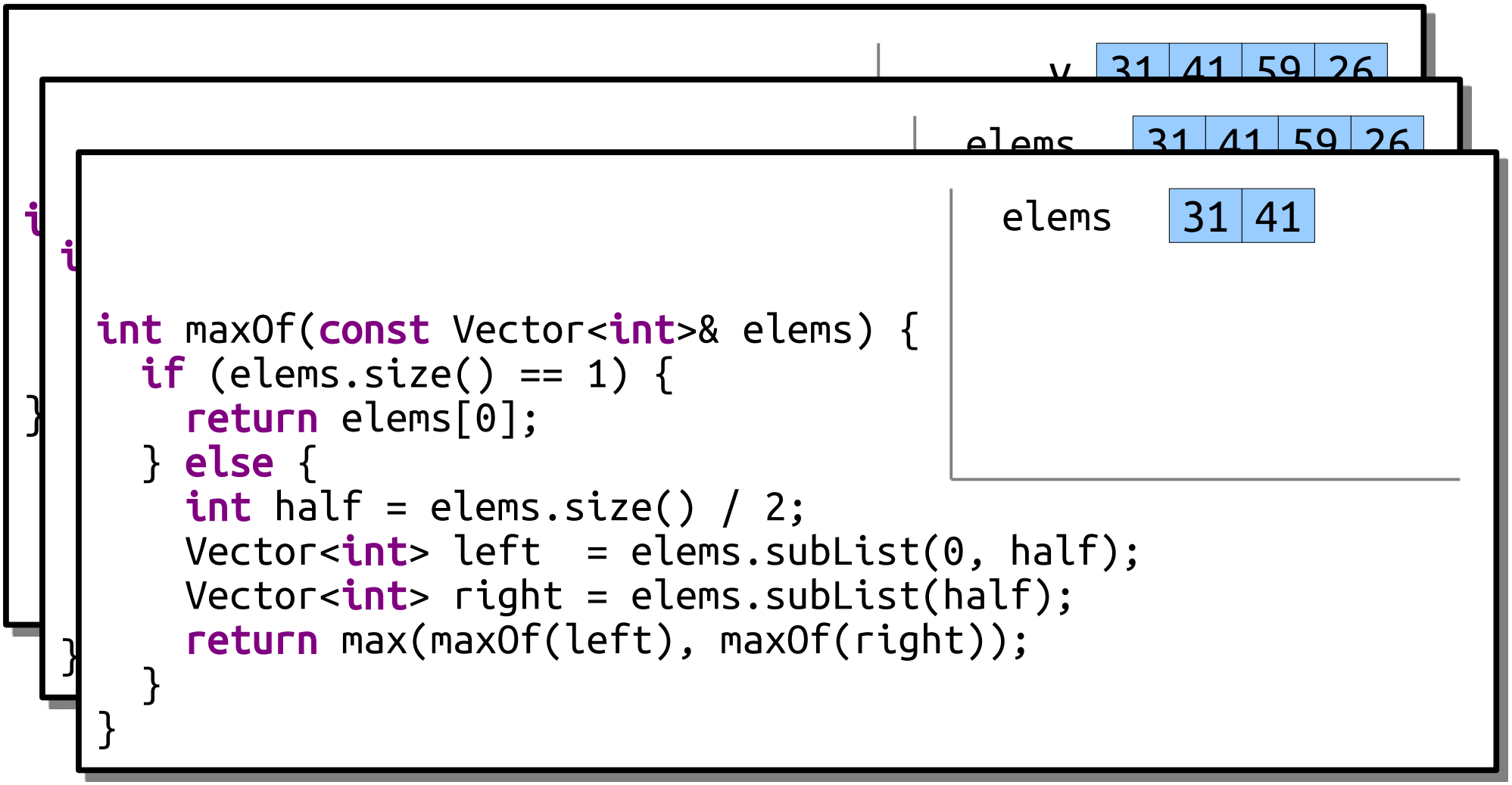


Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int half = elems.size() / 2;
    Vector<int> left  = elems.subList(0, half);
    Vector<int> right = elems.subList(half);
    return max(maxOf(left), maxOf(right));
  }
}
```

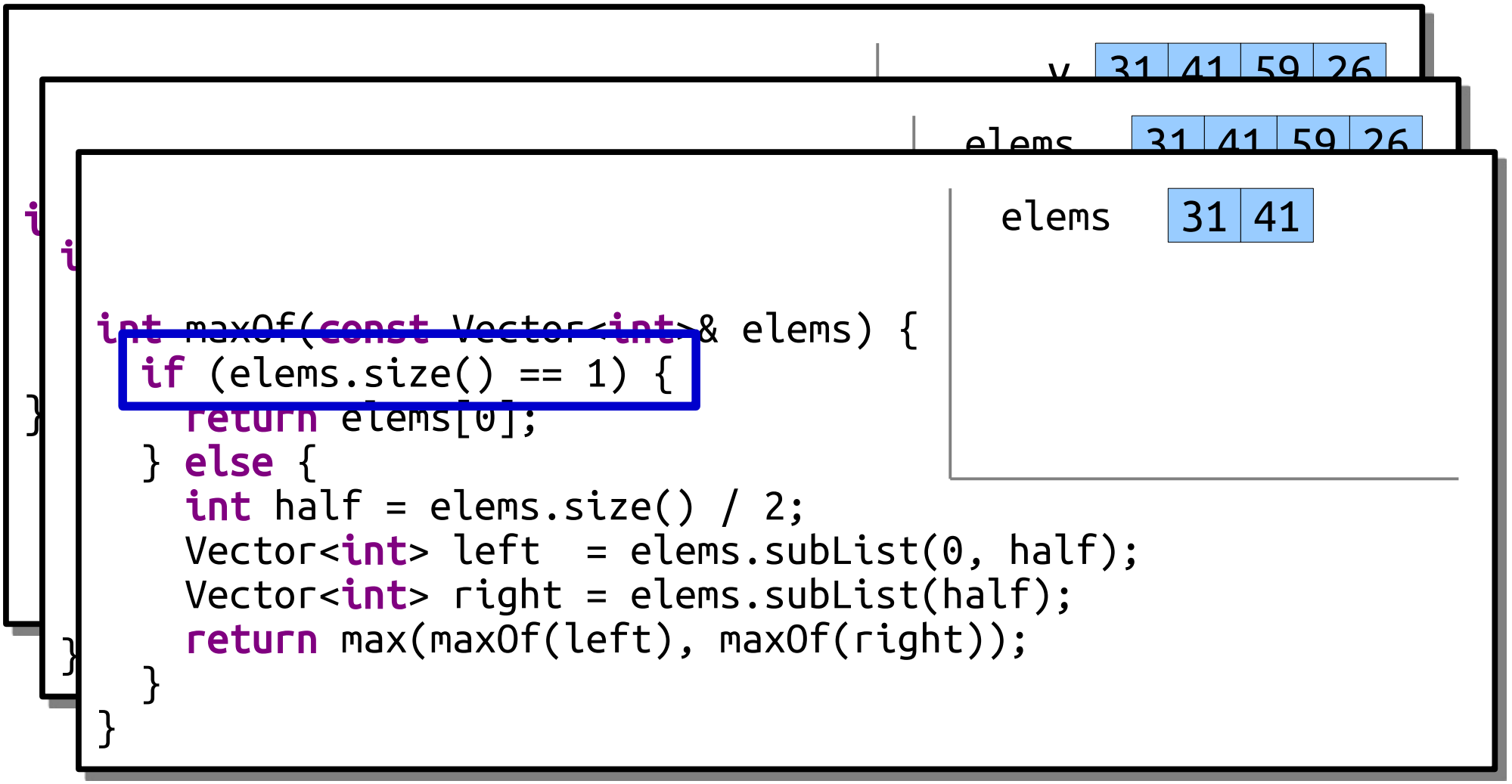


Tracing the Recursion

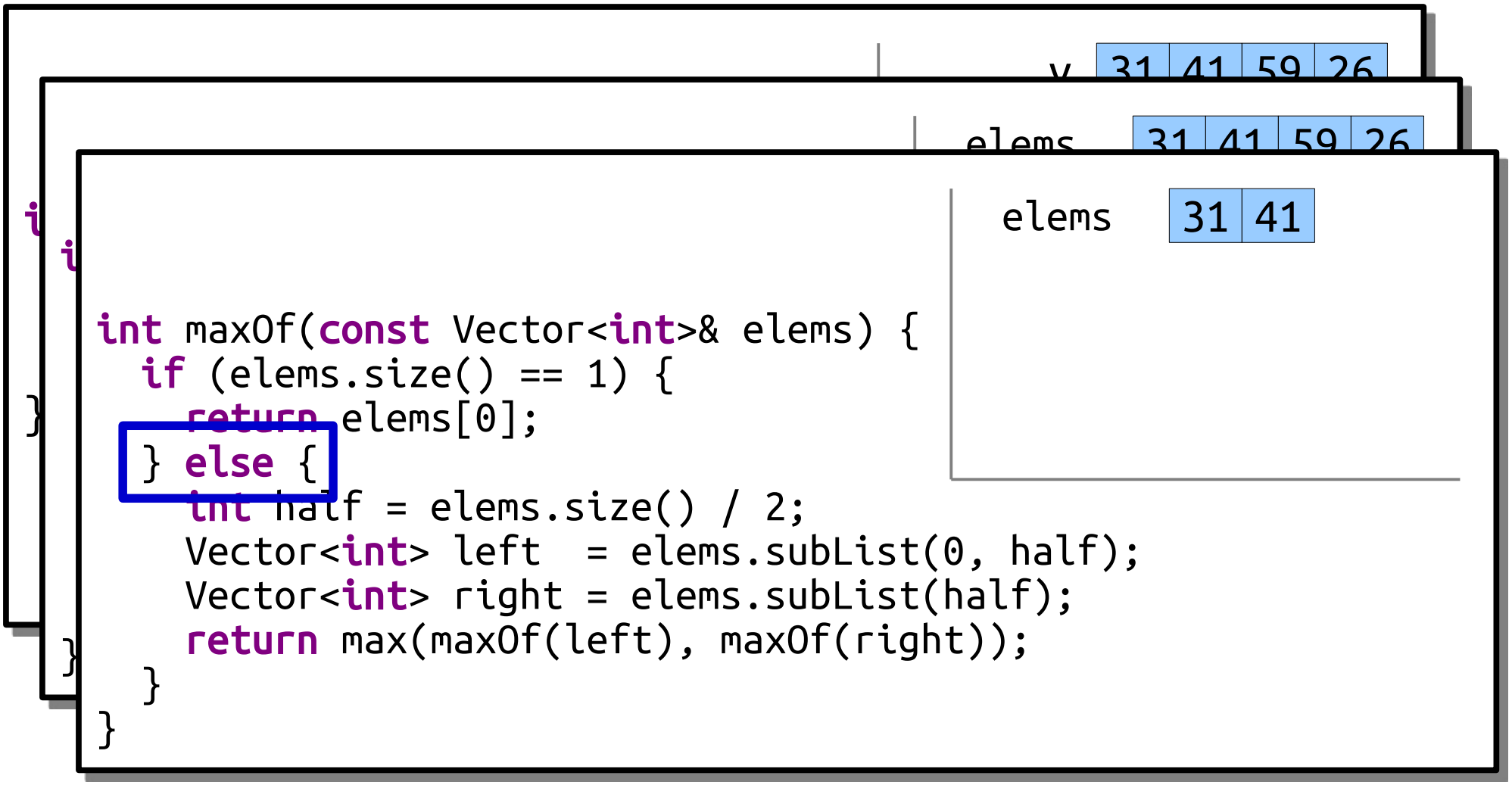



```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int half = elems.size() / 2;
    Vector<int> left  = elems.subList(0, half);
    Vector<int> right = elems.subList(half);
    return max(maxOf(left), maxOf(right));
  }
}
```

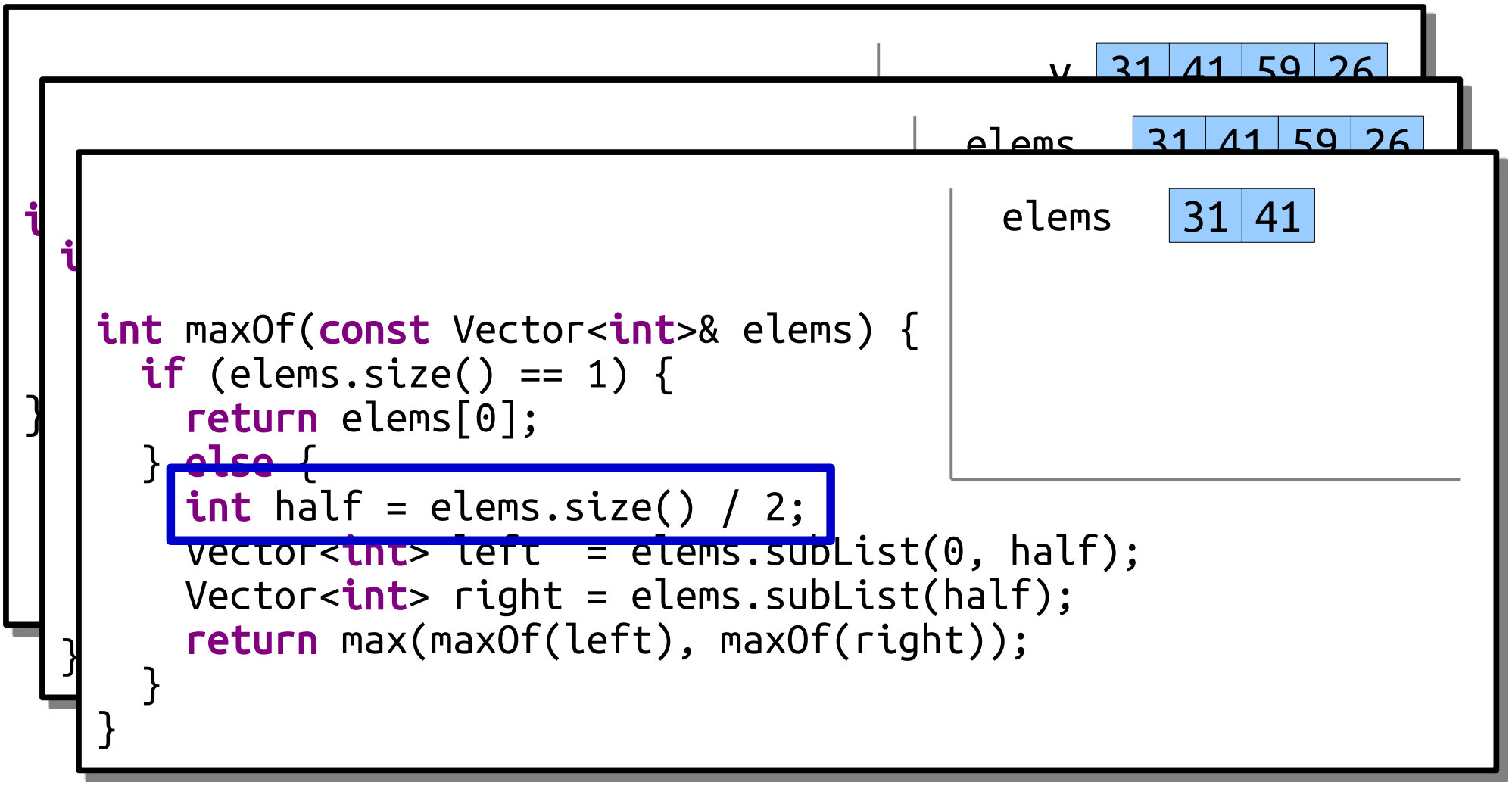
Tracing the Recursion



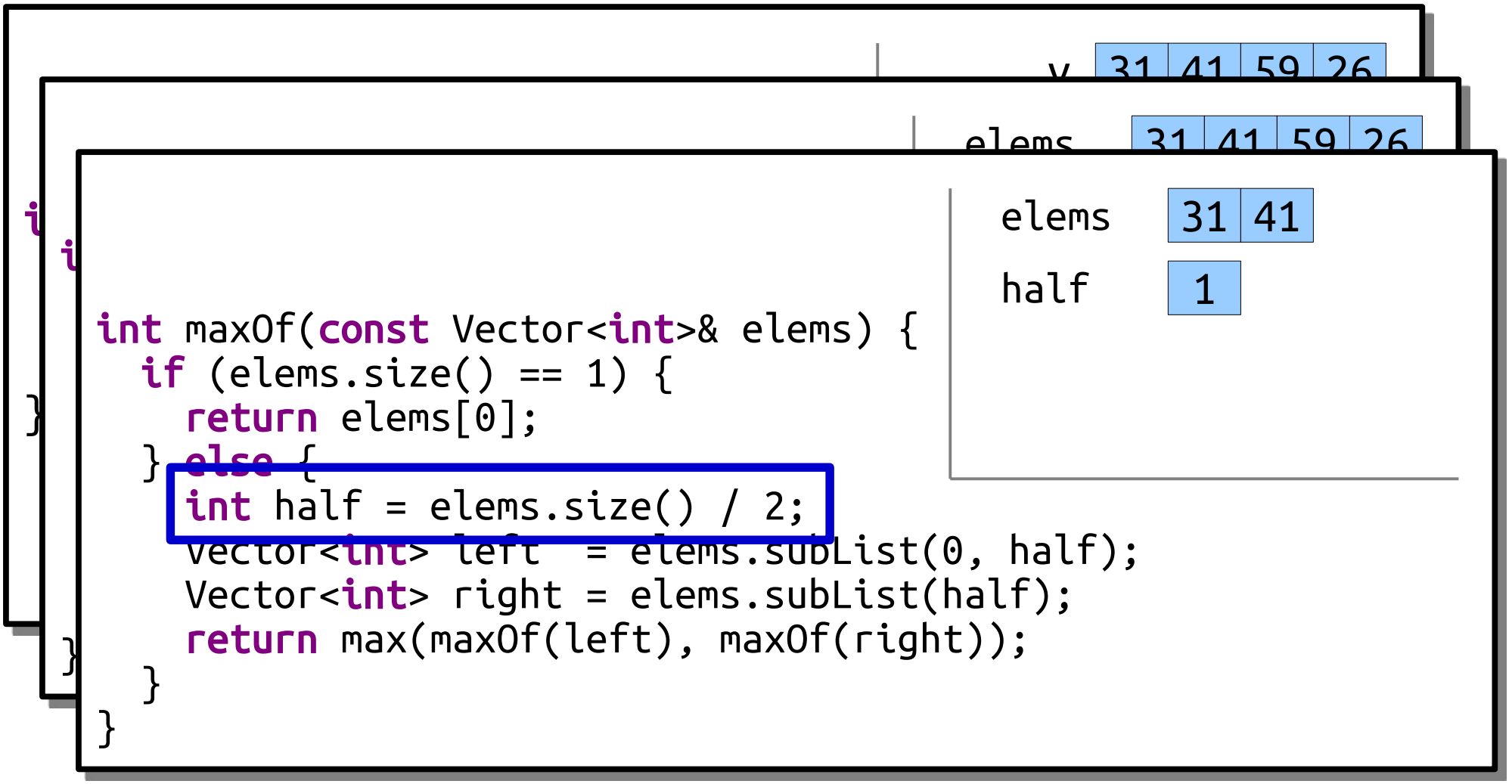
Tracing the Recursion



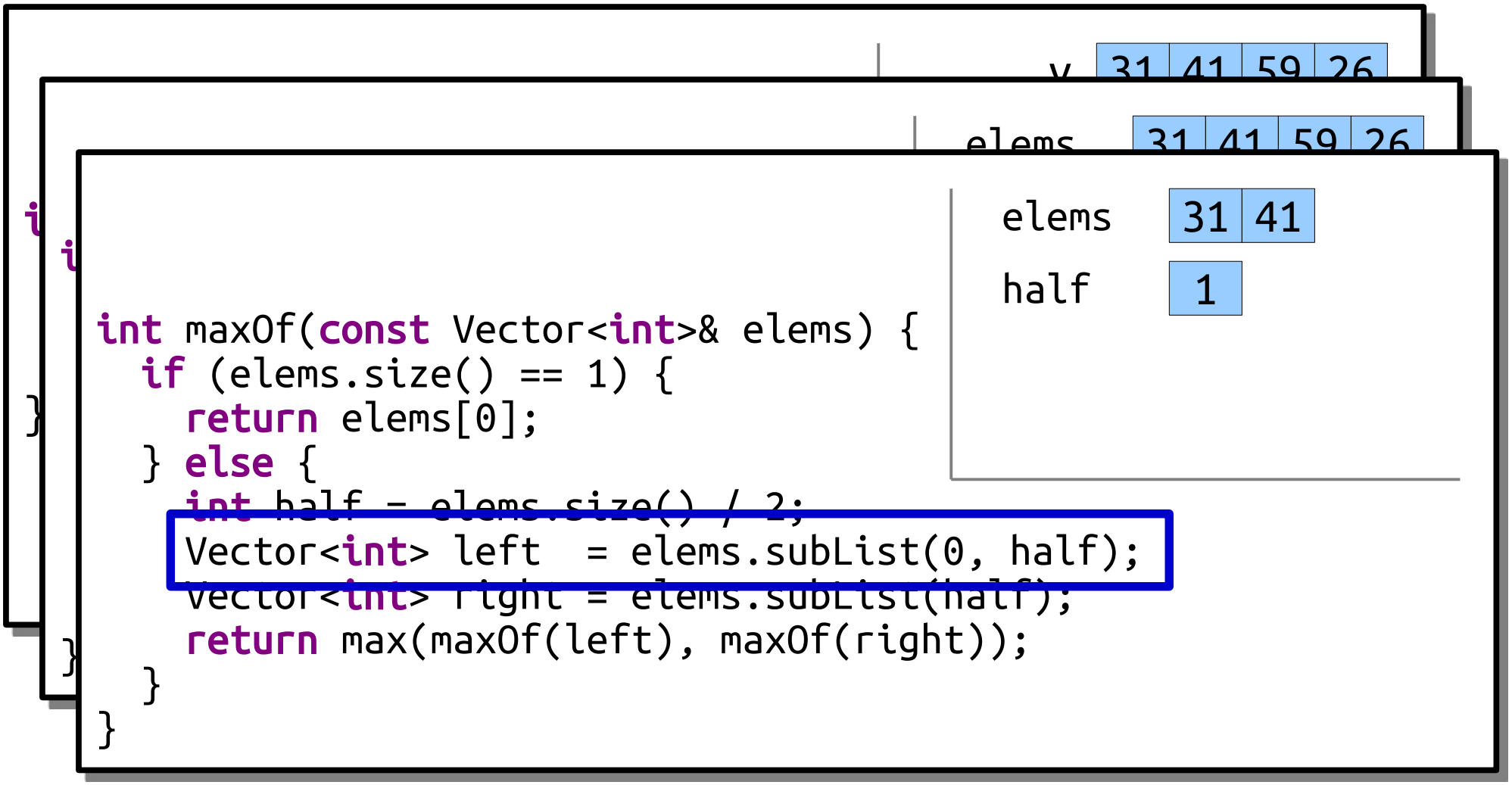
Tracing the Recursion



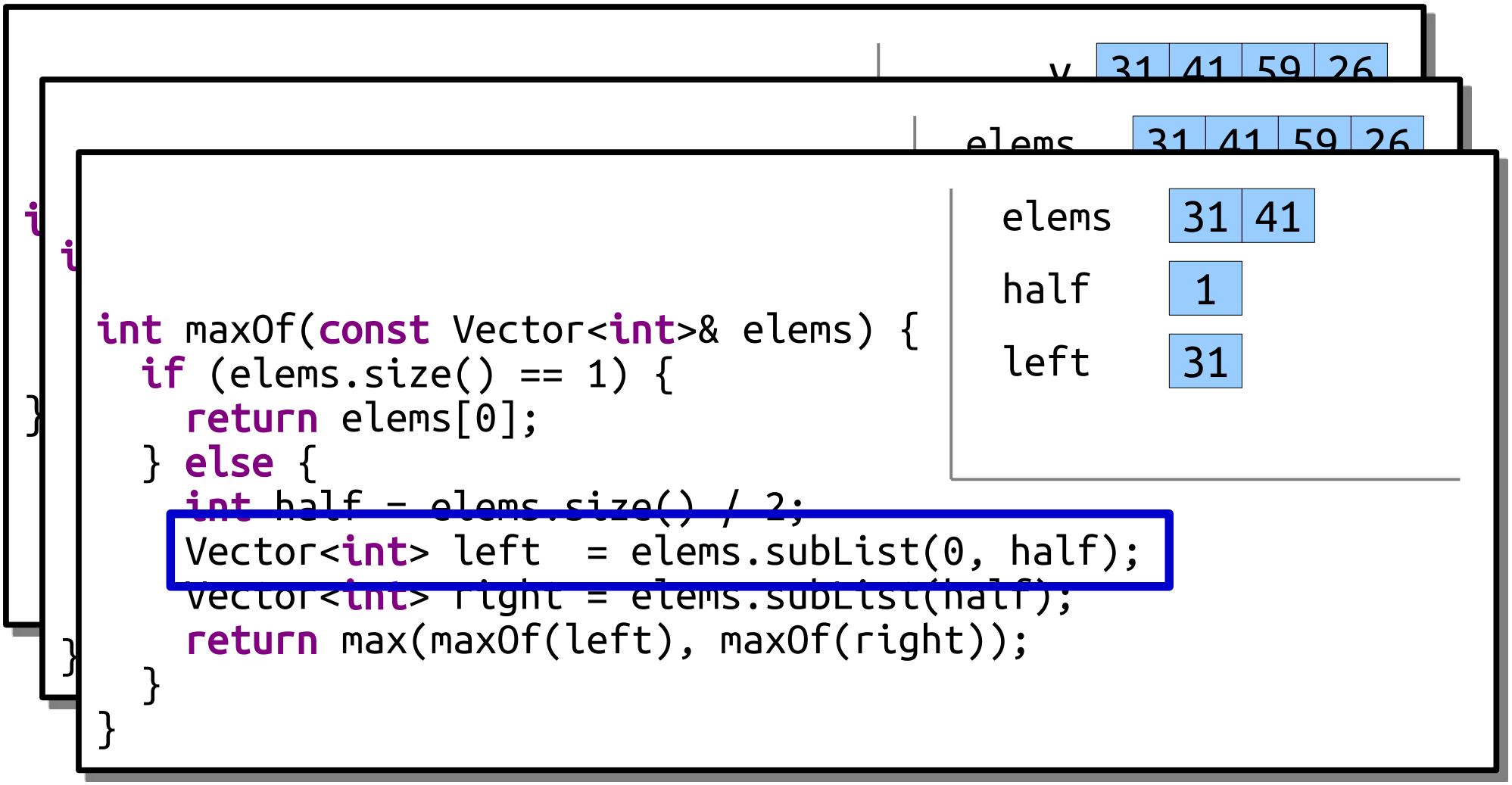
Tracing the Recursion



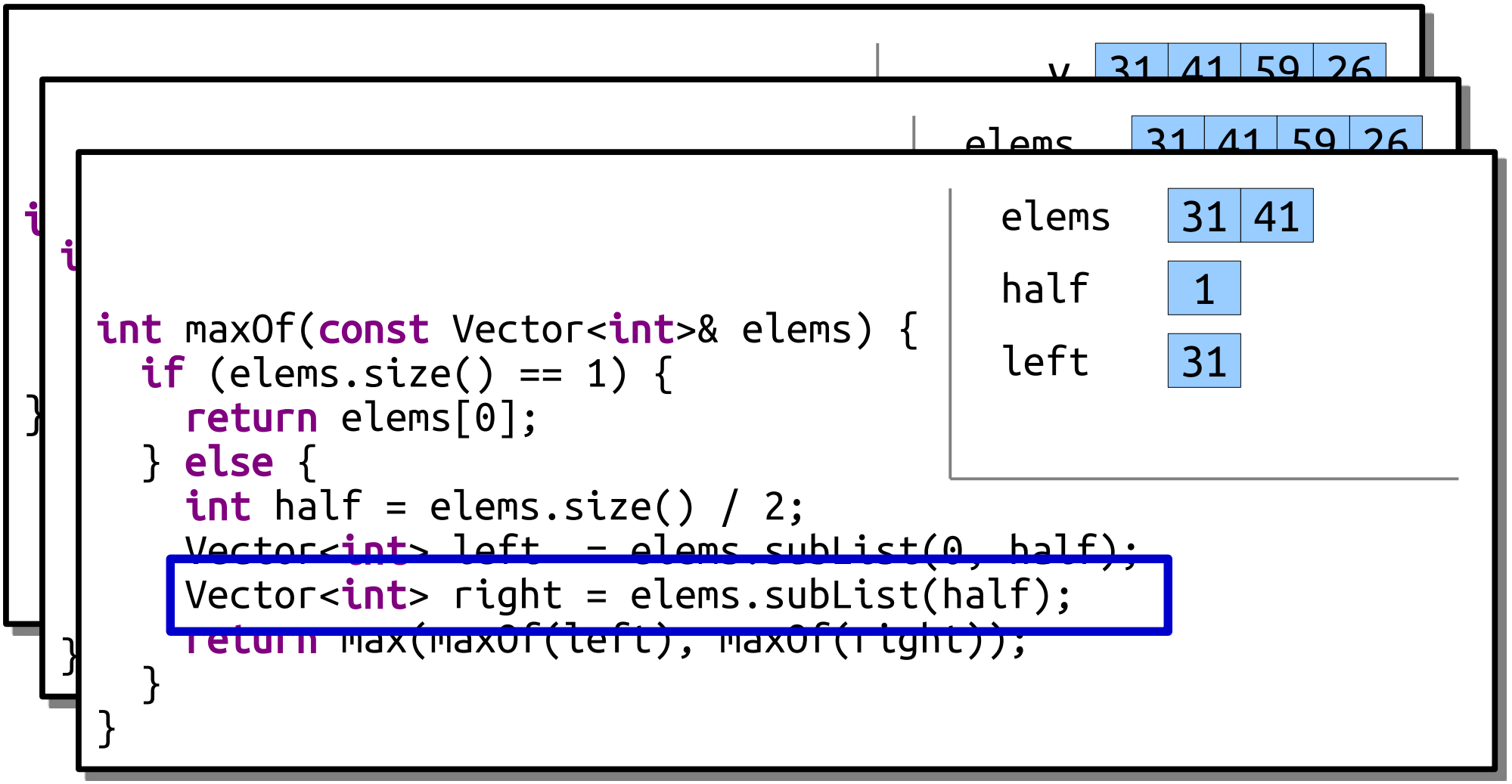
Tracing the Recursion



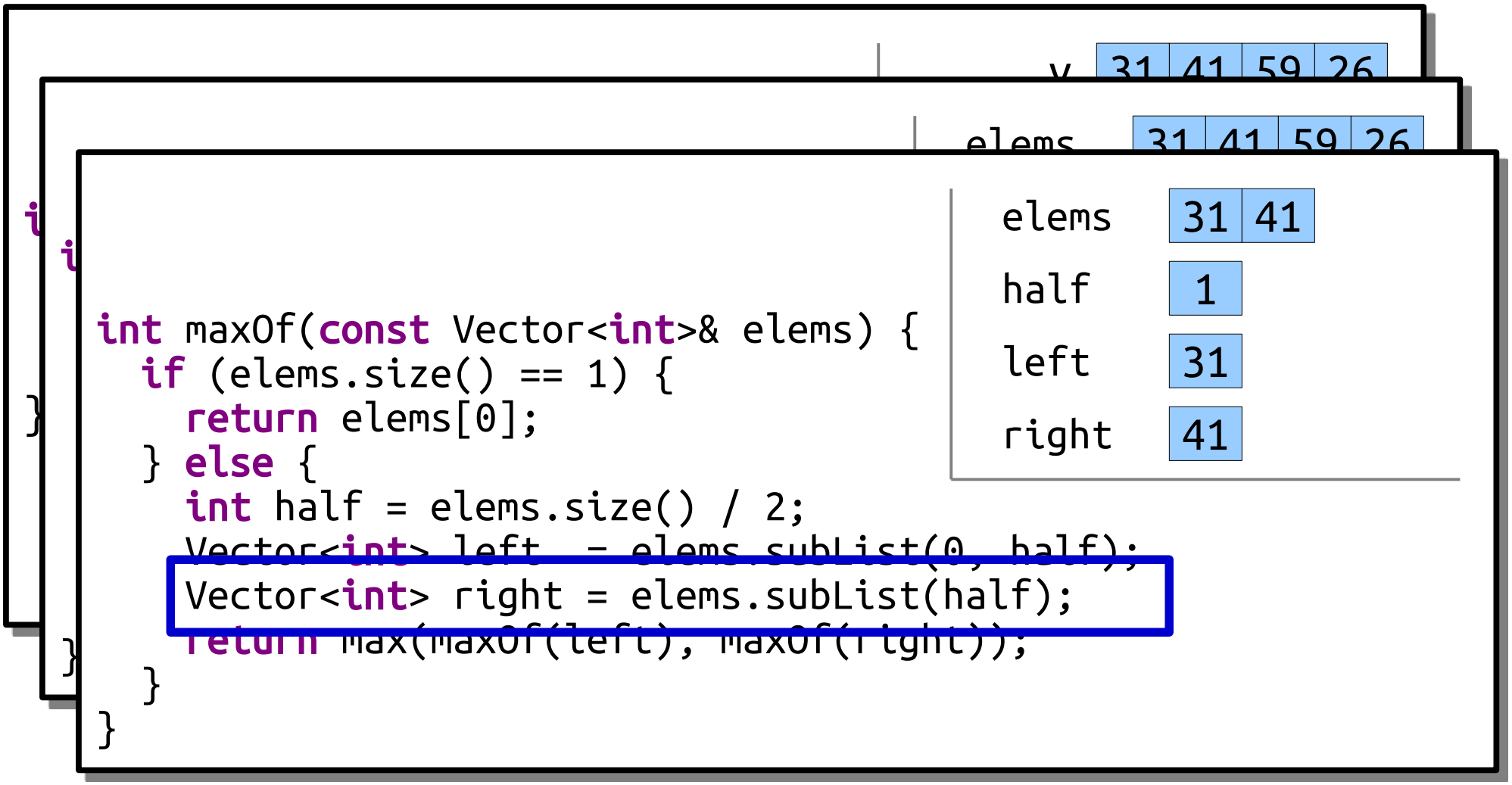
Tracing the Recursion



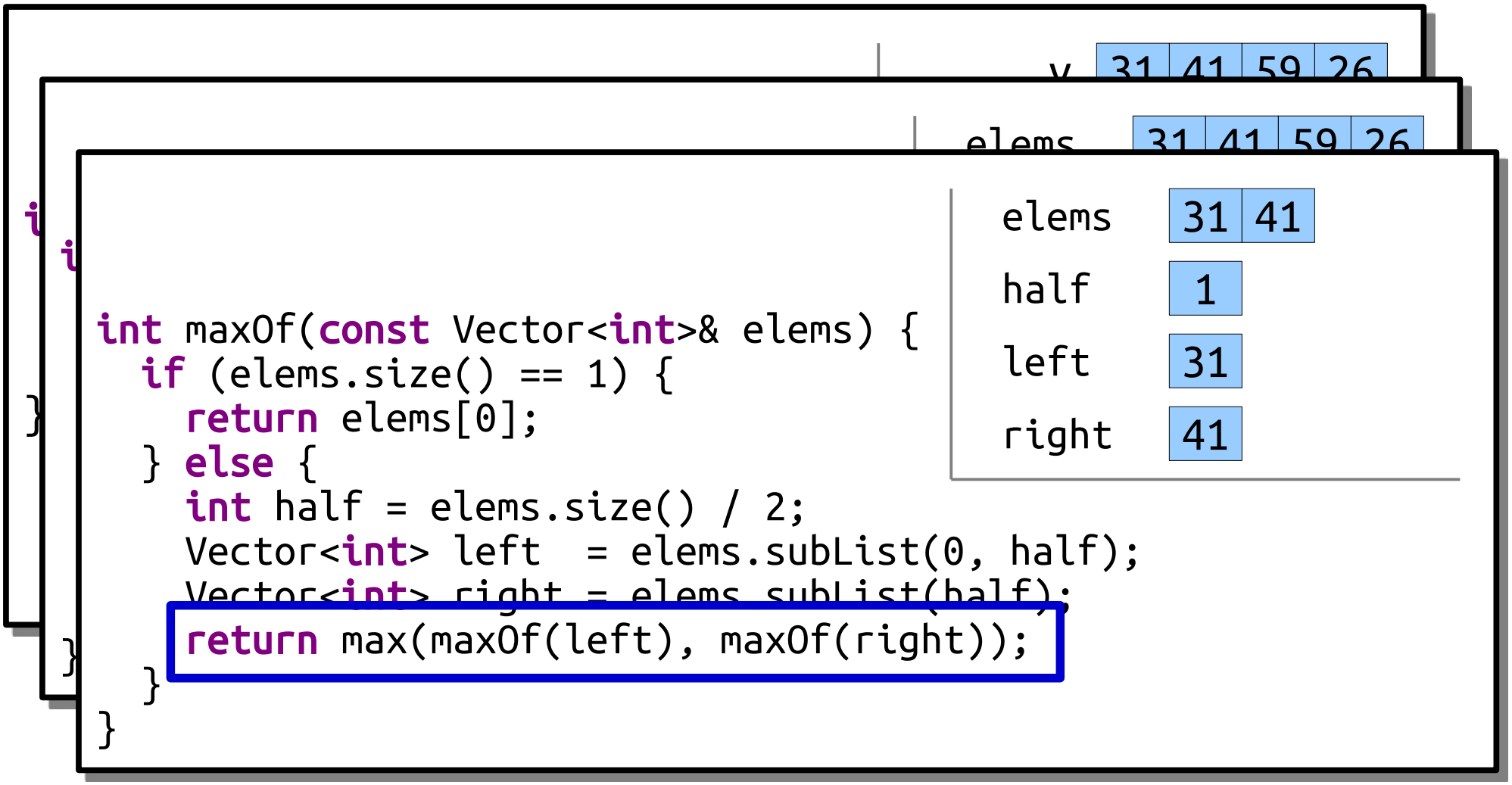
Tracing the Recursion



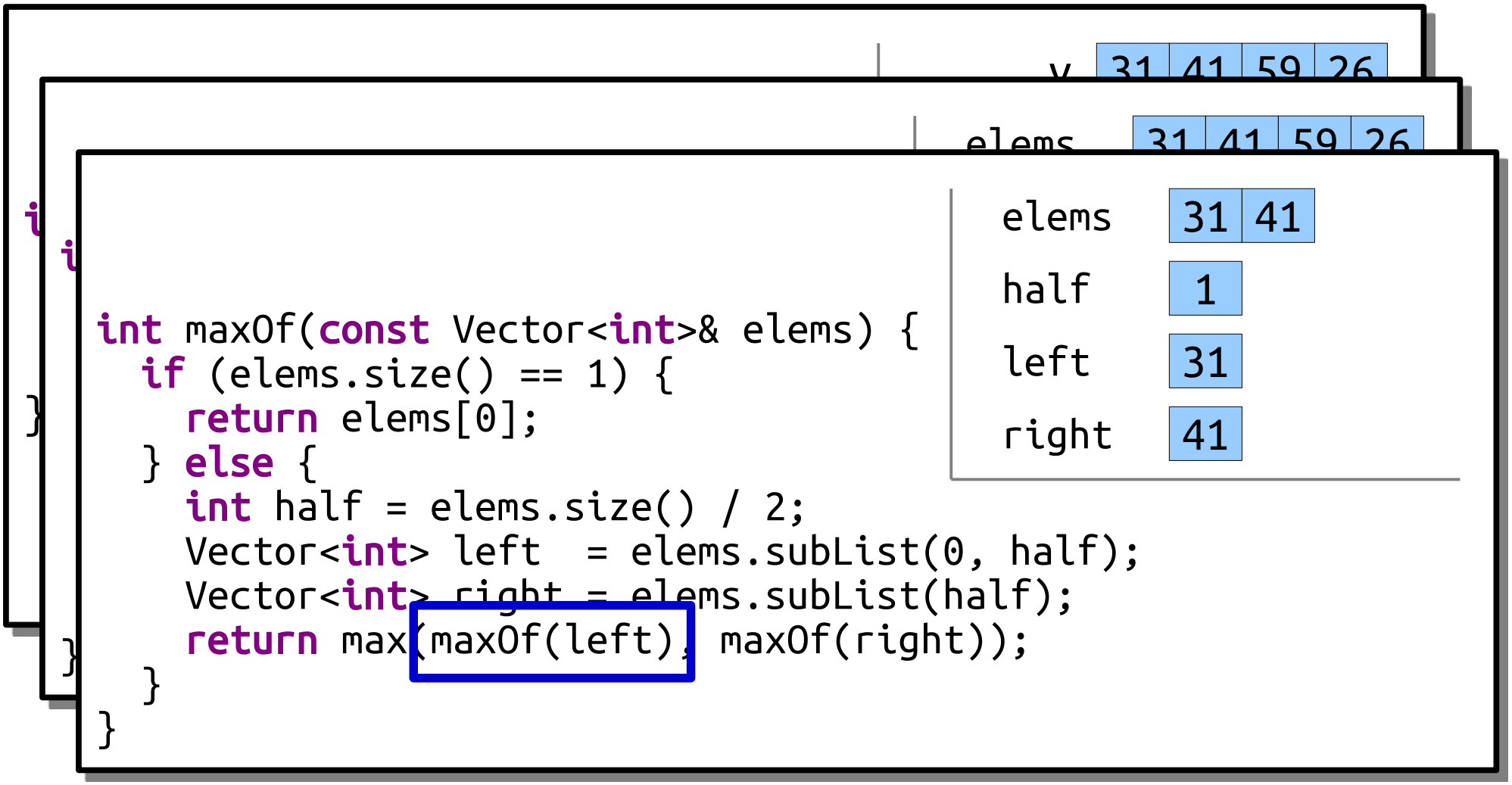
Tracing the Recursion



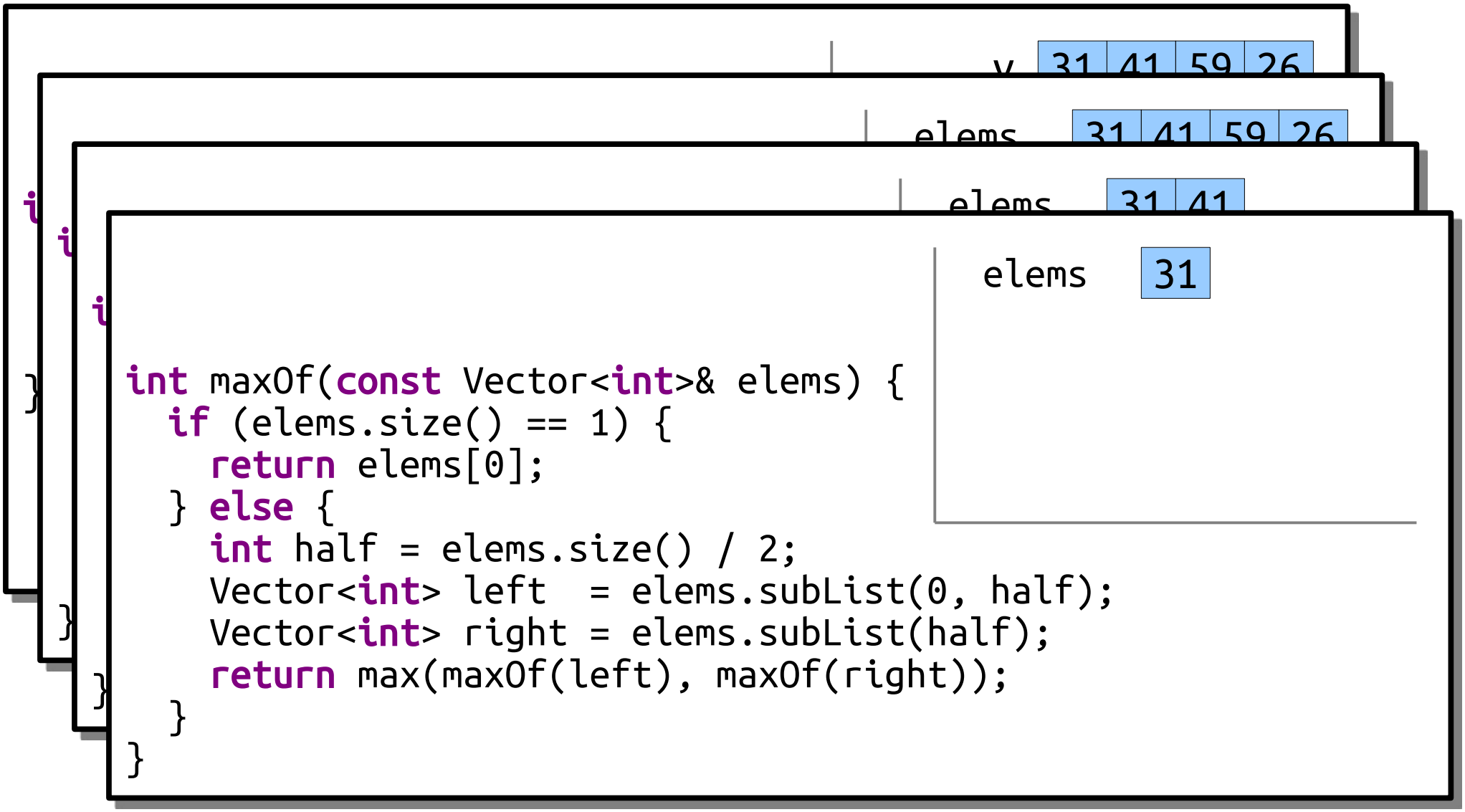
Tracing the Recursion



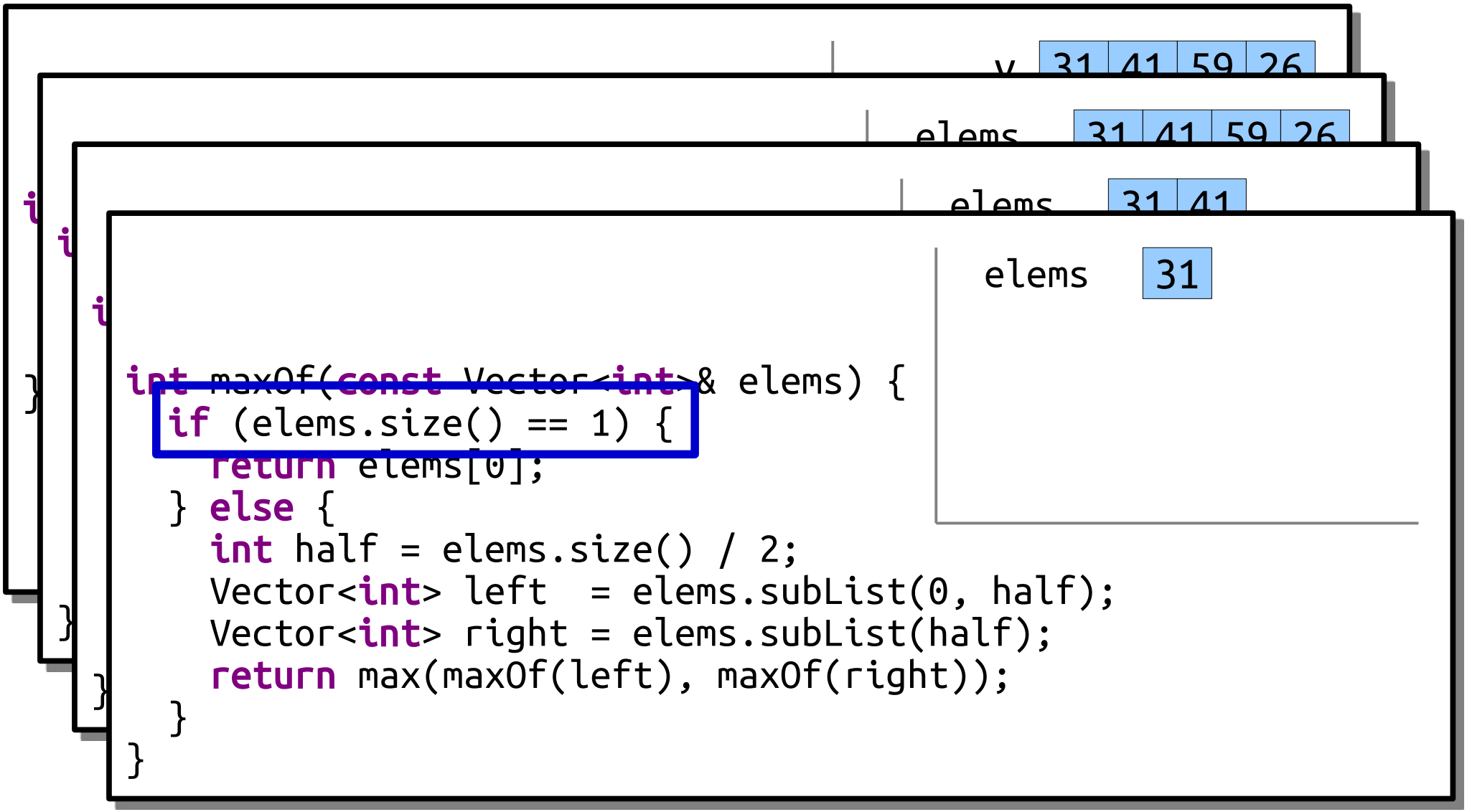
Tracing the Recursion



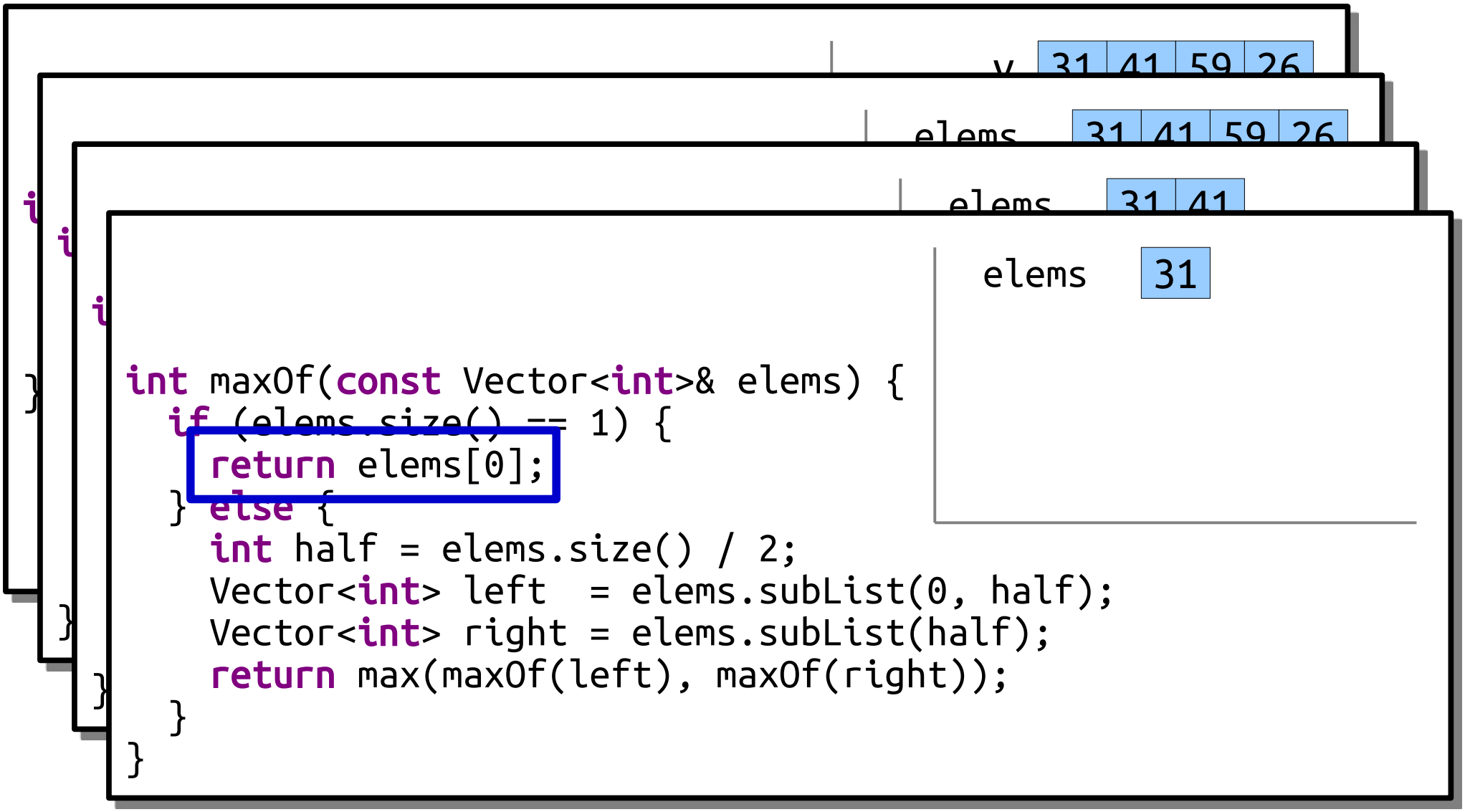
Tracing the Recursion



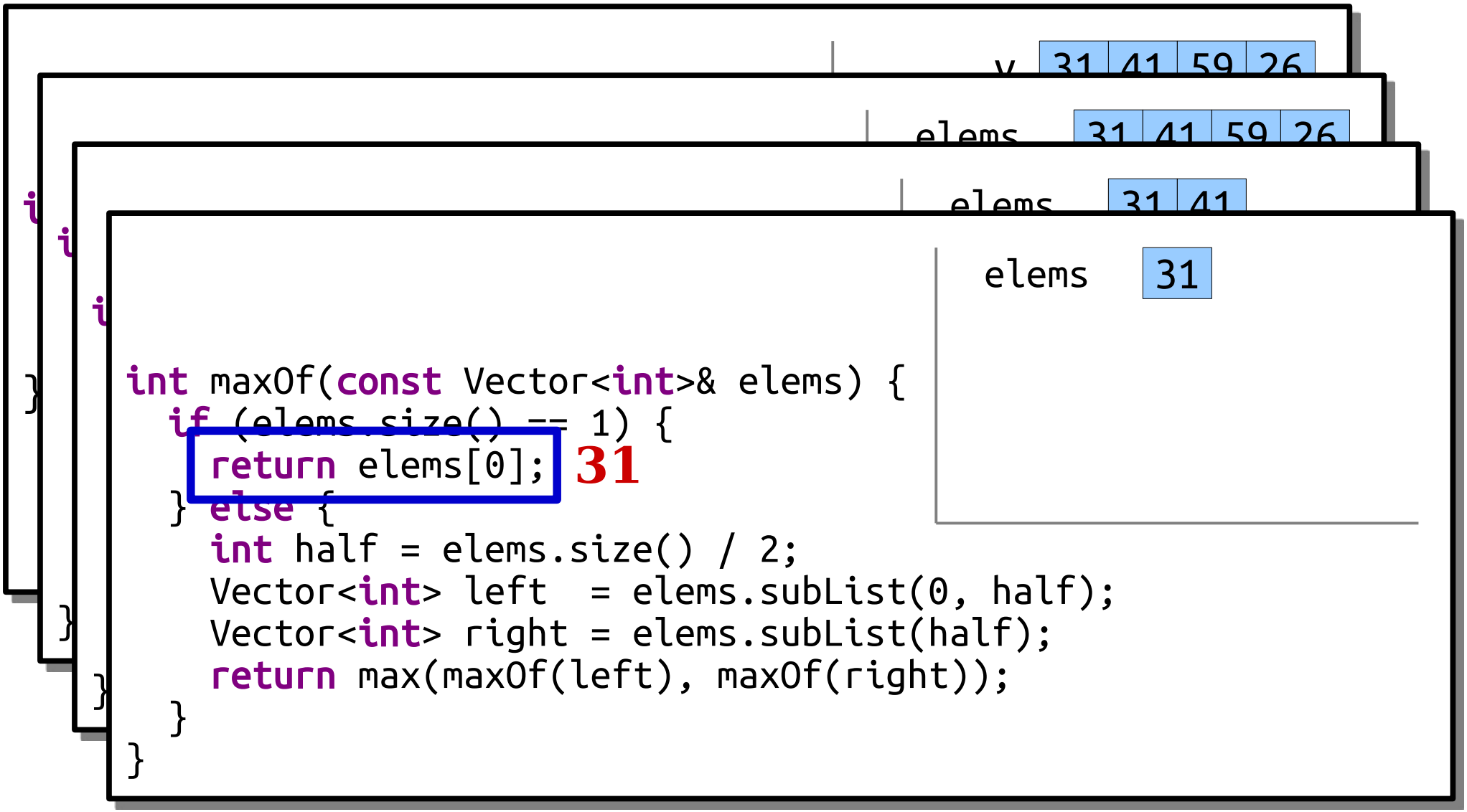
Tracing the Recursion



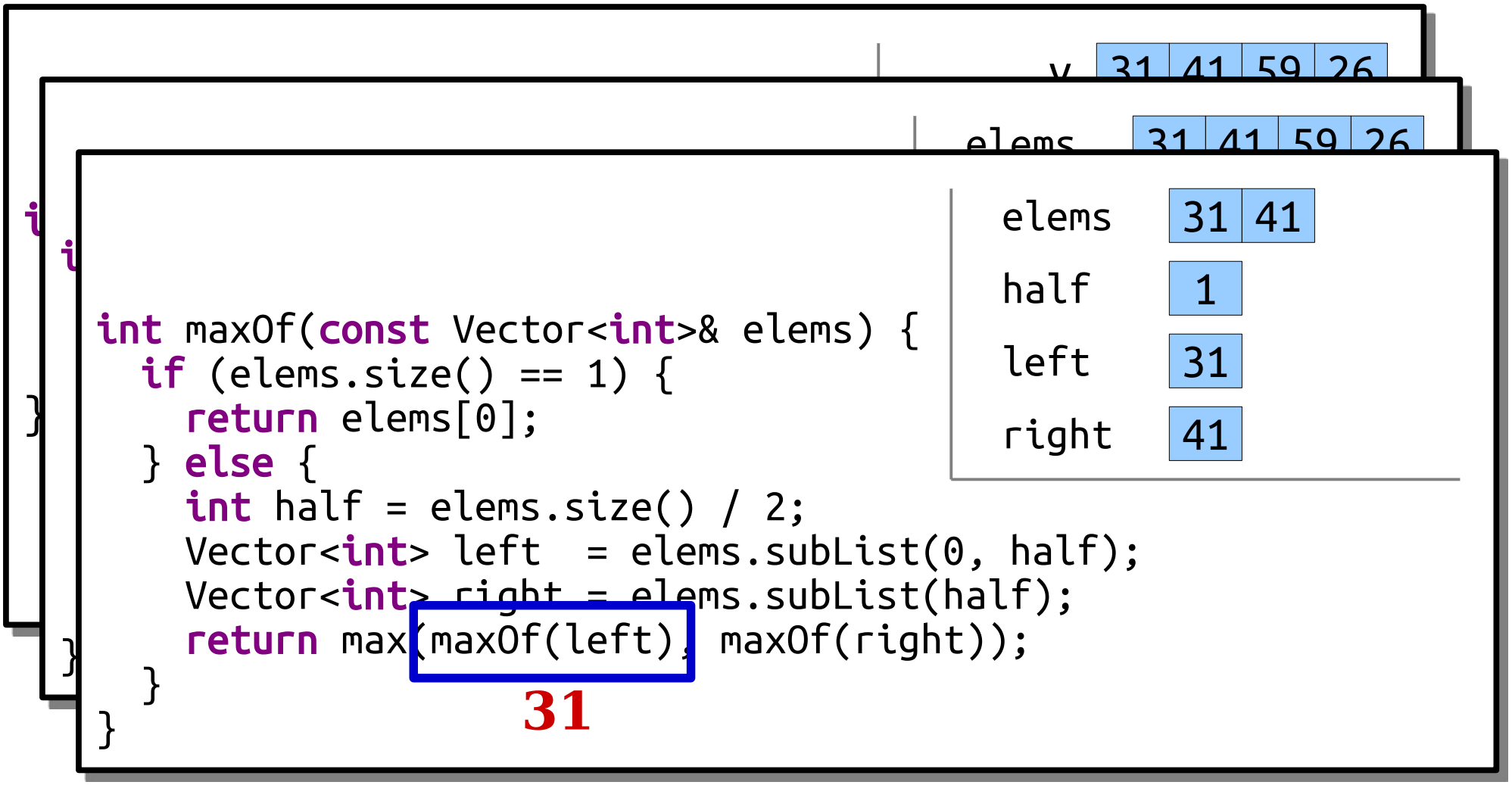
Tracing the Recursion



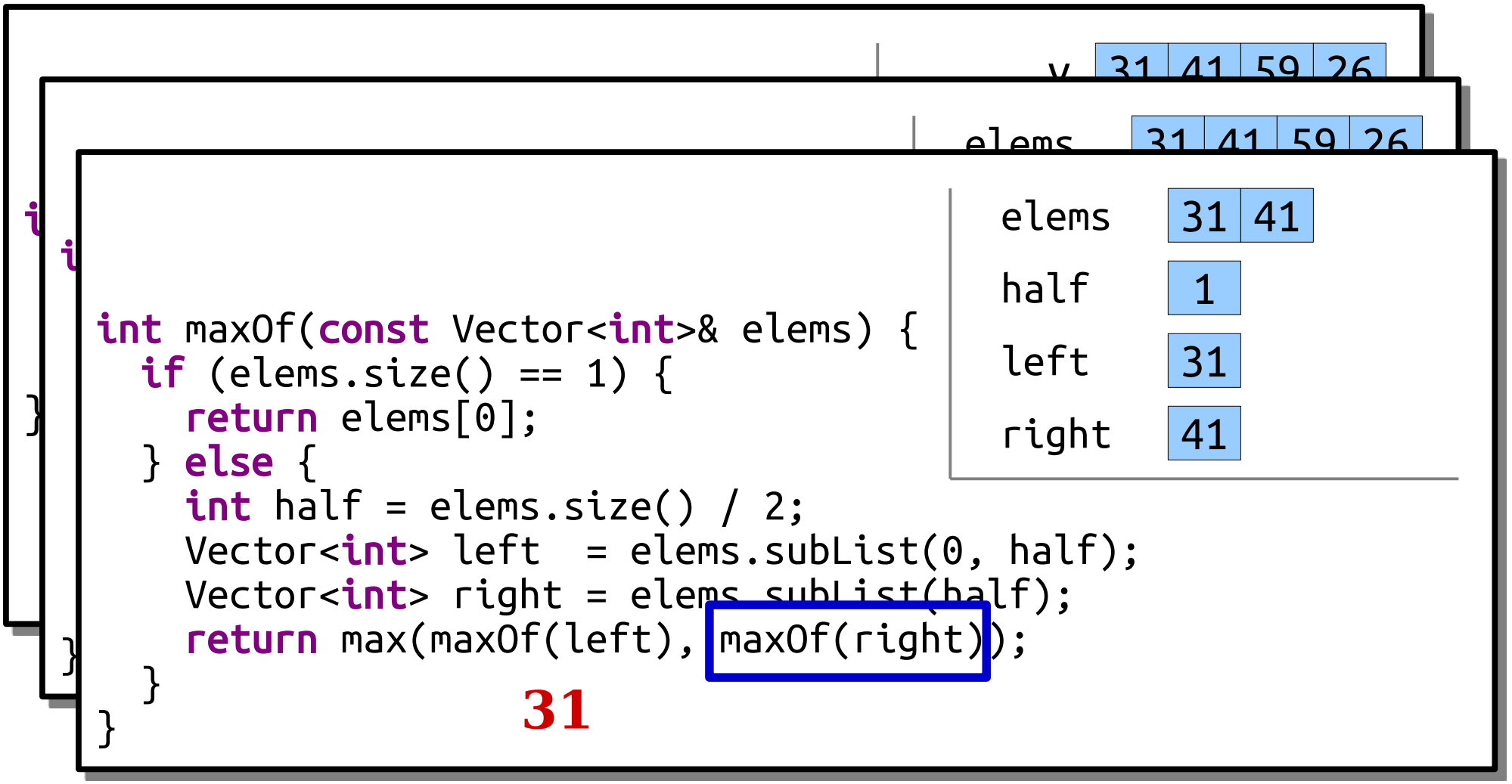
Tracing the Recursion



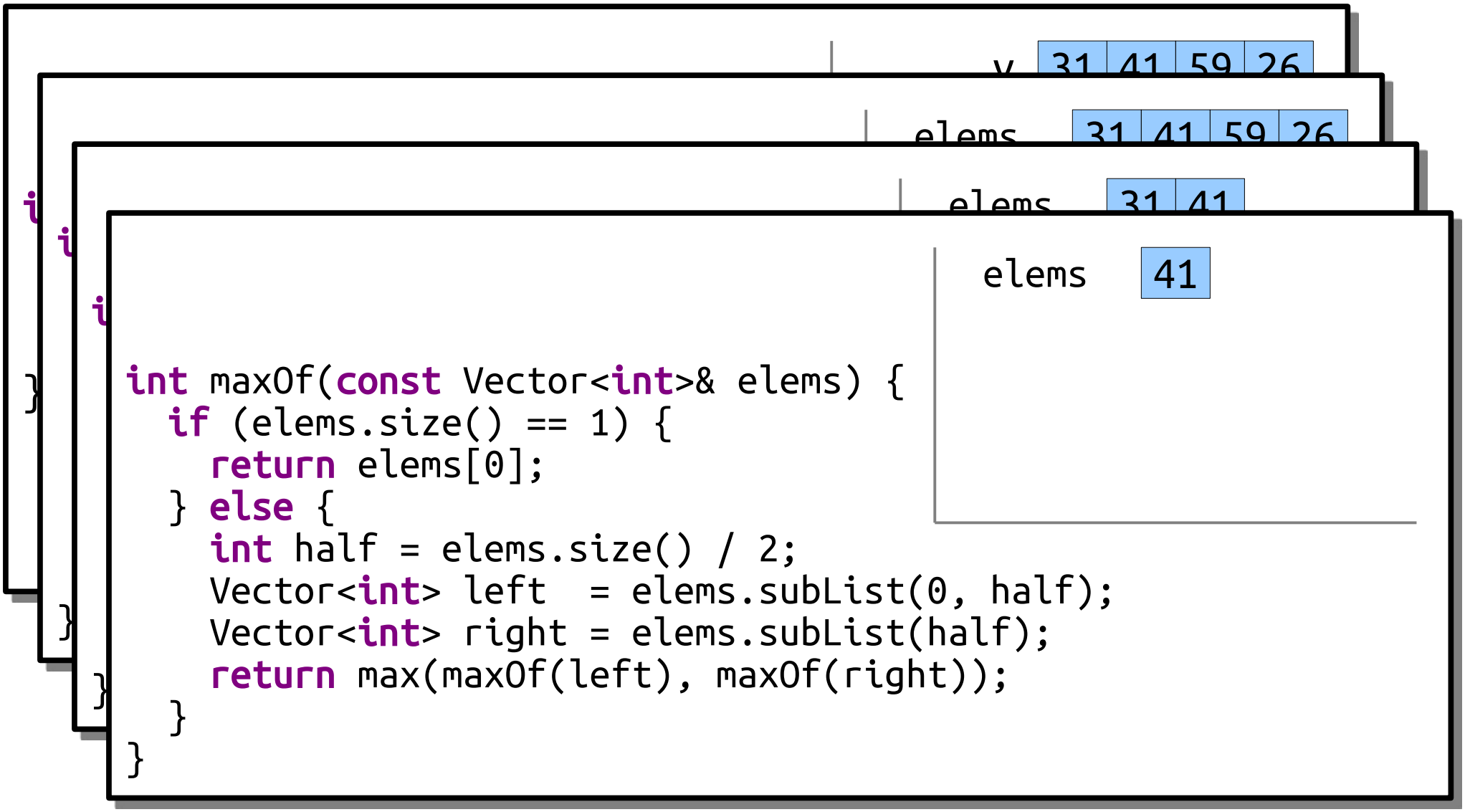
Tracing the Recursion



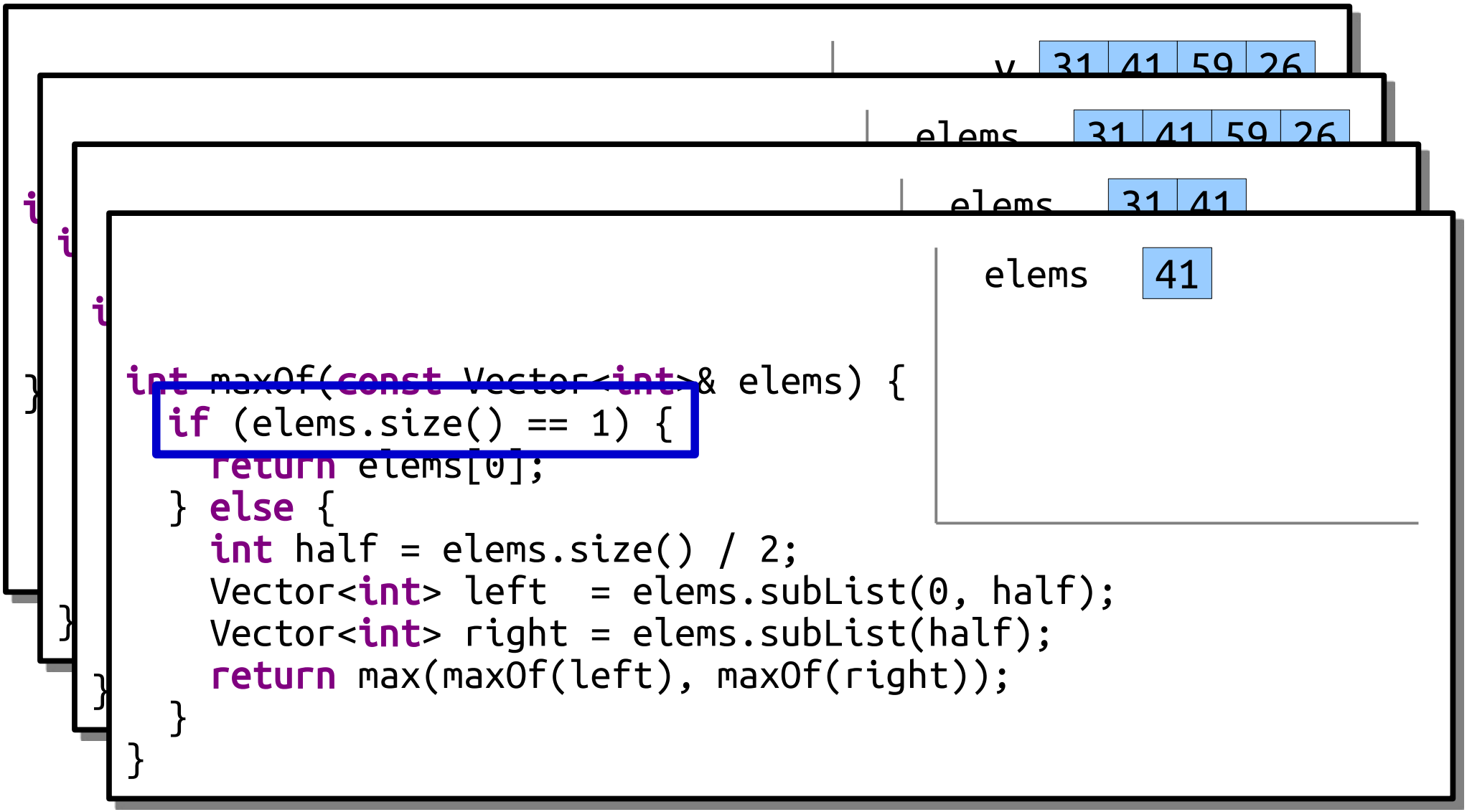
Tracing the Recursion



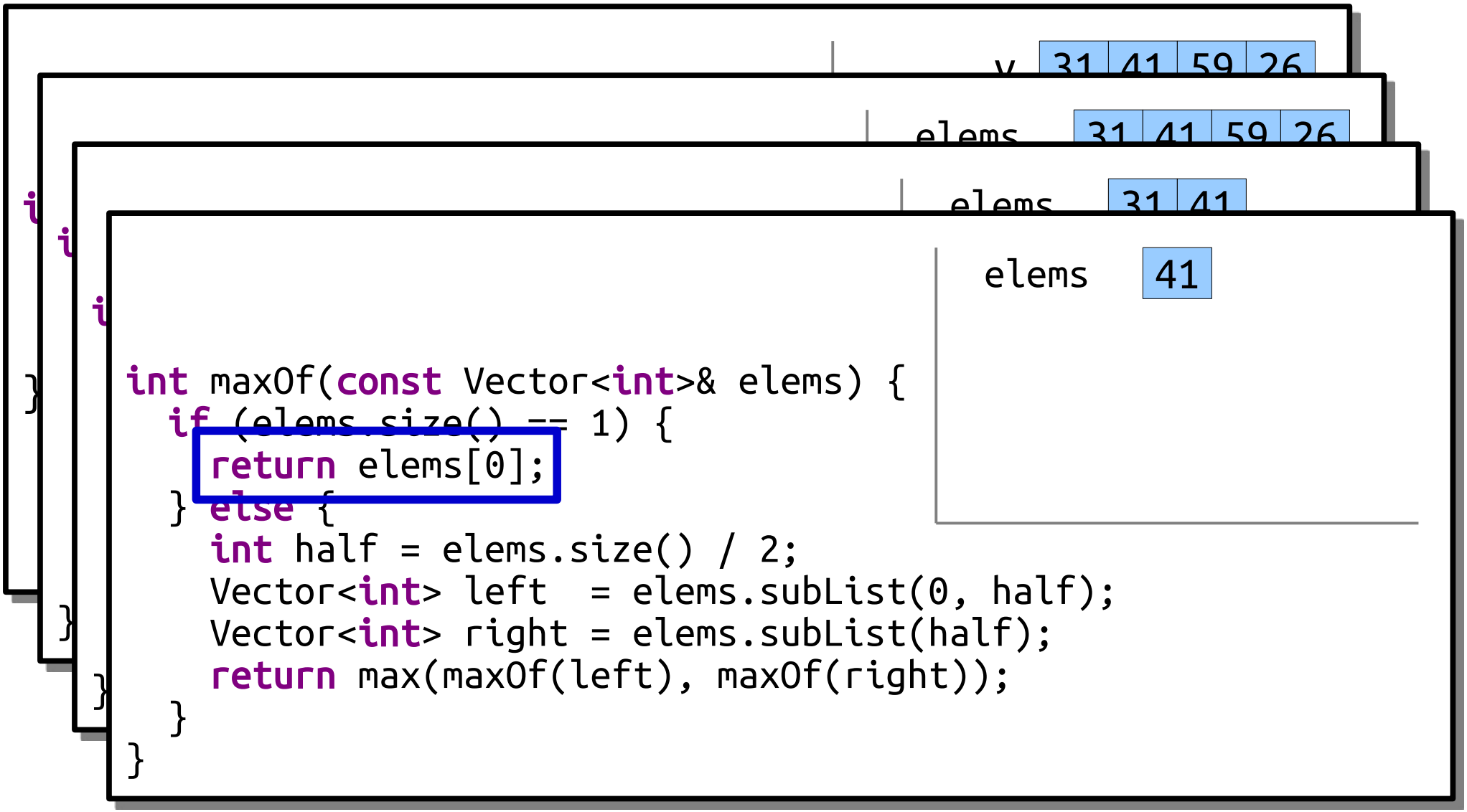
Tracing the Recursion



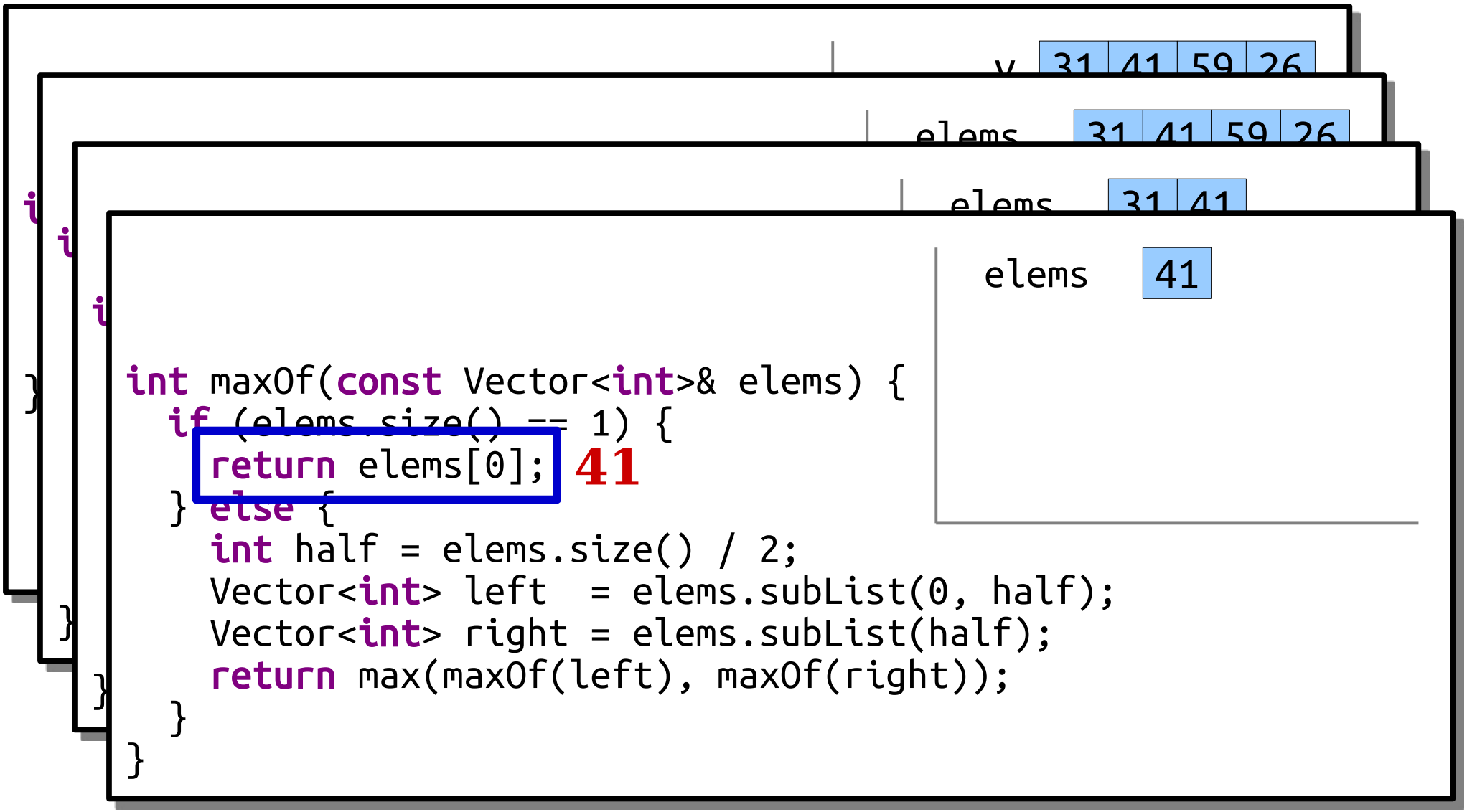
Tracing the Recursion



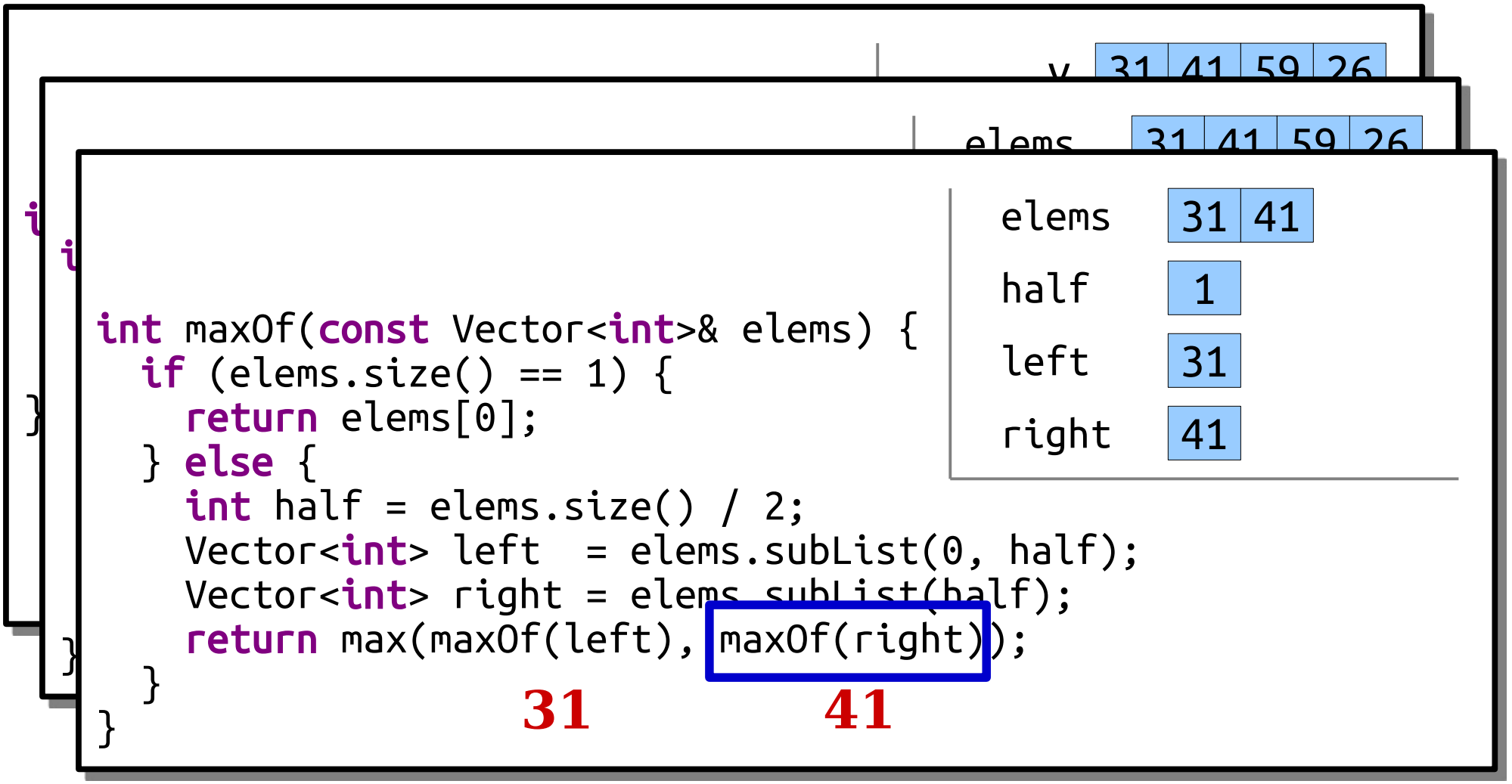
Tracing the Recursion



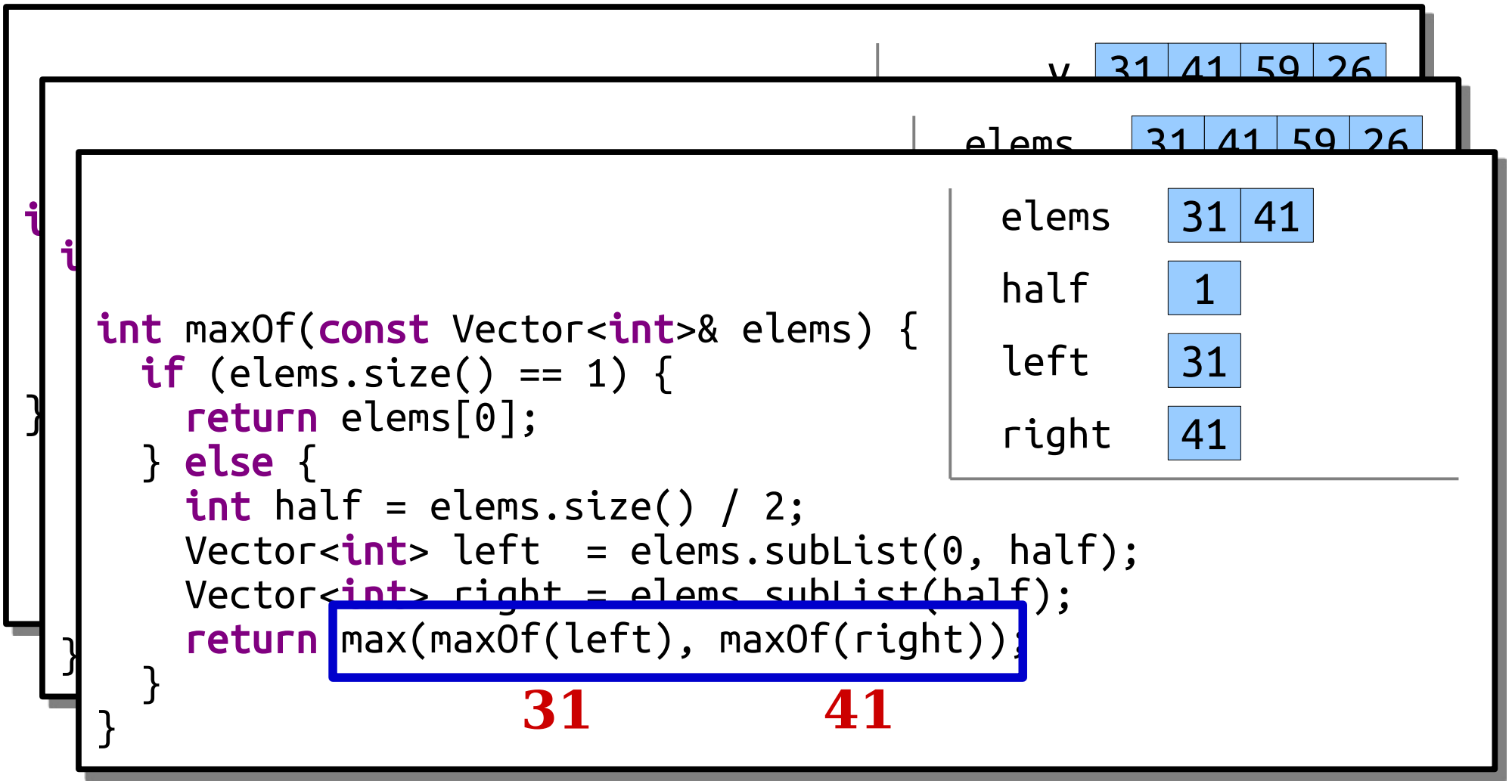
Tracing the Recursion



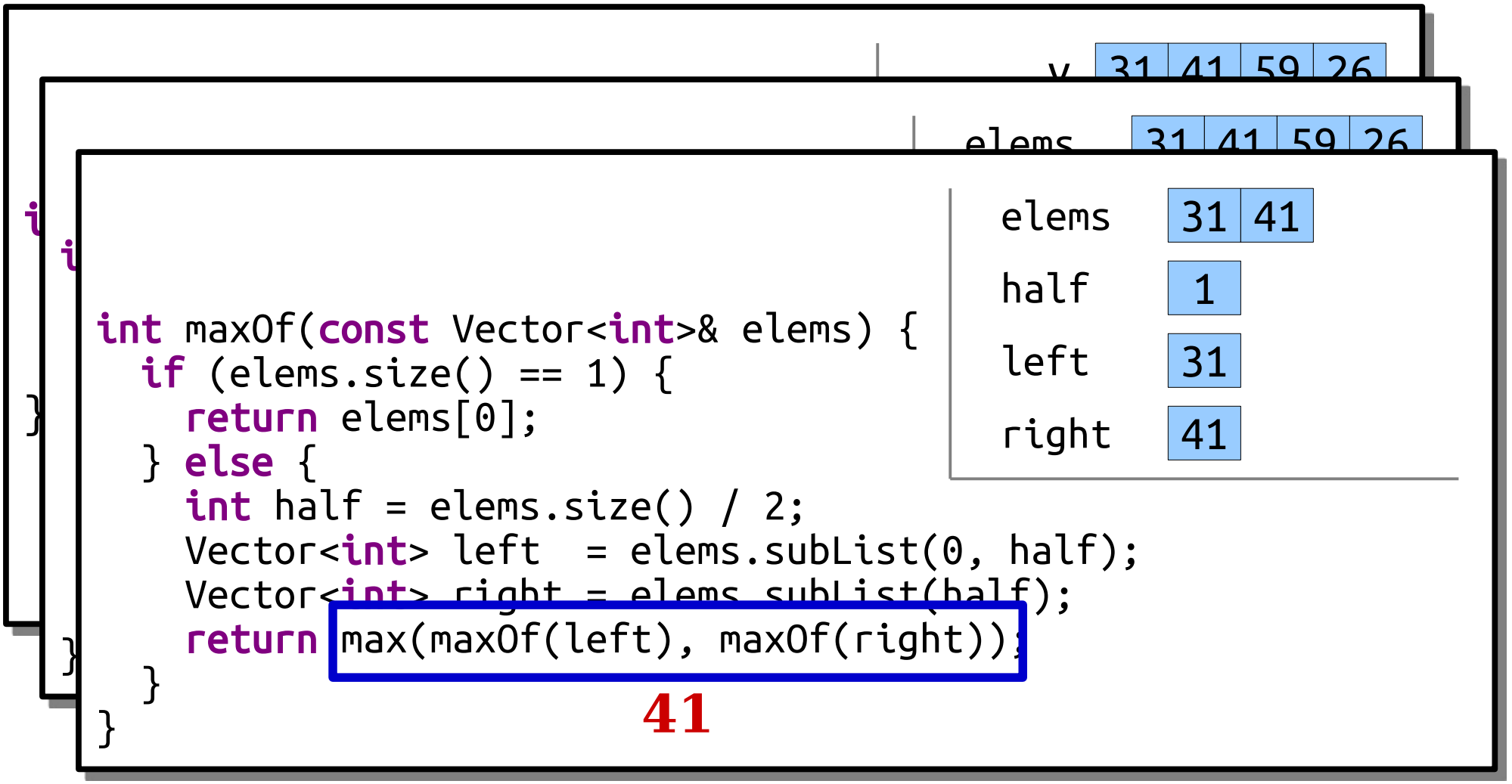
Tracing the Recursion



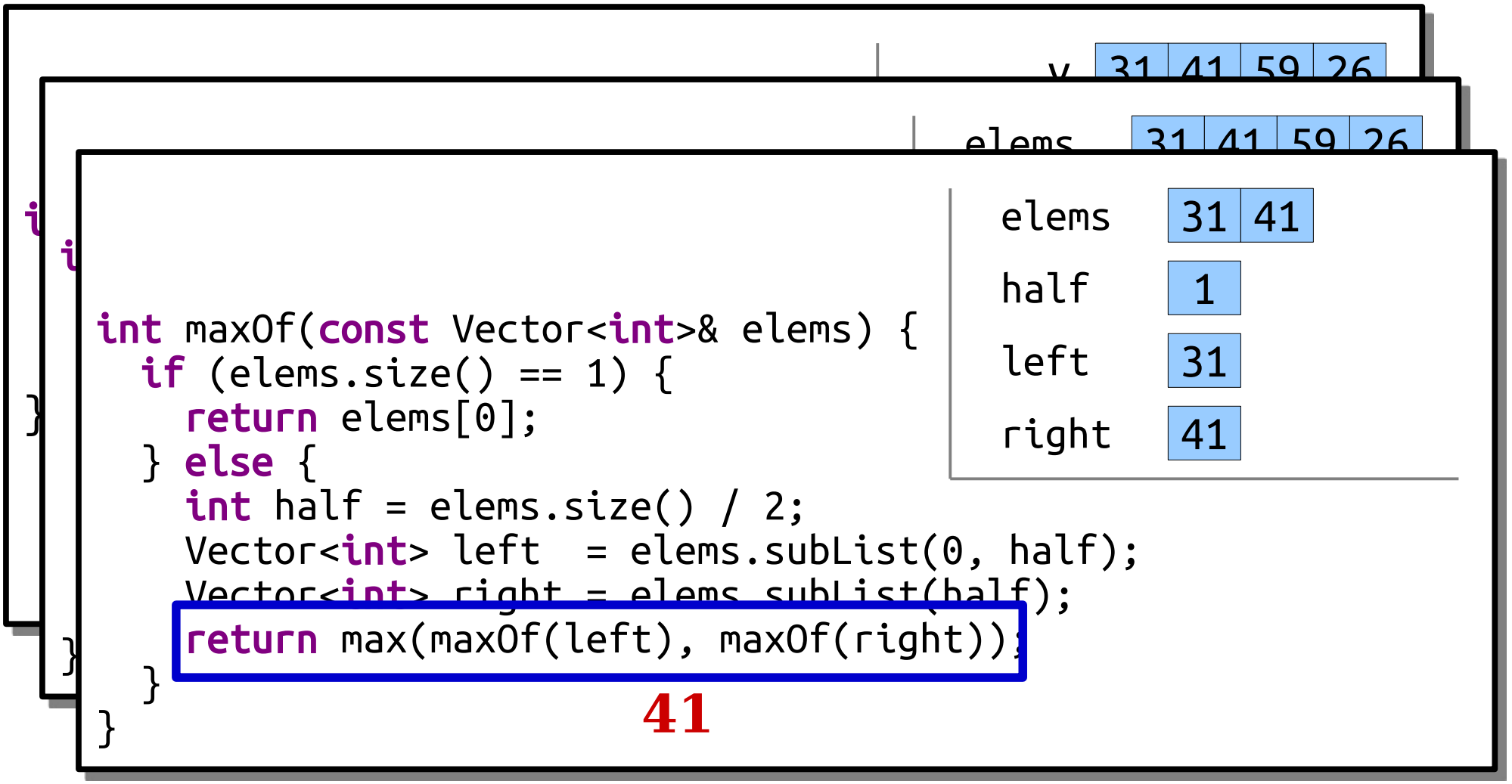
Tracing the Recursion



Tracing the Recursion



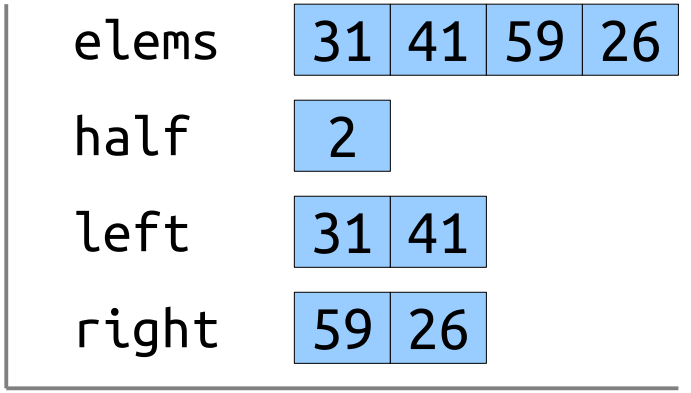
Tracing the Recursion



Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int half = elems.size() / 2;
    Vector<int> left  = elems.subList(0, half);
    Vector<int> right = elems.subList(half);
    return max(maxOf(left), maxOf(right));
  }
}
```

41



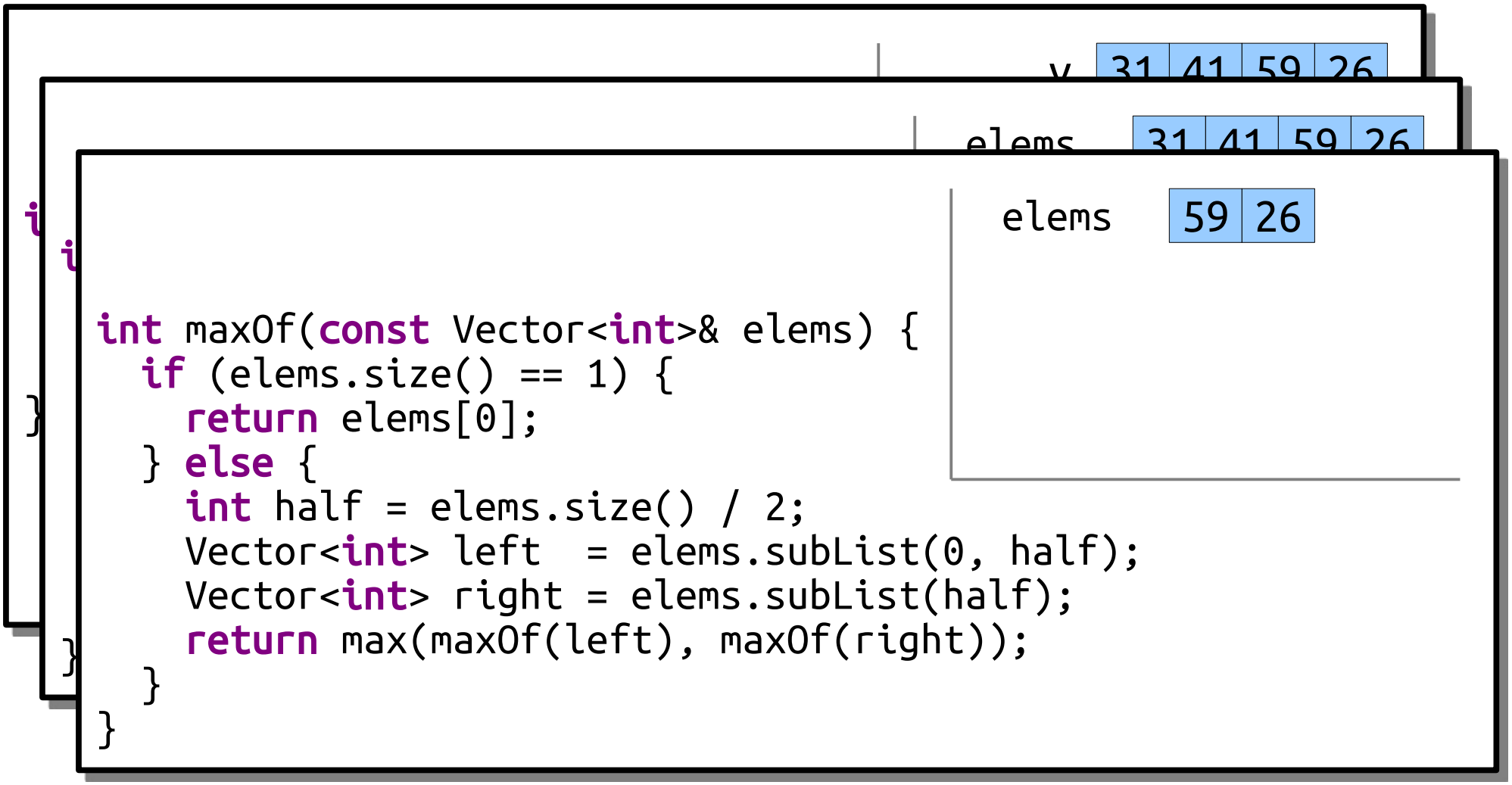
Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int half = elems.size() / 2;
    Vector<int> left  = elems.subList(0, half);
    Vector<int> right = elems.subList(half);
    return max(maxOf(left), maxOf(right));
  }
}
```

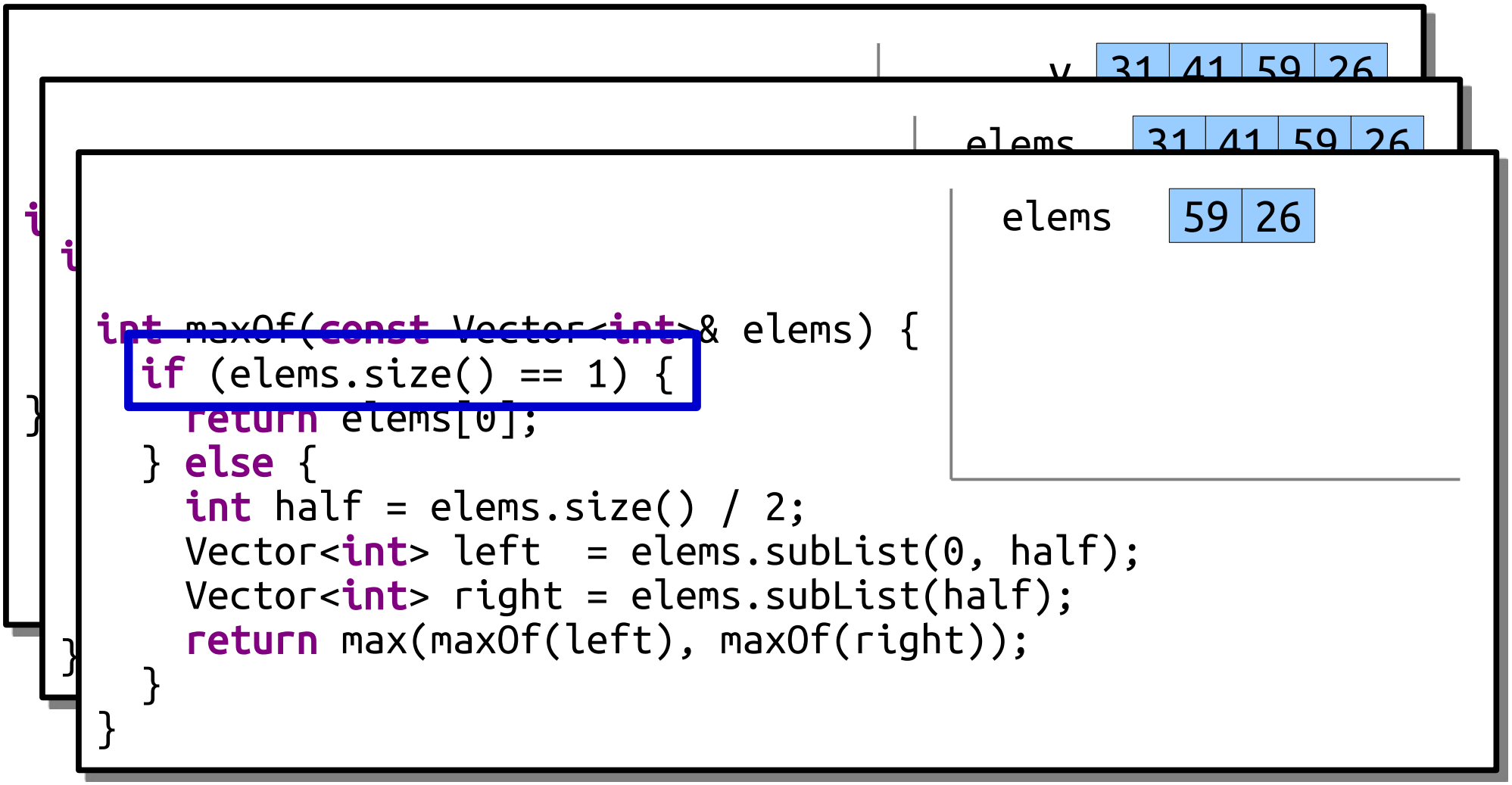
elems	31	41	59	26
half	2			
left	31	41		
right	59	26		

41

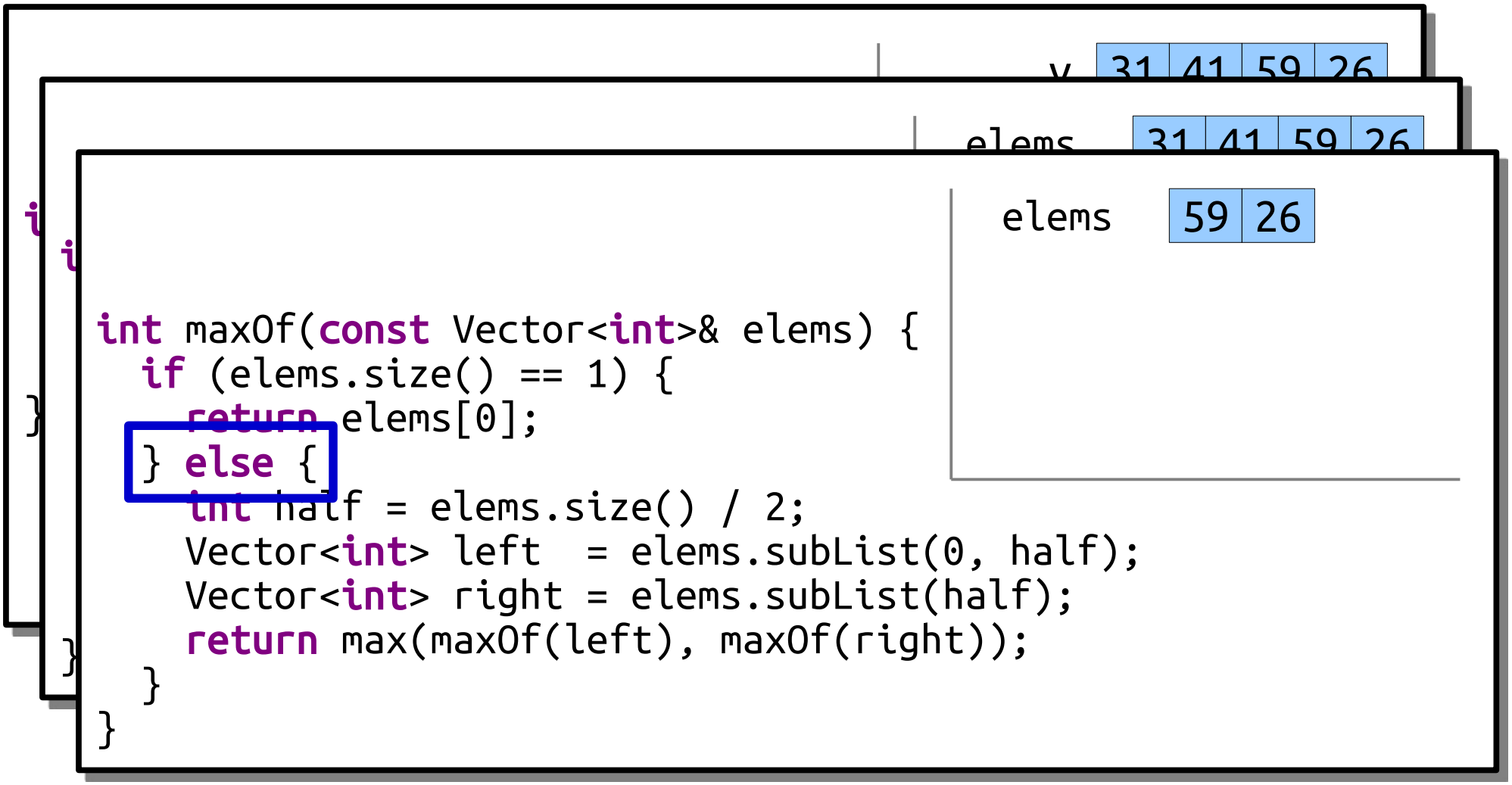
Tracing the Recursion



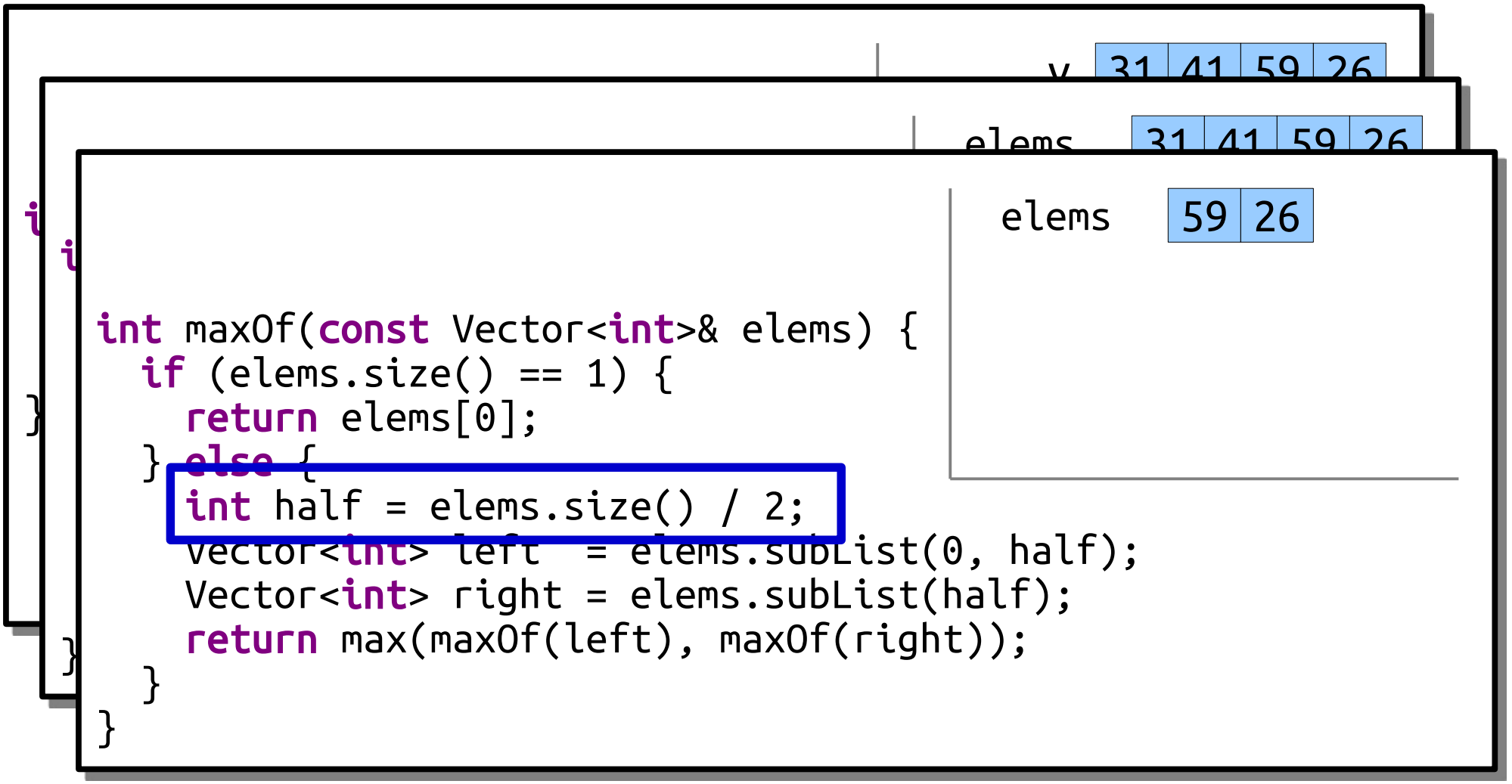
Tracing the Recursion



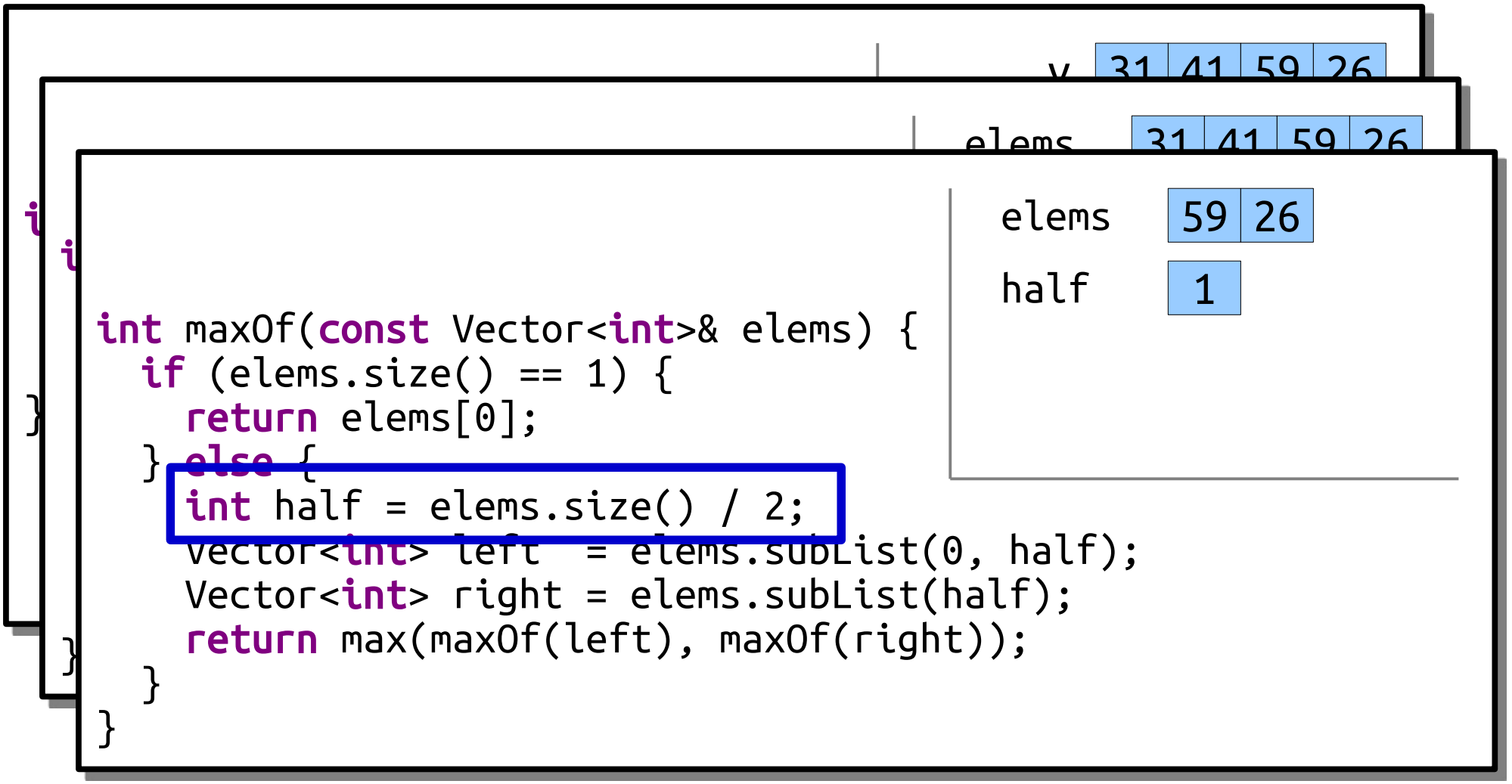
Tracing the Recursion



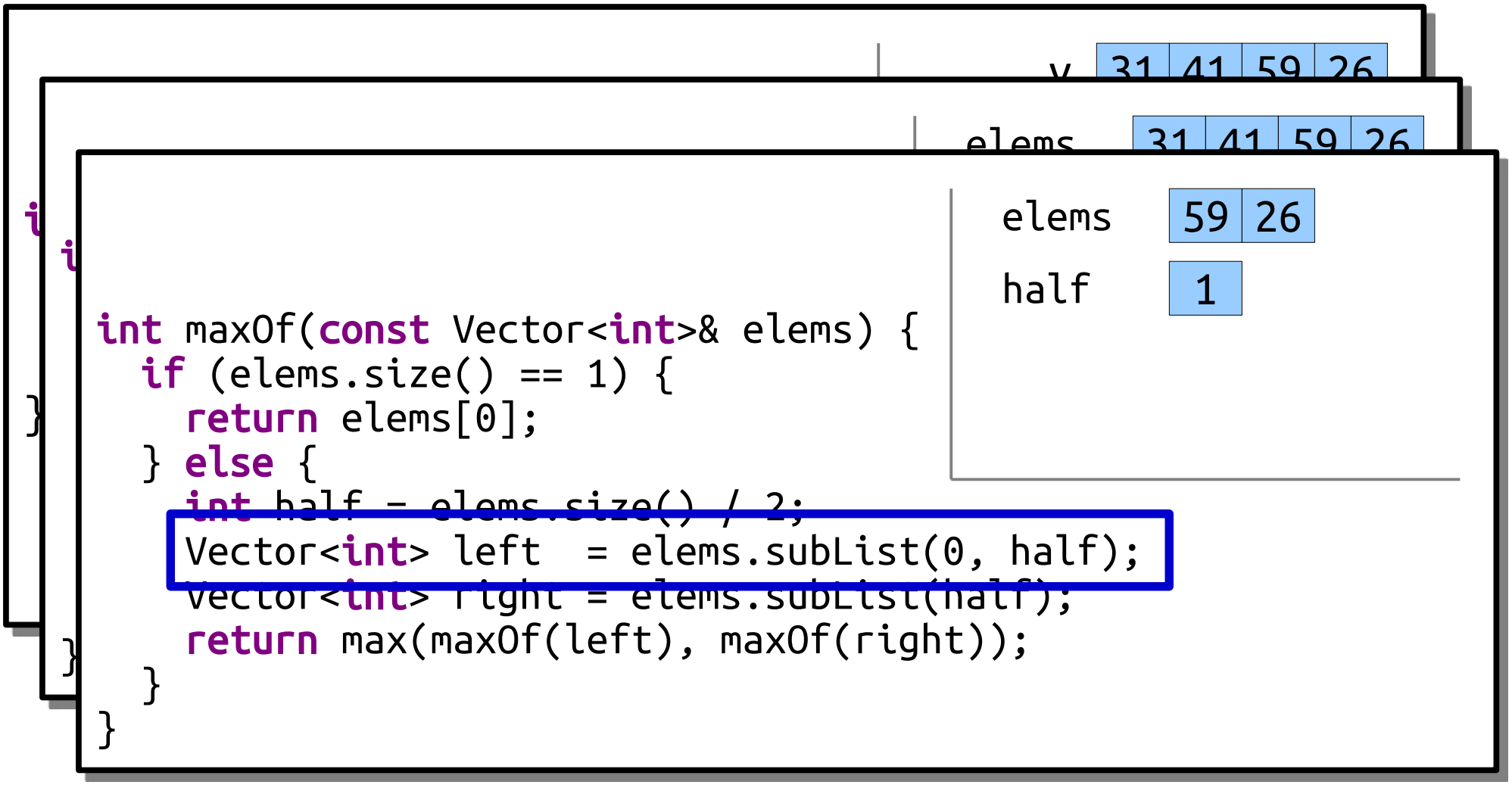
Tracing the Recursion



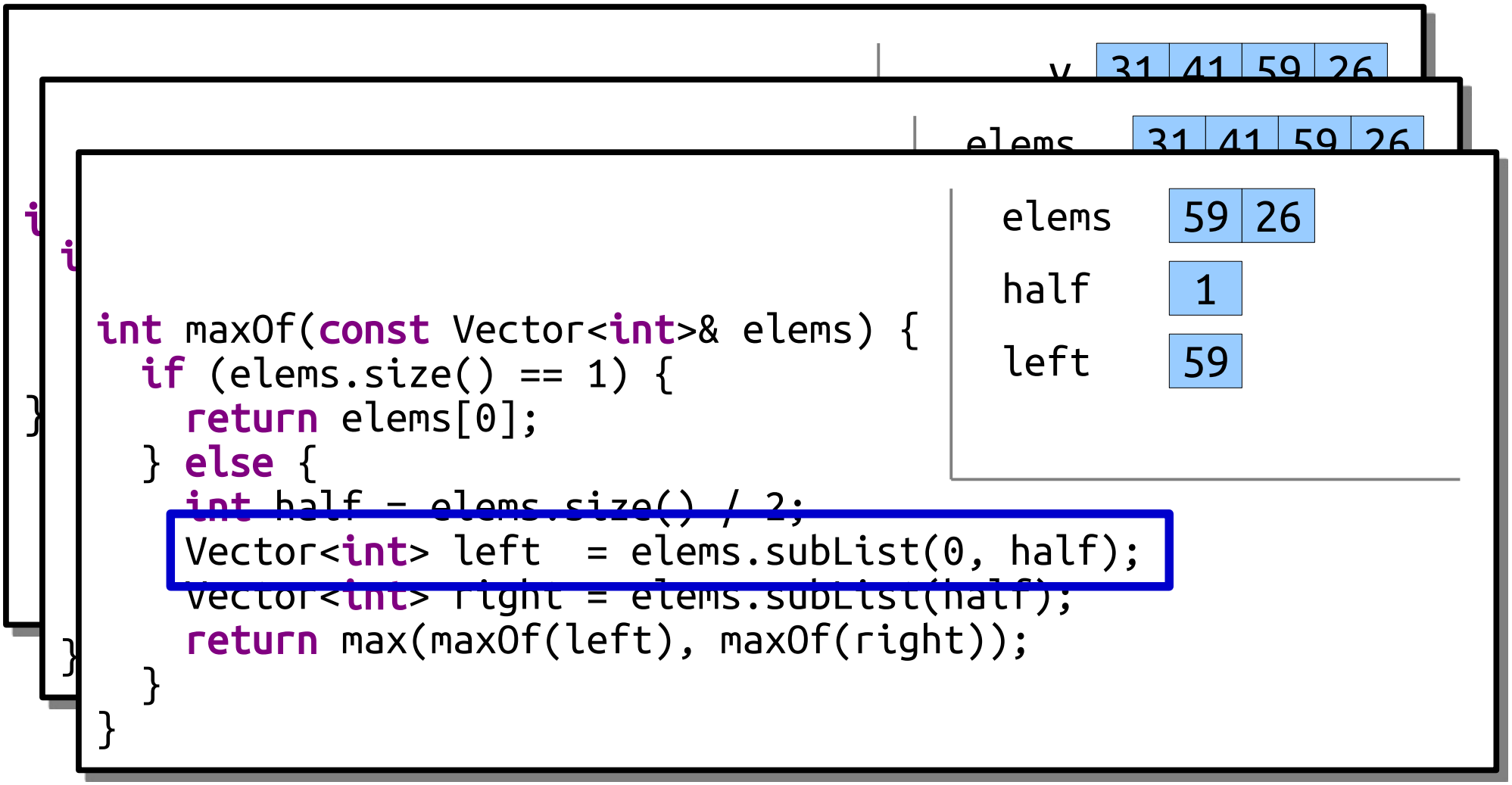
Tracing the Recursion



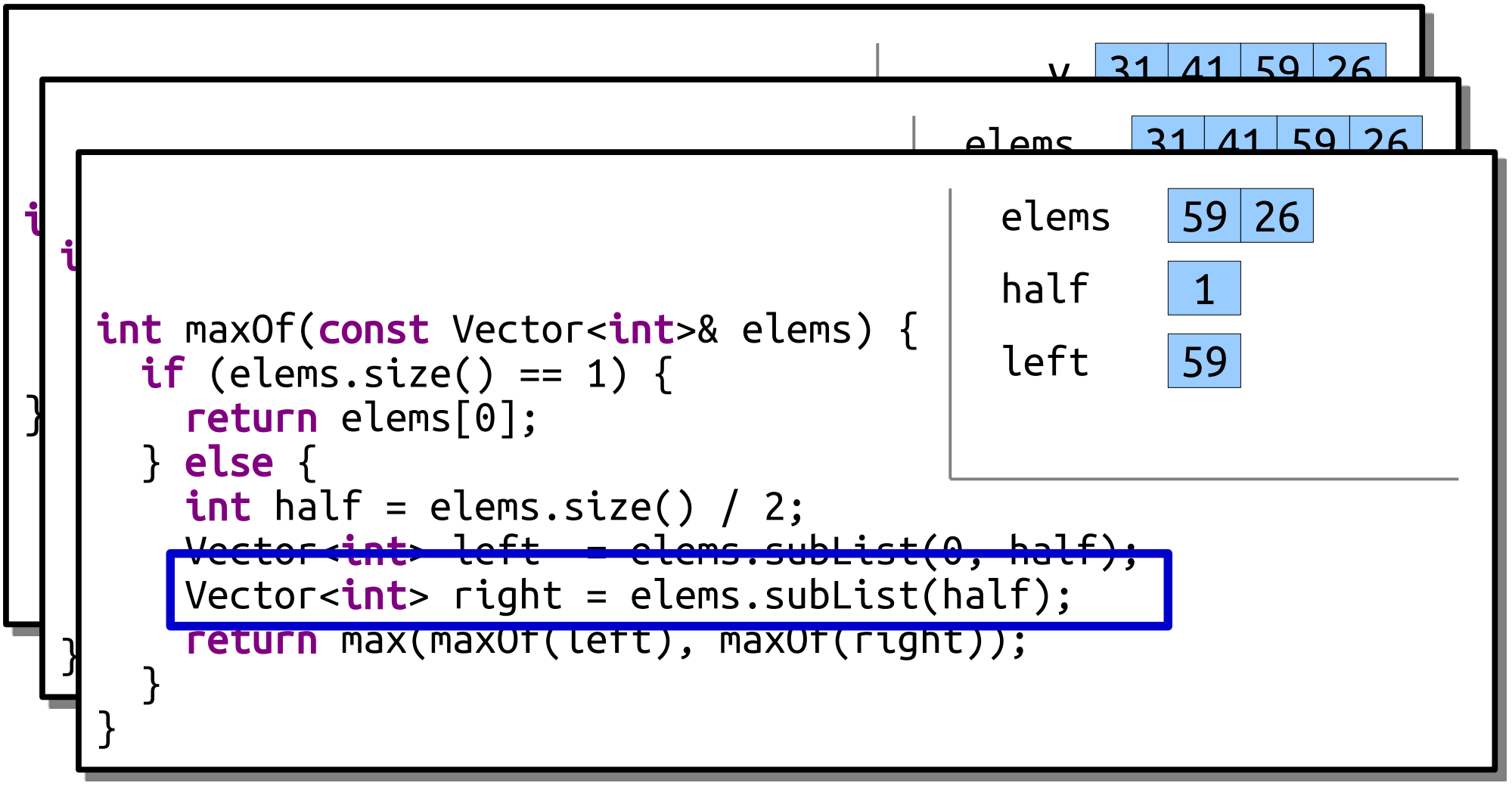
Tracing the Recursion



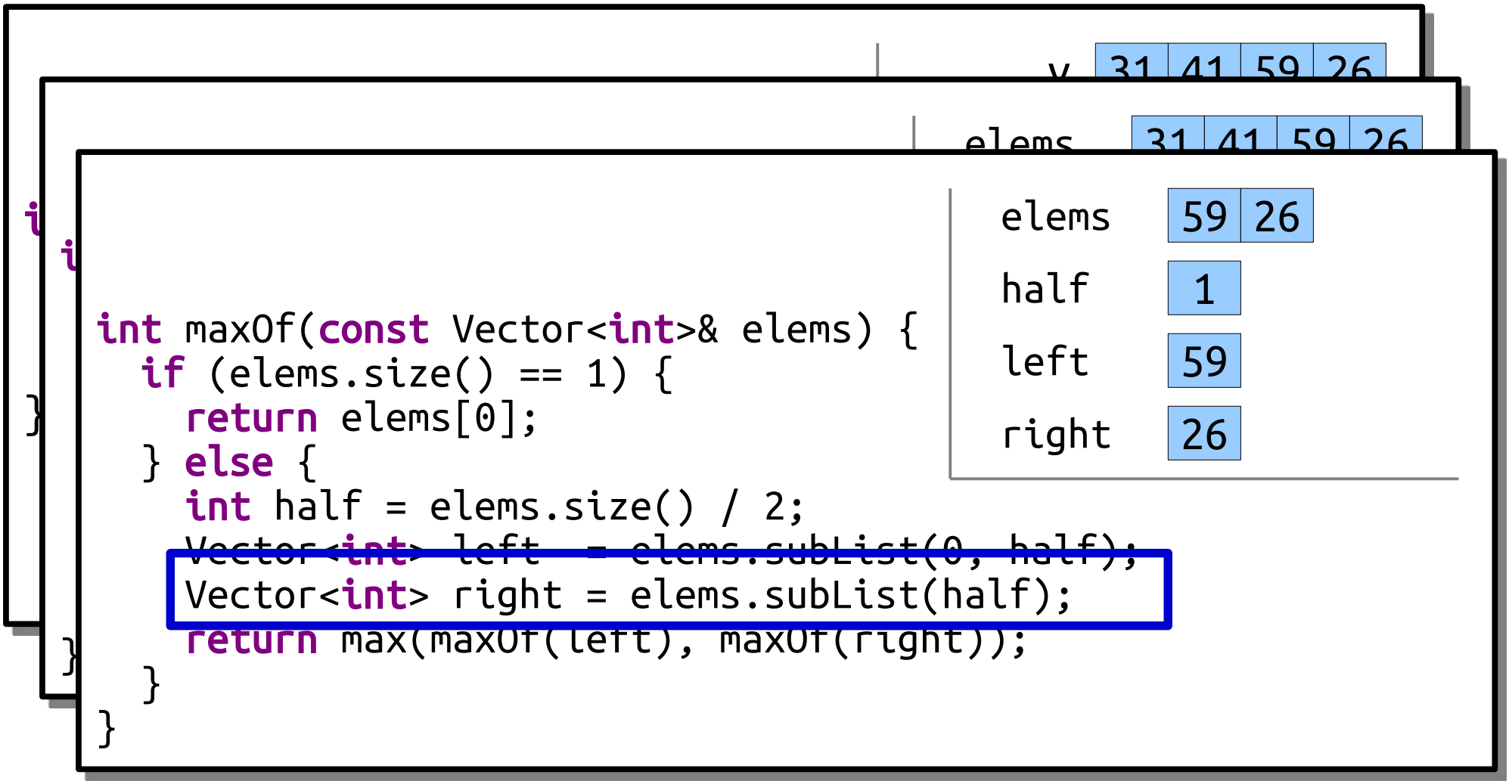
Tracing the Recursion



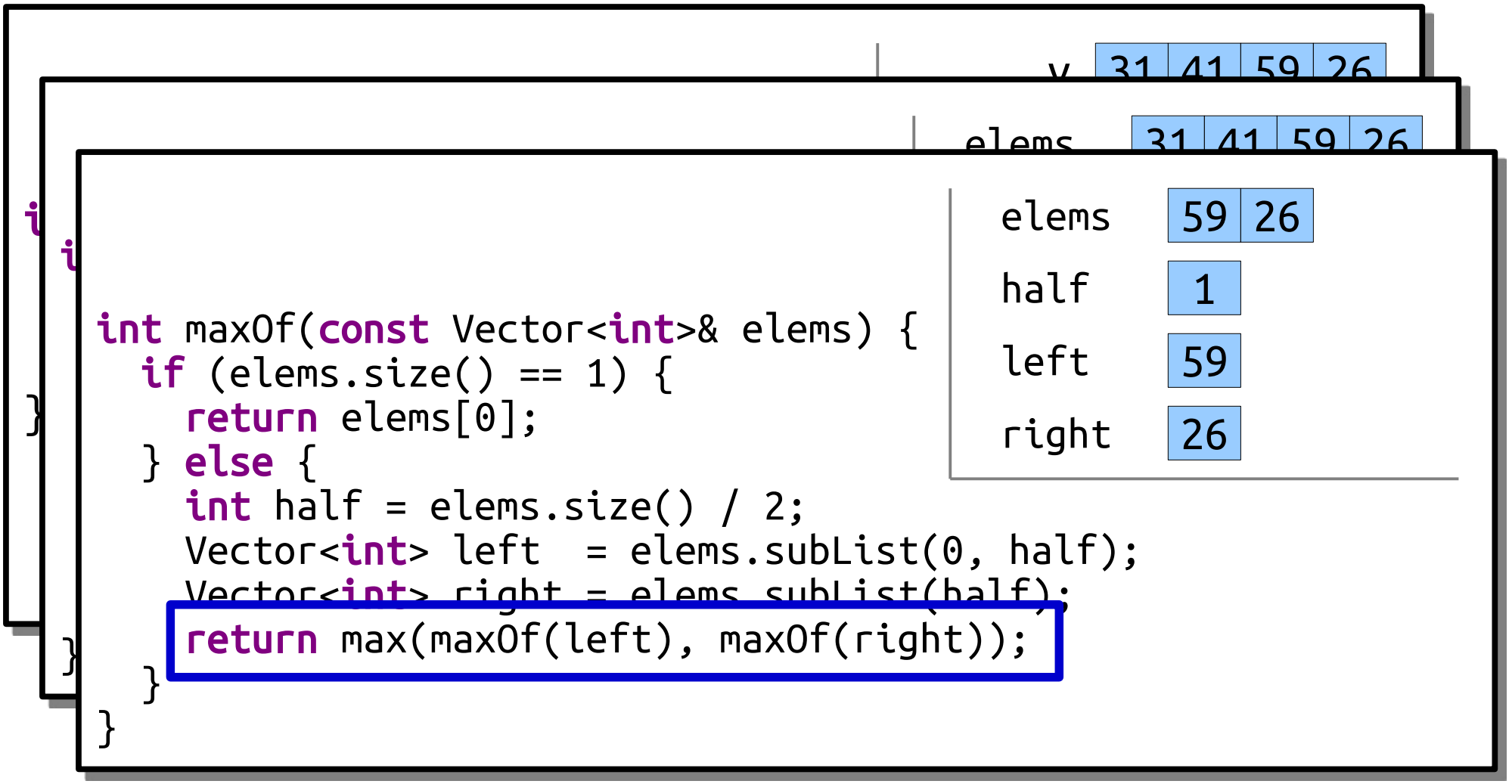
Tracing the Recursion



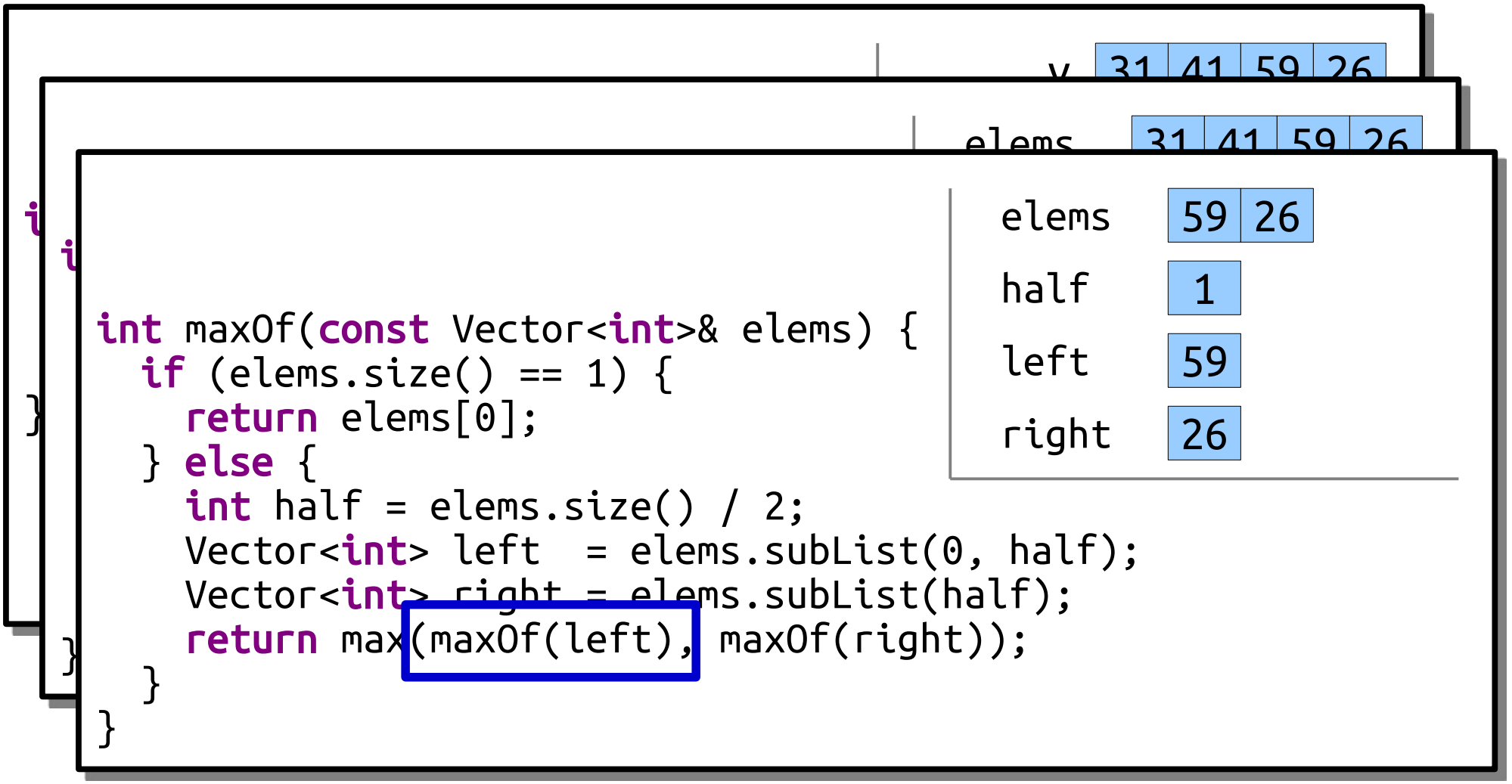
Tracing the Recursion



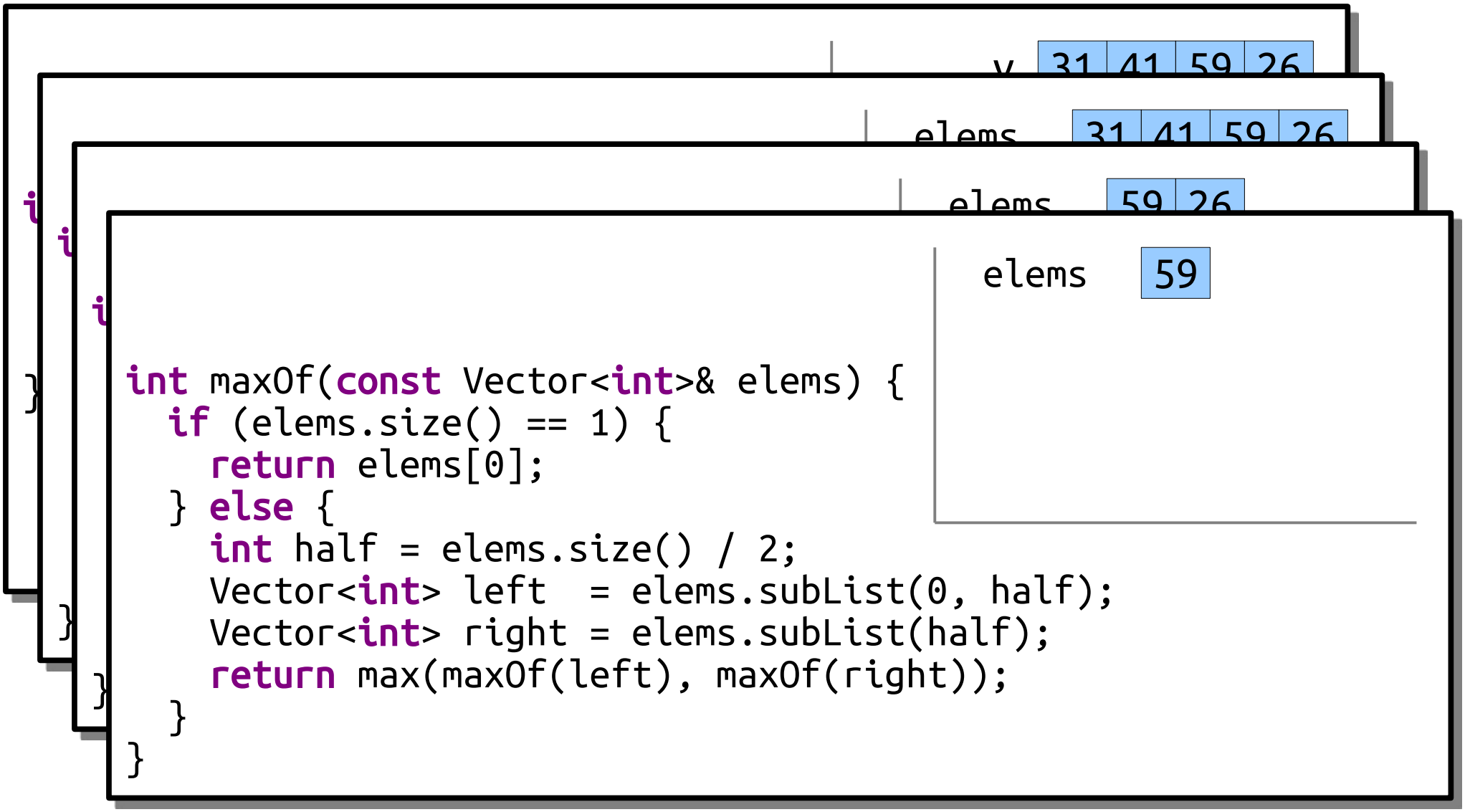
Tracing the Recursion



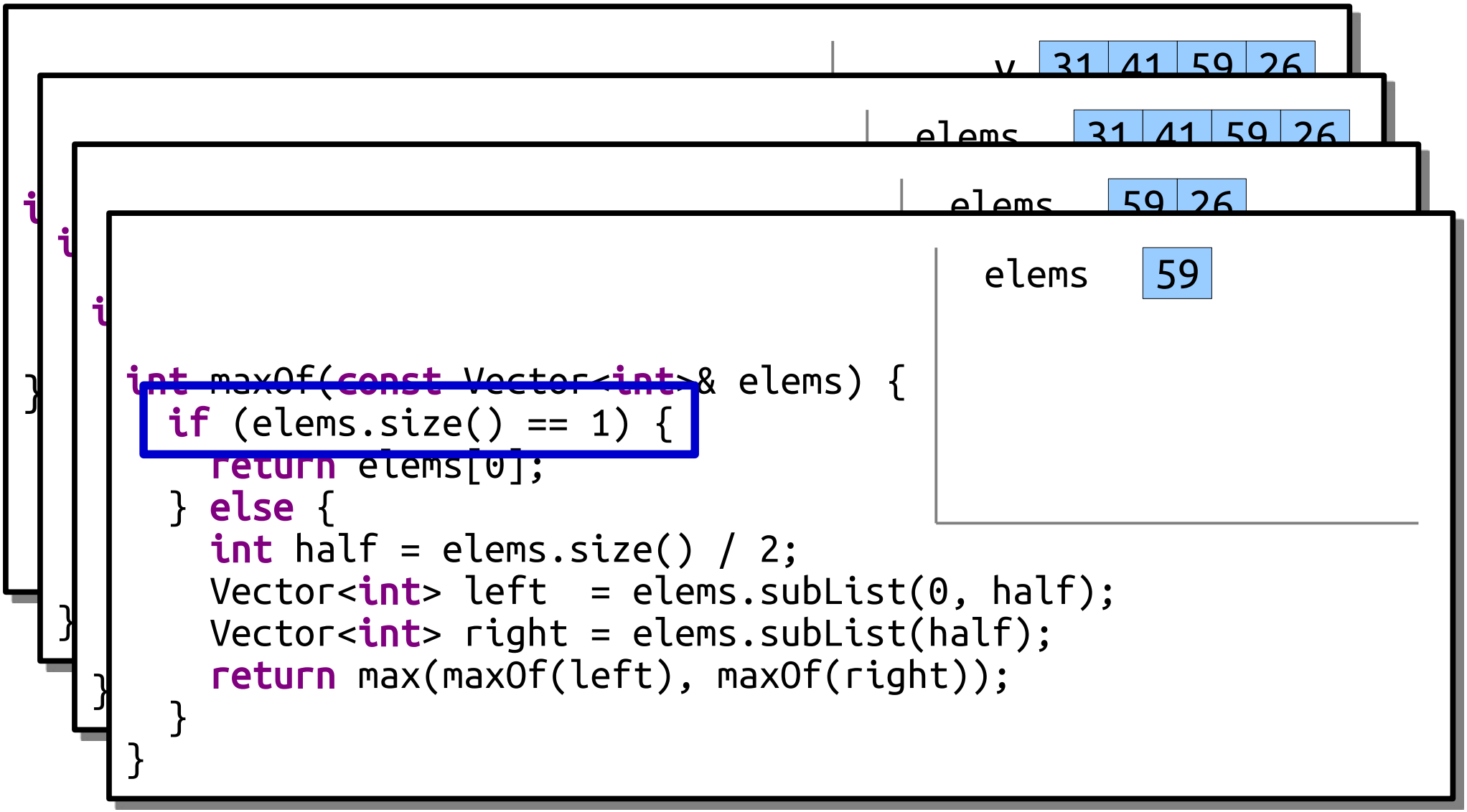
Tracing the Recursion



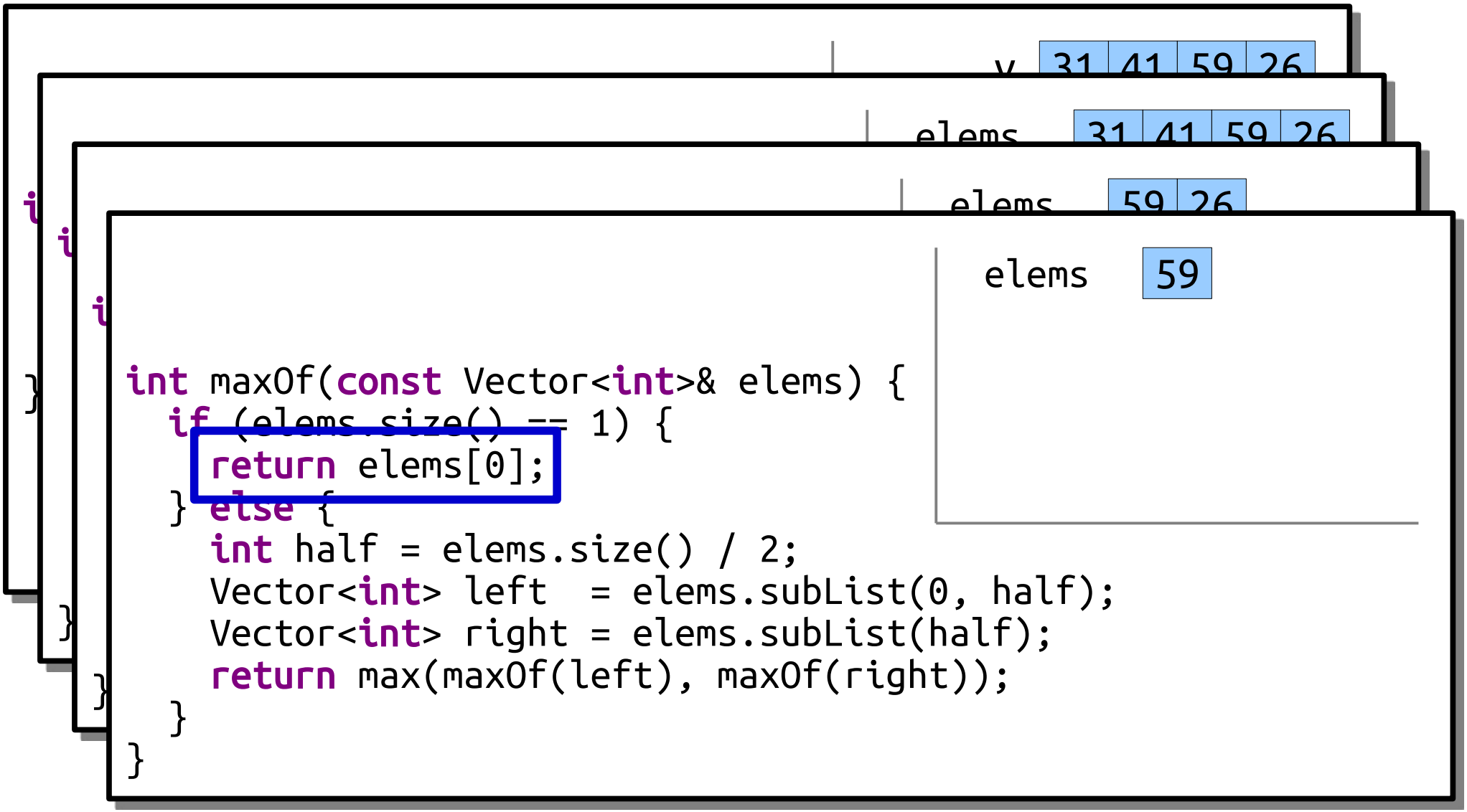
Tracing the Recursion



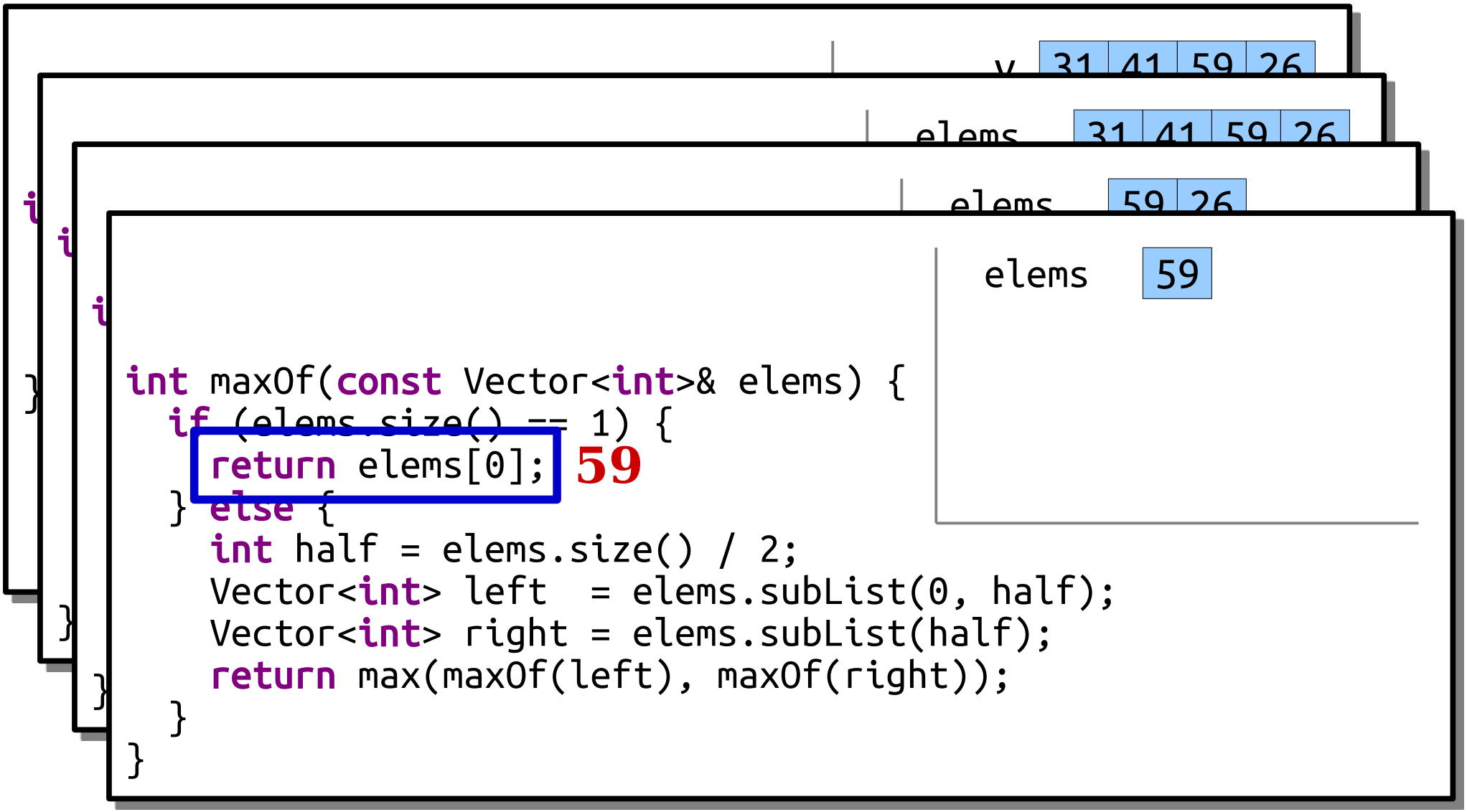
Tracing the Recursion



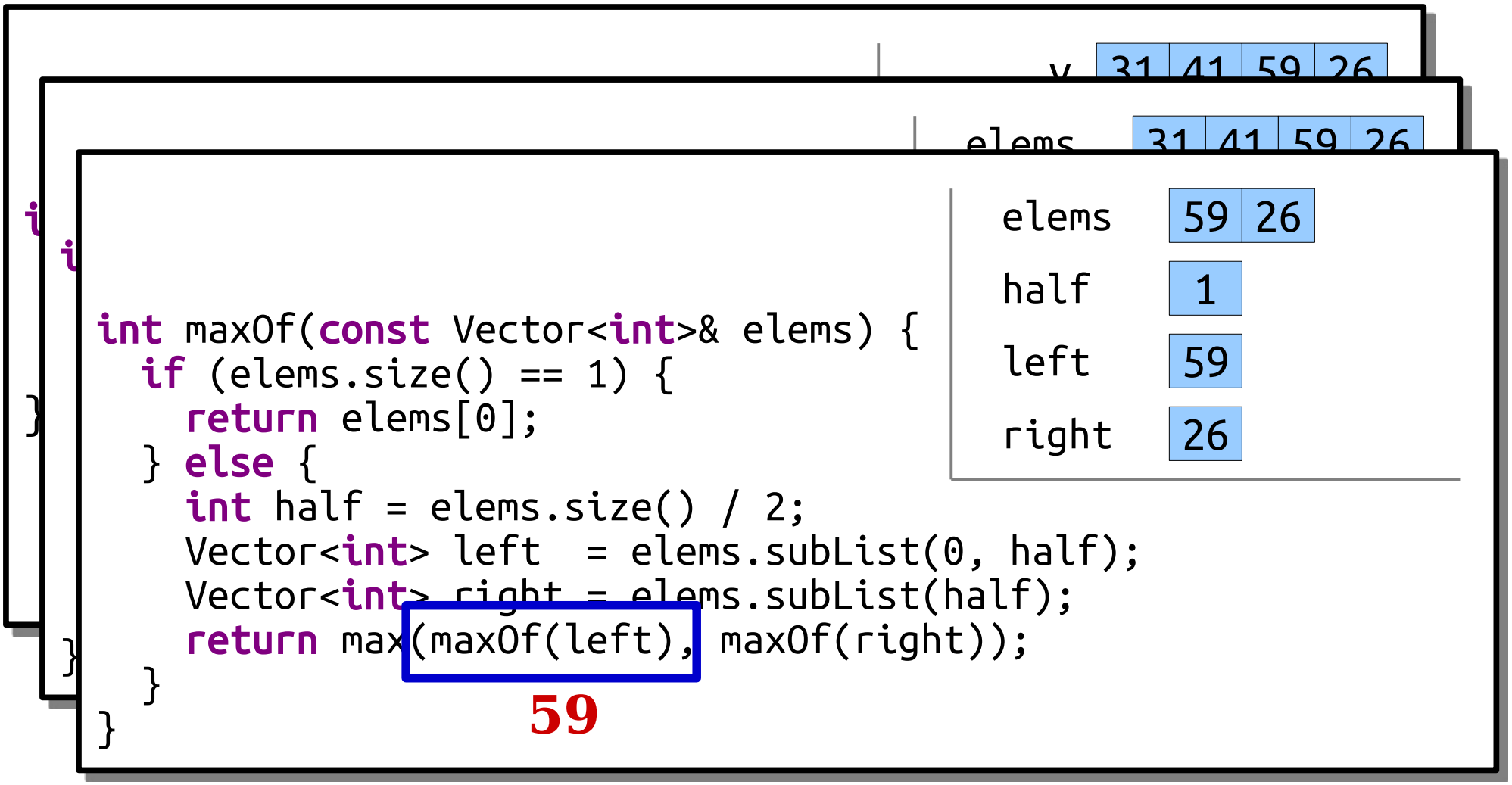
Tracing the Recursion



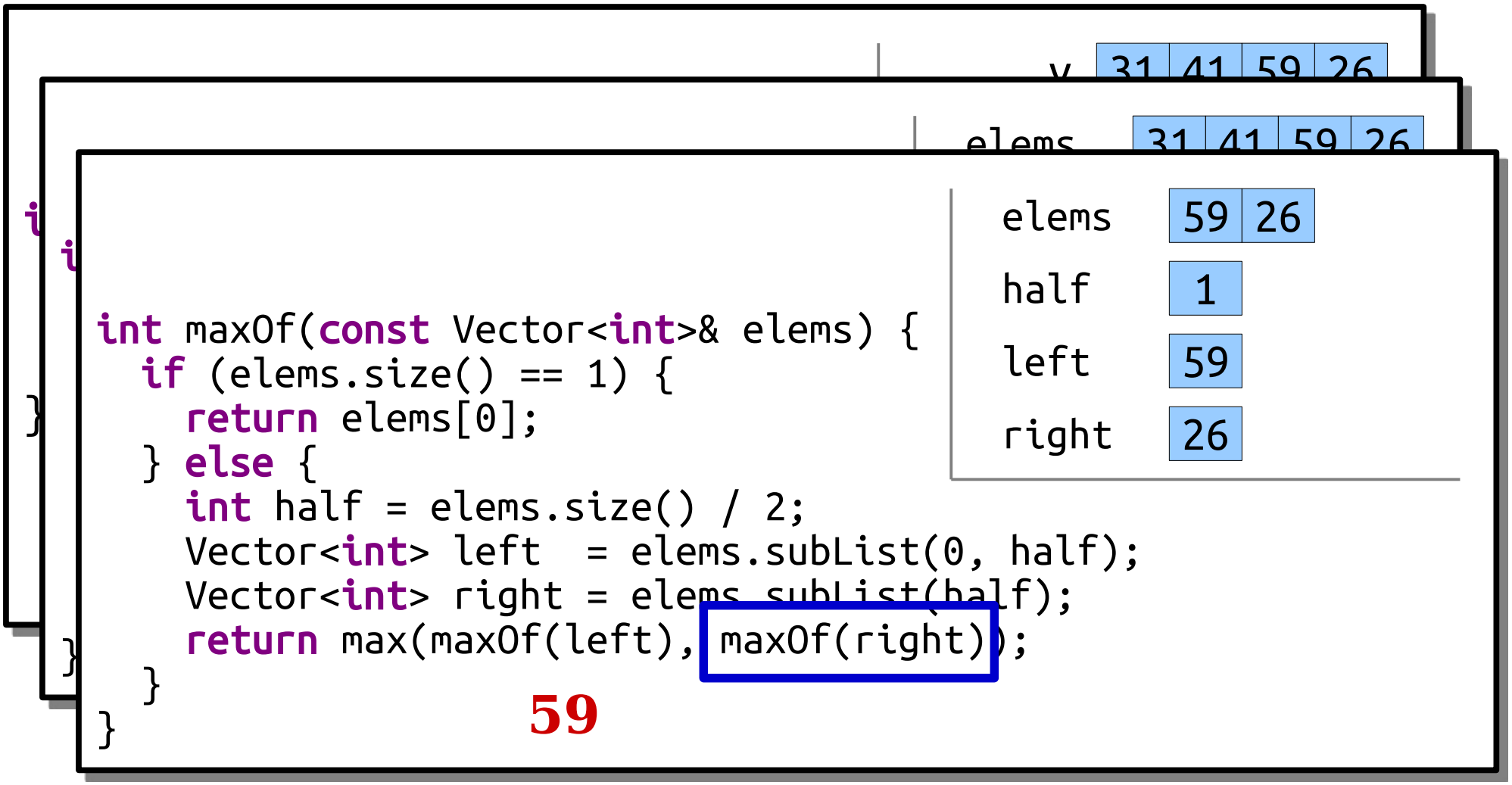
Tracing the Recursion



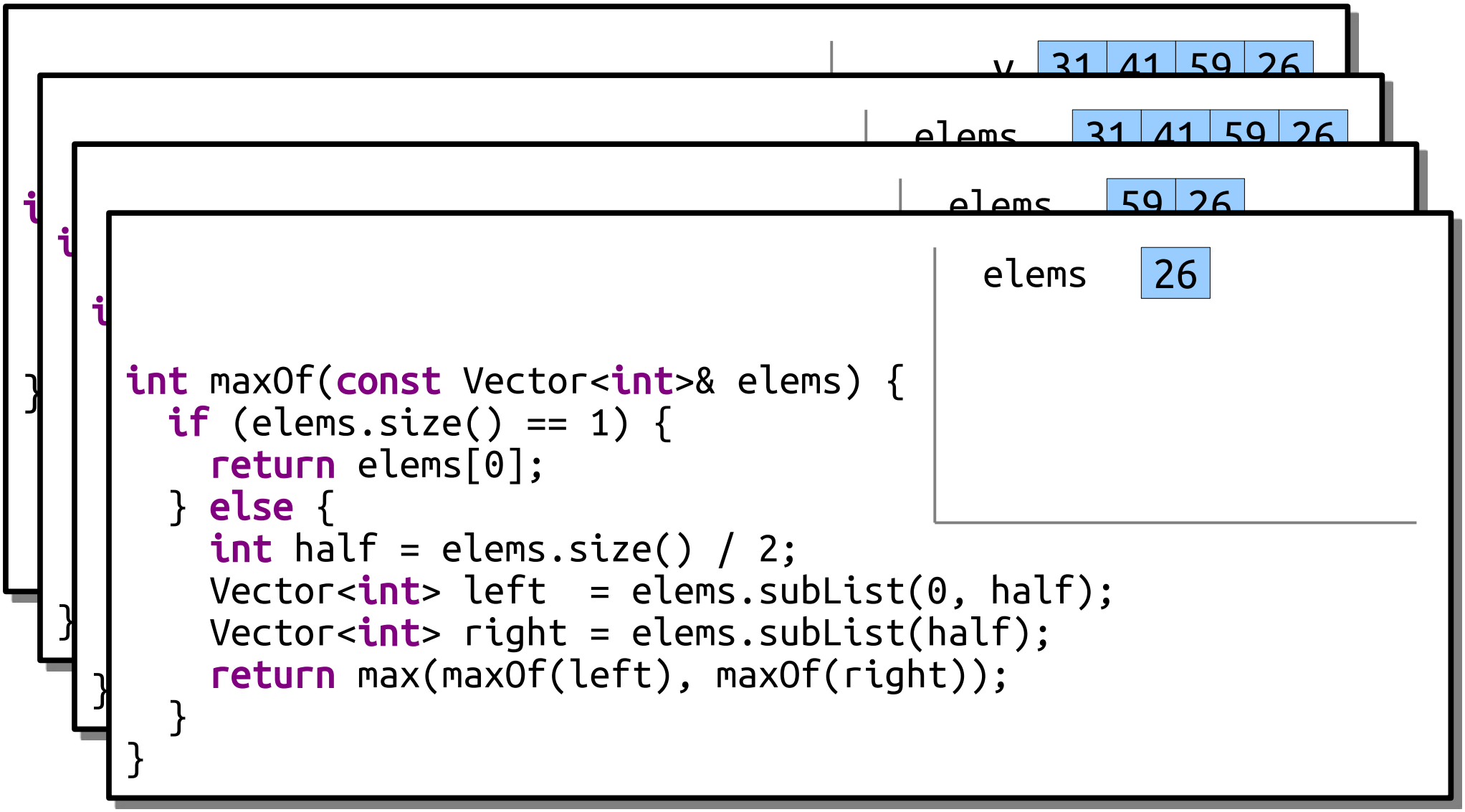
Tracing the Recursion



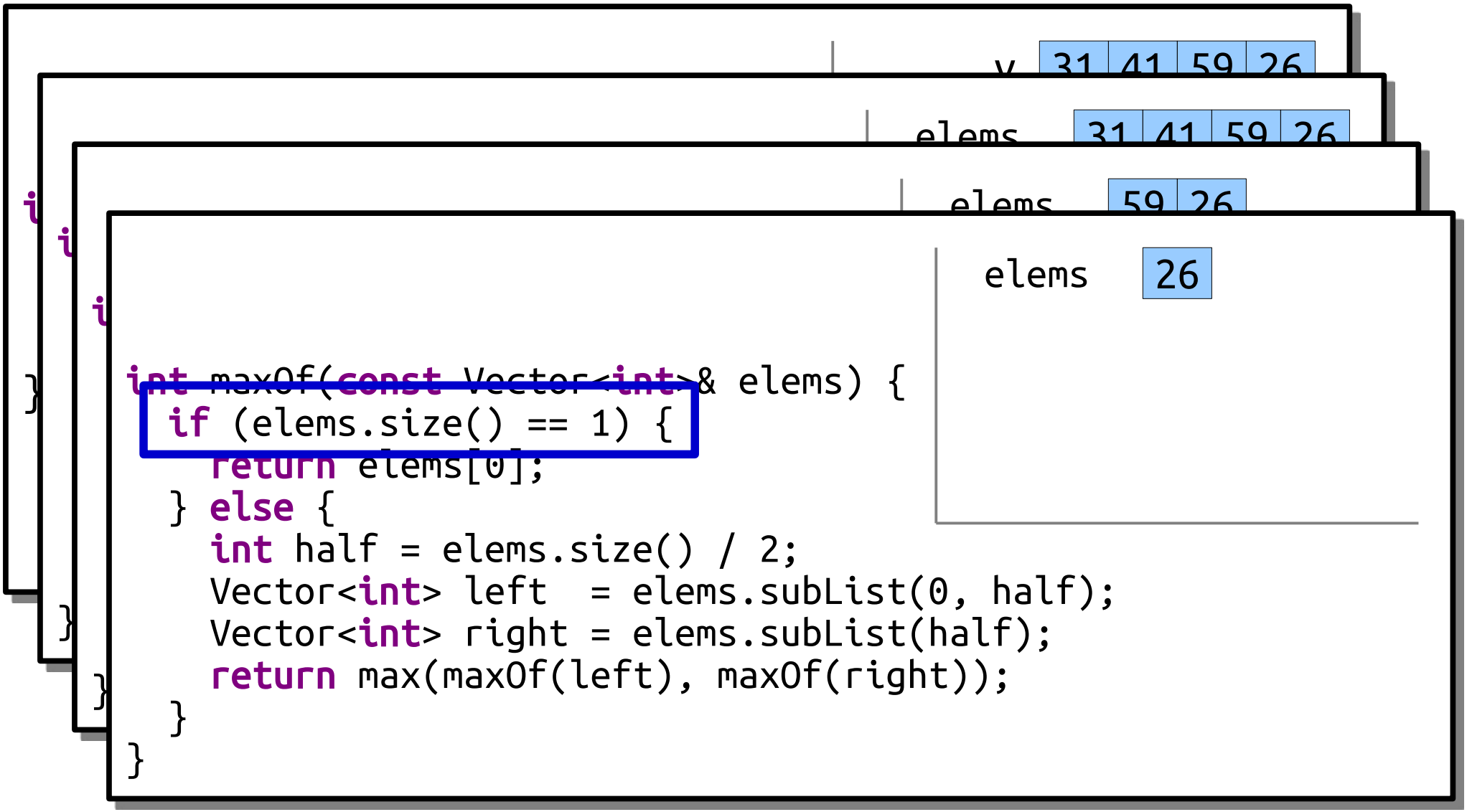
Tracing the Recursion



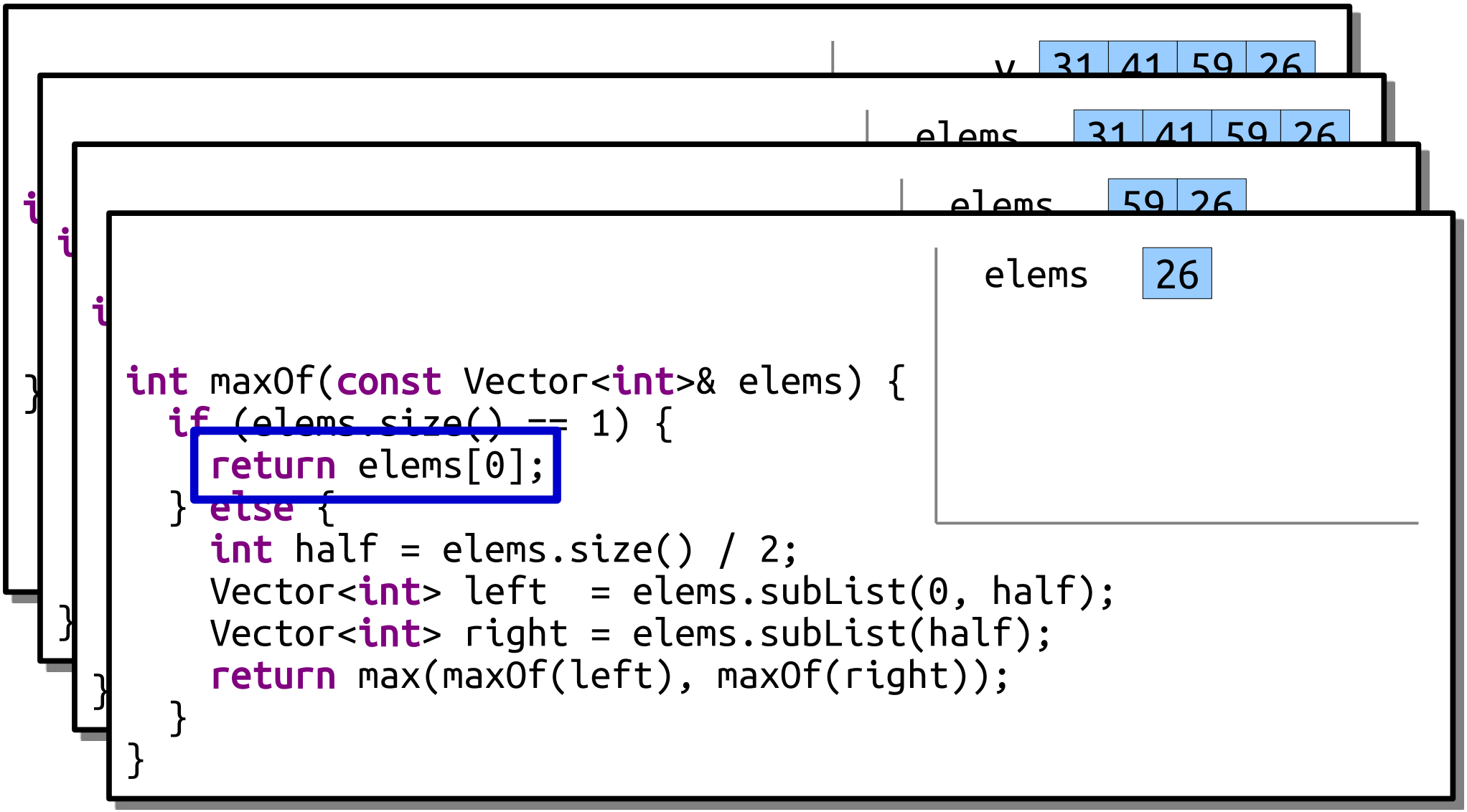
Tracing the Recursion



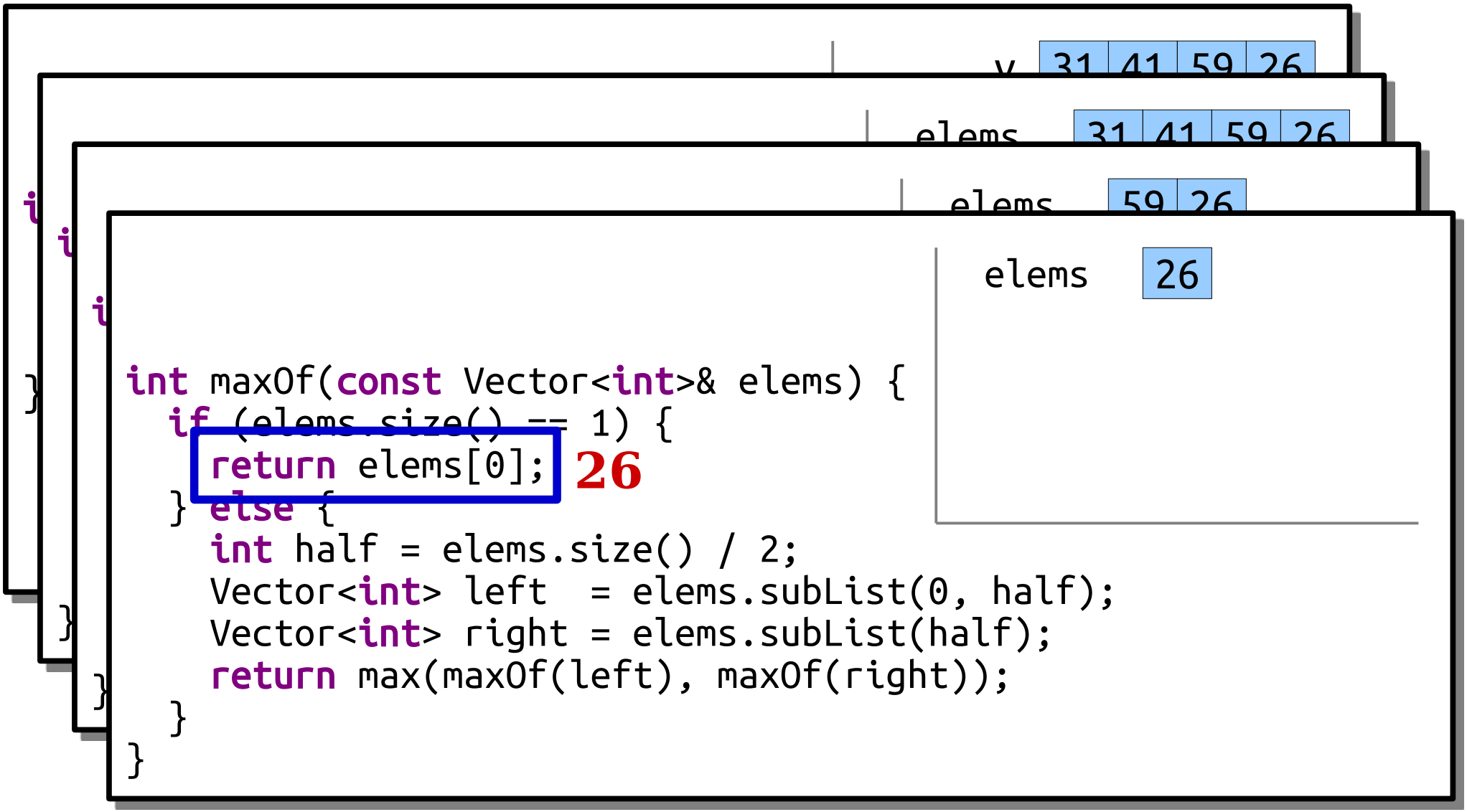
Tracing the Recursion



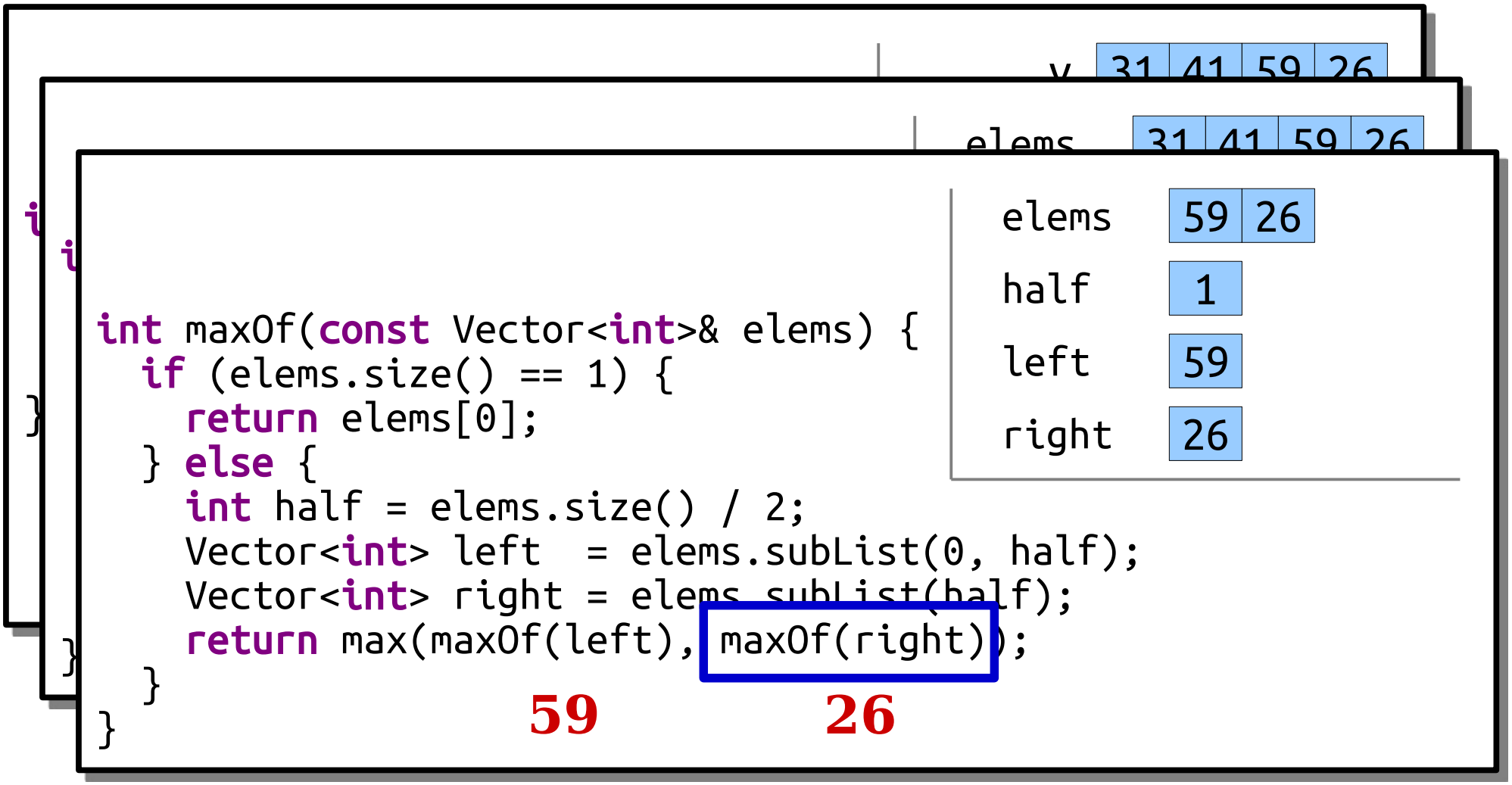
Tracing the Recursion



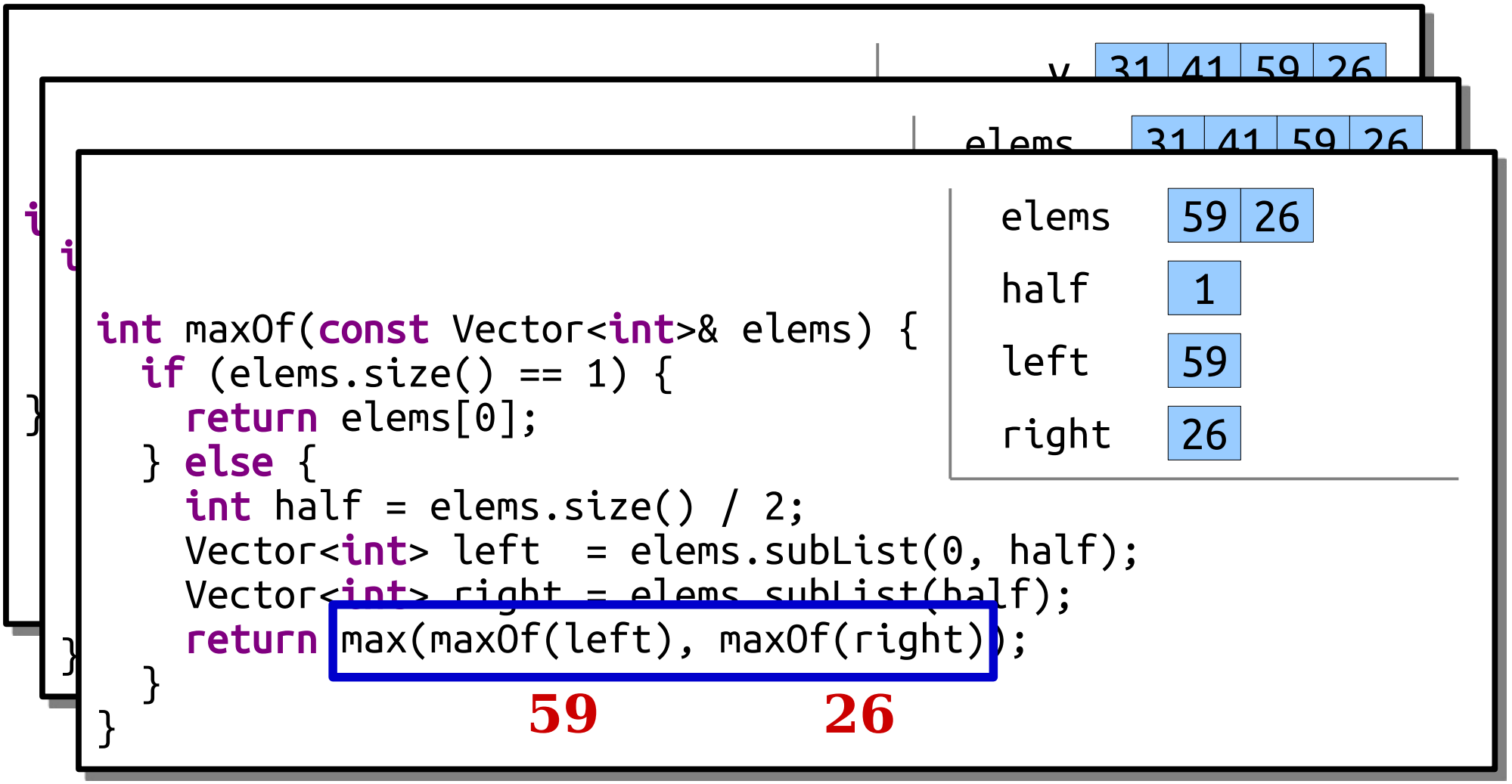
Tracing the Recursion



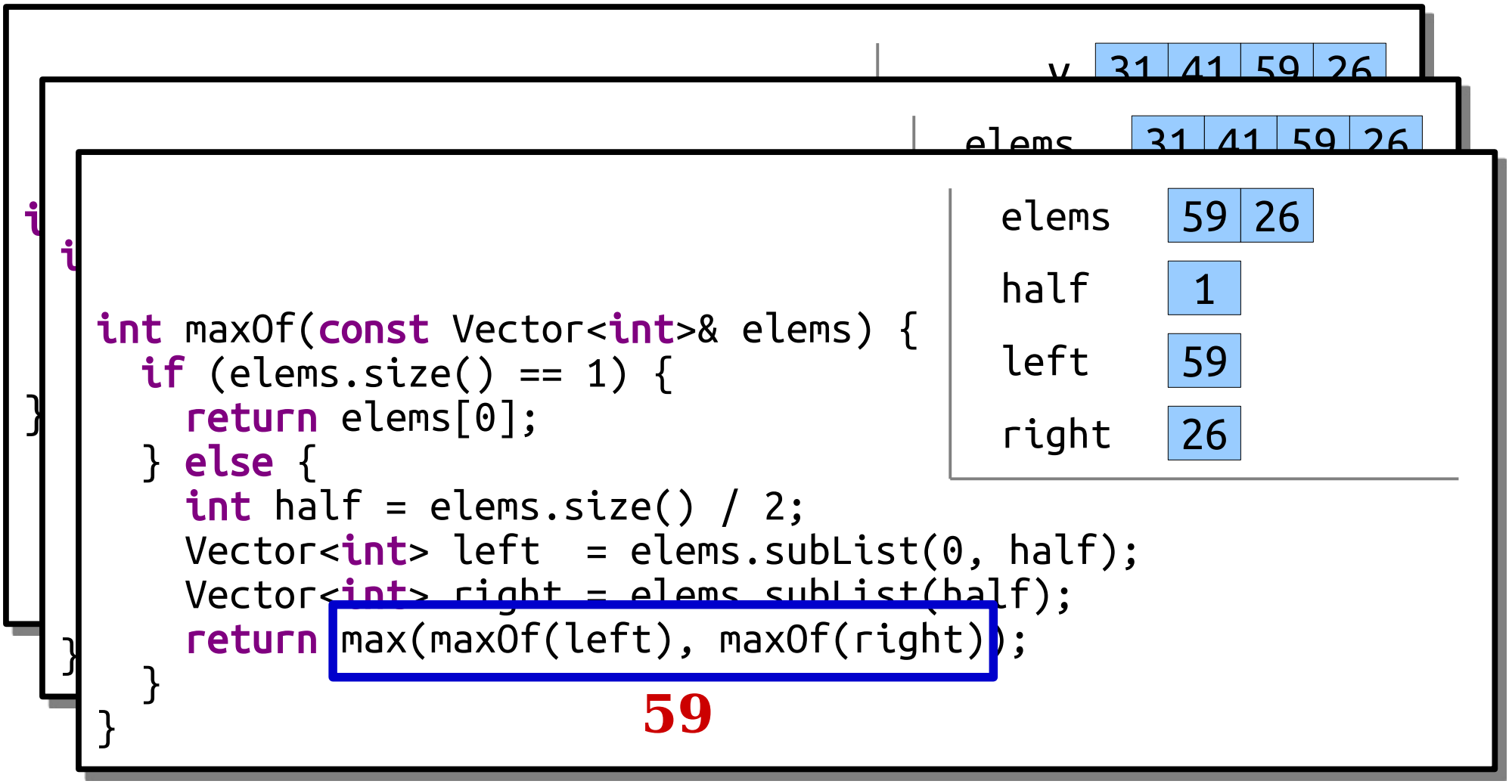
Tracing the Recursion



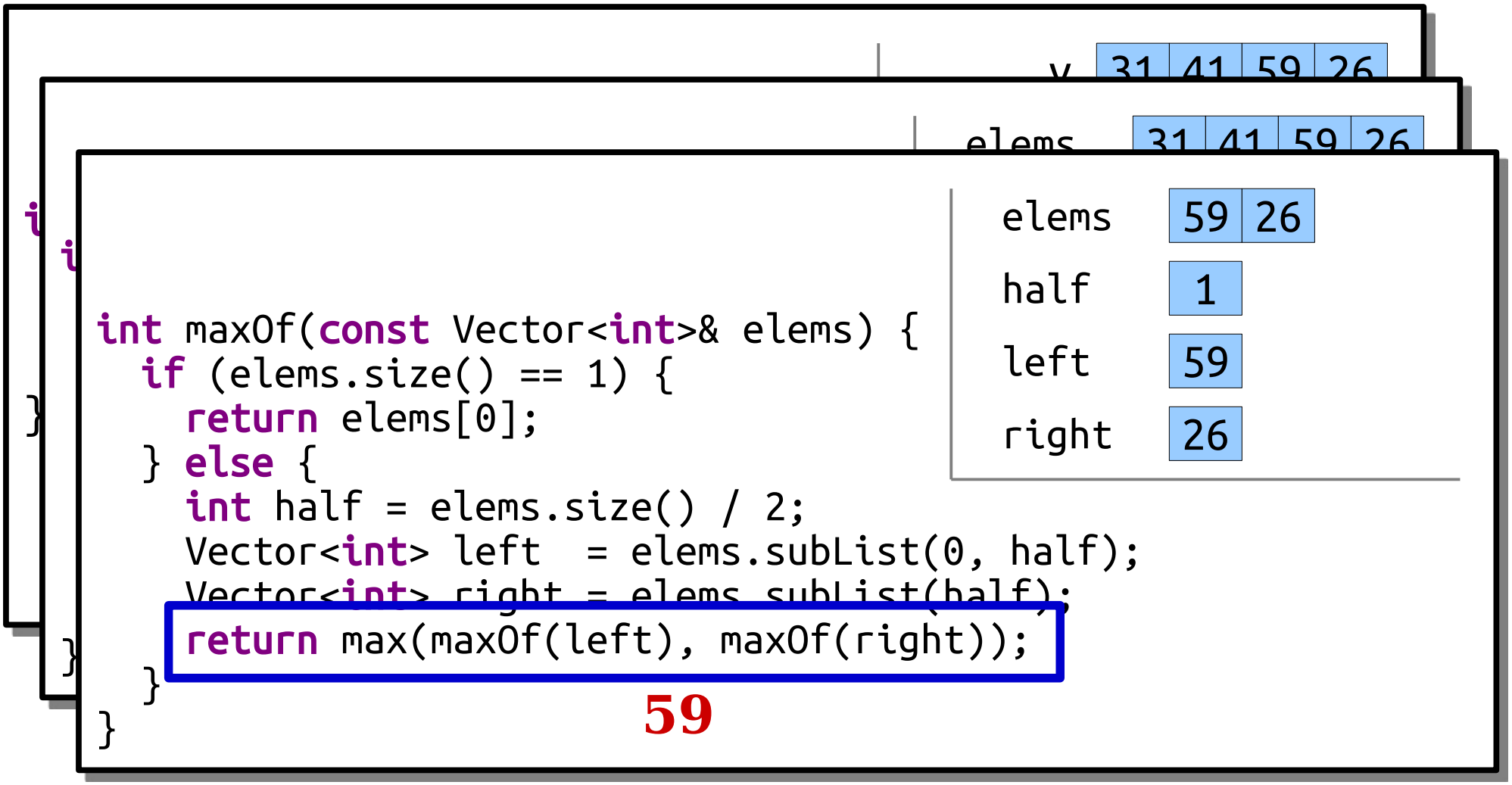
Tracing the Recursion



Tracing the Recursion



Tracing the Recursion



Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int half = elems.size() / 2;
    Vector<int> left  = elems.subList(0, half);
    Vector<int> right = elems.subList(half);
    return max(maxOf(left), maxOf(right));
  }
}
```

elems	31	41	59	26
half	2			
left	31	41		
right	59	26		

41 **59**

Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int half = elems.size() / 2;
    Vector<int> left  = elems.subList(0, half);
    Vector<int> right = elems.subList(half);
    return max(maxOf(left), maxOf(right));
  }
}
```

elems	31	41	59	26
half	2			
left	31	41		
right	59	26		

41 **59**

Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int half = elems.size() / 2;
    Vector<int> left  = elems.subList(0, half);
    Vector<int> right = elems.subList(half);
    return max(maxOf(left), maxOf(right));
  }
}
```

elems	31	41	59	26
half	2			
left	31	41		
right	59	26		

59

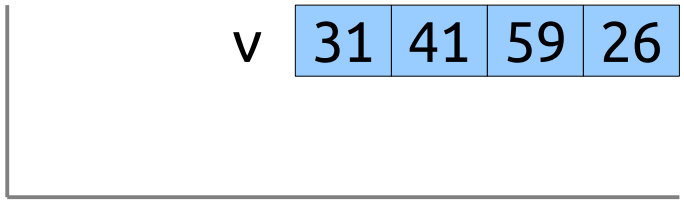
Tracing the Recursion

```
int maxOf(const Vector<int>& elems) {
  if (elems.size() == 1) {
    return elems[0];
  } else {
    int half = elems.size() / 2;
    Vector<int> left  = elems.subList(0, half);
    Vector<int> right = elems.subList(half);
    return max(maxOf(left), maxOf(right));
  }
}
```

elems	31	41	59	26
half	2			
left	31	41		
right	59	26		

59

Tracing the Recursion




```
int main() {
    Vector<int> v = { 31, 41, 59, 26 };
    cout << maxOf(v) << endl;
    return 0;
}
```

59