Programming Abstractions

CS106B

Cynthia Bailey Lee
Julie Zelenski

Topics:

- **Friday: 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)

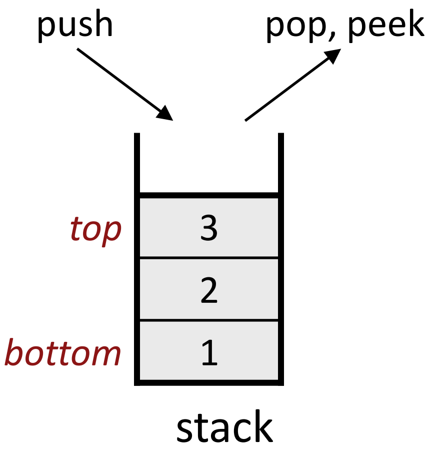
Stack Implementation

BEHIND THE SCENES TOUR!

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.

How Vector/Stack works

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 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);
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	42	-5	17	0	0	0	0	0	0	0
size	3	*capacity*	10							

How a Stack works

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 in which to put new elements later.
- Our stack will use the same array-based technique

To store our Stack, we will use a C/C++ array.

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

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*		*8*	*9*
value	42	-5	17	0	0	0	0	0	0	0
size	3	*capacity*	10							

Arrays in C++

BEHIND THE SCENES TOUR!

Two kinds of arrays in C/C++

```
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];`

```
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

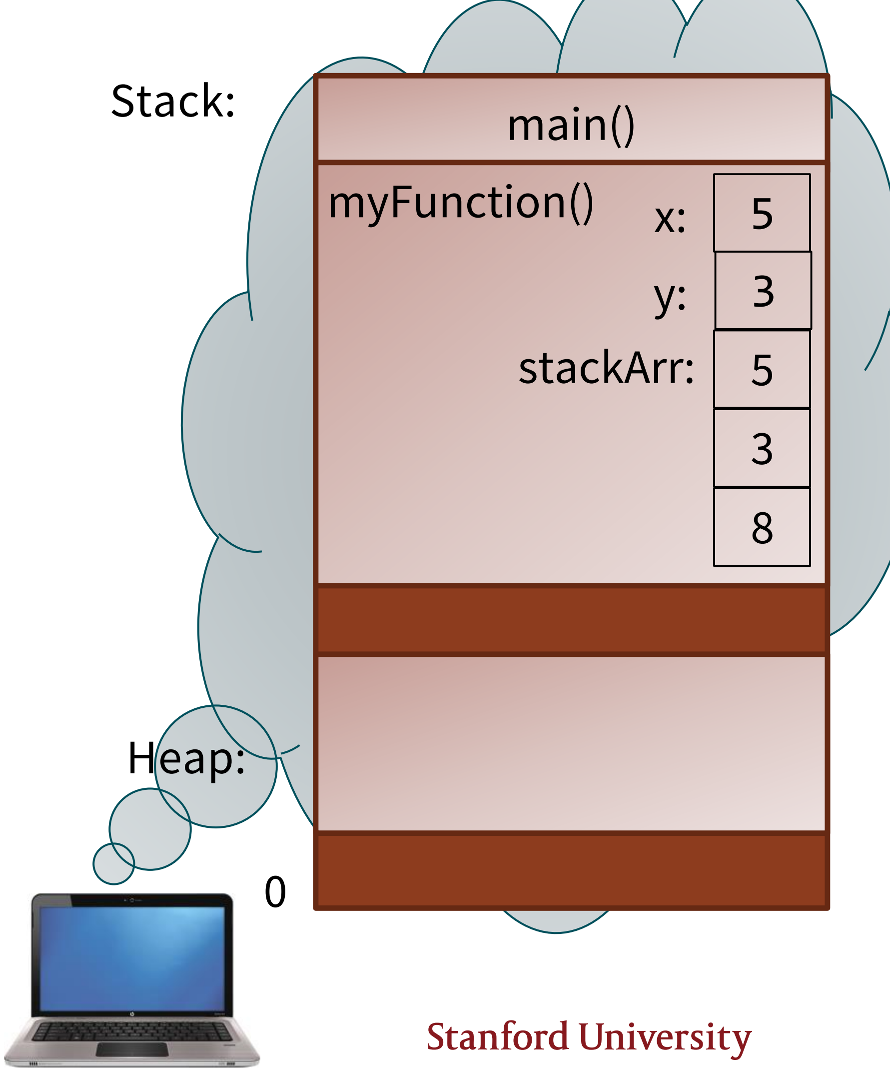
Example: `int* homeworkGrades = new int[7];`

Arrays in a memory diagram

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

    return y;
}
```

What happens when `myFunction()` returns?

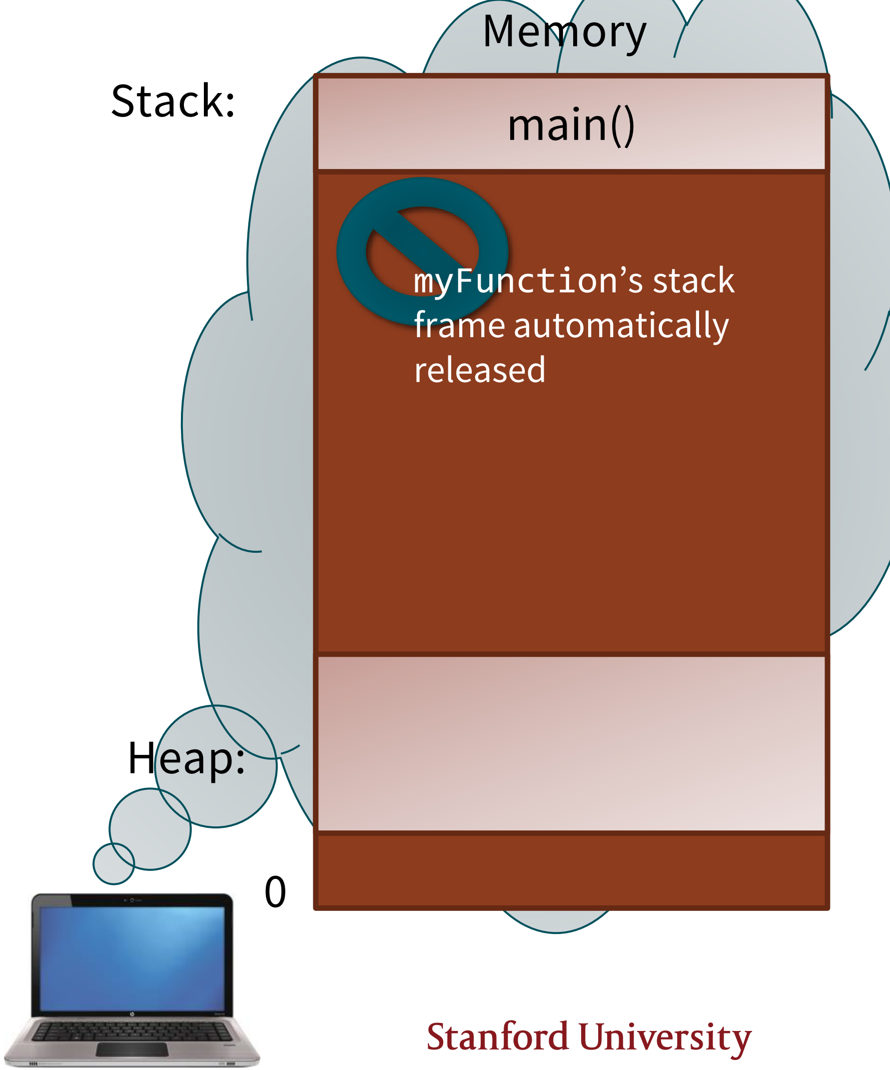


Arrays in a memory diagram

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

    return y; }
```

What happens when `myFunction()` returns?

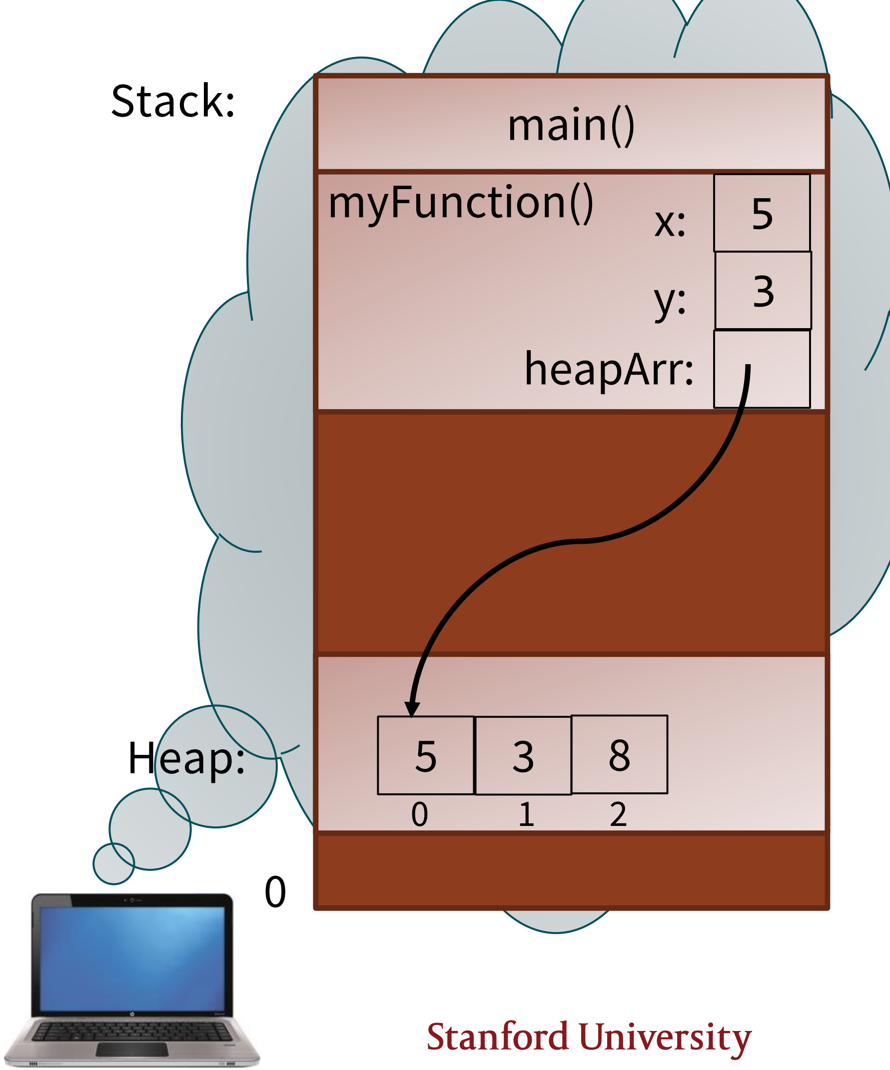


Arrays in a memory diagram

```
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;
}
```

What happens when `myFunction()` returns?

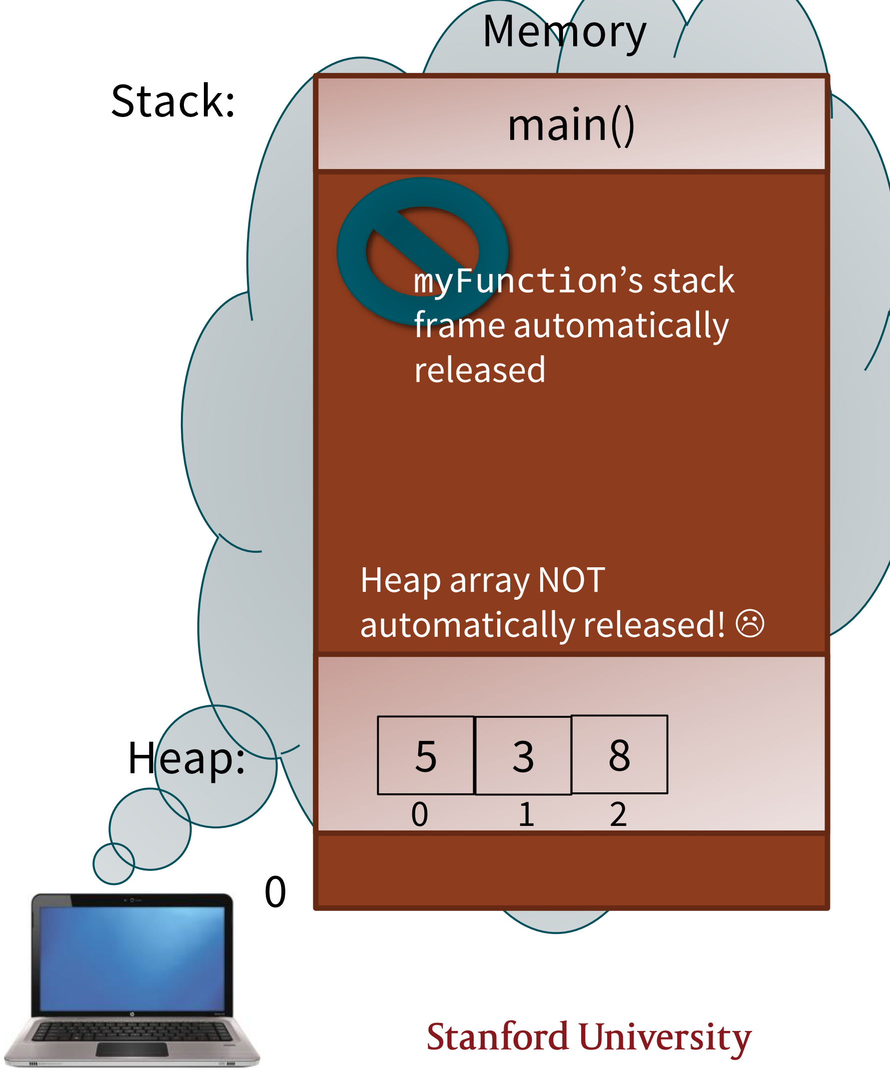


Arrays in a memory diagram

```
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;
}
```

What happens when `myFunction()` returns?

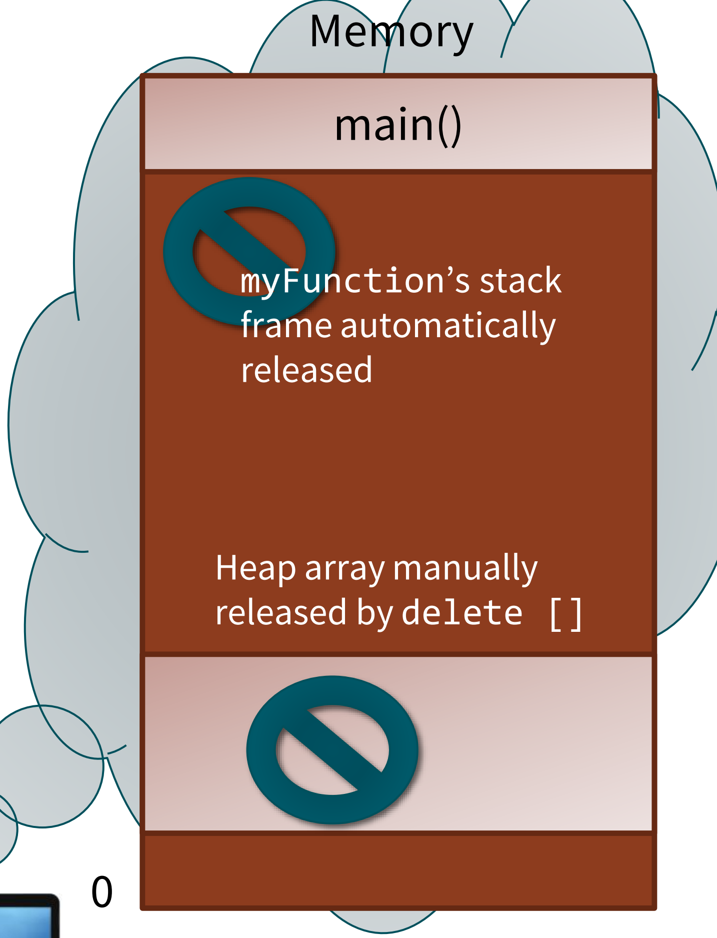


Always a pair: new and delete

```
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

```
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;
}
```

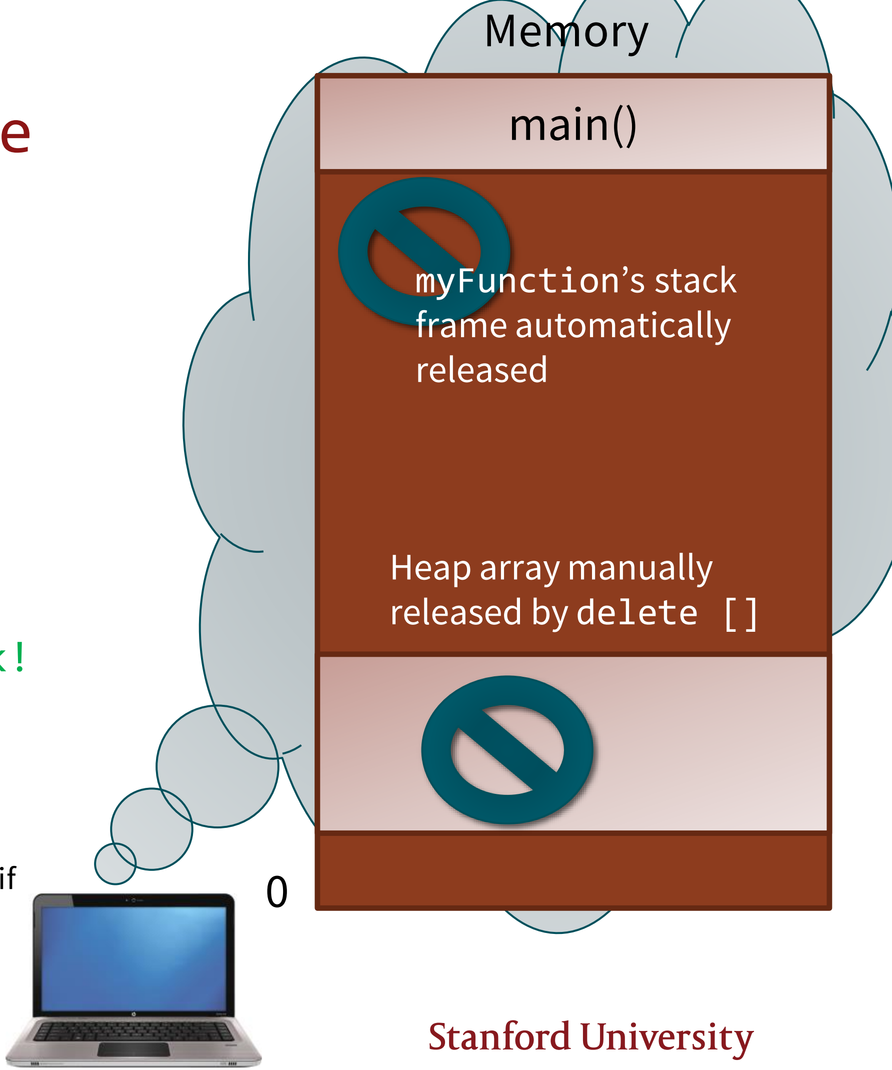


Always a pair: new and delete

```
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;
}
```

Q: "Why would you want to do that?"

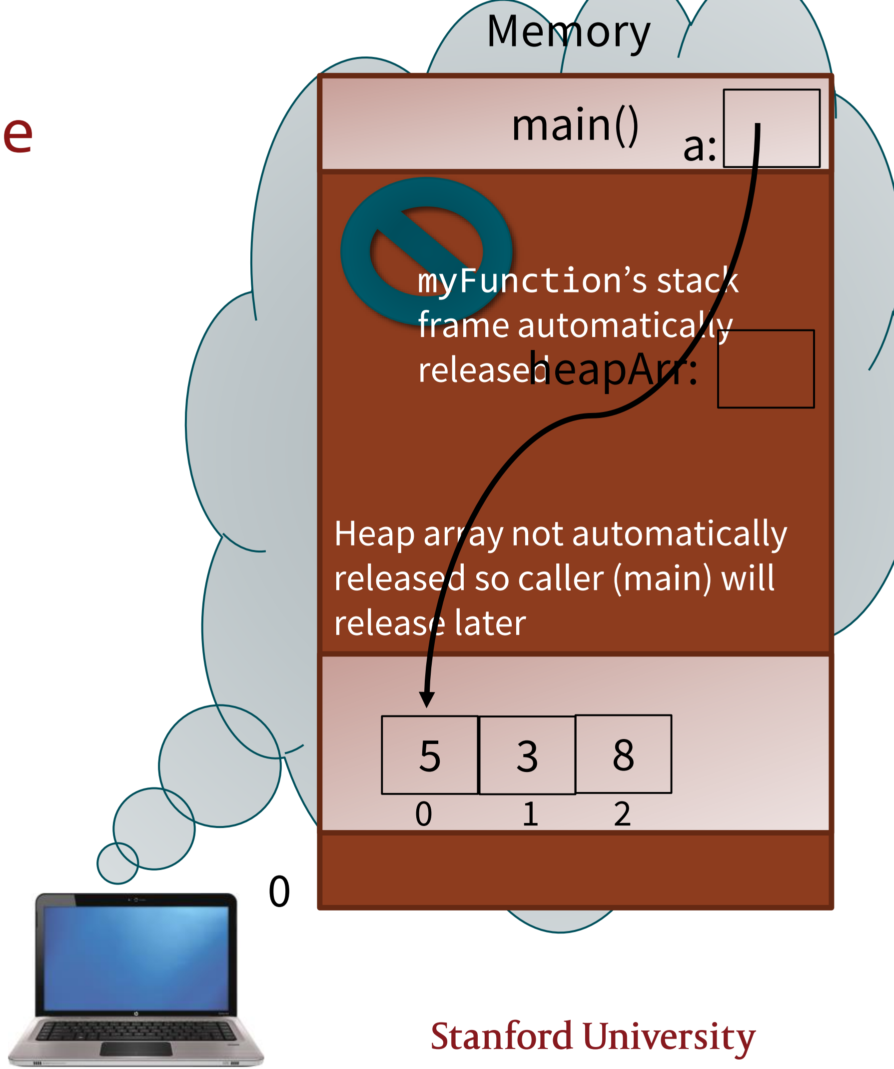
A: It's true that there's no point to using dynamic allocation if we are just deleting at the end of the function. Choose a static array instead to automatically release.
But what if we want to return the array?



Always a pair: new and delete

```
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 [] to be done later!
```



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* a1 = new int[3];
cout << a1[0];                // 2395876
cout << a1[1];                // -197630894

int* a2 = new int[3]();
cout << a2[0];                // 0
cout << a2[1];                // 0
```

Destructor (12.3)

```
// ClassName.h
~ClassName();
```

```
// ClassName.cpp
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

arraystack.h

```
#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
```

arraystack.cpp (part 1)

```
#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++;
}
```

arraystack.cpp (part 2)

```
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** *capacity* **20**

Overflow (extra) slides

FOR THE ADVANCED AND/OR CURIOUS STUDENT

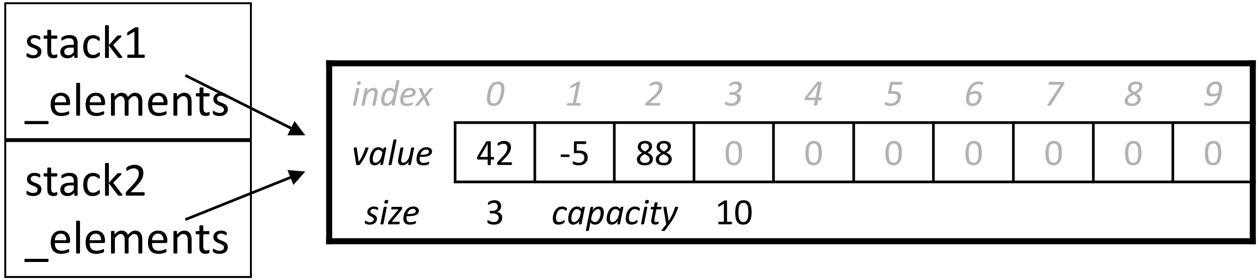
Shallow copy bug (12.7)

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.