

Containers

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++

Prediction Time!

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;  
}
```

137

value

How it Works

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

137

value

How it Works

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

137

137

netWorth

How it Works

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

137

137

netWorth

How it Works

```
int main() {  
    int value = 127;  
    void becomeWealthy(int netWorth) {  
        netWorth = 1000000000;  
    }  
}
```

137

kaching!

netWorth

How it Works

```
int main() {  
    int netWorth = 127;  
    void becomeWealthy(int netWorth) {  
        netWorth = 1000000000;  
    }  
}
```

137

kaching!

netWorth

How it Works

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

137

value

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;
}
```

137

value

How it Works Now

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

137

value

How it Works Now

```
int main() {  
    int value = 137;  
    becomeWealthy(value);  
    cout << value << endl;  
}  
void becomeWealthy(int& networth) {  
    networth = 1000000000;  
}
```

The diagram illustrates the state of memory during the execution of the provided C++ code. A large black rectangular frame represents the stack, with a grey bar at the bottom representing the heap. Inside the stack frame, the variable `value` is shown with the value `137` in a blue box. A red arrow points from this `value` box to a vertical grey bar labeled `value`, which is positioned to the right of the stack frame. This indicates that the local variable `value` in the `main` function points to the same memory location as the parameter `networth` in the `becomeWealthy` function. The `networth` variable is also shown with the value `1000000000` in a blue box, located further to the right. The `networth` label is placed below its corresponding box.

How it Works Now

```
int main() {  
    int value = 137;  
    becomeWealthy(value);  
    cout << value << endl;  
}  
void becomeWealthy(int& networth) {  
    networth = 1000000000;  
}
```

The diagram illustrates the state of variables in memory. A blue box labeled '137' represents the initial value of 'value'. A red arrow points from this box to a vertical grey bar labeled 'value' at its top. Inside a larger black-bordered box, another blue box labeled '1000000000' represents the modified value of 'networth' passed by reference. A red arrow points from this box to a vertical grey bar labeled 'networth' at its top. The code shows that the original variable 'value' retains its initial value of 137, while the variable 'networth' is modified to 1000000000.

How it Works Now

```
int main() {  
    int value = 137;  
    becomeWealthy(value);  
    cout << value << endl;  
}  
void becomeWealthy(int& networth) {  
    networth = 1000000000;  
}
```

The diagram illustrates the state of variables during the execution of the code. The variable `value` is shown outside the inner scope, while the variable `networth` is shown inside the inner scope. The assignment `networth = 1000000000;` is highlighted with a red box and accompanied by a blue callout bubble containing the text "kaching!". A red arrow points from this bubble to the `networth` parameter in the `becomeWealthy` function, indicating that the modification occurs within the local scope.

How it Works Now

```
int main() {  
    int value = 137;  
    becomeWealthy(value);  
    cout << value << endl;  
}  
void becomeWealthy(int& netWorth) {  
    netWorth = 1000000000;  
}
```

The diagram illustrates the state of variables during the execution of the code. A blue box highlights the line `becomeWealthy(value);`. To the right, a vertical grey bar represents memory space. On the left side of the bar, the variable `value` is shown with a value of `137`. On the right side, the variable `netWorth` is shown with a value of `1000000000`. A red arrow points from a blue box labeled `kaching!` towards the `netWorth` variable, indicating that the modification made in the function call is reflected back to the original variable in the main function's scope.

How it Works Now

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

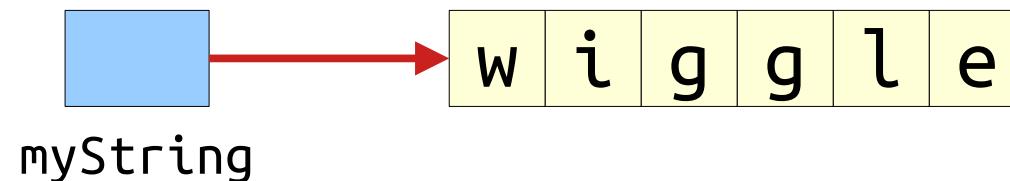
kaching!

value

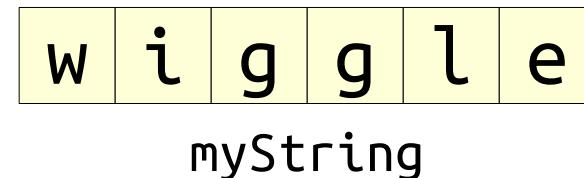
Prediction Time!

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;
}
```

wiggle

message

How it Works

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

wiggle

message

How it Works

```
int main() {  
    string text = "Hello";  
    wiggle(text);  
    cout << text;  
}  
  
void gollyGee(string text) {  
    text[0] = 'g';  
    wiggle(text);  
}
```

How it Works

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

How it Works

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

wiggle
giggle
text

How it Works

```
int main() {  
    string text = "wibble".  
    wiggle.  
    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;  
}
```

wiggle

value

How it Works Now

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";
    gotlyGee(message);
    cout << message << endl;
    return 0;
}
```

wiggle

message

How it Works Now

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

wiggle

message

How it Works Now

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

void gollyGee(string& text) {
    text[0] = 'g';
}
```

The diagram illustrates the state of variables in memory. In the `main()` function, the variable `message` is highlighted with a blue box and contains the value `wiggle`. In the `gollyGee()` function, the parameter `text` is also highlighted with a blue box and contains the value `wiggle`.

How it Works Now

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

void gollyGee(string& text) {
    text[0] = 'g';
}
```

The diagram illustrates the state of memory during the execution of the code. A blue box highlights the assignment `text[0] = 'g';`. A red arrow points from this box to a blue box labeled `wiggle`, which is part of the `message` variable in the `main` function. This visualizes how a change made in one part of the program (the `gollyGee` function) directly impacts another part (the `message` variable in `main`), demonstrating the concept of mutation.

How it Works Now

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

void gollyGee(string& text) {
    text[0] = 'g';
}
```

The diagram illustrates the state of memory for the variable `message`. A blue box highlights the assignment `text[0] = 'g';` in the `gollyGee` function. A red arrow points from this assignment to the first character of the `message` string, which is now labeled `giggle`. The original value `wiggle` is shown below it, indicating that the modification is non-destructive.

How it Works Now

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

void gollyGee(string& text) {
    text[0] = 'g';
}
```

The diagram illustrates the state of variables in memory. In the main function's stack frame, the variable `message` is highlighted with a blue box and contains the value `wiggle`. An arrow points from the word `giggle` to the `message` variable. In the `gollyGee` function's stack frame, there is a local variable `text` which also contains the value `wiggle`, indicated by another arrow.

How it Works Now

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

giggle

message

Container Types

Container Types

- A ***container type*** (also called an ***abstract data type*** or ***collection 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);
```

```
"""\n      Python Version\n"""
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);
```

```
"""\n      Python Version\n"""
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);
```

```
//      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);

```

```

/*      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);

```

```

"""" 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.

```

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);
```

```
/*      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);
```

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

v.append(271)

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

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

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.

```
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);
```

“ Java Version ”

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.

```
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); ←
```

```
"""\n      Python Version\n"""
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);
```

“ Java Script Version ”

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

```
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);
```

```
"""\n      Python Version\n"""
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;
}
```

```
"""\n      Python Version\n"""
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);
}
```

```
/*      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;
}
```

```
"""\n      Python Version\n"""
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);
}
```

```
/*      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;  
}
```

```
/*      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);  
}
```

```
"""" 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)
```

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

```
for (int i = 0; i < v.size(); i++) {  
    console.log(v[i]);  
}
```

```
/*      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;
}
```

```
"""\n      Python Version\n"""
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);
}
```

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

```
}
```

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

```
/*      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

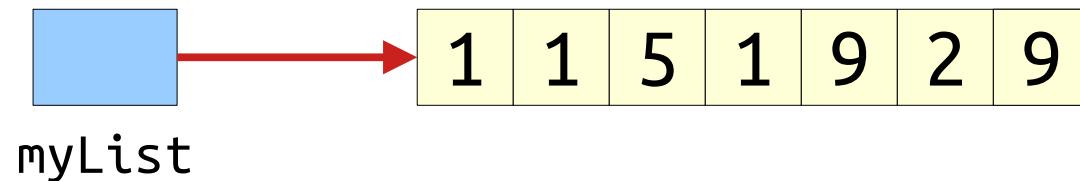
Stanford C++ Library Documentation

up on the course website.

Prediction Time!

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

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

1929	1955	1964
------	------	------

values

How it Works

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

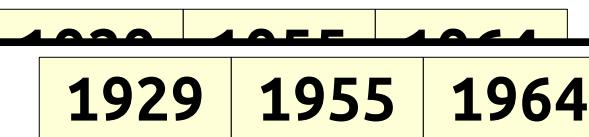
1929	1955	1964
------	------	------

values

How it Works

i

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



numbers

How it Works

i

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

1929	1955	1964
------	------	------

numbers

How it Works

i

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

1929	1963	1964
------	------	------

numbers

How it Works

i

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

1929	1963	1964
------	------	------

numbers

How it Works

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

1929	1955	1964
------	------	------

values

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

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

1929	1955	1964
------	------	------

values

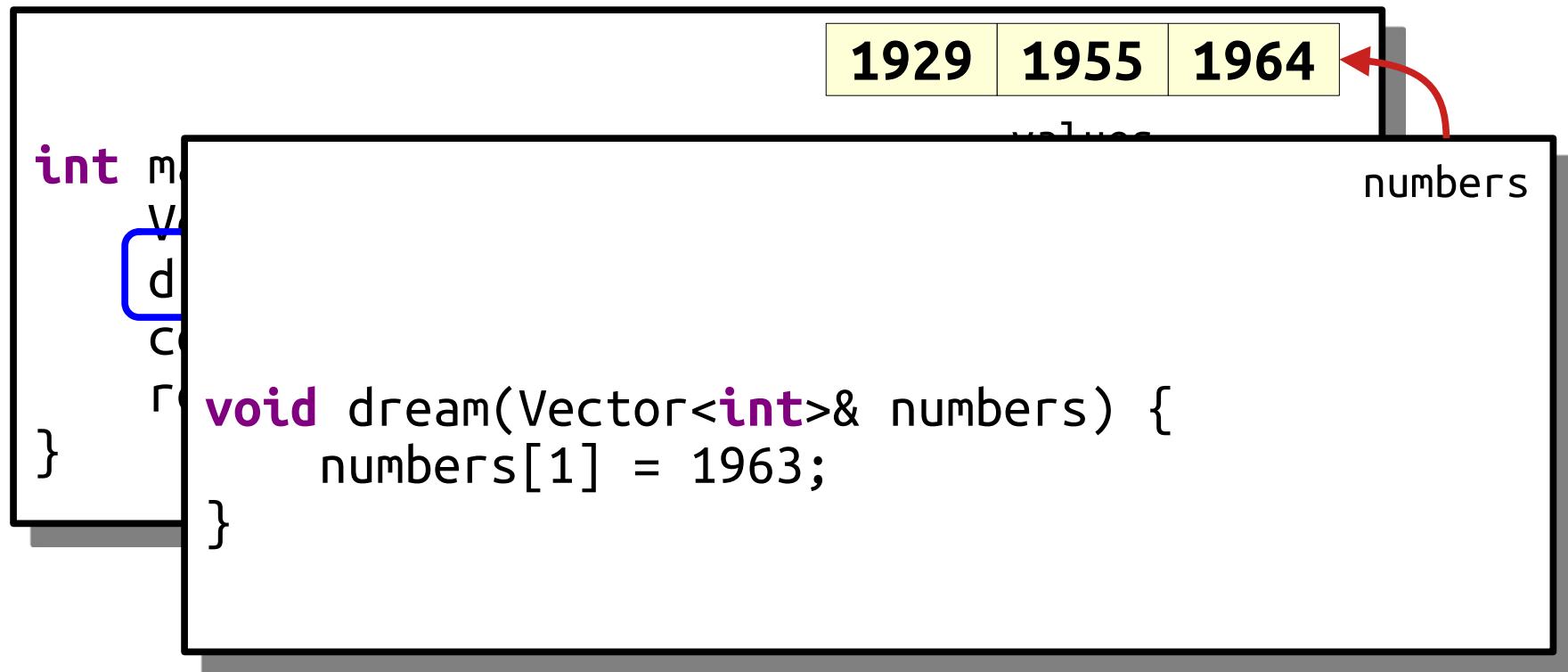
How it Works Now

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

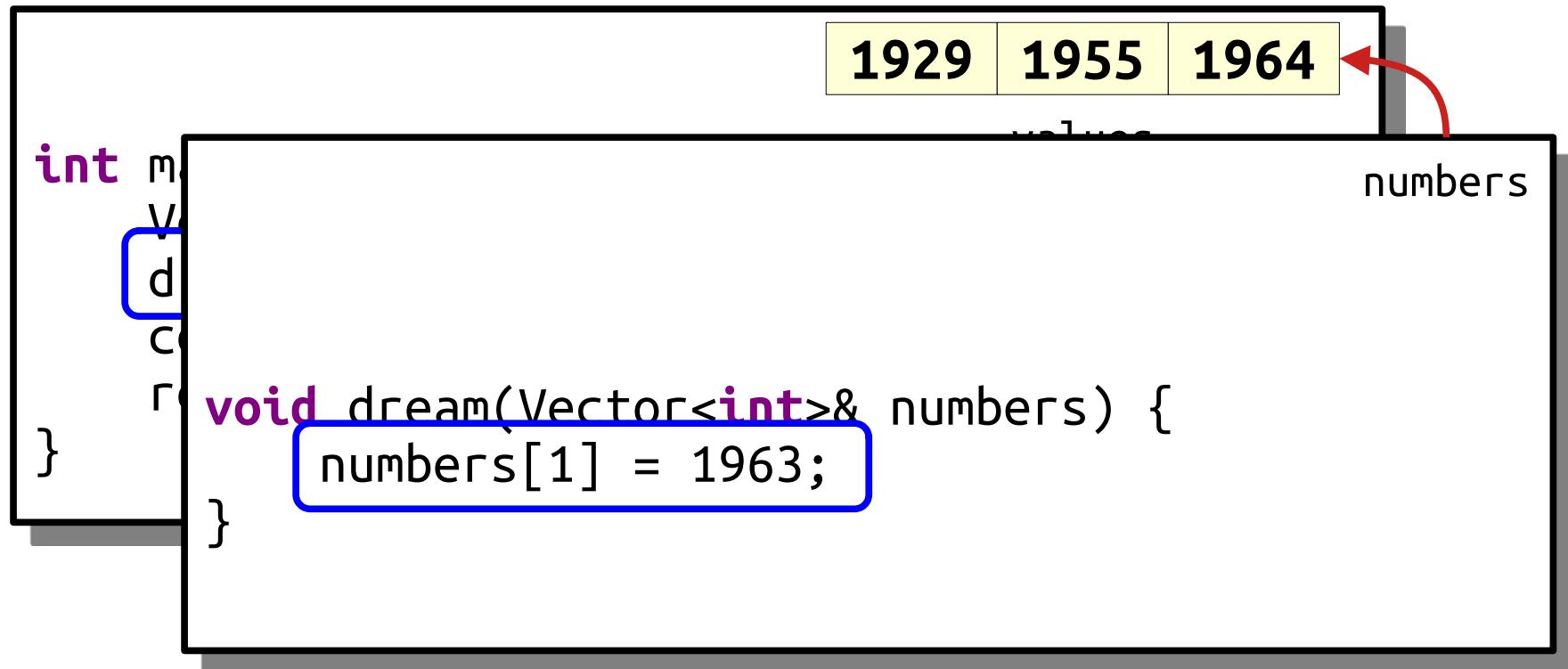
1929	1955	1964
------	------	------

values

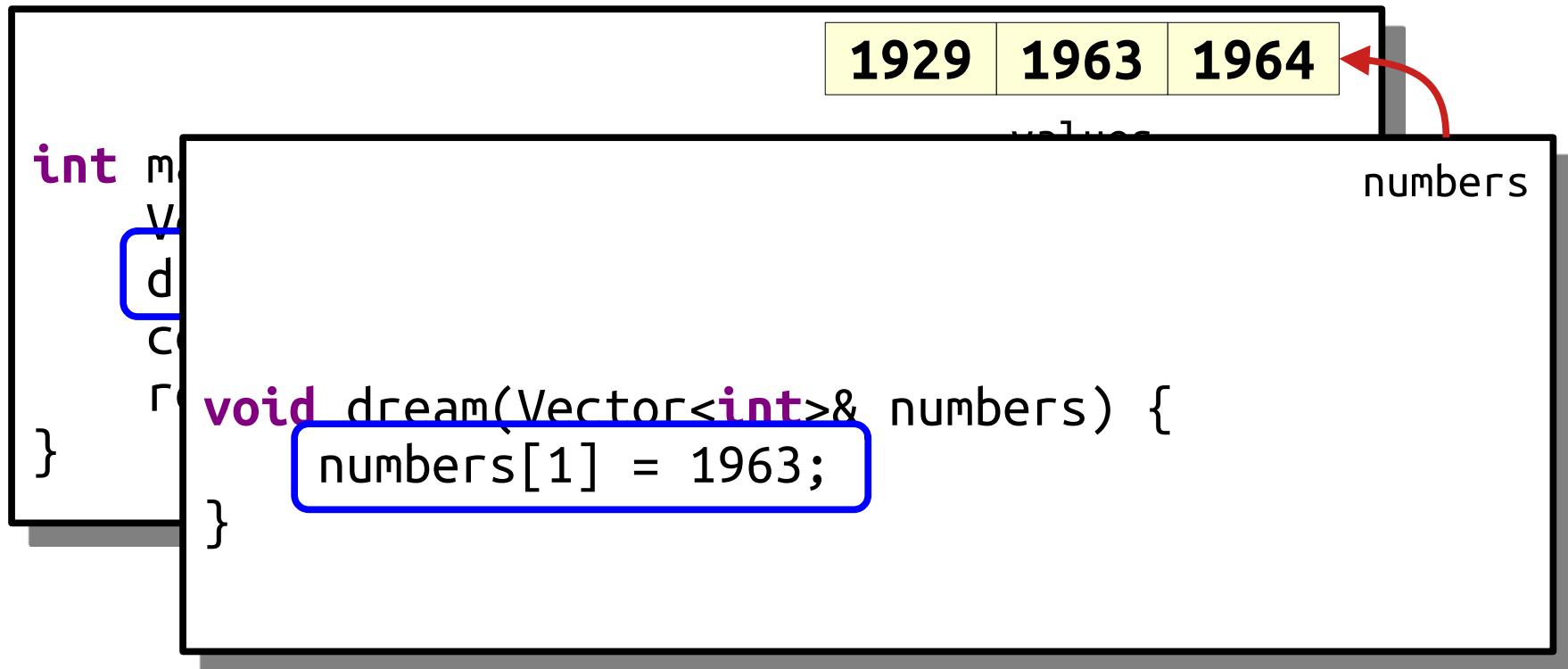
How it Works Now



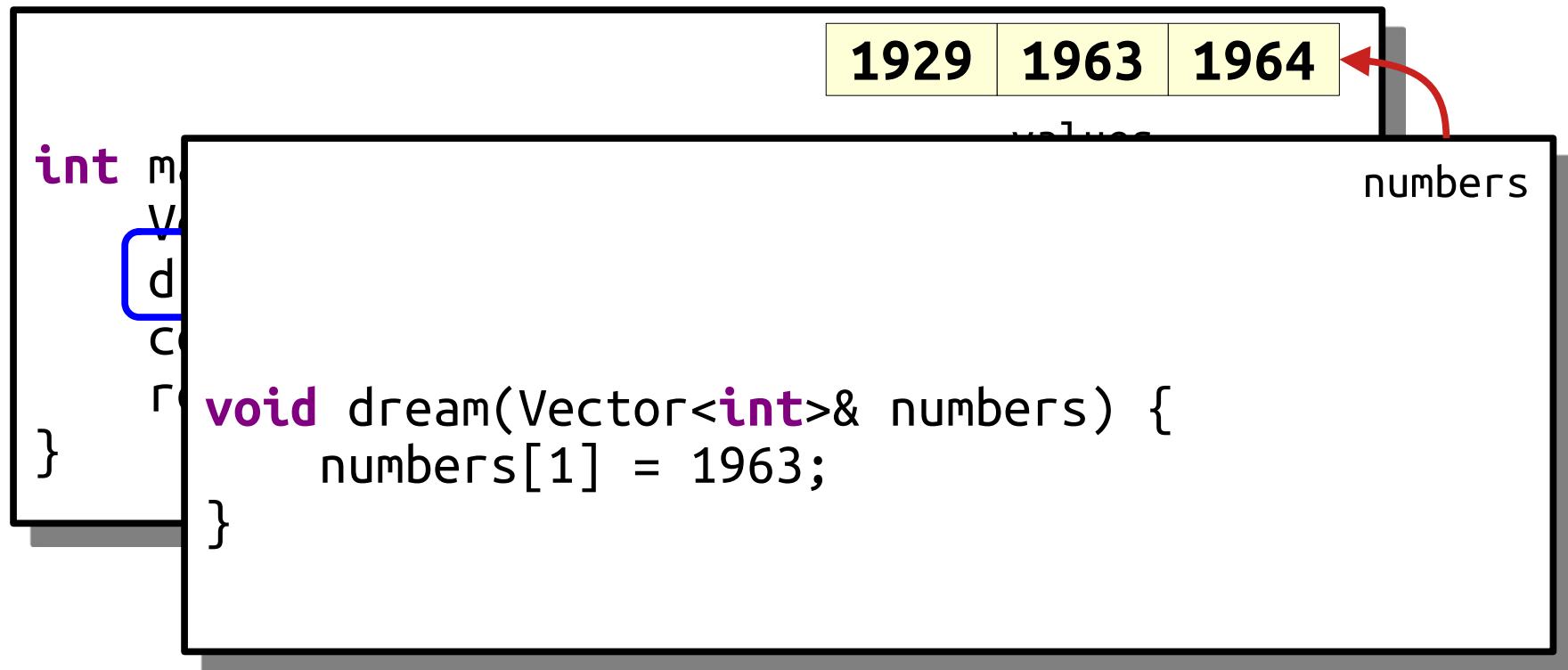
How it Works Now



How it Works Now



How it Works Now



How it Works Now

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

1929	1963	1964
------	------	------

values

“I’ll live forever! Once the technology is available, I’ll just upload my mind into the cloud.”

How people think it works:

```
void uploadToCloud(Mind& consciousness);
```

How it actually works:

```
void uploadToCloud(Mind consciousness);
```

Credit: Philip Heltweg

Time-Out for Announcements!

Sections

- Discussion sections start this week!
 - Didn't sign up by Sunday at 5PM? The signup link will reopen on Tuesday at 5PM, and you can choose any open section time.
 - If your section time doesn't work for you, you can switch into any section with available space starting Tuesday at 5PM. Visit cs198.stanford.edu to do this.
 - Still doesn't work for you? Ping Neel!
- You'll get your section assignment this Tuesday at 5:00PM.
- Each week we'll release a set of section problems on the course website. ***These are not graded***, but we recommend you read over them before your section.

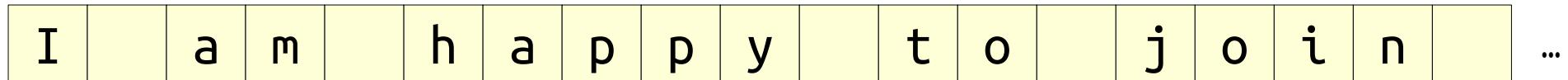
YEAH Hours

- We'll be holding special sessions called **Your Early Assignment Help Hours** (YEAH Hours) to give overviews of each of the assignments.
- The first one is today, **3PM - 4PM** in **200-034**.
- These are purely optional, but recommended if you have the bandwidth.

return;

A Question of Speed

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



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

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?

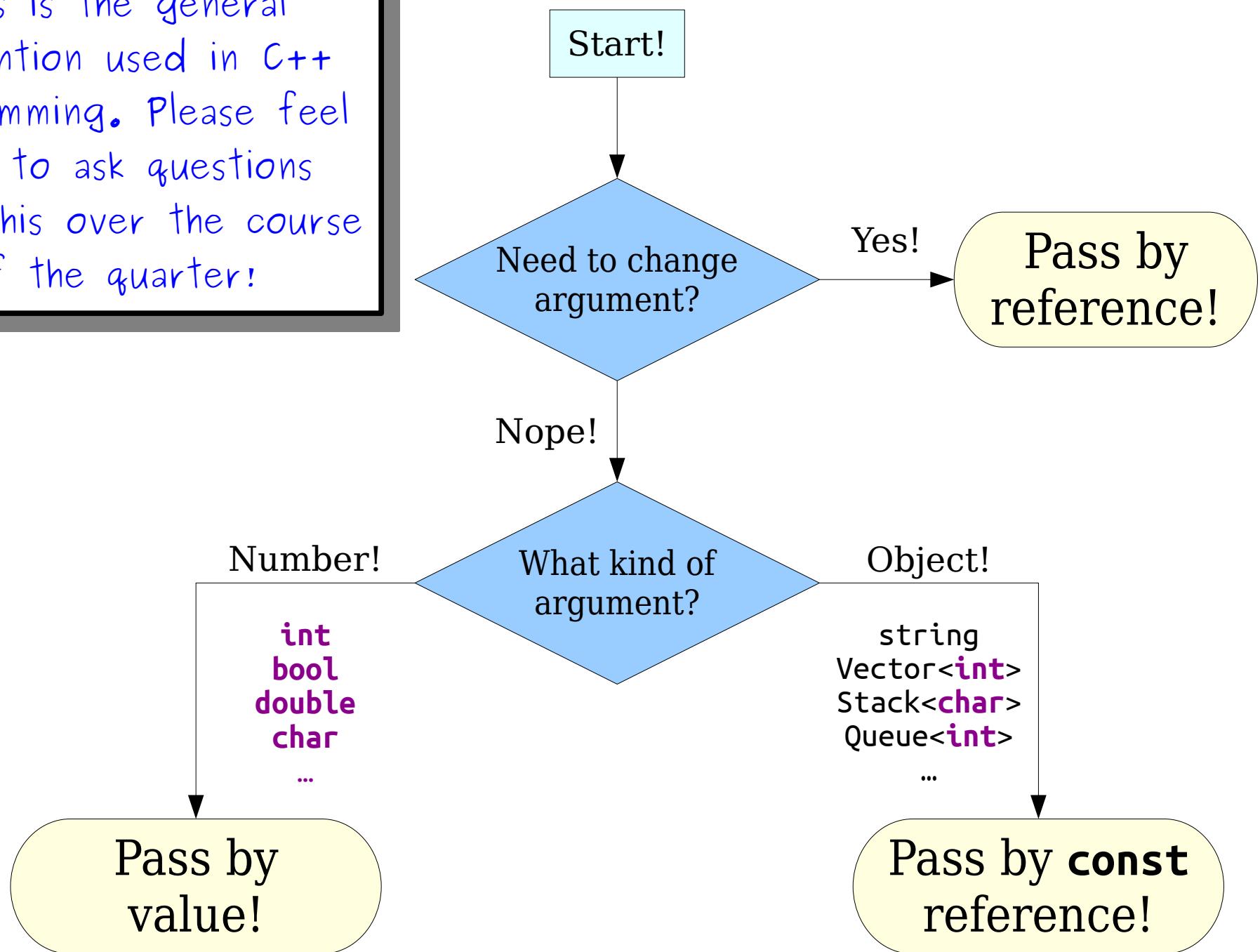
```
totallyNotSketchy(myMasterpiece);
```

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!



Recursion on Vectors

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>`, ...

137

elems[0]

... or the largest
element in this
`Vector<int>`.

271	828	182
-----	-----	-----

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 << max0T(v) << endl;
    return 0;
}
```

Tracing the Recursion

```
int main() {
    Vector<int> v = { 2, 7, 1 };
    cout << max0T(v) << endl;
    return 0;
}
```

v

2	7	1
---	---	---

Tracing the Recursion

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

v

2	7	1
---	---	---

Tracing the Recursion

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

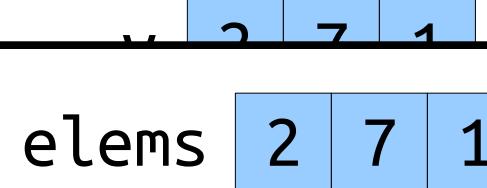
v

2	7	1
---	---	---

Tracing the Recursion

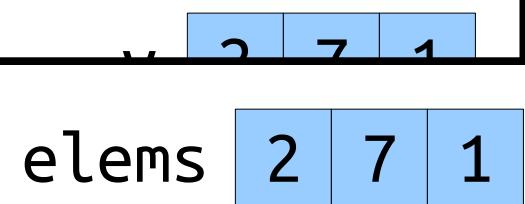
i

```
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

```
i 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

i

```
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 2 7 1

... 2 7 1

Tracing the Recursion

i

```
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));  
    }  
}
```

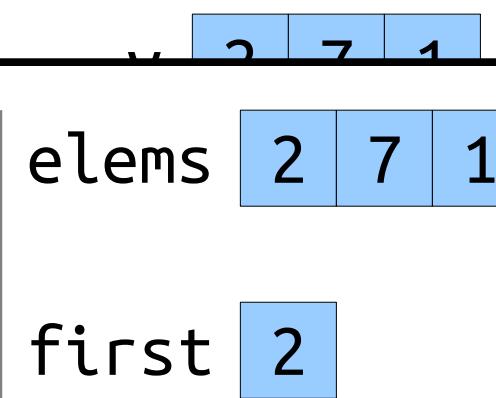
elems 2 7 1

2 7 1

Tracing the Recursion

i

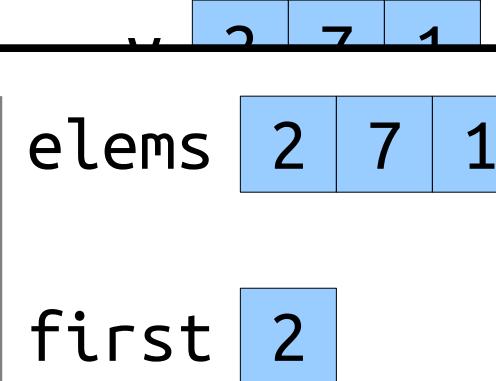
```
int maxOf(const Vector<int>& elems) {  
    if (elems.size() == 1) {  
        return elems[0];  
    } else {  
        int first = elems[0];  
        Vector<int> rest = elems.subList(1);  
        return max(first, maxOf(rest));  
    }  
}
```



Tracing the Recursion

i

```
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

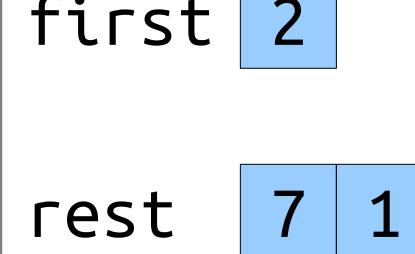
```
i int maxOf(const Vector<int>& elems) {  
    if (elems.size() == 1) {  
        return elems[0];  
    } else {  
        int first = elems[0];  
        Vector<int> rest = elems.subList(1);  
        return max(first, maxOf(rest));  
    }  
}
```



Tracing the Recursion

i

```
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

i

```
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

i

```
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

... 2 7 1

elems 2 7 1

first 2

rest 7 1

Tracing the Recursion

i

```
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

... 2 7 1

elems 2 7 1

first 2

rest 7 1

Tracing the Recursion

i

```
int maxOf(const Vector<int>& elems) {
```

```
    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

i

```
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

i

```
int maxOf(const Vector<int>& elems) {
```

```
    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

elems 2 7 1

elems 2 7 1

Tracing the Recursion

i

```
int maxOf(const Vector<int>& elems) {
```

```
    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

i

```
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

i

```
int maxOf(const Vector<int>& elems) {
```

```
    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 2 7 1

elems 2 7 1

elems 7 1

first 7

Tracing the Recursion

i

```
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

i

```
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 2 7 1

elems 2 7 1

elems 7 1

first 7

rest 1

Tracing the Recursion

i

```
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 2 7 1

elems 2 7 1

elems 7 1

first 7

rest 1

Tracing the Recursion

i

```
int maxOf(const Vector<int>& elems) {  
    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 2 7 1

elems 2 7 1

elems 7 1

first 7

rest 1

Tracing the Recursion

i

```
int maxOf(const Vector<int>& elems) {  
    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 2 7 1

elems 2 7 1

elems 7 1

first 7

rest 1

Tracing the Recursion

```
i int maxOf(const Vector<int>& elems) {  
    int maxOf(const Vector<int>& elems) {  
        }  
        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));  
            }  
        }  
    }
```

The diagram illustrates the state of the recursive call stack. It consists of three horizontal bars, each representing a call frame. The top bar contains the elements [2, 7, 1]. The middle bar contains the elements [7, 1]. The bottom bar contains the element [1]. This visualizes how the input vector is reduced from three elements to one element as the recursion progresses.

Tracing the Recursion

```
i int maxOf(const Vector<int>& elems) {  
    int maxOf(const Vector<int>& elems) {  
        }  
        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));  
            }  
        }  
    }  
}
```

The diagram illustrates the state of the 'elems' vector at different stages of the recursion:

- Initial call: elems = [2, 7, 1]
- First recursive call: elems = [7, 1]
- Second recursive call: elems = [1]

Tracing the Recursion

```
i int maxOf(const Vector<int>& elems) {  
    int maxOf(const Vector<int>& elems) {  
        }  
        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

```
i int maxOf(const Vector<int>& elems) {  
    int maxOf(const Vector<int>& elems) {  
        }  
        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));  
            }  
        }  
    }  
}
```

The diagram illustrates the state of the `elems` parameter across three recursive calls. In the first call, `elems` is a vector containing 2, 7, and 1. In the second call, `elems` is a vector containing 7 and 1. In the final call, `elems` contains only the value 1.

Tracing the Recursion

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



Tracing the Recursion

i

```
int maxOf(const Vector<int>& elems) {  
    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 1

elems 2 7 1

elems 2 7 1

elems 7 1

first 7

rest 1

Tracing the Recursion

i

```
int maxOf(const Vector<int>& elems) {  
    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 1

elems 2 7 1

elems 2 7 1

elems 7 1

first 7

rest 1

Tracing the Recursion

i

```
int maxOf(const Vector<int>& elems) {  
    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 2 7 1

elems 2 7 1

elems 7 1

first 7

rest 1

Tracing the Recursion

i

```
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 2 7 1

elems 2 7 1

elems 7 1

first 7

rest 1

Tracing the Recursion

i

```
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

... 2 7 1

elems 2 7 1

first 2

rest 7 1

Tracing the Recursion

```
i 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));  
    }  
}
```

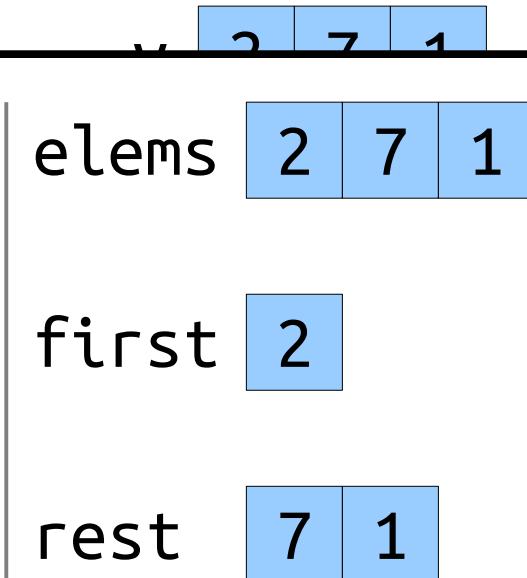
2 7



Tracing the Recursion

```
i 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

i

```
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  
}
```

v

2	7	1
---	---	---

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.***
 - If you're following our recommended timetable, aim to have Debugger Warmups and Fire completed tonight, and start working on Only Connect by Wednesday.
- ***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.
- ***Fun With Words***
 - Simple programs + rich data = cool demos.

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>`, ...

137	271
-----	-----

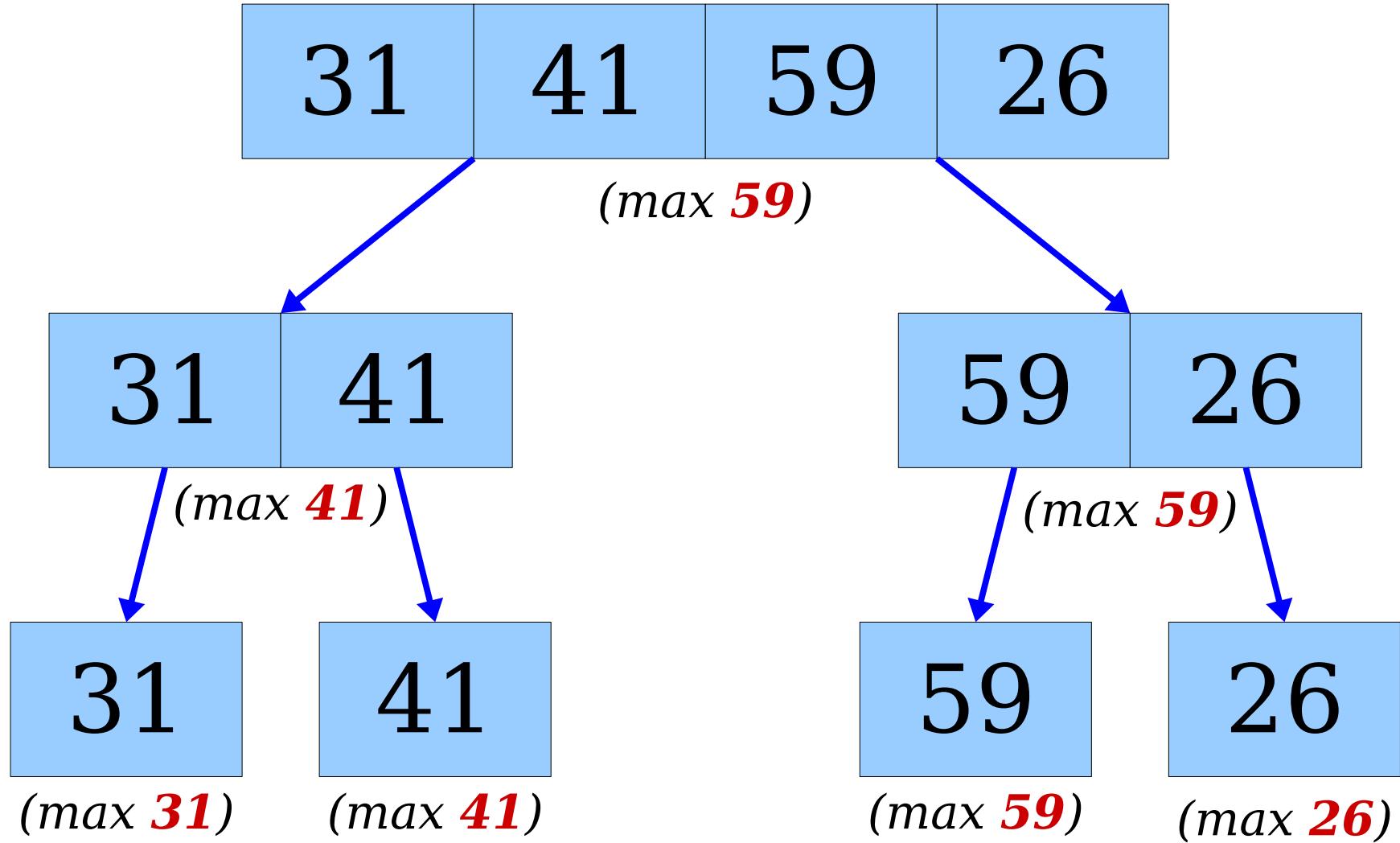
... or the largest
element in this
`Vector<int>`.

828	182
-----	-----

`elems.subList(0, elems.size() / 2)`

`elems.subList(elems.size() / 2)`

max0f 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

v 31 41 59 26

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

Tracing the Recursion

v 31 41 59 26

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

Tracing the Recursion

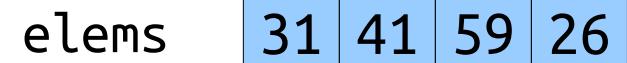
v 31 41 59 26

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

Tracing the Recursion

i

```
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

i

```
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));
    }
}
```

v 31 41 59 26

elems 31 41 59 26

Tracing the Recursion

i

```
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

i

```
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

i

```
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));  
    }  
}
```

v 31 41 59 26

elems	31	41	59	26
half	2			

Tracing the Recursion

i

```
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.SUBLIS(half);  
        return max(maxOf(left), maxOf(right));  
    }  
}
```

v 31 41 59 26

elems	31	41	59	26
half	2			

Tracing the Recursion

i

```
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.SUBLIS(half);  
        return max(maxOf(left), maxOf(right));  
    }  
}
```

v 31 41 59 26

elems	31	41	59	26
half	2			
left	31	41		

Tracing the Recursion

i

```
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));  
    }  
}
```

v 31 41 59 26

elems	31	41	59	26
half	2			
left	31	41		

Tracing the Recursion

i

```
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));  
    }  
}
```

v 31 41 59 26

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

Tracing the Recursion

i

```
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		

Tracing the Recursion

i

```
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));  
    }  
}
```



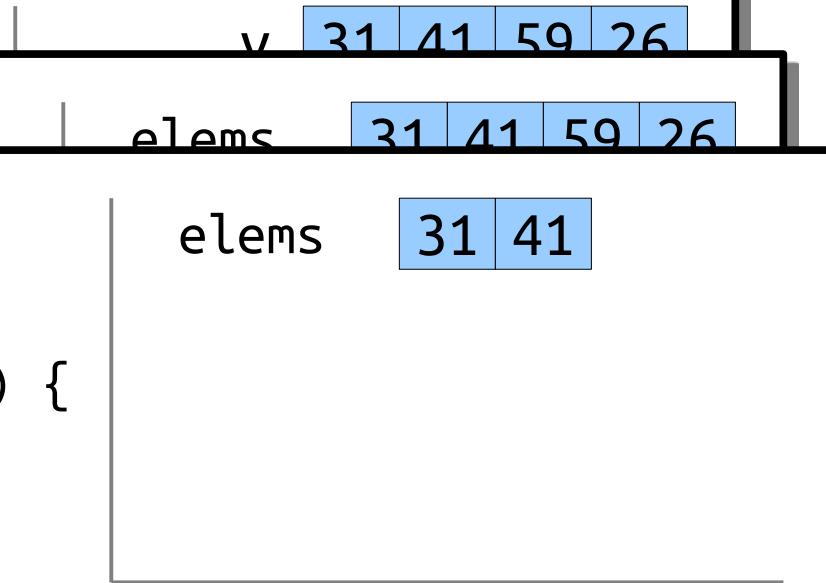
elems	31 41 59 26
half	2
left	31 41
right	59 26

Tracing the Recursion

i

i

```
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

i i

```
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

i

i

```
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

i

i

```
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));  
    }  
}
```



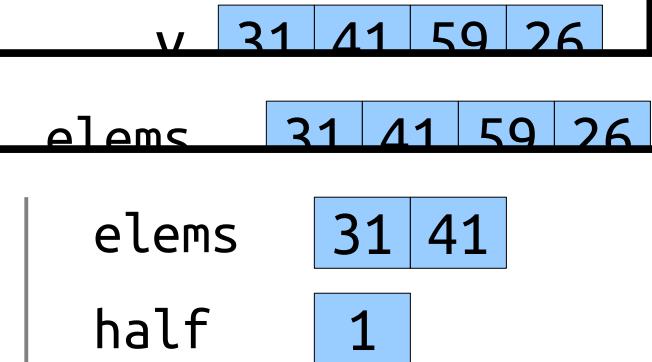
elems

elems

31 41

Tracing the Recursion

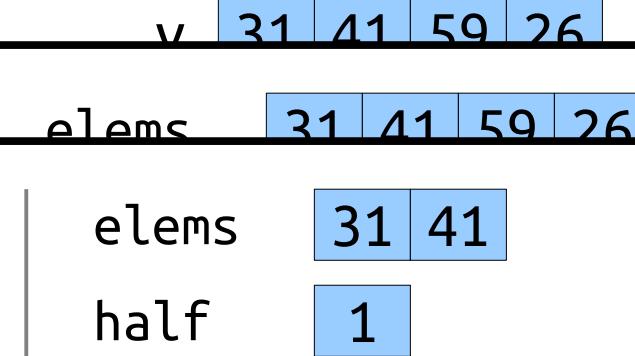
```
i i  
int maxOf(const Vector<int>& elems) {  
    if (elems.size() == 1) {  
        return elems[0];  
    } else {  
        int half = elems.size() / 2;  
        vector<int> left = elems.subList(0, half);  
        Vector<int> right = elems.subList(half);  
        return max(maxOf(left), maxOf(right));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



Tracing the Recursion

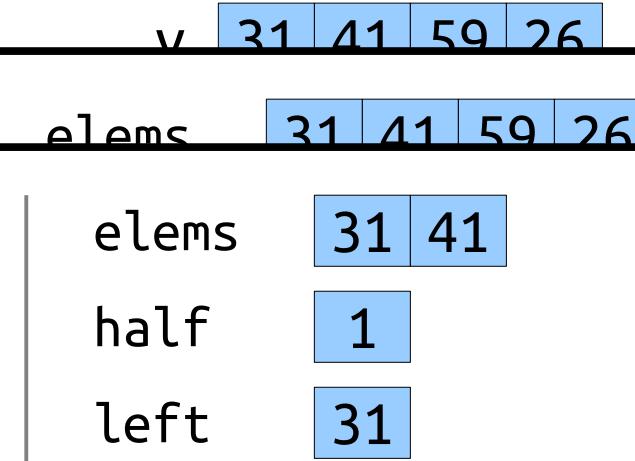
i
i

```
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

```
i i  
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));  
    }  
}
```

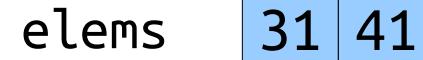


Tracing the Recursion

i

i

```
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));  
    }  
}
```

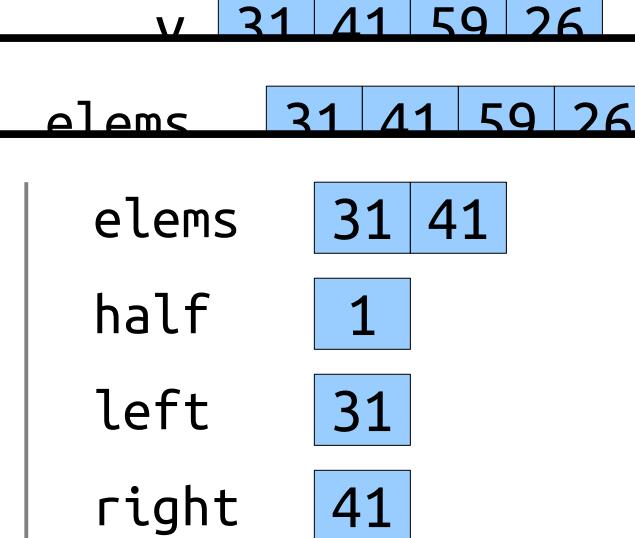


Tracing the Recursion

i

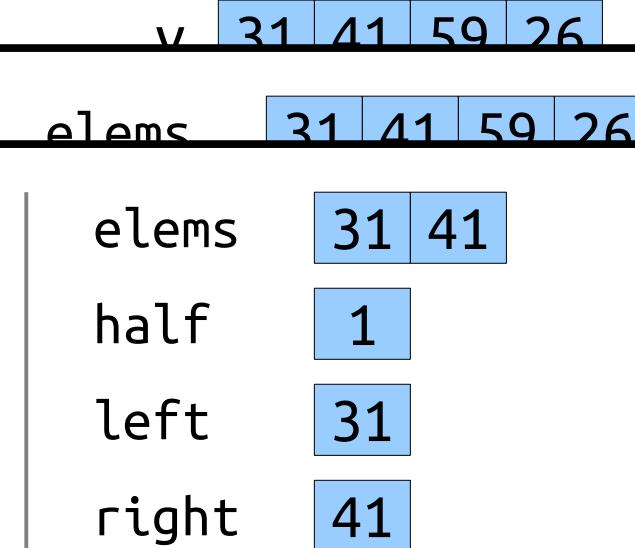
i

```
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));  
    }  
}
```



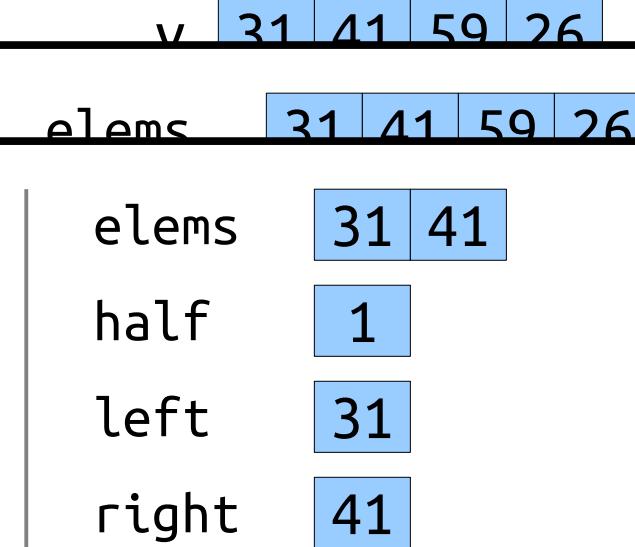
Tracing the Recursion

```
i i  
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

```
i i  
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]; 31
    } 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));
    }
}
```



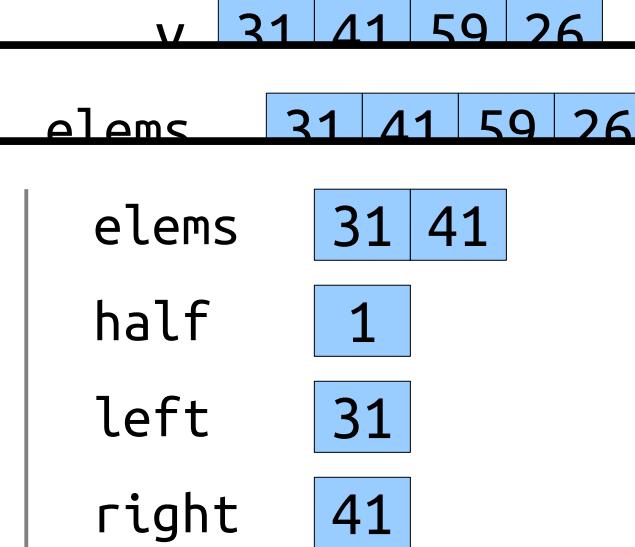
Tracing the Recursion

i

i

```
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));  
    }  
}
```

31



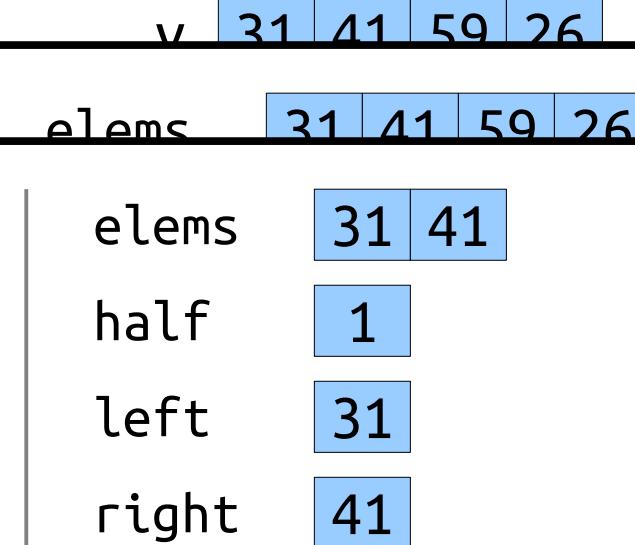
Tracing the Recursion

i

i

```
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));  
    }  
}
```

31



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]; 41
    } 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

i

i

```
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));  
    }  
}
```

31

41



1

31

41

Tracing the Recursion

i

i

```
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));  
    }  
}
```

31

41

v 31 41 59 26

elems 31 41 59 26

elems 31 41

1

left 31

41

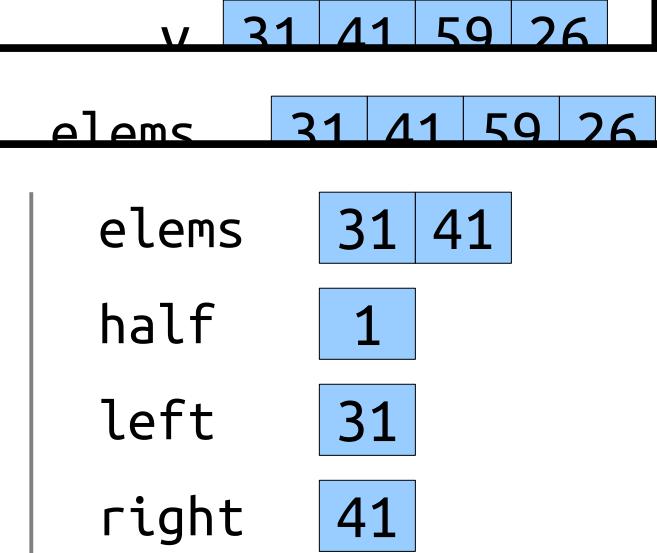
Tracing the Recursion

i

i

```
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));  
    }  
}
```

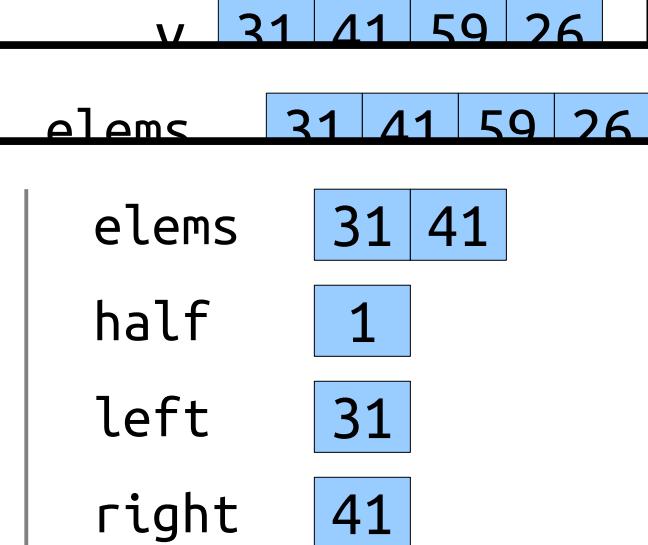
41



Tracing the Recursion

```
i i  
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

i

```
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

v 31 41 59 26

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

Tracing the Recursion

i

```
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

v 31 41 59 26

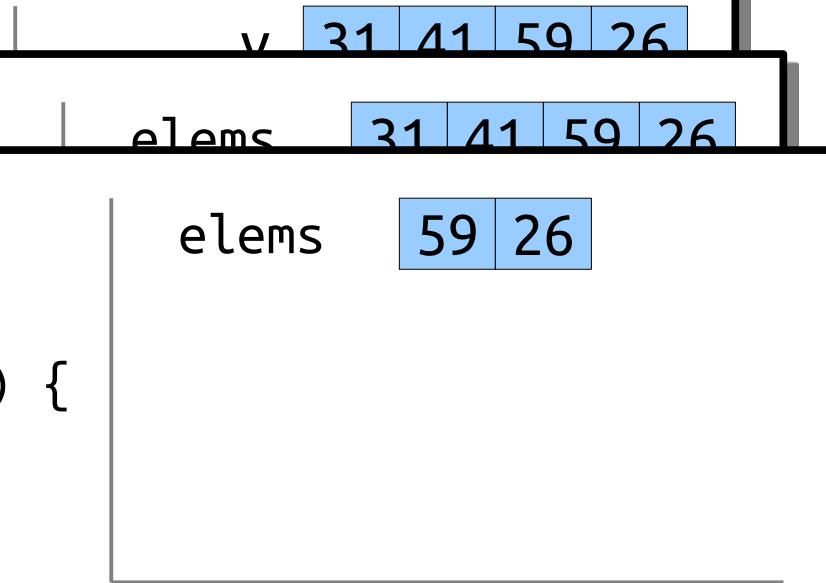
elems	31	41	59	26
half	2			
left	31	41		
right	59	26		

Tracing the Recursion

i

i

```
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

i
i

```
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));  
    }  
}
```

v 31 41 59 26

elems 31 41 59 26

elems 59 26

Tracing the Recursion

i i

```
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

i

i

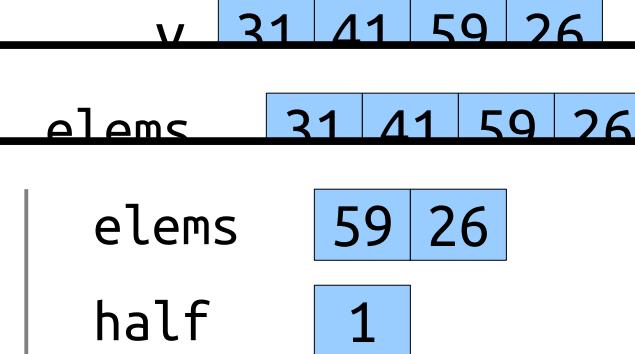
```
int maxOf(const Vector<int>& elems) {  
    if (elems.size() == 1) {  
        return elems[0];  
    } else {  
        int half = elems.size() / 2;  
        vector<int> left = elems.subList(0, half);  
        Vector<int> right = elems.subList(half);  
        return max(maxOf(left), maxOf(right));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



Tracing the Recursion

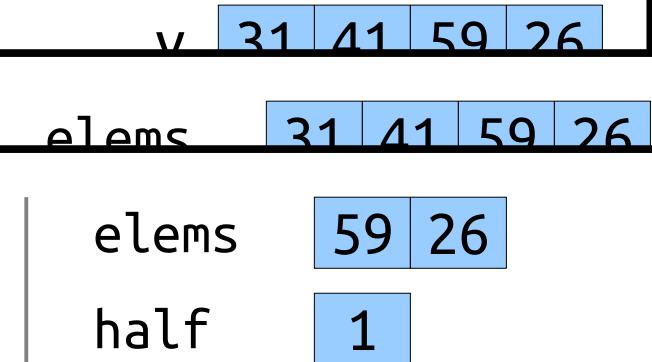
i
i

```
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

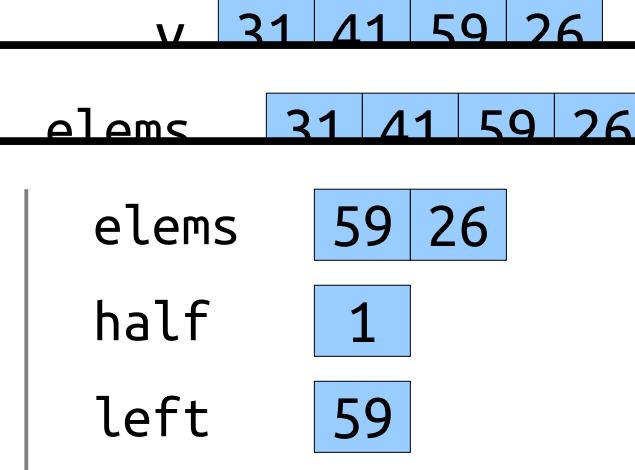
```
i i  
int maxOf(const Vector<int>& elems) {  
    if (elems.size() == 1) {  
        return elems[0];  
    } else {  
        int half = elems.size() / 2;  
        Vector<int> left  = elems.subList(0, half);  
        vector<int> right = elems.subList(half);  
        return max(maxOf(left), maxOf(right));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



Tracing the Recursion

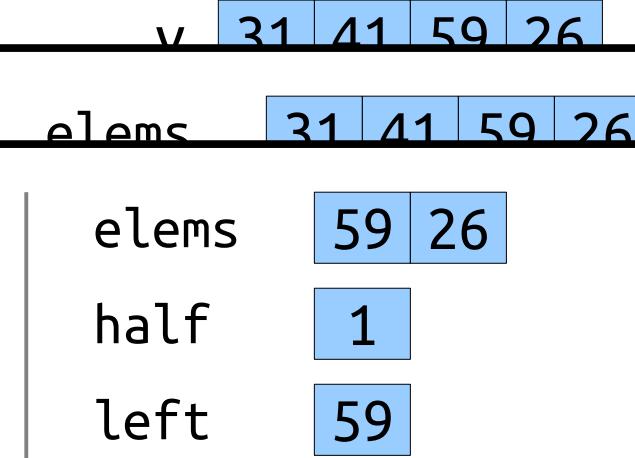
i
i

```
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

```
i i  
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));  
    }  
}
```

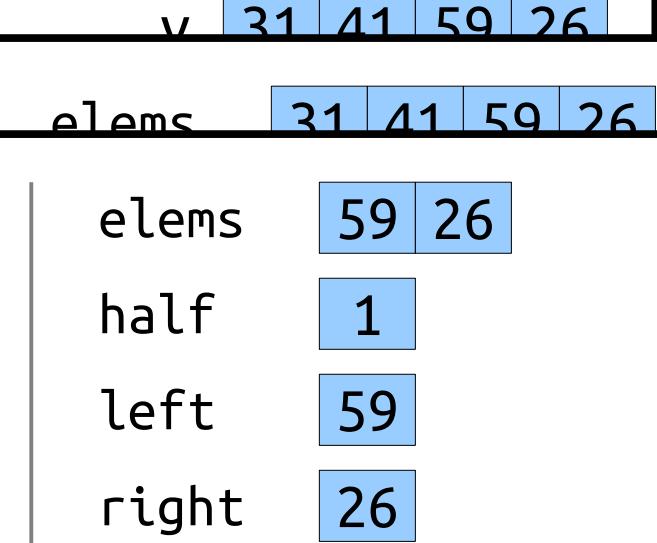


Tracing the Recursion

i

i

```
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

i

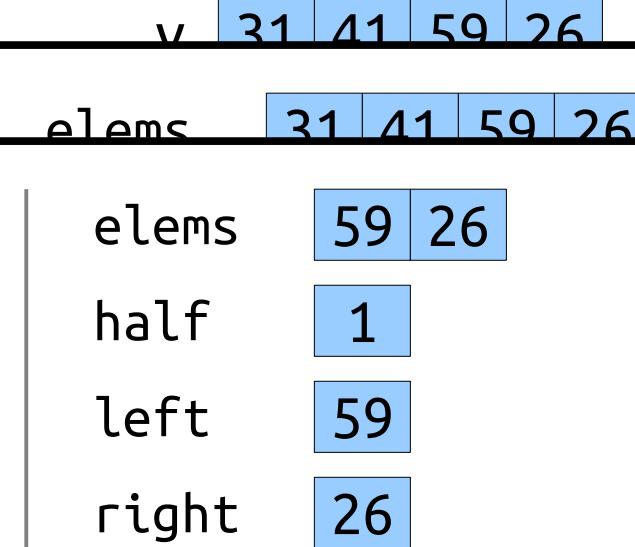
i

```
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

```
i i  
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]; 59
    } 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));
    }
}
```



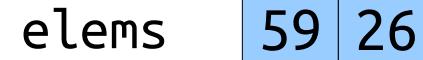
Tracing the Recursion

i

i

```
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));  
    }  
}
```

59



Tracing the Recursion

i

i

```
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));  
    }  
}
```

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));
    }
}
```



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]; 26
    } 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

i

i

```
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));  
    }  
}
```

59

26

v 31 41 59 26

elems 31 41 59 26

elems 59 26

half 1

left 59

right 26

Tracing the Recursion

i

i

```
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));  
    }  
}
```

59 26

v 31 41 59 26

elems 31 41 59 26

elems 59 26

half 1

left 59

right 26

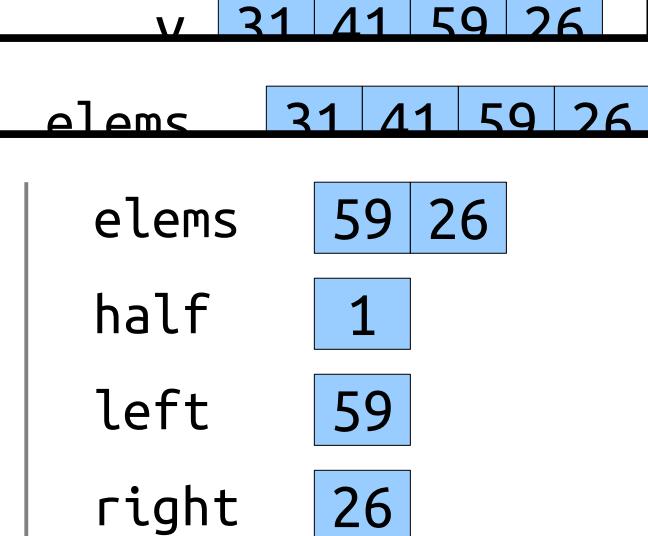
Tracing the Recursion

i

i

```
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));  
    }  
}
```

59



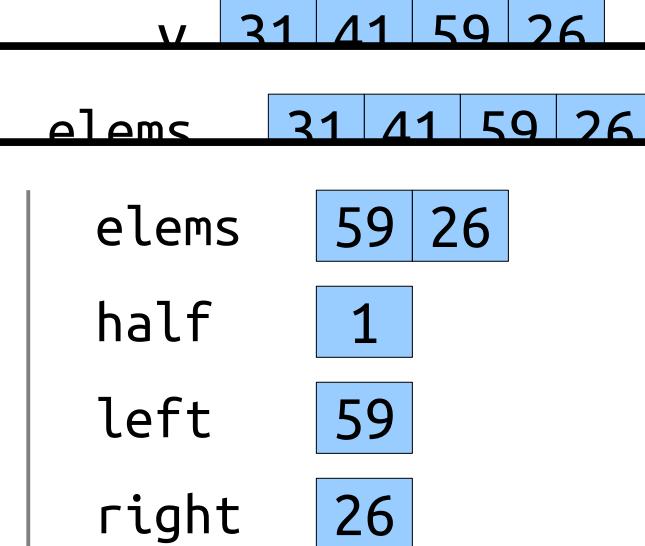
Tracing the Recursion

i

i

```
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));  
    }  
}
```

59



Tracing the Recursion

i

```
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 **59**

v 31 41 59 26

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

Tracing the Recursion

i

```
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

59

v 31 41 59 26

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

Tracing the Recursion

i

```
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));  
    }  
}
```

v 31 41 59 26

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

59

Tracing the Recursion

i

```
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));  
    }  
}
```

59



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

Tracing the Recursion

v 31 41 59 26

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