Implementing Abstractions
Part Two
Friday Four Square!
4:15PM, Outside Gates
Dynamic Memory Allocation

```c
int* ptr;
```
Dynamic Memory Allocation

int* ptr;

? ptr
Dynamic Memory Allocation

```cpp
int* ptr;
ptr = new int[5];
```
Dynamic Memory Allocation

```cpp
int* ptr;
ptr = new int[5];
```
Dynamic Memory Allocation

```cpp
int* ptr;
ptr = new int[5];
```
Dynamic Memory Allocation

```cpp
int* ptr;
ptr = new int[5];
ptr[0] = 137;
```
Dynamic Memory Allocation

```cpp
int* ptr;
ptr = new int[5];

ptr[0] = 137;
```
Dynamic Memory Allocation

```cpp
int* ptr;
ptr = new int[5];
ptr[0] = 137;
ptr[2] = 42;
```
Dynamic Memory Allocation

```c
int* ptr;
ptr = new int[5];

ptr[0] = 137;
ptr[2] = 42;
```
Cleaning Up

- Unlike other languages like Java, in C++, you are responsible for deallocating any memory allocated with `new[]`.
- You can deallocate memory with the `delete[]` operator:
  ```
  delete[] ptr;
  ```
- This destroys the array pointed at by the given pointer, not the pointer itself.
Cleaning Up

- Unlike other languages like Java, in C++, you are responsible for deallocating any memory allocated with `new[]`.

- You can deallocate memory with the `delete[]` operator:

  ```
  delete[] ptr;
  ```

- This destroys the array pointed at by the given pointer, not the pointer itself.
Cleaning Up

- Unlike other languages like Java, in C++, you are responsible for deallocating any memory allocated with `new[]`.

- You can deallocate memory with the `delete[]` operator:

  ```cpp
delete[] ptr;
  ```

- This destroys the array pointed at by the given pointer, not the pointer itself.
Cleaning Up

• Unlike other languages like Java, in C++, you are responsible for deallocating any memory allocated with `new[]`.

• You can deallocate memory with the `delete[]` operator:

  ```cpp
delete[] ptr;
  ```

• This destroys the array pointed at by the given pointer, not the pointer itself.
Implementing Stack
An Initial Idea

- A bounded stack.
- Allocate a fixed-size array for elements.
- Add elements to the array when they're pushed.
- Remove elements from the array when they're popped.
- Report an error if we exceed the size of the array.
An Initial Idea

- **element array**
- **allocated length**: 4
- **logical length**: 0
An Initial Idea

137

- element array
- allocated length: 4
- logical length: 1
An Initial Idea

137  42

- element array
- allocated length: 4
- logical length: 2
An Initial Idea

- Element array
- Allocated length: 4
- Logical length: 3

Numbers:
- 137
- 42
- 2718
An Initial Idea

- **element array**
- **allocated length**
- **logical length**

- 137
- 42
- 2718
- 512
An Initial Idea

137  42  2718

- element array
- allocated length: 4
- logical length: 3
An Initial Idea

- Element array
- Allocated length: 4
- Logical length: 2
- Array: [137, 42]
An Initial Idea

- Element array: 137, 42, 161
- Allocated length: 4
- Logical length: 3
An Initial Idea

137  42  161  314

- element array
- allocated length: 4
- logical length: 4
Running out of Space

- Our current implementation very quickly runs out of space to store elements.
- What should we do when this happens?
An Initial Idea

137  42

element array

allocated length  4

logical length  2
An Initial Idea

137  42  161

- element array
- allocated length: 4
- logical length: 3
An Initial Idea

137  42  161  314

- element array
- allocated length: 4
- logical length: 4
An Initial Idea

Element array
allocated length
logical length

137 42 161 314

4 4 4 4
An Initial Idea

- Element array
- Allocated length: 4
- Logical length: 4

137 42 161 314
An Initial Idea

- Element array
- Allocated length: 4
- Logical length: 4

[Diagram showing an array with elements 137, 42, 161, 314]
An Initial Idea

- element array
- allocated length: 4
- logical length: 4

Array:
137 42 161 314
An Initial Idea

- Element array
- Allocated length: 4
- Logical length: 4
An Initial Idea

137  42  161  314

137  42  161  314

element array
allocated length
logical length

4
4
An Initial Idea

137  42  161  314

- element array
- allocated length: 4
- logical length: 4
An Initial Idea

137  42  161  314

element array
allocated length  4
logical length  4
An Initial Idea

Element array:
- Allocated length: 5
- Logical length: 4

Numbers:
- 137
- 42
- 161
- 314
An Initial Idea

137 42 161 314 159

element array
allocated length 5
logical length 5
An Initial Idea

- Element array:
  - Allocated length: 5
  - Logical length: 5

- Array elements:
  - 137
  - 42
  - 161
  - 314
  - 159
An Initial Idea

- Element array
- Allocated length: 5
- Logical length: 5

```
137 42 161 314 159
```
An Initial Idea

- Element array
- Allocated length: 5
- Logical length: 5
An Initial Idea

Element array

Allocated length

Logical length

<table>
<thead>
<tr>
<th>137</th>
<th>42</th>
<th>161</th>
<th></th>
<th></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>137</th>
<th>42</th>
<th>161</th>
<th>314</th>
<th>159</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>5</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>5</th>
</tr>
</thead>
</table>
An Initial Idea

137 42 161 314 159

element array
 allocated length
  5
logical length
  5
An Initial Idea

- Element array
- Allocated length: 5
- Logical length: 5

Numbers in the diagram:
- 137
- 42
- 161
- 314
- 159
An Initial Idea

A visual representation of an array with elements and allocated length. The array contains the following elements:

- 137
- 42
- 161
- 314
- 159

The allocated length is 5, and the logical length is also 5 as indicated by the boxes with the number 5.
An Initial Idea

137 42 161 314 159

element array
allocated length 5
logical length 5
An Initial Idea

```
137  42  161  314  159
```

- **element array**
- **allocated length**: 5
- **logical length**: 5
An Initial Idea

- Element array
  - Allocated length: 6
  - Logical length: 5

Array elements: 137, 42, 161, 314, 159
An Initial Idea

137  42  161  314  159  265

- element array
- allocated length: 6
- logical length: 6
Ready... set... grow!
Analyzing Our Approach

- We now have a working solution, but is it an *efficient* solution?
- Let's analyze the big-O complexity of the five operations.
Analyzing Our Approach

- We now have a working solution, but is it an *efficient* solution?
- Let's analyze the big-O complexity of the five operations.
  - `size`: $O(1)$
  - `isEmpty`: $O(1)$
  - `push`: $O(n)$
  - `pop`: $O(1)$
  - `top`: $O(1)$
What This Means

- What is the complexity of pushing $n$ elements and then popping them?
What This Means

• What is the complexity of pushing \( n \) elements and then popping them?

• Cost of the pushes:
  • \( 1 + 2 + 3 + 4 + \ldots + n \)
What This Means

• What is the complexity of pushing $n$ elements and then popping them?

• Cost of the pushes:
  • $1 + 2 + 3 + 4 + \ldots + n = \mathcal{O}(n^2)$
What This Means

• What is the complexity of pushing $n$ elements and then popping them?
• Cost of the pushes:
  • $1 + 2 + 3 + 4 + \ldots + n = O(n^2)$
• Cost of the pops:
  • $1 + 1 + 1 + 1 + \ldots + 1$
What This Means

• What is the complexity of pushing \( n \) elements and then popping them?

• Cost of the pushes:
  • \( 1 + 2 + 3 + 4 + \ldots + n = \mathcal{O}(n^2) \)

• Cost of the pops:
  • \( 1 + 1 + 1 + 1 + \ldots + 1 = \mathcal{O}(n) \)
What This Means

• What is the complexity of pushing \( n \) elements and then popping them?

• Cost of the pushes:
  • \( 1 + 2 + 3 + 4 + \ldots + n = O(n^2) \)

• Cost of the pops:
  • \( 1 + 1 + 1 + 1 + \ldots + 1 = O(n) \)

• Total cost:
What This Means

- What is the complexity of pushing $n$ elements and then popping them?

- Cost of the pushes:
  - $1 + 2 + 3 + 4 + \ldots + n = O(n^2)$

- Cost of the pops:
  - $1 + 1 + 1 + 1 + \ldots + 1 = O(n)$

- Total cost: $O(n^2)$
Validating Our Model
Speeding up the Stack
A Better Idea

Element array: (137, 42, 161, 314)

- Allocated length: 4
- Logical length: 4
A Better Idea

Element array

Allocated length

Logical length

137 42 161 314
A Better Idea

- element array
- allocated length: 4
- logical length: 4

137 42 161 314
A Better Idea

- Element array
- Allocated length: 4
- Logical length: 4

137 42

137 42 161 314
A Better Idea

- Element array
- Allocated length: 4
- Logical length: 4
A Better Idea

137  42  161  314

element array
allocated length
logical length

4
A Better Idea

element array
allocated length 4
logical length 4

137 42 161 314

137 42 161 314
A Better Idea

Element array

Allocated length

Logical length

137 42 161 314
A Better Idea

137  42  161  314

- element array
- allocated length: 4
- logical length: 4
A Better Idea

element array

allocated length 6

logical length 4

137 42 161 314
A Better Idea

Element array

Allocated length

Logical length

137  42  161  314  159
A Better Idea

137 42 161 314 159 265

element array
allocated length
logical length
What Just Happened?

- Half of our pushes are now “easy” pushes, and half of our pushes are now “hard” pushes.
- Hard pushes still take time $O(n)$.
- Easy pushes only take time $O(1)$.
- Worst-case is still $O(n)$.
- What about the average case?
Analyzing the Work
Analyzing the Work
Analyzing the Work

We cut down the amount of work by roughly one half!
A Different Analysis
A Different Analysis
A Different Analysis
A Different Analysis
A Different Analysis
A Different Analysis
A Different Analysis
We cut down the amount of work by roughly one half!
Let's Check it Out!
A Much Better Idea

137  42

- element array
- allocated length: 2
- logical length: 2
A Much Better Idea

137  42

allocated length

element array

allocated length

2

logical length

2
A Much Better Idea

137

137 42

element array
allocated length
logical length
2
2
A Much Better Idea
A Much Better Idea

137 42

---

137 42

---

element array
allocated length 2
logical length 2
A Much Better Idea

Element array

Allocated length

Logical length

137 42
A Much Better Idea

allocated length

element array

allocated length

logical length

137 42
A Much Better Idea

137 42

allocated length

element array

4

allocated length

logical length

2
A Much Better Idea

allocated length

logical length

element array

137 42 271

4

3
A Much Better Idea

137  42  271  828

- element array
- allocated length: 4
- logical length: 4
A Much Better Idea

- Element array
- Allocated length: 4
- Logical length: 4

137 42 271 828
A Much Better Idea

137 42 271 828

element array
allocated length
logical length

4

4
A Much Better Idea

- Element array
- Allocated length: 4
- Logical length: 4

```
137  42  271  828
```

```
137  42
```
A Much Better Idea

Element array

Allocated length

Logical length

137 42 271

828
A Much Better Idea

allocated length

logical length

element array

137  42  271  828

allocated length

logical length

137  42  271  828
A Much Better Idea

- allocated length
- logical length

Element array

- 137
- 42
- 271
- 828

---

- 137
- 42
- 271
- 828

---

- 4

---

- 4
A Much Better Idea

137 | 42 | 271 | 828

element array
allocated length
logical length
A Much Better Idea

element array
allocated length
logical length

137  42  271  828
A Much Better Idea

<table>
<thead>
<tr>
<th>element array</th>
<th>allocated length</th>
<th>logical length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>

| 137 | 42 | 271 | 828 |

Diagram showing a better idea with allocated length and logical length.
A Much Better Idea

```
137  42  271  828  182
```

- element array
- allocated length: 8
- logical length: 5
A Much Better Idea

**element array**

allocated length: 8

logical length: 6

| 137 | 42  | 271 | 828 | 182 | 845 |
A Much Better Idea

```
137  42  271  828  182  845  904
```

- **element array**
- **allocated length**: 8
- **logical length**: 7
A Much Better Idea

element array
allocated length
logical length

allocated length
Let's Give it a Try!
How do we analyze this?
Spreading the Work
Spreading the Work
Spreading the Work
Spreading the Work
Spreading the Work
Spreading the Work
Spreading the Work
Spreading the Work
Spreading the Work
Spreading the Work
Spreading the Work

On average, we do just 3 units of work!

This is $O(1)$ work on average!
Sharing the Burden

- We still have “heavy” pushes taking time $O(n)$ and “light” pushes taking time $O(1)$.
- Worst-case time for a push is $O(n)$.
- Heavy pushes become so rare that the average time for a push is $O(1)$.
- Can we confirm this?
Amortized Analysis

• The analysis we have just done is called an **amortized analysis**.

• Reason about the total amount of work done, not the word done per operation.

• In an amortized sense, our implementation of the stack is extremely fast!

• This is one of the most common approaches to implementing **Stack**.