Implementing Abstractions
Part Two
Previously, on CS106B...
A Bounded Stack

element array
allocated size
logical size

allocated size
logical size
A Bounded Stack

The stack’s *allocated size* is the number of slots in the array. Remember – arrays in C++ cannot grow or shrink.
A Bounded Stack

The stack’s \textit{allocated size} is the number of slots in the array. Remember – arrays in C++ cannot grow or shrink.

The stack’s \textit{logical size} is the number of elements actually stored in the stack. This lets us track how much space we’re actually using.
A Bounded Stack

- Element array: 137
- Allocated size: 4
- Logical size: 1
A Bounded Stack

- Element array
- Allocated size: 4
- Logical size: 2

Numbers: 137, 42
A Bounded Stack

- Element array: [137, 42, 2718]
- Allocated size: 4
- Logical size: 3
A Bounded Stack

137  42  2718  512

- element array
- allocated size: 4
- logical size: 4
A Bounded Stack

The diagram illustrates a bounded stack with the following attributes:

- **Element Array**: Contains elements 137, 42, 2718, and 512.
- **Allocated Size**: 4
- **Logical Size**: 3

The stack is bounded by these parameters, limiting the capacity and memory usage.
A Bounded Stack

Arrays cannot grow or shrink, so this older value is still technically there in the array. We’re just going to pretend it isn’t.
A Bounded Stack

allocated size

logical size

element array

allocated size

logical size

137  42  2718  512
A Bounded Stack

- **Element Array**: 137, 42, 161, 512
- **Allocated Size**: 4
- **Logical Size**: 3
A Bounded Stack

[Diagram with numbers 137 42 161 314 and labeled boxes for element array, allocated size, and logical size]
New Stuff!
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

Element array

Allocated size: 4

Logical size: 2

<table>
<thead>
<tr>
<th>137</th>
<th>42</th>
<th></th>
</tr>
</thead>
</table>
An Initial Idea

- element array: 137, 42, 161
- allocated size: 4
- logical size: 3
An Initial Idea

137 | 42 | 161 | 314

- **Element Array**
- **Allocated Size**: 4
- **Logical Size**: 4
An Initial Idea

- Element array: allocated size
- Logical size: 4
- Array values: 137, 42, 161, 314
An Initial Idea

- Element array
- Allocated size: 4
- Logical size: 4

137  42  161  314

- Red box: Allocated size
- Yellow box: Logical size
An Initial Idea

- Element array
- Allocated size: 4
- Logical size: 4

137 42
137 42 161 314
An Initial Idea

137  42  161

137  42  161  314

element array
allocated size
4
logical size
4
An Initial Idea

element array
allocated size
logical size

137 42 161 314

137 42 161 314
An Initial Idea

Dynamic Deallocation!

element array
allocated size
logical size

4

4

4

137 42 161 314
An Initial Idea

137  42  161  314

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

element array
allocated size
logical size

137  42  161  314
An Initial Idea

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

137 42 161 314
An Initial Idea

Element array
allocated size
logical size

137 42 161 314 159

5 5
An Initial Idea

allocated size

logical size

element array

allocated size

logical size

137 42 161 314 159
An Initial Idea

allocated size

logical size

element array

allocated size

logical size

137 42 161 314 159
An Initial Idea

- **Element array:**
  - Allocated size: 5
  - Logical size: 5

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

- Element array
- Allocated size: 5
- Logical size: 5

The diagram illustrates an initial idea with an element array, showing allocated and logical sizes.
An Initial Idea

137 42 161 314

137 42 161 314 159

element array
allocated size
logical size

5

5
An Initial Idea

137  42  161  314  159

137  42  161  314  159

allocated size

element array

allocated size

logical size

5
An Initial Idea

```
137  42  161  314  159
```

Dynamic Deallocation!

```
element
array
```
```
allocated
size
```
```
logical
size
```

```
delete[]
```
An Initial Idea

137  42  161  314  159

- element array
- allocated size: 5
- logical size: 5
An Initial Idea

element array
allocated size
logical size

allocated size
logical size

137 42 161 314 159
An Initial Idea

137 | 42 | 161 | 314 | 159

- element array
- allocated size: 6
- logical size: 5
An Initial Idea

137  42  161  314  159  265

element array
allocated size 6
logical size 6
Ready... set... grow!
class OurStack {
public:
    OurStack();
    ~OurStack();

    void push(int value);
    int pop();
    int peek() const;

    int size() const;
    bool isEmpty() const;

private:

    int* elesms;
    int allocatedSize;
    int logicalSize;
};
class OurStack {
public:
    OurStack();
    ~OurStack();

    void push(int value);
    int pop();
    int peek() const;

    int size() const;
    bool isEmpty() const;

private:
    void grow();

    int* elems;
    int allocatedSize;
    int logicalSize;
};

This is a private member function. It’s a helper function only the implementation can call.
An Initial Idea

- Allocated size: 4
- Logical size: 4
- Elements: 137, 42, 161, 314
An Initial Idea

void OurStack::grow() {
  allocatedSize++;
}

void OurStack::grow() {
  allocatedSize++;
  int* newElems = new int[allocatedSize];
  for (int i = 0; i < size(); i++) {
    newElems[i] = elems[i];
  }
  delete[] elems;
  elems = newElems;
}
An Initial Idea

void OurStack::grow() {
    allocatedSize++;
    int * newElems = new int[allocatedSize];
    for (int i = 0; i < size(); i++) {
        newElems[i] = elems[i];
    }
    delete [] elems;
    elems = newElems;
}
An Initial Idea

void OurStack::grow() {
    allocatedSize++;
    int* newElems = new int[allocatedSize];
    for (int i = 0; i < size(); i++) {
        newElems[i] = elems[i];
    }
    delete[] elems;
    elems = newElems;
}
An Initial Idea

void OurStack::grow() {
    allocatedSize++;
    int* newElems = new int[allocatedSize];
    for (int i = 0; i < size(); i++) {
        newElems[i] = elems[i];
    }
    delete[] elems;
    elems = newElems;
}
void OurStack::grow() {
    allocatedSize++;

    int* newElems = new int[allocatedSize];

    for (int i = 0; i < size(); i++) {
        newElems[i] = elems[i];
    }
}

An Initial Idea

void OurStack::grow() {
    allocatedSize++;

    int* newElems = new int[allocatedSize];

    for (int i = 0; i < size(); i++) {
        newElems[i] = elems[i];
    }

    delete [] elems;
    elems = newElems;
}
void OurStack::grow() {
    allocatedSize++;
    int* newElems = new int[allocatedSize];
    for (int i = 0; i < size(); i++) {
        newElems[i] = elems[i];
    }
    delete[] elems;
    elems = newElems;
}
void OurStack::grow() {
    allocatedSize++;

    int* newElems = new int[allocatedSize];

    for (int i = 0; i < size(); i++) {
        newElems[i] = elems[i];
    }

    delete[] elems;
    elems = newElems;
}
An Initial Idea

void OurStack::grow() {
    allocatedSize++;

    int* newElems = new int[allocatedSize];

    for (int i = 0; i < size(); i++) {
        newElems[i] = elems[i];
    }

    delete[] elems;
    elems = newElems;
}
An Initial Idea

If a and b are pointers, then

```cpp
void OurStack::grow() {
    allocatedSize++;
    int* newElems = new int[allocatedSize];
    for (int i = 0; i < size(); i++) {
        newElems[i] = elems[i];
    }
    delete[] elems;
    elems = newElems;
}
```
void OurStack::grow() {
    allocatedSize++;

    int* newElems = new int[allocatedSize];

    for (int i = 0; i < size(); i++) {
        newElems[i] = elems[i];
    }

    delete[] elems;
    elems = newElems;
}
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. As usual, let $n$ denote the number of items in the stack when the operation is performed.
  - size:
  - isEmpty:
  - push:
  - pop:
  - peek:
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. As usual, let $n$ denote the number of items in the stack when the operation is performed.
  - size: $O(1)$
  - isEmpty: $O(1)$
  - push:
  - pop:
  - peek:
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. As usual, let \( n \) denote the number of items in the stack when the operation is performed.
  - size: \( O(1) \)
  - isEmpty: \( O(1) \)
  - push: \( O(n) \)
  - pop:
  - peek:
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. As usual, let $n$ denote the number of items in the stack when the operation is performed.
  - size: $O(1)$
  - isEmpty: $O(1)$
  - push: $O(n)$
  - pop: $O(1)$
  - peek: $O(1)$
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:
  - $1 + 2 + 3 + 4 + \ldots + n$

\[ 1 + 2 + 3 + 4 + \ldots + n = O(n^2) \]

\[ 1 + 1 + 1 + 1 + \ldots + 1 = O(n) \]

Total cost: $O(n^2)$
What This Means

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

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

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

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

• Cost of the pops:
  • $1 + 1 + 1 + 1 + ... + 1$
What This Means

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

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

• Cost of the pops:
  • \( 1 + 1 + 1 + 1 + ... + 1 = O(n) \)
What This Means

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

- What is the complexity of pushing $n$ elements and then popping them?
- Cost of the pushes:
  - $1 + 2 + 3 + 4 + \ldots + n = \mathcal{O}(n^2)$
- Cost of the pops:
  - $1 + 1 + 1 + 1 + \ldots + 1 = \mathcal{O}(n)$
- Total cost: $\mathcal{O}(n^2)$
Validating Our Model
Speeding up the Stack
Key Idea: *Plan for the Future*
A Better Idea

allocated size

logical size

element array

allocated size

logical size

137 42 161 314
A Better Idea

element array
allocated size
logical size

4

137 42 161 314
A Better Idea

137

137 42 161 314

element
array

allocated
size

logical
size
A Better Idea

allocated size

logical size

element array

allocated size

logical size

4

4

137 42

137 42 161 314
A Better Idea

allocated size

logical size

element array

allocated size

logical size

137 42 161

137 42 161 314
A Better Idea

Element array

Allocated size

Logical size

137 42 161 314
A Better Idea

Dynamic Deallocation!

<table>
<thead>
<tr>
<th>element array</th>
<th>delete[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocated size</td>
<td>4</td>
</tr>
<tr>
<td>logical size</td>
<td>4</td>
</tr>
</tbody>
</table>
A Better Idea

Element array

Allocated size

Logical size

137  42  161  314
A Better Idea

element array

allocated size

logical size

137  42  161  314
A Better Idea

- Element array: 137, 42, 161, 314
- Allocated size: 6
- Logical size: 4
A Better Idea

```
137  42  161  314  159
```

### element array

- allocated size: 6
- logical size: 5
A Better Idea

137  42  161  314  159  265

- element array
- allocated size: 6
- logical size: 6
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
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
A Different Analysis

We cut down the amount of work by roughly one half!
How does it stack up?
A Much Better Idea

137  42

element array
allocated size
logical size

2

2
A Much Better Idea

element array

allocated size

logical size

137

42

2
A Much Better Idea

137

137 42

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

allocated size

element array

allocated size

logical size

137 42

137 42

2

2
A Much Better Idea

Dynamic Deallocation!

- Element array
- Allocated size: 2
- Logical size: 2

137 42
A Much Better Idea

137 42

- element array
- allocated size
- logical size

2
A Much Better Idea

- element array
- allocated size: 2
- logical size: 2

137  42
A Much Better Idea

Element array

Allocated size

Logical size

137 42
A Much Better Idea

137  42  271

- element array
- allocated size: 4
- logical size: 3
A Much Better Idea

137  42  271  828

element array
allocated size 4
logical size 4
A Much Better Idea
A Much Better Idea

element array
allocated size
logical size

137

42

271

828

4

4
A Much Better Idea

```plaintext
137  42
```

```
137  42  271  828
```

```
element
array
```

```
allocated
size
```

```
logical
size
```

```
4
```

```
4
```
A Much Better Idea

- **element array**: [137, 42, 271, 828]
- **allocated size**: 4
- **logical size**: 4
## A Much Better Idea

<table>
<thead>
<tr>
<th>137</th>
<th>42</th>
<th>271</th>
<th>828</th>
</tr>
</thead>
</table>

- **element array**
- **allocated size**: 4
- **logical size**: 4
A Much Better Idea

Dynamic Deallocation!

 Allocate size

Delete

Element array

4

4

Allocated size

Logical size

4
A Much Better Idea

allocated size

logical size

137  42  271  828

element array

allocated size

logical size

4
A Much Better Idea

element array

allocated size

logical size

137  42  271  828
A Much Better Idea

137 | 42 | 271 | 828 |   |   |   |   

allocated size

logical size

element array

allocated size

8

logical size

4
A Much Better Idea

element array
allocated size
logical size

137 42 271 828 182
A Much Better Idea

![Element Array Diagram]

- **Element Array**: 137, 42, 271, 828, 182, 845
- **Allocated Size**: 8
- **Logical Size**: 6
A Much Better Idea

137  42  271  828  182  845  904

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

Element array: 8

Allocated size: 8

Logical size: 8

Array: [137, 42, 271, 828, 182, 845, 904, 5]
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 $O(1)$.
- Heavy pushes become so rare that the average time for a push is $O(1)$.
- Cost of $n$ pushes:
  - $1 + 1 + 1 + ... + 1 = O(n)$.
- Cost of $n$ pops:
  - $1 + 1 + 1 + ... + 1 = O(n)$.
- Total work done: $O(n)$.
- Can we confirm this?
Amortized Analysis

- The analysis we have just done is called an *amortized analysis*.
- We reason about the total work done, not the work 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.
Your Action Items

- **Keep working on Assignment 5**
  - Haven’t started yet? Not a problem! You’ve got time if you make slow and steady progress from here on out.
  - Need help? Stop by the LaIR!
Next Time

• *Hash Functions*
  • A magical and wonderful gift from the world of mathematics.

• *Hash Tables*
  • How do we implement HashMap and HashSet?