Programming Abstractions

CS106B

Cynthia Bailey Lee

Julie Zelenski

Today’s Topics

- Drill down on memory and pointers
  - Uninitialized memory
  - Different pointer types
  - C++ structs and pointers
- IMPORTANT: the Midterm is Tuesday.
  - Check your room assignment on the course website.
  - Information about topics, rules, etc, on the course website.
  - If you have a special situation or accommodation and don’t have an email confirming your separate time/place, *we do not have you in our records*, so it is critical that you reach out to Jonathan *immediately*.
- Apply to be a section leader! Applications due Saturday Nov 2.

- **For important announcements, be sure to see the weekly announcements post on the Ed Q&A board! https://edstem.org**
- **Also on Ed: live lecture Q&A with Chris & Jonathan**

Recap from Last Time

STACK AND HEAP ARRAYS

Two kinds of arrays in 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];`

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

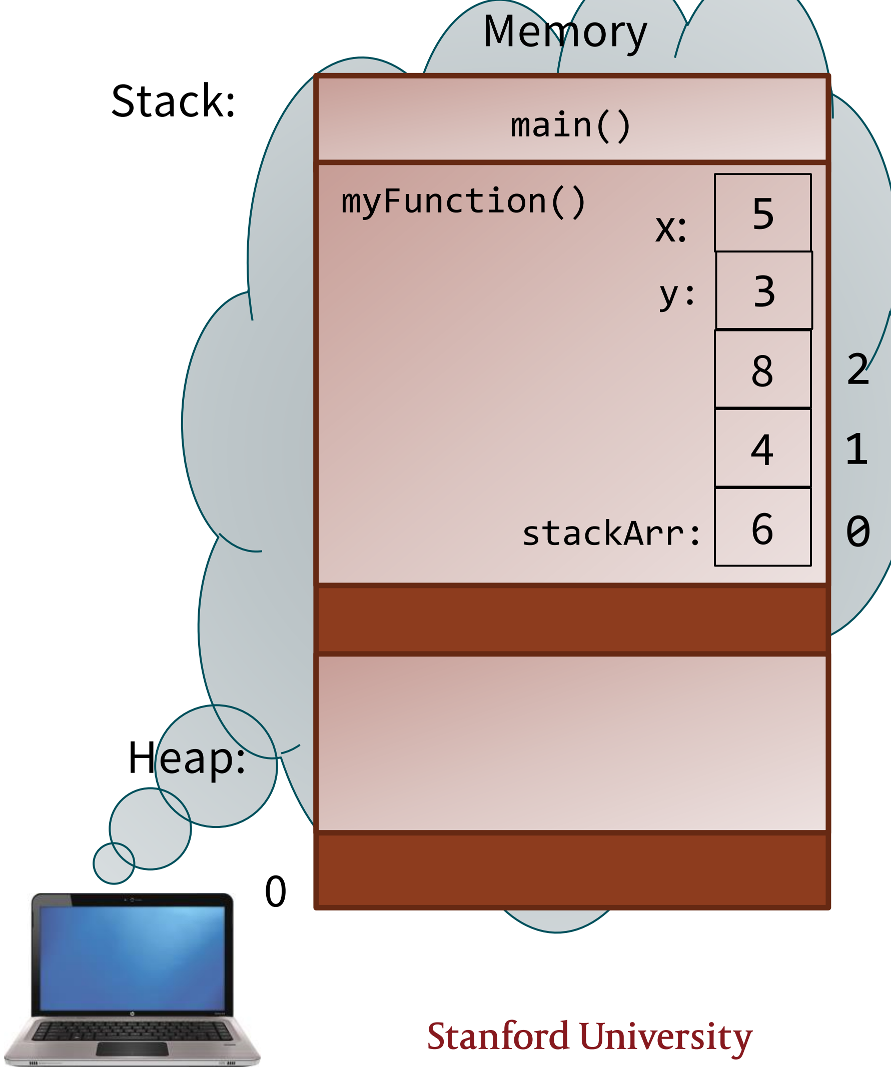
- **Dynamically allocated array (AKA heap allocated)**
- The variable that refers to the array is called a pointer, and it is on the stack
- But the actual array is stored in the heap!

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

Stack array memory diagram

```
int myFunction() {
   int x = 5;
   int y = 3;
   int stackArr[3];
   stackArr[0] = x + 1; // 6
   stackArr[1] = y + 1; // 4
   stackArr[2] = x + y; // 8

   return y;
}
```

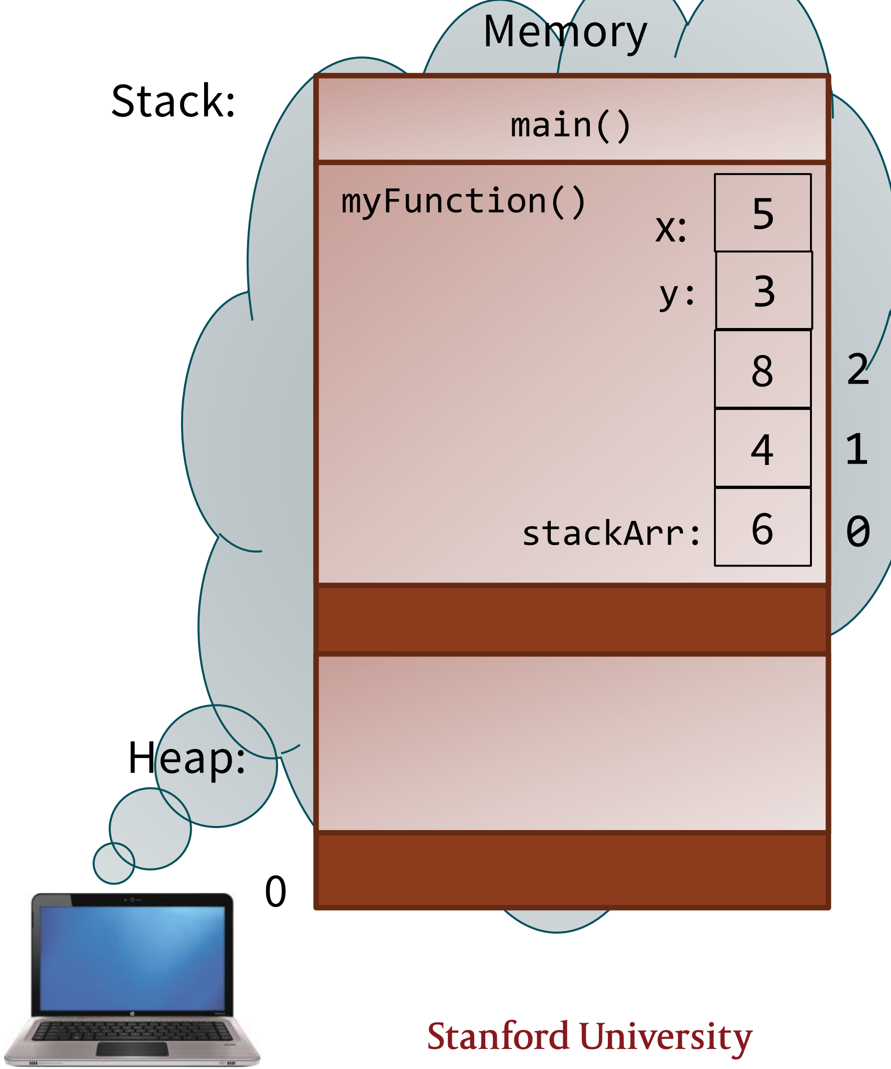


Stack array memory diagram

```
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;
}
```

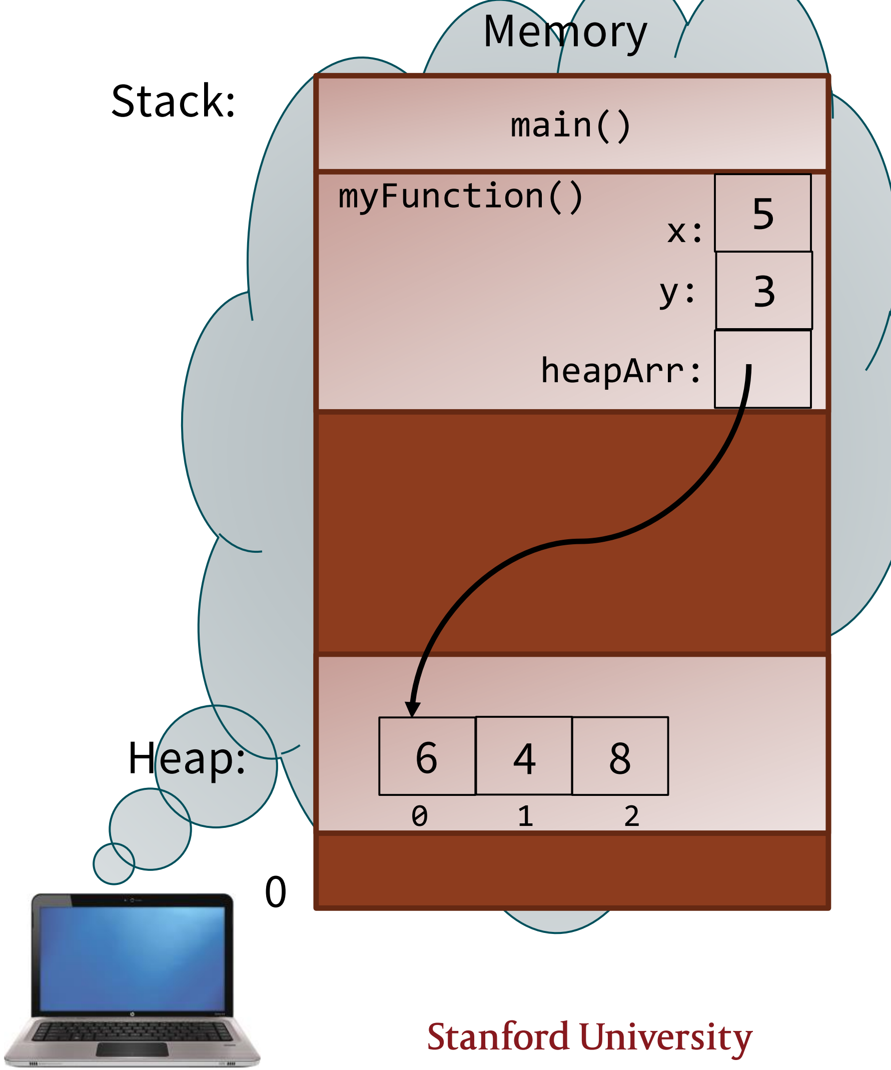
What happens when `myFunction()` returns?



Heap array memory diagram

```
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;
}
```

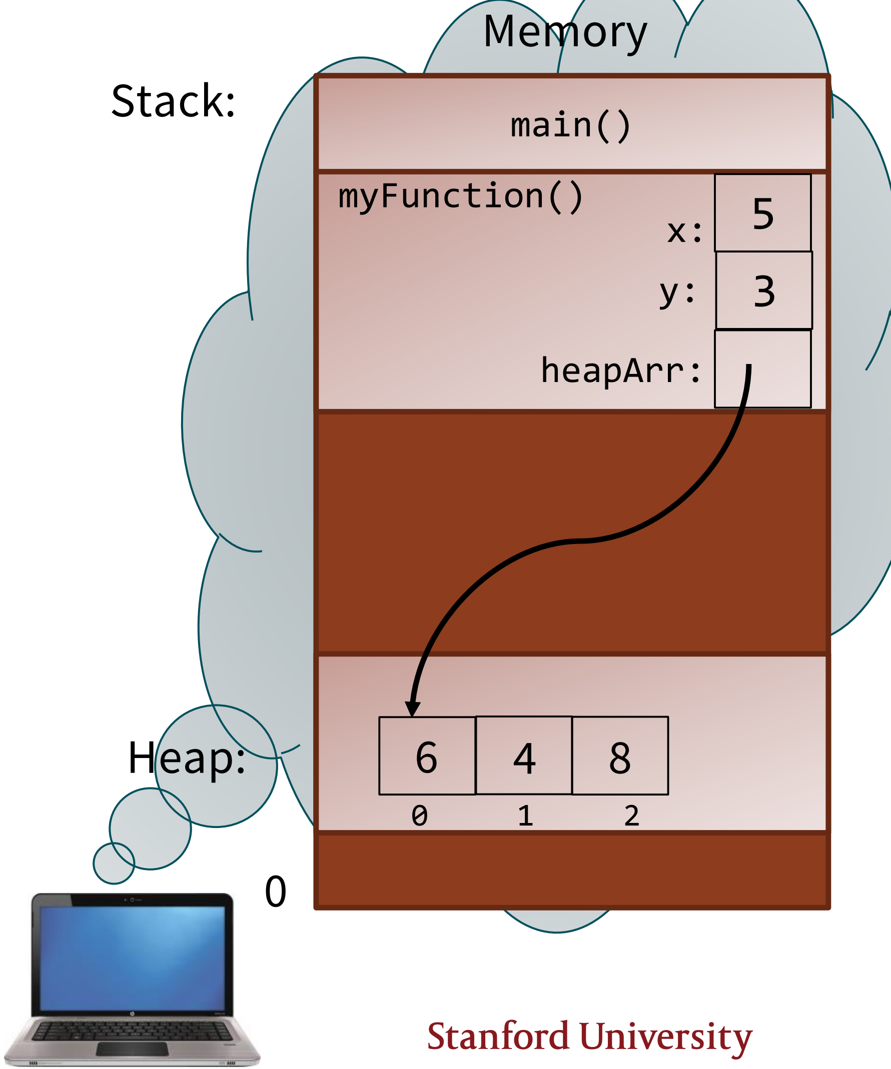


Heap array memory diagram

```
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?



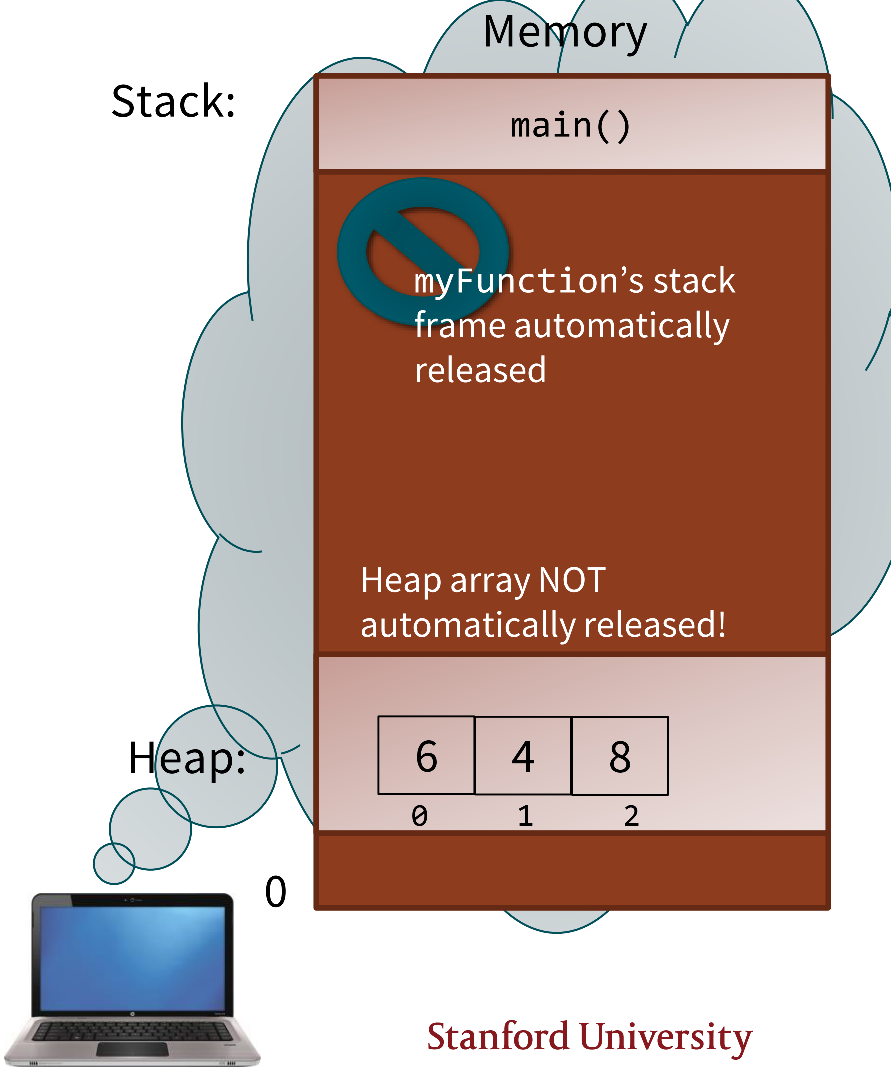
Stanford University

Heap array memory diagram

```
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?



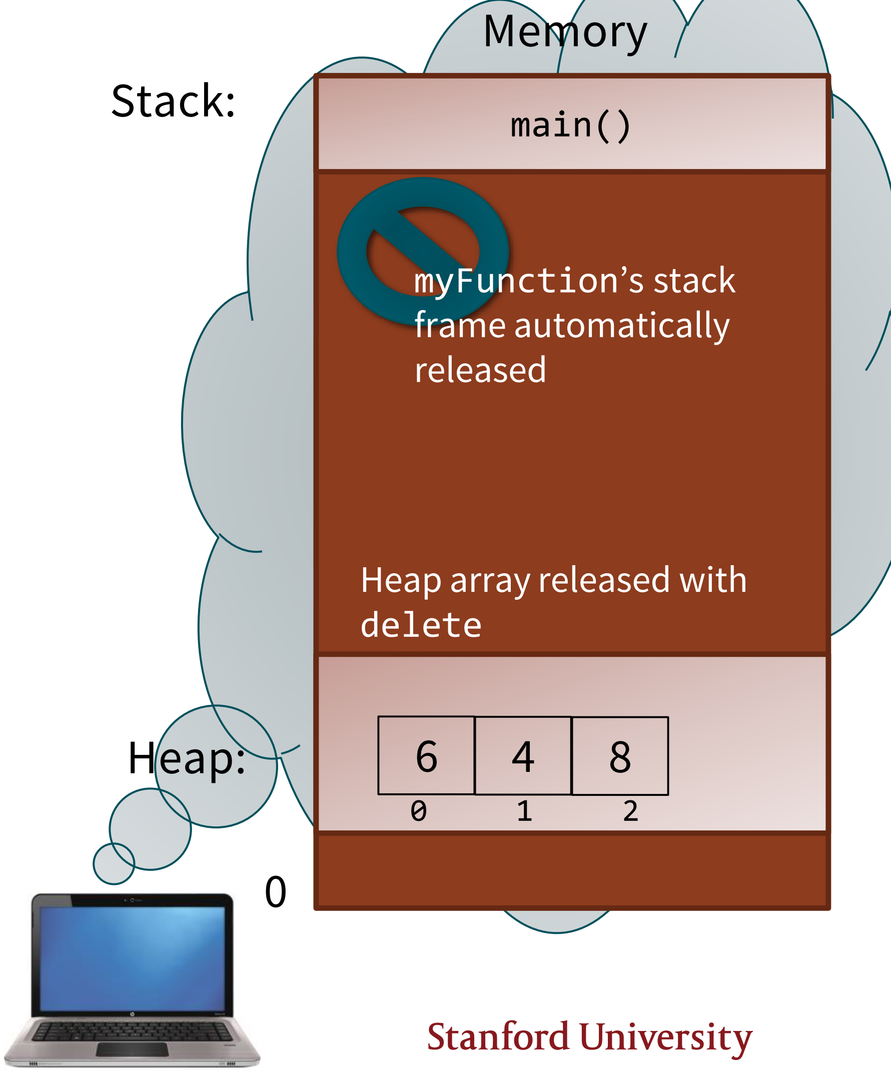
myFunction’s stack frame automatically released

Heap array NOT automatically released!

Heap array memory diagram

```
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?



myFunction’s stack frame automatically released

Heap array released with `delete`

Uninitialized Memory

TWO CODE DEMOS

How to fix the uninitialized memory danger

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

- In general, memory stores uninitialized (“random”/garbage) values
- If ( ) are written after [ ], all elements are zeroed out
  - Slower but good if needed

```
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
```

Pointers

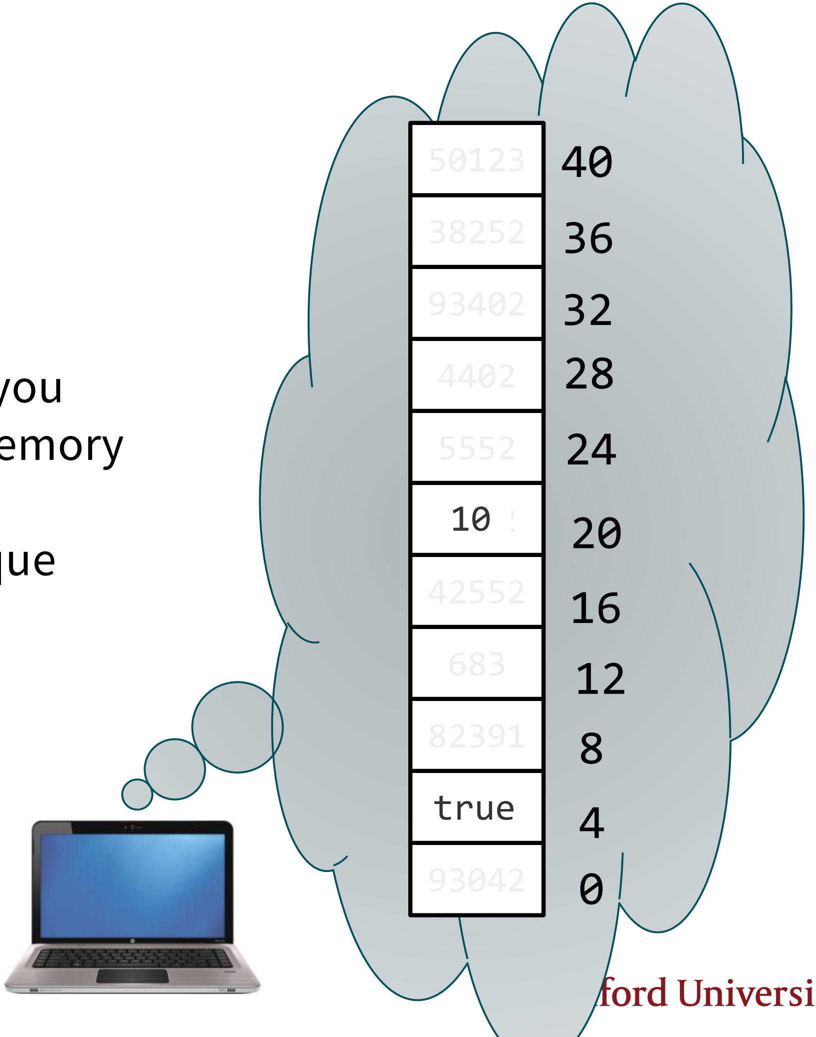
TAKING A DEEPER LOOK AT THE SYNTAX OF THAT ARRAY ON THE HEAP

Memory addresses

```
bool kitkat = true;
int candies = 10;
```

Whenever you declare a variable, you allocate a bucket (or more) of memory for the value of that variable

Each bucket of memory has a unique address



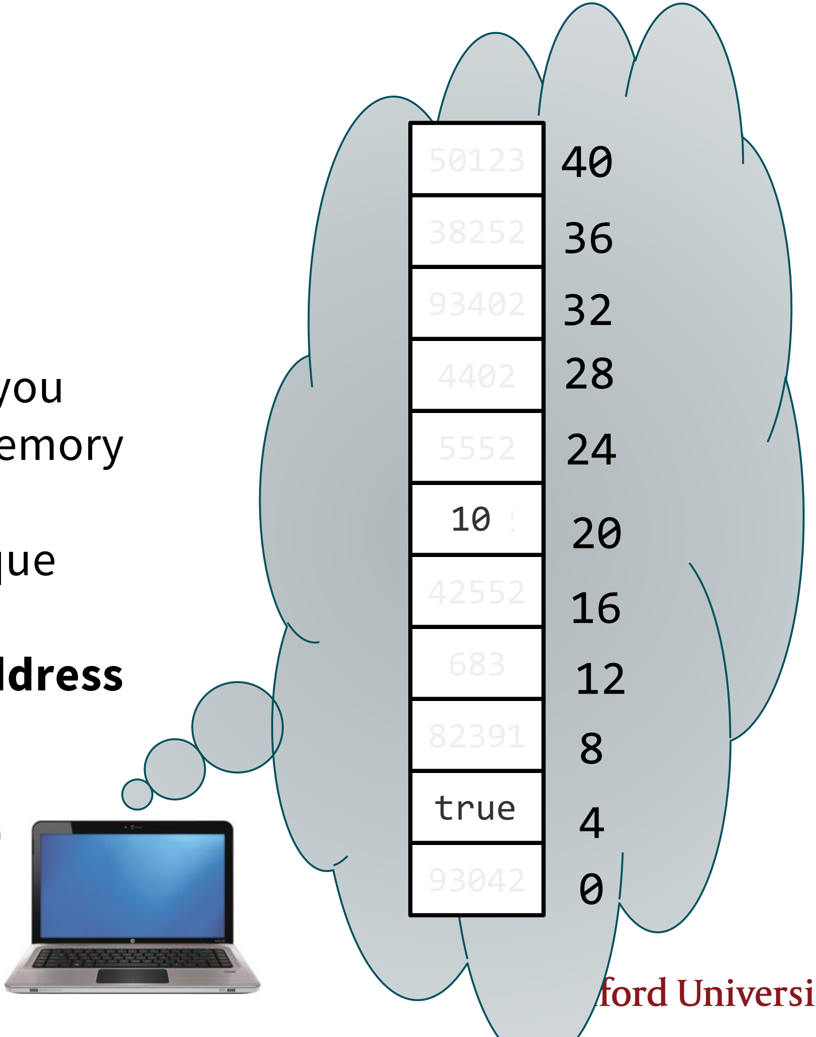
50123	40
38252	36
93402	32
4402	28
5552	24
10	20
42552	16
683	12
82391	8
true	4
93042	0

Memory addresses

```
bool kitkat = true;
int candies = 10;
```

- Whenever you declare a variable, you allocate a bucket (or more) of memory for the value of that variable
- Each bucket of memory has a unique address
- **You can ask for any variable's address using the & operator.**

```
cout << &candies << endl;    // 20
cout << &kitkat << endl;     // 4
```



50123	40
38252	36
93402	32
4402	28
5552	24
10	20
42552	16
683	12
82391	8
true	4
93042	0

Memory addresses

```
bool kitkat = true;
int candies = 10;
```

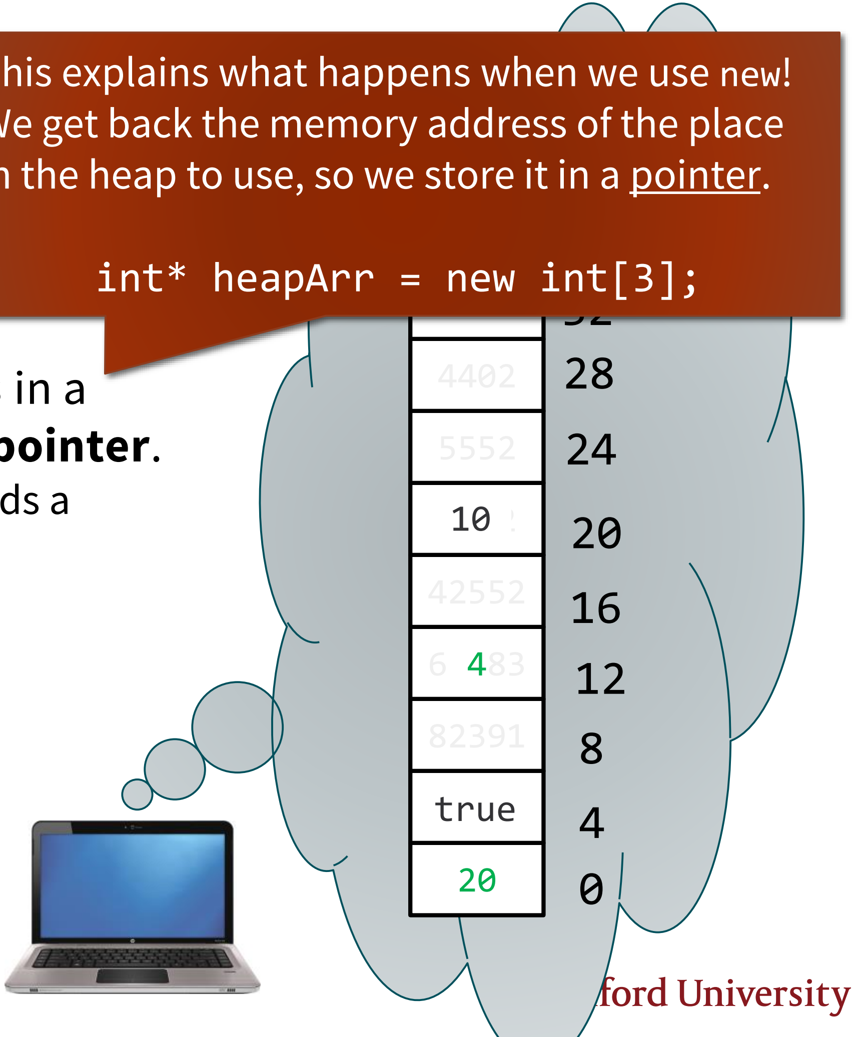
You can **store memory addresses** in a special type of variable called a **pointer**.

- i.e. A pointer is a variable that holds a memory address.

```
int* ptrC = &candies;     // 20
bool* ptrB = &kitkat;     // 4
```

This explains what happens when we use new! We get back the memory address of the place in the heap to use, so we store it in a pointer.

```
int* heapArr = new int[3];
```



4402	28
5552	24
10	20
42552	16
6 483	12
82391	8
true	4
20	0

Memory addresses

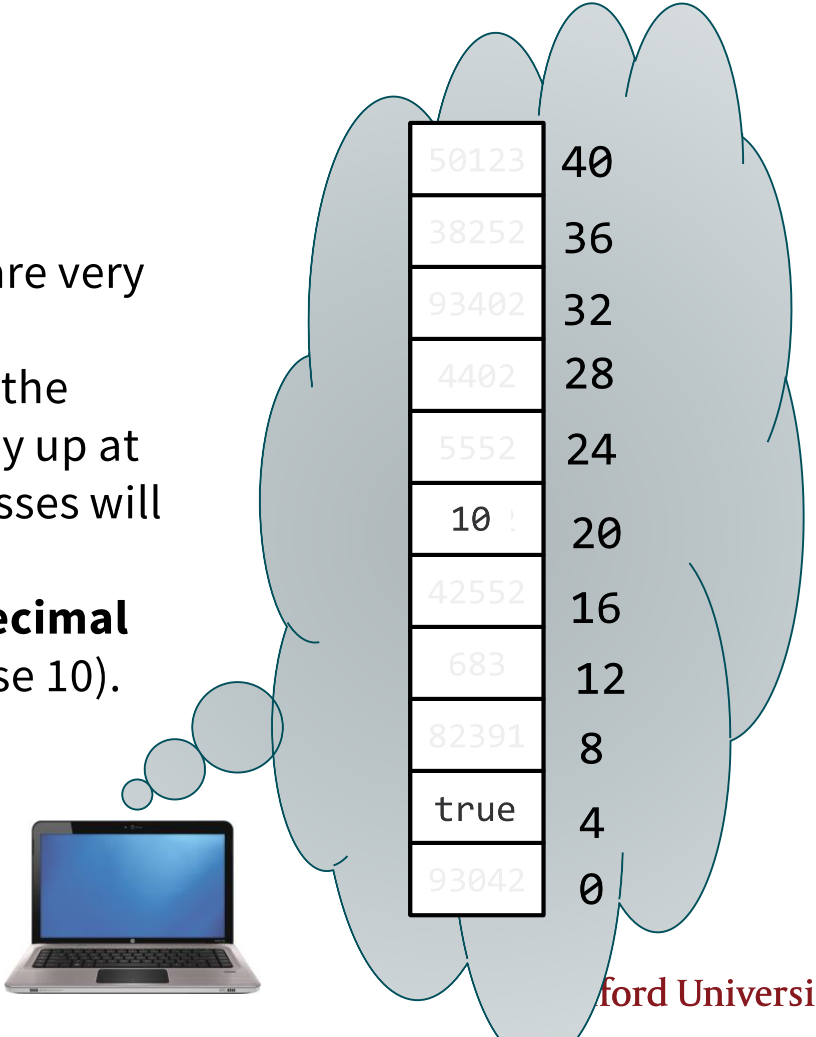
In our example here, the memory addresses of our local variables are very small numbers.

Remember that in a real situation, the stack part of memory is waaaaaay up at the end of memory, so the addresses will be quite large!

We typically **write them in hexadecimal (base 16)** instead of deciaml (base 10).

Example:

```
0x7ffee40f1494
```



50123	40
38252	36
93402	32
4402	28
5552	24
10	20
42552	16
683	12
82391	8
true	4
93042	0

Memory addresses

- “Pointer” isn’t one type in C++ but many—it depends on what it points to.
- You can declare a pointer using * and the type pointed-to:
  - `int*`
  - `bool*`
  - `string*`
  - `double*`
  - `Queue<GridLocation>*`
  - `int**` ← Yes this is possible (!!), you’ll see this in CS107.

Memory addresses

- “Pointer” isn’t one type in C++ but many—it depends on what it points to.
- You can declare a pointer using * and the type pointed-to:
  - `int*`
  - `bool*`
  - `string*`
  - `double*`
  - `Queue<GridLocation>*`
  - `int**` ← Yes this is possible (!!), you’ll see this in CS107.

Does this imply that we can use `new` with class types like Queue, to put the entire Queue object in heap memory? Yep, we sure can!

Uninitialized Pointers and `nullptr`

MORE C++ DETAILS

What is in an uninitialized pointer variable?

- We saw that, in general, memory stores uninitialized (“random”/garbage) values
- What is an uninitialized pointer?
  - Just some number, the “arrow” of the pointer points to some “random” location
  - This is REALLY BAD for bugs in code ☹ ☹ ☹
  - You could change the value of any other variable in your code on accident
  - Extremely hard to debug ☹ ☹ ☹

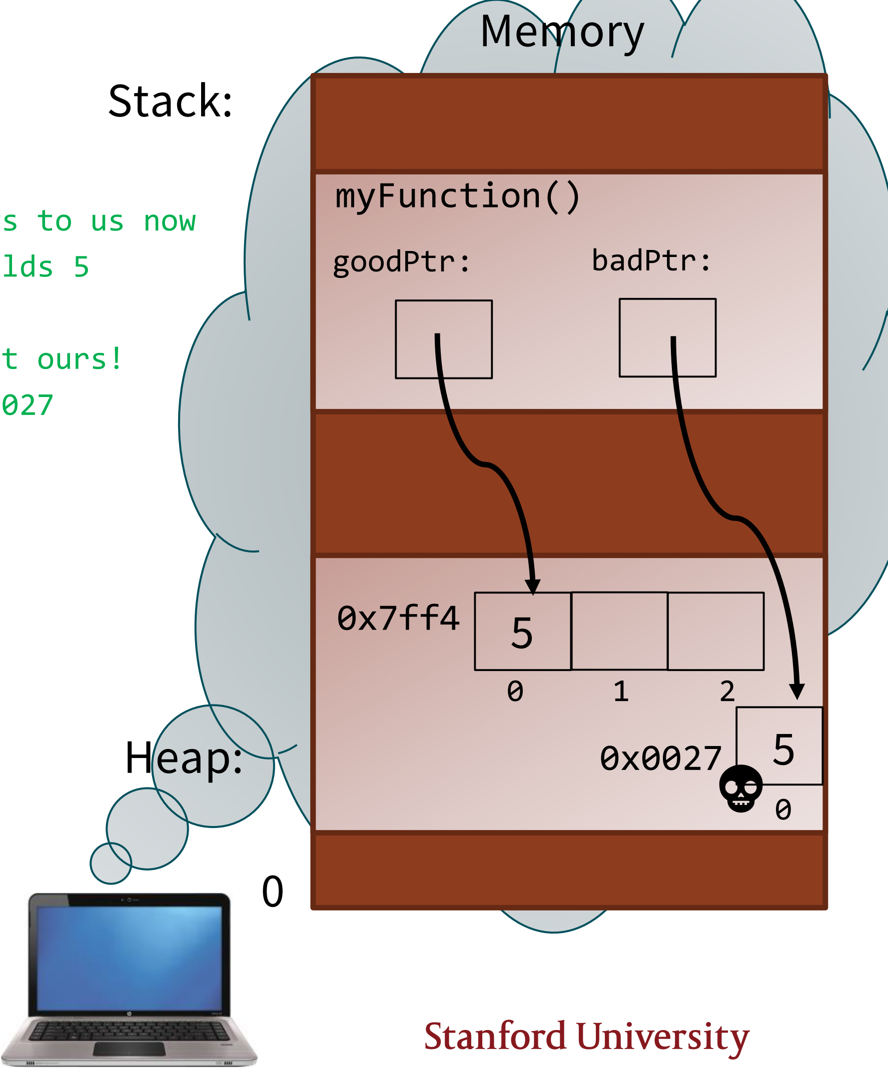
```
int* goodPtr = new int[3];   // address 0x7ff4 belongs to us now
goodPtr[0] = 5;              // address 0x7ff4 now holds 5

int* badPtr;                // uninitialized – address 0x0027 not ours!
badPtr[0] = 5;              // RIP whoever was using address 0x0027
```

Uninitialized pointers

```
int* goodPtr = new int[3]; // address 0x7ff4 belongs to us now
goodPtr[0] = 5;            // address 0x7ff4 now holds 5

int* badPtr;   // uninitialized – address 0x0027 not ours!
badPtr[0] = 5; // RIP whoever was using address 0x0027
```



Initializing pointer variables: `nullptr`

- We’ve seen that uninitialized pointers are REALLY BAD ☹ ☹ ☹
- nullptr is a special value that we can use to initialize pointers
  - Guaranteed to never be a usable memory address that belongs to anyone
  - (it’s actually just the number zero, but don’t use `0` in your code)

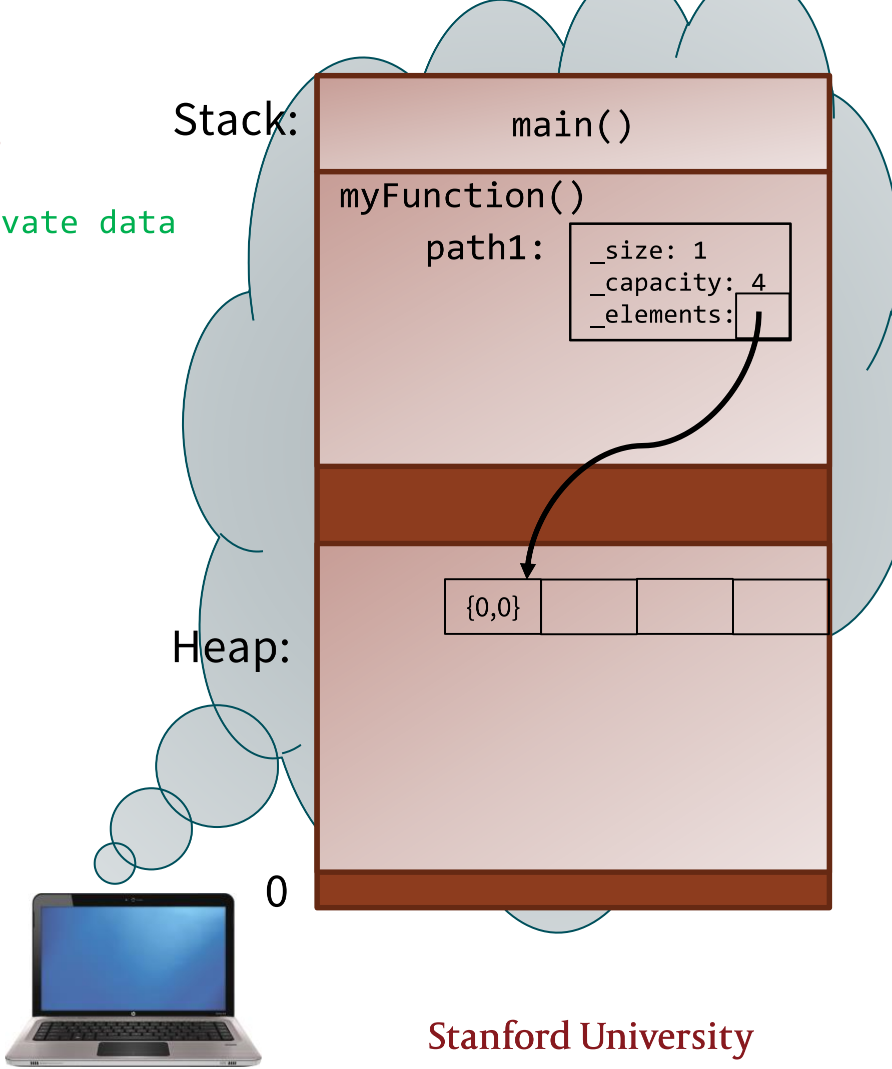
```
int* ptr = nullptr; // good value to use for now
ptr[0] = 5;         // this will give an error—THAT'S USEFUL
if (ptr != nullptr) { // nullptr is good to test for
    ptr[0] = 5;
} else {
    // don’t use ptr!
}
```

More on Dynamically-Allocated Memory

NEW AND DELETE FOR THINGS OTHER THAN ARRAYS

Dynamically-allocated objects

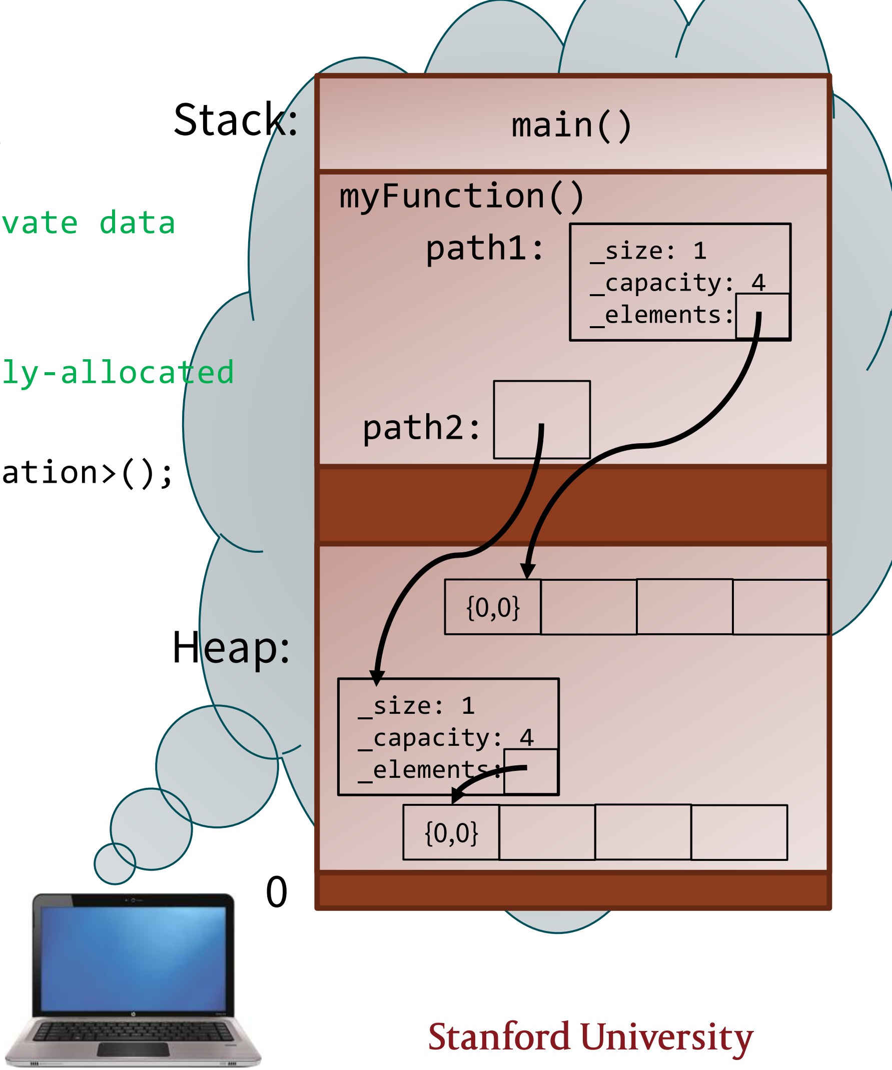
```
// Stack object with dynamically-allocated private data
Queue<GridLocation> path1;
path1.enqueue(loc);
```



Stanford University

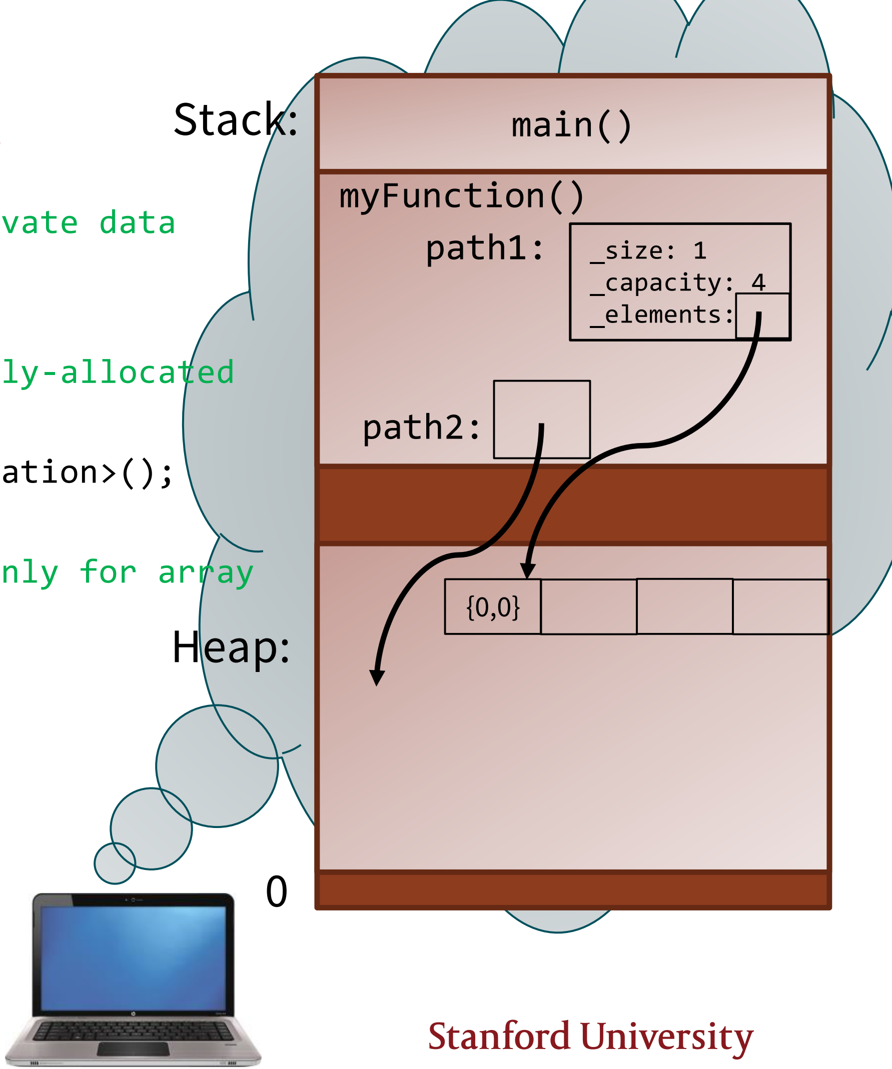
Dynamically-allocated objects

```
// Stack object with dynamically-allocated private data
Queue<GridLocation> path1;
path1.enqueue(loc);
// Dynamically-allocated object with dynamically-allocated
// private data
Queue<GridLocation>* path2 = new Queue<GridLocation>();
path2->enqueue(loc);  // note ->
```



Dynamically-allocated objects

```
// Stack object with dynamically-allocated private data
Queue<GridLocation> path1;
path1.enqueue(loc);
// Dynamically-allocated object with dynamically-allocated
// private data
Queue<GridLocation>* path2 = new Queue<GridLocation>();
path2->enqueue(loc);  // note ->
delete path2;         // don't use [] that’s only for array
```

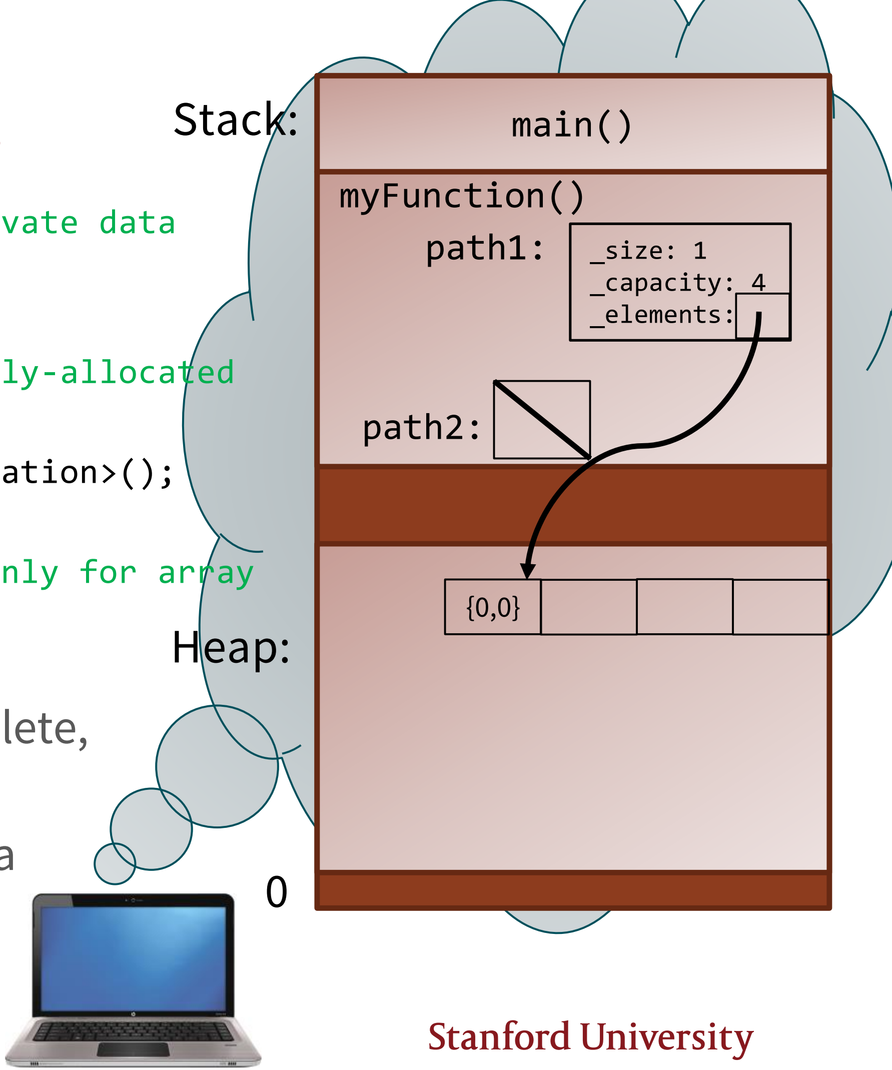


Stanford University

Dynamically-allocated objects

```
// Stack object with dynamically-allocated private data
Queue<GridLocation> path1;
path1.enqueue(loc);
// Dynamically-allocated object with dynamically-allocated
// private data
Queue<GridLocation>* path2 = new Queue<GridLocation>();
path2->enqueue(loc);  // note ->
delete path2;         // don't use [] that’s only for array
path2 = nullptr;
```

- Tip: set pointers to `nullptr` right after a delete, that way you don’t accidentally use them
- In memory diagrams, we draw `nullptr` as a slash through the box



Stack:
main()
myFunction()
path1:
_size: 1
_capacity: 4
_elements:
path2:
Heap:
{0,0}
0