Collections, Part Two
Stack
Stack

• A **Stack** is a data structure representing a stack of things.
• Objects can be **pushed** on top of the stack or **popped** from the top of the stack.
• No other objects in the stack are visible.
• Example: Function calls
Stack

- A **Stack** is a data structure representing a stack of things.
- Objects can be **pushed** on top of the stack or **popped** from the top of the stack.
- No other objects in the stack are visible.
- Example: Function calls
A **Stack** is a data structure representing a stack of things. Objects can be **pushed** on top of the stack or **popped** from the top of the stack. No other objects in the stack are visible. Example: Function calls
Stack

• A **Stack** is a data structure representing a stack of things.
• Objects can be **pushed** on top of the stack or **popped** from the top of the stack.
• No other objects in the stack are visible.
• Example: Function calls
Stack

- A **Stack** is a data structure representing a stack of things.
- Objects can be **pushed** on top of the stack or **popped** from the top of the stack.
- No other objects in the stack are visible.
- Example: Function calls
Stack

- A **Stack** is a data structure representing a stack of things.
- Objects can be **pushed** on top of the stack or **popped** from the top of the stack.
- No other objects in the stack are visible.
- Example: Function calls
Wait a Minute...

- But couldn't we just use a Vector for this?
- To push, just append:
  \[ v += \text{elem} \]
- To pop, remove the last element:
  \[ v\.removeAt(v\.length() - 1); \]
Stacks Matter

• There are several major advantages to using a stack.

• **Conceptual simplicity:**
  - Describing a problem as a stack rather than a vector more precisely describes the problem.
  - Recognizing this use pattern sheds light on the structure of multiple related problems.

• **Implementation efficiency:**
  - Stacks can be implemented slightly more efficiently than vectors; more on that later.
Balancing Parentheses

int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```

^
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```

^
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```

^
Balancing Parentheses

int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
^
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```

^
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```

^  

(}
Balancing Parentheses

int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }

^
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```

Balancing Parentheses

int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
Balancing Parentheses

```c
int foo() {
    if (x * (y + z[1]) < 137) {
        x = 1;
    }
}
```
Balancing Parentheses

int foo() { if (x * (y + z[1]) < 137) { x = 1; } }

^
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }

Balancing Parentheses
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) ^ { x = 1; } }
```
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```

Balancing Parentheses

int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
Balancing Parentheses

int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

int foo() { if (x * (y + z[1]) < 137) { x = 1; } ^

{ }

{ 

{ }

}
Balancing Parentheses

int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
Balancing Parentheses

int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
Balancing Parentheses

int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }

Balancing Parentheses
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```

```c
^ { }
```
Balancing Parentheses

int foo() { if (x * (y + z[1]) < 137) { x = 1; } }

^
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Balancing Parentheses

```c
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```
Queue
Queue

- A **Queue** is a data structure representing a waiting line.
- Objects can be **enqueued** to the back of the line or **dequeued** from the front of the line.
- No other objects in the queue are visible.
- Example: A checkout counter.
Queue

- A **Queue** is a data structure representing a waiting line.
- Objects can be **enqueued** to the back of the line or **dequeued** from the front of the line.
- No other objects in the queue are visible.
- Example: A checkout counter.
Queue

- A **Queue** is a data structure representing a waiting line.
- Objects can be **enqueued** to the back of the line or **dequeued** from the front of the line.
- No other objects in the queue are visible.
- Example: A checkout counter.
Queue

- A **Queue** is a data structure representing a waiting line.
- Objects can be **enqueued** to the back of the line or **dequeued** from the front of the line.
- No other objects in the queue are visible.
- Example: A checkout counter.
Queue

- A **Queue** is a data structure representing a waiting line.
- Objects can be **enqueued** to the back of the line or **dequeued** from the front of the line.
- No other objects in the queue are visible.
- Example: A checkout counter.
Queue

• A **Queue** is a data structure representing a waiting line.

• Objects can be **enqueued** to the back of the line or **dequeued** from the front of the line.

• No other objects in the queue are visible.

• Example: A checkout counter.
Queue

- A **Queue** is a data structure representing a waiting line.
- Objects can be **enqueued** to the back of the line or **dequeued** from the front of the line.
- No other objects in the queue are visible.
- Example: A checkout counter.
Queue

- A **Queue** is a data structure representing a waiting line.
- Objects can be **enqueued** to the back of the line or **dequeued** from the front of the line.
- No other objects in the queue are visible.
- Example: A checkout counter.
Listing All Strings

- Suppose we want to generate all strings of letters A, B, and C of length at most three.
- How might we do this?
How to Remember All This?
An Amazingly Useful Link

http://www.stanford.edu/class/cs106b/materials/cppdoc/
Map
Map

- The `Map` class represents a set of key/value pairs.
- Each key is associated with a unique value.
- Given a key, can look up the associated value.
Map

- The **Map** class represents a set of key/value pairs.
- Each key is associated with a unique value.
- Given a key, can look up the associated value.
Map

• The **Map** class represents a set of key/value pairs.

• Each key is associated with a unique value.

• Given a key, can look up the associated value.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CS106B</td>
<td>Awesome!</td>
</tr>
<tr>
<td>Ibex</td>
<td>Cute!</td>
</tr>
</tbody>
</table>
Map

- The **Map** class represents a set of key/value pairs.
- Each key is associated with a unique value.
- Given a key, can look up the associated value.

<table>
<thead>
<tr>
<th>CS106B</th>
<th>Awesome!</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ibex</td>
<td>Cute!</td>
</tr>
<tr>
<td>This Slide</td>
<td>Self Referential</td>
</tr>
</tbody>
</table>
Map

- The **Map** class represents a set of key/value pairs.
- Each key is associated with a unique value.
- Given a key, can look up the associated value.

<table>
<thead>
<tr>
<th>CS106B</th>
<th>Awesome!</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ibex</td>
<td><strong>Very</strong> Cute!</td>
</tr>
<tr>
<td>This Slide</td>
<td>Self Referential</td>
</tr>
</tbody>
</table>
Using the Map

- To use the map, you must specify both the key type and the value type:
  \[
  \text{Map<} \text{KeyType, ValueType}> \text{ map;}
  \]
- You can add or change a key/value pair by writing
  \[
  \text{map[} \text{key}] = \text{value;}
  \]
- You can read the value associated with a key by writing
  \[
  \text{map[} \text{key]}
  \]
  If no value exists, a new key/value pair is automatically added for you. The value is initialized to a sensible default.
- You can check whether a key exists in the map by calling
  \[
  \text{map.containsKey(} \text{key)}
  \]
What states have the most cities/towns in them?

What states have the fewest?
**foreach**

- You can loop the elements of any collection class using the **foreach** macro:

```cpp
foreach (type var in collection) {
    /* ... do something with var ... */
}
```

- **foreach** is **not** a part of standard C++; it's a **macro** that we've built to keep things simple.

- Oh, and the implementation of **foreach** will make you go blind. You've been warned.
Ordering in foreach

• When using foreach to iterate over a collection:
  • In a Vector, string, or array, the elements are retrieved in order.
  • In a Map, the keys are returned in sorted order.
  • In a Set or Lexicon (more on them later), the values are returned in sorted order.
  • In a Grid, the elements of the first row are returned in order, then the second row, etc. (this is called row-major order).