Topics:

- **Last time: Classes, Part 1**
  - BankAccount class
  - Ball class

- **Today: Classes, Part 2**
  - More practice making our own classes!
  - This time we will implement one of our ADTs from earlier in the quarter!!
    - A simple **Stack ADT** with unlimited capacity
  - In doing so, we need to learn about:
    - **C/C++ arrays**
    - **Dynamic memory allocation** (this is a huge topic in itself—much of CS107 is about this)
Implementing a classic ADT: Stack

Today let's learn how to write a Stack class

- We will implement a stack
- Not *quite* like the one in Stanford library—for simplicity this *only stores* `int`
- The stack will use an array to store its elements
- The capacity will grow as needed

Recall the basic stack operations:
- **push**: Add an element to the top.
- **pop**: Remove the top element.
- **peek**: Examine the top element.
Inside our Stack

Inside a Stack (also true of Queue and Vector) is an **array** storing the elements you have added.

- Typically the array is larger than the data added so far, so that it has some extra slots ready to go to put new elements later.
- Our stack will use the same array-based technique

```
// Diagram shows the internal state of the Stack class
// after 3 ints are pushed
Stack<int> s;
s.push(42);
s.push(-5);
s.push(17);
```

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>42</td>
<td>-5</td>
<td>17</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>size</td>
<td>3</td>
<td>capacity</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Inside our Stack

Inside a Stack (also true of Queue and Vector) is an **array** storing the elements you have added.

- Typically the array is larger than the data added so far, so that it has some extra slots ready to go to put new elements later.
- Our stack will use the same array-based technique

```
// Diagram shows the internal state of the Stack
// after 3 ints are pushed
Stack<int> s;
s.push(42);
s.push(-5);
s.push(17);
```

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>42</td>
<td>-5</td>
<td>17</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>size</td>
<td>3</td>
<td>capacity</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Inside our Stack

Inside a Stack (also true of Queue and Vector) is an array storing the elements you have added.

- Typically the array is larger than the data added so far, so that it has some extra slots ready to go to put new elements later.
- Our stack will use the same array-based technique

```cpp
// Diagram shows the internal state of the Stack class
// after 3 ints are pushed
Stack<int> s;
s.push(42);
s.push(-5);
s.push(17);
```

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>values</td>
<td>42</td>
<td>-5</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>capacity</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Quick check: which end will we consider the “top” of the stack?

Our class member variables will include size and capacity. And this storage area that is a C/C++ array.
Arrays in C++

BEHIND THE SCENES TOUR!
Basic Array in C/C++

```c
#define type name[length];
```

- An array is has enough space for multiple values of a type
  - If a regular variable is a single-family home, arrays are an apartment building
  - Similar concept as a Vector, but much more basic
    - Can’t ever be resized
    - No methods
    - Really just several adjacent spaces of the same type

Example:
```c
int homeworkGrades[7];
homeworkGrades[0] = 90;
homeworkGrades[3] = 95;
```
Arrays in a memory diagram

int myFunction() {
    int x = 5;
    int y = 3;
    int stackArr[3];
    stackArr[0] = x + 1;
    stackArr[1] = y + 1;
    stackArr[2] = x + y;

    return y;
}

What happens when myFunction() returns?
Arrays in a memory diagram

```c
int myFunction() {
    int x = 5;
    int y = 3;
    int stackArr[3];
    stackArr[0] = x + 1;
    stackArr[1] = y + 1;
    stackArr[2] = x + y;

    return y;
}
```

What happens when myFunction() returns?
A second kind of array in C/C++

```c
type name[length];
```

- **Basic array (AKA statically allocated or stack allocated)**
  - Stored in the stack frame alongside other local variables

  Example: `int homeworkGrades[7];`

```c
type* name = new type[length];
```

- **Dynamically allocated array (AKA heap allocated)**
  - The variable that refers to the array is called a pointer, and it is on the stack
  - But the actual array is stored in the heap!

  Example: `int* homeworkGrades = new int[7];`

- Literally the word “new”!
Arrays in a memory diagram

```cpp
int myFunction() {
    int x = 5;
    int y = 3;
    int* heapArr = new int[3];
    heapArr[0] = x + 1;
    heapArr[1] = y + 1;
    heapArr[2] = x + y;

    return y;
}
```

What happens when myFunction() returns?
Arrays in a memory diagram

```c
int myFunction() {
    int x = 5;
    int y = 3;
    int* heapArr = new int[3];
    heapArr[0] = x + 1;
    heapArr[1] = y + 1;
    heapArr[2] = x + y;

    return y;
}
```

What happens when myFunction() returns?
Arrays in a memory diagram

```cpp
int myFunction() {
    int x = 5;
    int y = 3;
    int* heapArr = new int[3];
    heapArr[0] = x + 1;
    heapArr[1] = y + 1;
    heapArr[2] = x + y;
    delete [] heapArr;
    return y;
}
```

What happens when `myFunction()` returns?
Dynamic Memory Allocation

Keywords new and delete
Memory leaks

- The pointer variable that points to heap allocated memory is like the string on a helium balloon.
- If you let go of the string (or lose that pointer variable), the balloon still exists out there somewhere, but it’s never yours to play with ever again. 😭
  - Also it’s polluting the environment.
Always a pair: new and delete

- Think of **new** as making a hotel room reservation.
  - **new** int[5]
  - “I’d like 5 connecting rooms, each big enough for 1 int value, please.”

- Think of **delete** as checking out of the hotel room.
  - **delete** [] arr
  - “My trip is done. Stop charging me for these rooms, and you can give them to other guests.”
Always a pair: new and delete

Many things can go wrong with dynamic memory that are analogous to the hotel situation:

- Leave town but forget to check out—you’ll keep getting charged for the room and it can’t go to another guest
  - When you forget delete, you get a memory leak
- Check out of the room but then try to go back in—another guest might already be using it and will be very angry!
  - After you call delete, be sure not to try to use that memory again!

```cpp
int* arr = new int[10];
...
delete [] arr;
arr[0] = 5;  // no!!
```
Always a pair: `new` and `delete`

```c
int myFunction() {
    int x = 5;
    int y = 3;
    int* heapArr = new int[3];
    heapArr[0] = x + 1;
    heapArr[1] = y + 1;
    heapArr[2] = x + y;
    delete [] heapArr;  // fixed leak!
    return y;
}
```
Always a pair: new and delete

```c
int myFunction() {
    int x = 5;
    int y = 3;
    int* heapArr = new int[3];
    heapArr[0] = x + 1;
    heapArr[1] = y + 1;
    heapArr[2] = x + y;
    delete [] heapArr; // fixed leak!
    return y;
}
```

Q: “Why would you want to do that?”
A: It’s true that there’s no point to using dynamic allocation if we are just deleting at the end of the function. Choose a static array instead to automatically release. Dynamic allocation is for when you want the data to last so you can keep using it.
Destructor (12.3)

// ClassName.h  // ClassName.cpp
~ClassName();  ClassName::~ClassName() { ... }  

Destructor: Called when the object is deleted by the program
- When the object goes out of {} scope; opposite of a constructor
- (or when you expressly call “delete” on the object, if heap-allocated)
- Useful if your object needs to do anything important as it dies, such as freeing any array memory used by its fields
#ifndef _arraystack_h
#define _arraystack_h

class ArrayStack {
public:
    ArrayStack();
    ~ArrayStack();
    void push(int n);
    int pop();
    int peek() const;
    bool isEmpty() const;

private:
    int* _elements;
    int _capacity;
    int _size;

    void checkResize();
};

#endif
#include "arraystack.h"

ArrayStack::ArrayStack() {
    _elements = new int[10];
    _capacity = 10;
    _size = 0;
}

ArrayStack::~ArrayStack() {
    delete[] _elements;
}

bool ArrayStack::isEmpty() const {
    return _size == 0;
}

void ArrayStack::push(int n) {
    _elements[_size] = n;
    _size++;
}
```
int ArrayStack::pop() {
    if (isEmpty()) {
        throw "Can't pop from an empty stack!";
    }
    int result = _elements[_size - 1];
    _elements[_size - 1] = 0;
    _size--;
    return result;
}

int ArrayStack::peek() const {
    if (isEmpty()) {
        throw "Can't peek from an empty stack!";
    }
    return _elements[_size - 1];
}
```
Resize when out of space

// grows array to twice the capacity if needed
void ArrayStack::checkResize() {
    if (_size == _capacity) {
        // create bigger array and copy data over
        int* bigger = new int[2 * _capacity]();
        for (int i = 0; i < _capacity; i++) {
            bigger[i] = _elements[i];
        }
        delete[] _elements;
        _elements = bigger;
        _capacity *= 2;
    }
}

| index |  0  |  1  |  2  |  3  |  4  |  5  |  6  |  7  |  8  |  9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|-------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|----|----|----|----|----|----|----|----|----|
| value |  3  |  8  |  9  |  7  |  5  | 12  |  4  |  8  |  1  |  6  | 75 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |
| size  | 11  | 12  | 13  | 14  | 15  | 16  | 17  | 18  | 19  | 20  | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
Overflow (extra) slides

FOR THE ADVANCED AND/OR CURIOUS STUDENT
Shallow copy bug (12.7)

If one stack is assigned to another, they will share one array.

- ArrayStack stack1;
- ArrayStack stack2 = stack1;

A change to one will affect the other. (That's bad!)

- stack2.pop();
- stack1.push(88);

When they fall out of scope, memory could get freed twice (error!)
Deep copy

To correct the shallow copy bug, we must define:

- a copy constructor (constructor that takes a list as a parameter)
  \[\text{ArrayStack}(\text{const ArrayStack}\& \ stack);\]

- an assignment operator (overloaded = op between two lists)
  \[\text{ArrayStack}\& \ \text{operator } = (\text{const ArrayStack}\& \ stack);\]

  - in both of these, we will make a deep copy of the array of elements.

Rule of Three: In C++, when you define one of these three items in your class, you probably should define all three:

- 1) copy constructor  2) assignment operator  3) destructor
Advanced: Forbid copying

One quick fix is to just forbid your objects from being copied.

- Declare a private copy constructor and = operator in the .h file.
- Don't give them any actual definition/body in the .cpp file.

```cpp
// in arraystack.h
private:
    ArrayStack(const ArrayStack& stack);
    ArrayStack& operator =(const ArrayStack& stack);
```

- Now if the client tries stack2 = stack1; it will not compile.
- Solves the shallow copy problem, but restrictive and less usable.