Topics:

- Big-O extra review and tips (for midterm)

- Classes, Part 2
  - More practice making our own classes
  - This time we will implement one of our ADTs from earlier in the quarter
    - In section you did a **queue** with **limited** capacity
    - In lecture we’ll do a **stack** with **unlimited** capacity
Big-O

Applying to algorithms
Applying Big-O to Algorithms

- Code example:

```cpp
for (int i = data.size() - 1; i >= 0; i--){
    for (int j = 0; j < data.size(); j++){
        cout << data[i] << data[j] << endl;
    }
}
```

is $O(n^2)$, where $n$ is `data.size()`.

*Pro tip: for loops inside loops, MULTIPLY the individual loop counts*
Applying Big-O to Algorithms

- Code example:

```cpp
for (int i = data.size() - 1; i >= 0; i -= 3){
    for (int j = 0; j < data.size(); j += 3){
        cout << data[i] << data[j] << endl;
    }
}
```

is $O(n^2)$, where $n$ is $\text{data.size()}$.

Pro tip: for loops inside loops, MULTIPLY the individual loop counts
Applying Big-O to Algorithms

- Code example:
  
  ```cpp
  for (int i = data.size() - 1; i >= 0; i -= 3) {
    for (int j = 0; j < data.size(); j += 3) {
      cout << data[i] << data[j] << endl;
    }
  }
  cout << BinarySearch(data, 37) << endl;
  ```

  is \( O(n^2) \), where \( n \) is `data.size()`.

  *Pro tip: for code in sequence (not “nested” inside), ADD the individual counts*
Applying Big-O to Algorithms

- Code example:
  ```cpp
  for (int i = 0; i < 5; i++) {
      cout << data[i] << endl;
  }
  ``

  is $O(1)$ (this is called “constant time”), where $n$ is `data.size()`.

(OK to make assumption that `data` is at least as large as needed so this will not crash)
Implementing an ADT: Stack

Plus pointers and dynamic memory!
Implementing a classic ADT: Stack

Today let's learn how to write a class representing an ADT

- 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.
How Vector/Stack works

Inside a 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 in which 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>
Two kinds of arrays in C/C++

type* name = new type[length];

› A dynamically allocated (AKA heap-allocated) array
› The variable that refers to the array is called a pointer
› The memory allocated for the array must be manually released, or else the program will have a memory leak

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

type name[length];

› A statically allocated (stack-allocated) array; can never be resized.
› Memory does not need to be freed; will be automatically released.

› Example: int homeworkGrades[7];
Danger in C/C++: uninitialized memory!

type* name = new type[length];    // uninitialized
type* name = new type[length]();  // initialized with zeroes

› If () are written after [], all elements are zeroed out (slower but good if needed)
› If () are missing, the elements store uninitialized (“random”/garbage) values

int* a = new int[3];
cout << a[0];                       // 2395876
cout << a[1];                       // -197630894

int* a2 = new int[3]();
cout << a[0];                       // 0
cout << a[1];                       // 0
Arrays in a memory diagram

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

    return y; // bad -- memory leak!
}
```

What happens when `myFunction()` returns?
int myFunction() {
    int x = 5;
    int y = 3;
    int* heapArr = new int[3];
    heapArr[0] = x;
    heapArr[1] = y;
    heapArr[2] = x + y;

    return y; // bad -- memory leak!
}

What happens when myFunction() returns?
Always a pair: new and delete

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

Memory

main()

- myFunction’s stack frame automatically released
- Heap array manually released by delete []
**Always a pair: new and delete**

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

“Why would you want to do that?”

No point if we are just deleting at the end of the function when it would automatically release anyway. But what if we want to return the array?
Always a pair: new and delete

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

delete [] heapArr; // to be done later!
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;
    }
}
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)
  `ArrayStack(const ArrayStack& stack);`

- an assignment operator (overloaded `=` op between two lists)
  `ArrayStack& operator =(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.

// 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.