Memory and Pointers

Amrita Kaur

July 27, 2023

Contributions made from previous CS106B Instructors

Announcements

- Midterm regrade requests will be processed by Saturday

What’s a Priority Queue?

- A queue that sorts its elements based on their priority
- Like regular queues, you can only access the element at the front
  - No indices
- Good way to model things like:
  - ER waiting rooms
  - Organ matches
  - Vaccine availability
  - Airplane boarding groups
  - Social media feed
  - College admissions
  - Welfare allocation

Priority Queue Operations

- `peek()` - returns the element with the highest priority in the queue without removing it
- `enqueue(elem, priority)` - inserts elem with given priority
- `dequeue()` - removes and returns the element with the highest priority from the queue

Priority Queue Operations

- `peek()` - returns the element with the highest priority in the queue without removing it
- `enqueue(elem, priority)` - inserts elem with given priority
- `dequeue()` - removes and returns the element with the highest priority from the queue
- `size()` - returns the number of elements in the queue
- `isEmpty()` - returns `true` if there are no elements in the queue, `false` otherwise
- `clear()` - empties the queue

Priority Queue Implementations

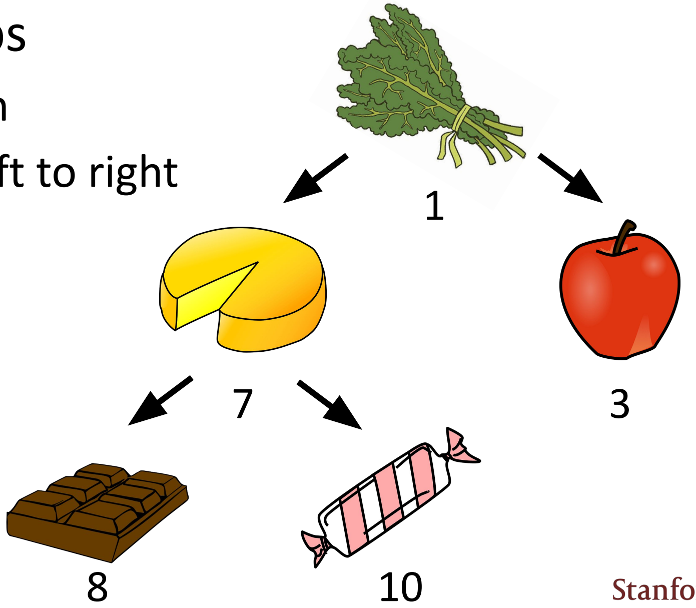
- Using a sorted array
  - `peek() - O(1)`
  - `enqueue(elem, priority) - O(n)`
  - `dequeue() - O(1)`

Priority Queue Implementations

- Using a sorted array
  - `peek() - O(1)`
  - `enqueue(elem, priority) - O(n)`
  - `dequeue() - O(1)`
- Using a binary heap
  - `peek() - O(1)`
  - `enqueue(elem, priority) - O(log n)`
  - `dequeue() - O(log n)`

What’s a Binary Heap?

- A heap is a tree-based data structure that satisfies the “heap property”: parents have a higher priority than their children
- For now, we’ll focus on *binary* heaps
  - Each parent has exactly two children
  - Exception: last level, which we fill left to right



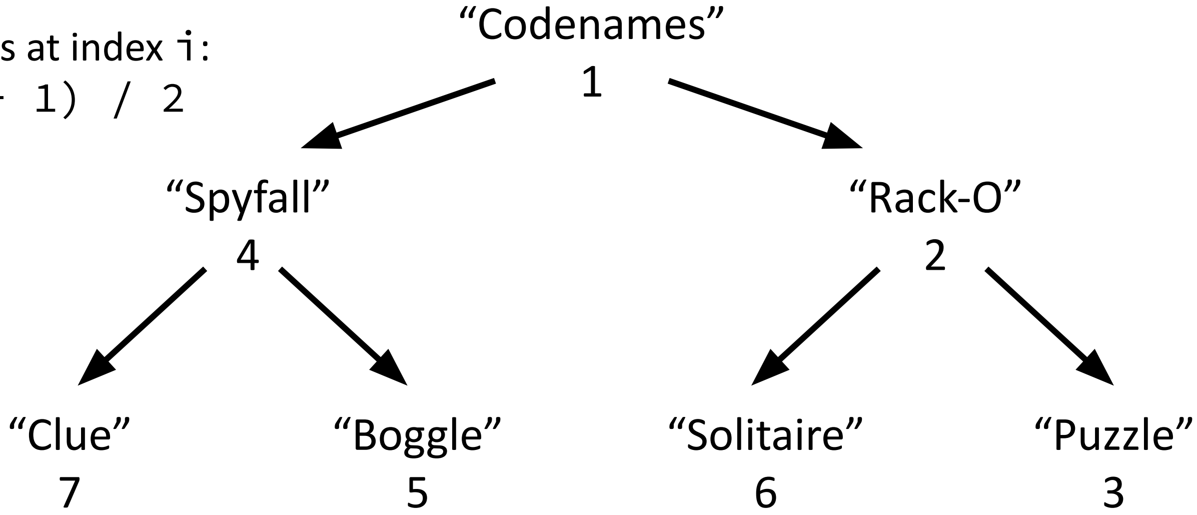
Formula: if parent is at index `i`:

Left child is at `2 * i + 1`

Right child is at `2 * i + 2`

Formula: if child is at index `i`:

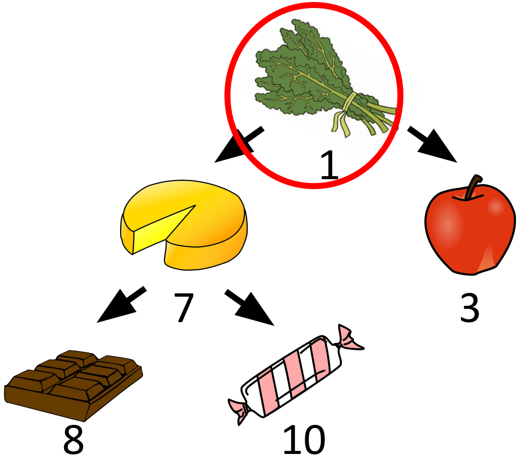
Parent is at `(i - 1) / 2`


“Codenames”
1
“Spyfall”
4
“Rack-O”
2
“Clue”
7
“Boggle”
5
“Solitaire”
6
“Puzzle”
3

{“Codenames”, 1}	{“Spyfall”, 4}	{“Rack-O”, 2}	{“Clue”, 7}	{“Boggle”, 5}	{“Solitaire”, 6}	{“Puzzle”, 3}
0	1	2	3	4	5	6

PQ Heap - `peek()`

- Return the highest priority element, without removing it
- This is `O(1)`, we just check what's at the first index of our array



{"kale", 1}	{"cheese", 7}	{"apple", 3}	{"cocoa", 8}
0	1	2	3

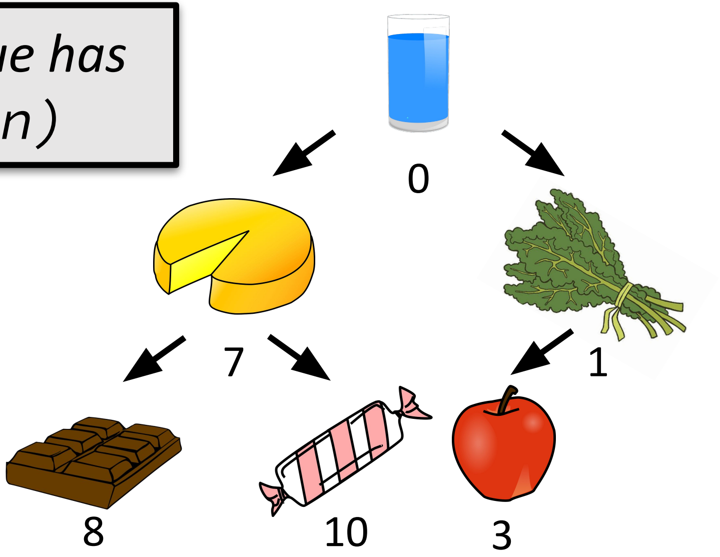
PQ Heap - enqueue()

To enqueue a new element into our PQ Heap, we “bubble up”:

1. Insert element at the end of array
2. If this element has a greater priority than its parent, swap parent and child element
3. Repeat 2 until heap property is satisfied or we reach the root!

PQ Heap - enqueue()

PQ Heap enqueue has runtime O(log n)



{"water", 0}	{"cheese", 7}	{"kale", 1}	{"cocoa", 8}	{"candy", 10}	{"apple", 3}
0	1	2	3	4	5

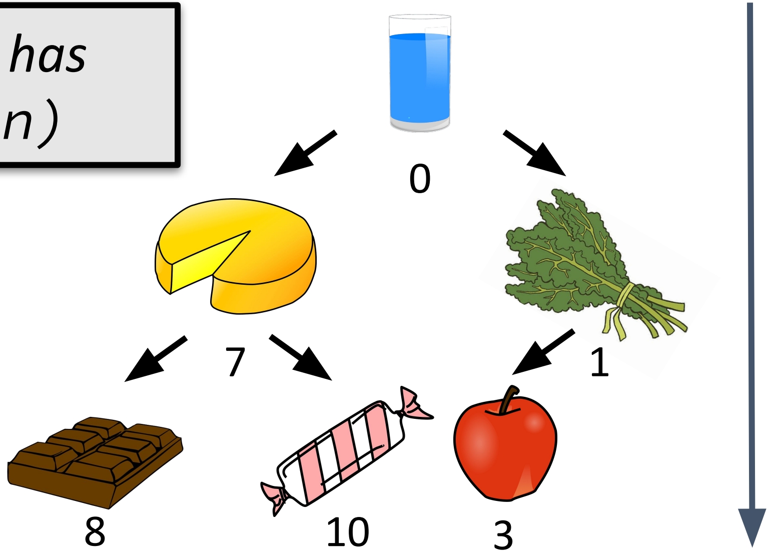
PQ Heap - dequeue()

To dequeue the highest priority element in our PQ Heap:

1. Remove element from the beginning (index 0) of our array
2. Move last element in array to index 0
3. Swap with higher priority child until heap property is satisfied

PQ Heap - dequeue()

PQ Heap dequeue has runtime O(log n)



Worst case, we bubble down from the top to the bottom of the tree

{“water”, 0}	{“cheese”, 7}	{“kale”, 1}	{“cocoa”, 8}	{“candy”, 10}	{“apple”, 3}
0	1	2	3	4	5

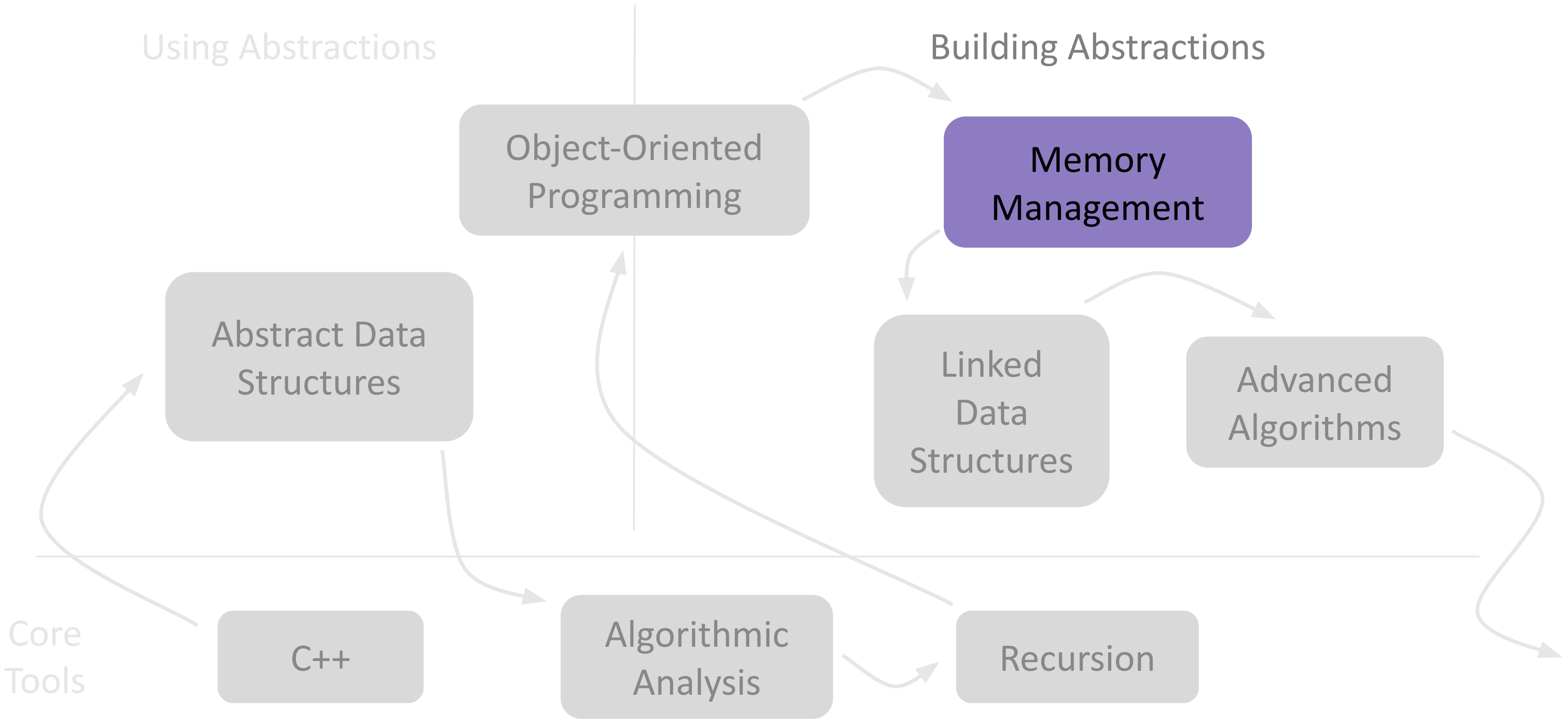
PQ Heap Runtimes

- `peek() - O(1)`
- `enqueue(elem, priority) - O(log n)`
- `dequeue() - O(log n)`

Notice how implementing the same data structure with a heap versus sorted array leads to different runtimes.

Stay tuned for Assignment 4!

Roadmap


Using Abstractions
Building Abstractions
Object-Oriented Programming
Memory Management
Abstract Data Structures
Linked Data Structures
Advanced Algorithms
Core Tools
C++
Algorithmic Analysis
Recursion

Memory Organization

What is computer memory?

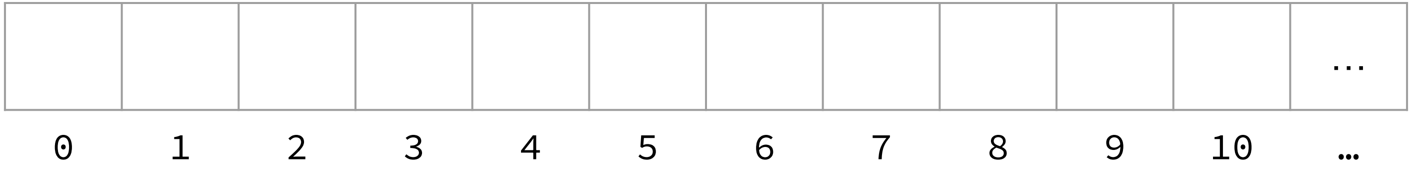
- The programs we write all make use of a specific component of the computer's hardware called Random Access Memory (RAM)
  - This is what we are referring to when we talk about "computer memory"
  - C++ gives us a variety of fundamental ways to access computer hardware from our code
  - This is where both the stack and heap are!

Why is computer memory important?

- We've already seen the power and importance of being able to dynamically allocate arrays and use these as data storage fundamentals for ADT classes
- Being able to directly work with computer memory opens up the doors to more interesting data storage and organization techniques (beyond arrays)
- After today's lecture, we'll spend the next two weeks talking about linked data structures (which are a powerful, alternative way to impose structure and meaning on data that is scattered over different places in computer memory)
  - In order to understand linked data structures, we first need to develop our toolbox of working directly with computer memory in C++!

How is computer memory organized?

- Memory in your computer is just a giant array!
  - Can think of it as a long row of boxes, with each box having a value in it and an associated index



											...
0	1	2	3	4	5	6	7	8	9	10	…

How is computer memory organized?

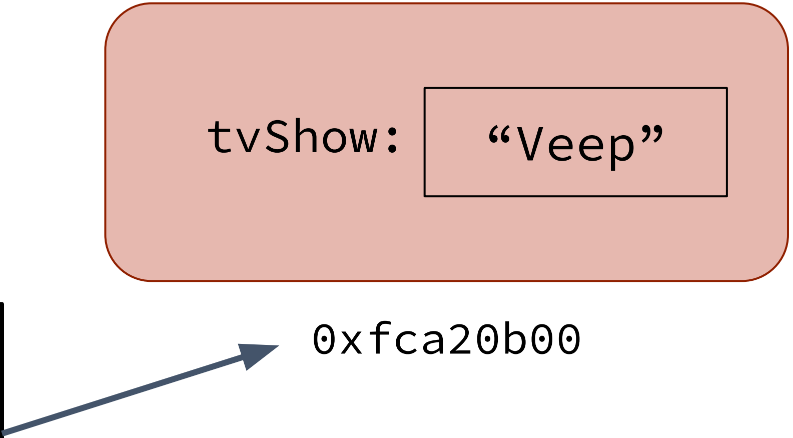
- Memory in your computer is just a giant array!
  - Can think of it as a long row of boxes, with each box having a value in it and an associated index

											...
0	1	2	3	4	5	6	7	8	9	10	…

- How can we communicate with the computer to find exactly which box we want to access/store information in?
  - We'll give each box an associated numerical location, called a **memory address**.

Memory Addresses

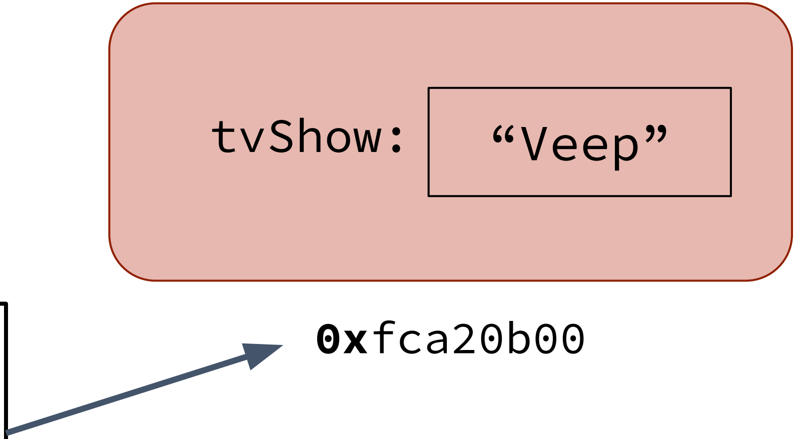
```
string tvShow = “Veep”;
```



This is the **memory address** of `tvShow`. This special numerical value acts as the unique identifier for this variable across the entire pool of the computer's memory.

Memory Addresses

```
string tvShow = “Veep”;
```



This is the **memory address** of `tvShow`. This special numerical value acts as the unique identifier for this variable across the entire pool of the computer's memory.

The Hexadecimal Number System

- We typically represent numbers using the decimal (base-10) number system
  - Each place value represents a factor of ten (ones, tens, hundreds, etc.)
  - 10 possible digits for each place value
- In computer systems,it is often more convenient to express numbers using the hexadecimal (base-16) number system.
  - Each place value represents a factor of 16 (16^0, 16^1 , 16^2, etc.)
  - 16 possible "digits" for each place value.
    - 10 numerical digits (0-9) and the letters ‘a’ to ‘f’
    - `0 1 2 3 4 5 6 7 8 9 a(10) b(11) c(12) d(13) e(14) f(15)`
- The prefix `0x` is used to communicate that a number is being expressed in hexadecimal

Memory Organization Recap

- Every location in memory, and therefore every variable, has an address.
- Every address corresponds to a unique location in memory.
- The computer generates/knows the address of every variable in your program.
- Given a memory address, the computer can find out what value is stored at that location.

Pointer

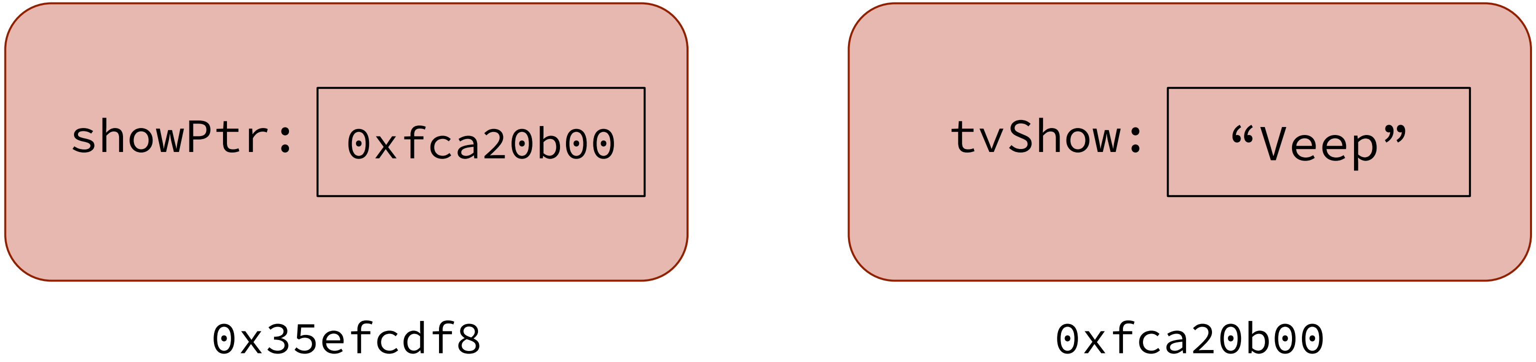
Pointer

- Data type that allows us to work directly with computer memory addresses
- Just like all other data types, pointers take up space in memory and store specific values
- Always stores a **memory address**, telling us where in the computer to look for a certain value
- They quite literally "point" to another location on your computer

What is a pointer?

A memory address!!

Introduction to Pointers



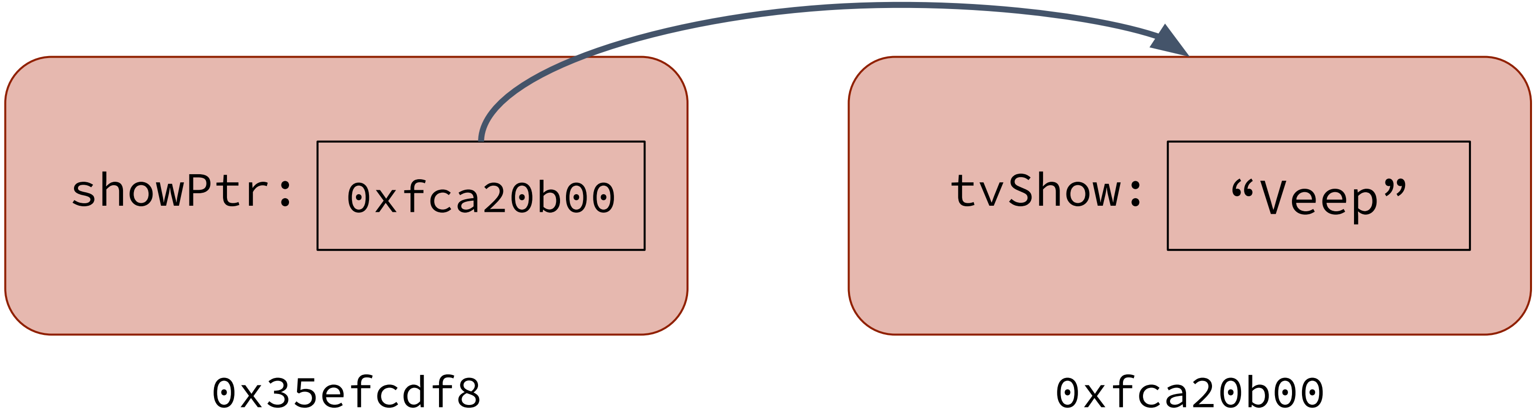
showPtr: 0xfca20b00

0x35efcdf8

tvShow: “Veep”

0xfca20b00

Introduction to Pointers



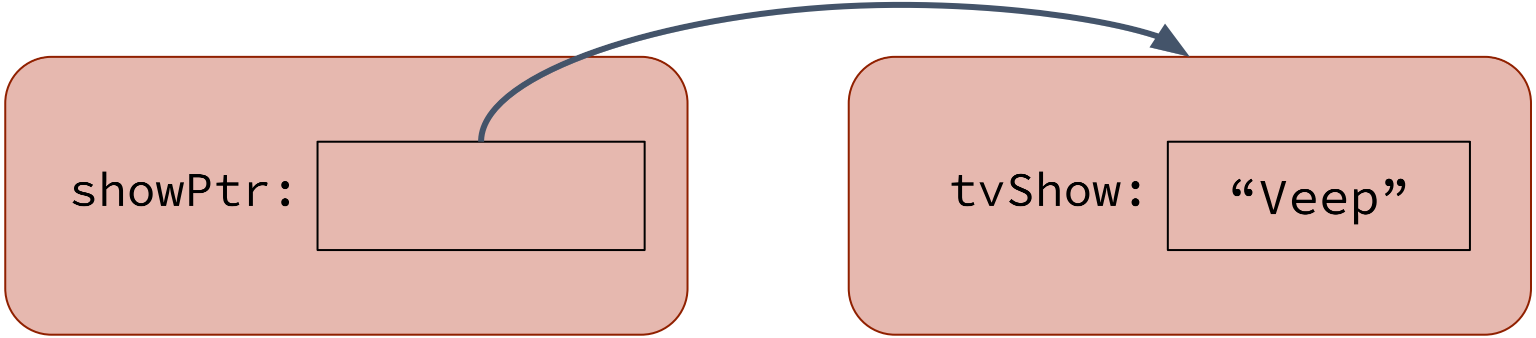
showPtr: 0xfca20b00

tvShow: “Veep”

0x35efcdf8

0xfca20b00

Introduction to Pointers



What is a pointer?

A memory address!!

Pointer Syntax

- Pointer syntax can get really tricky!
- Our goal in this class is to give you a brief, holistic overview. To truly become a master of pointers, take CS107 :)
- We'll talk about 4 main components of pointer syntax

Pointer Syntax, Part 1

- To declare a pointer of a particular type, use the `*`(asterisk) symbol:

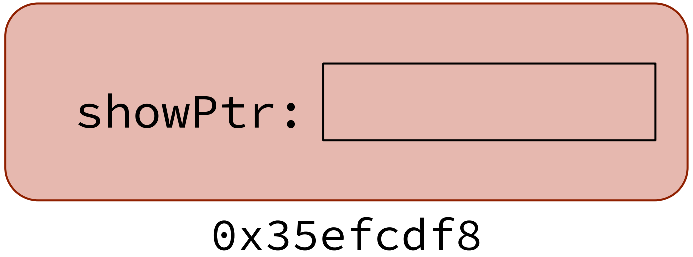
```
string* showPtr;     // declare a pointer to a string
int* agePtr;         // declare a pointer to an int
char* letterPtr;     // declare a pointer to a char
```

- The type “pointer to T,” denoted `T*`, is different from the type of the pointee itself, `T`
  - The type for `showPtr` is `string*` and not `string`

Pointer Syntax, Part 2

- To get the address of another variable, use the & (ampersand) operator.
- This is **not** the same as using a reference parameter. Same symbol, different meanings!

Pointer Syntax, Part 2

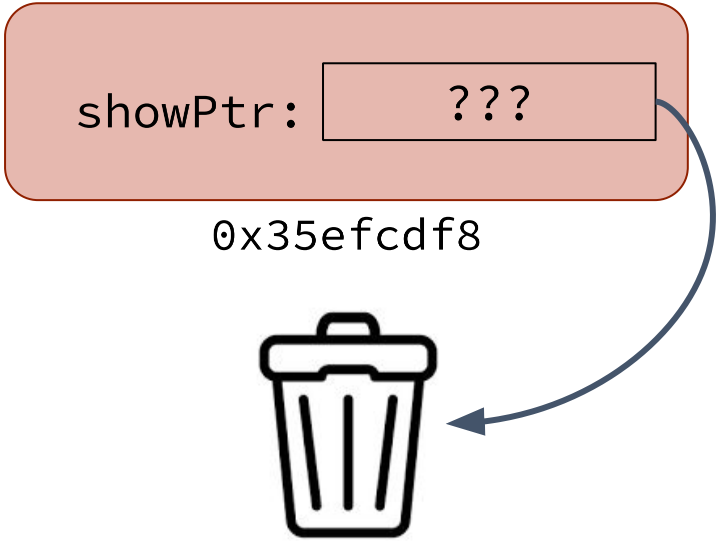


showPtr:

0x35efcdf8

```
string* showPtr;
```

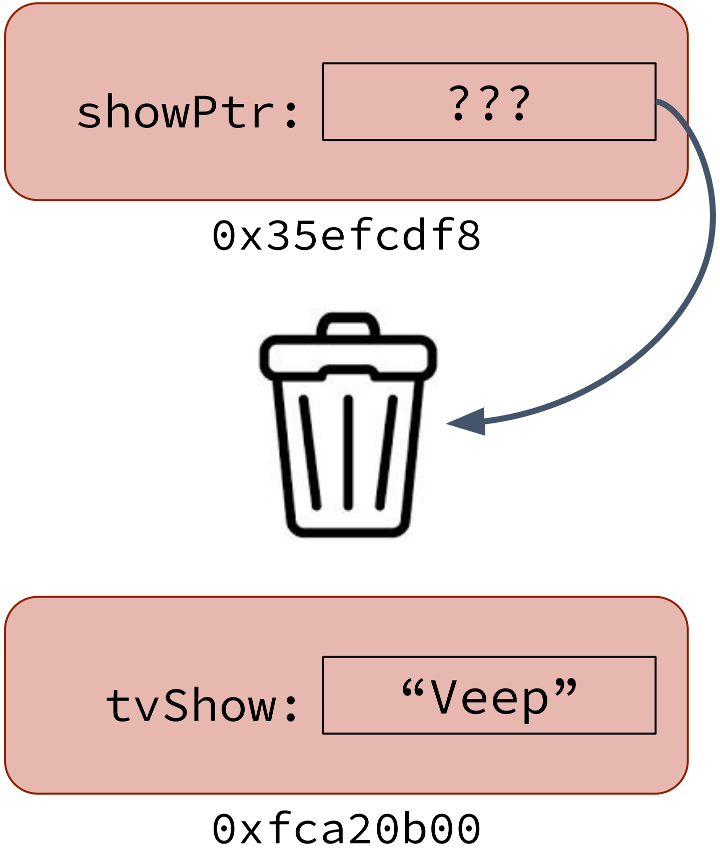
Pointer Syntax, Part 2



```
string* showPtr;
```

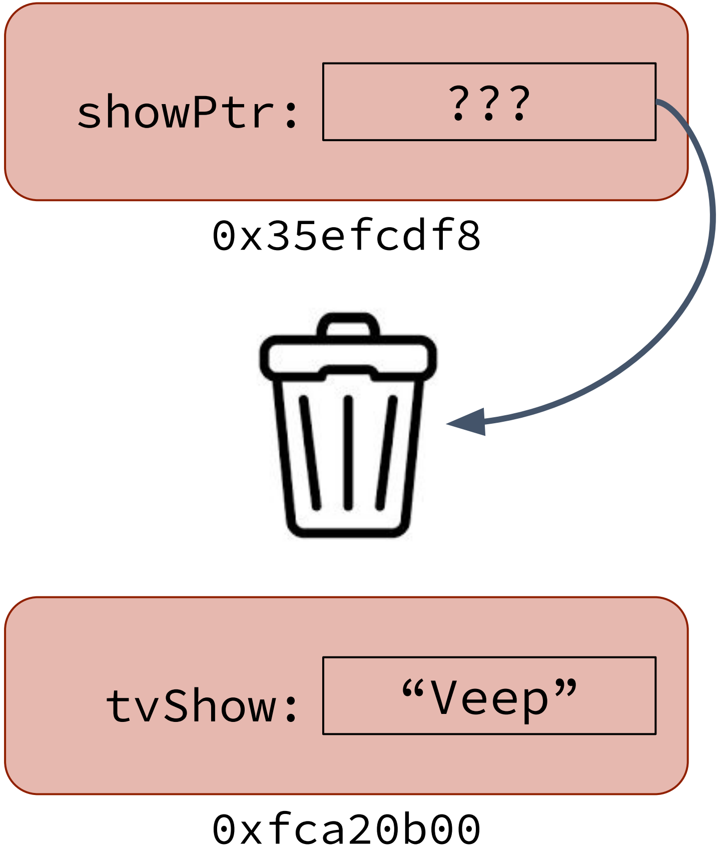
Pointer Syntax, Part 2

```
string* showPtr;
string tvShow = “Veep”;
```



Pointer Syntax, Part 2

```
string* showPtr;
string tvShow = “Veep”;
showPtr = &tvShow;
```



showPtr: ???

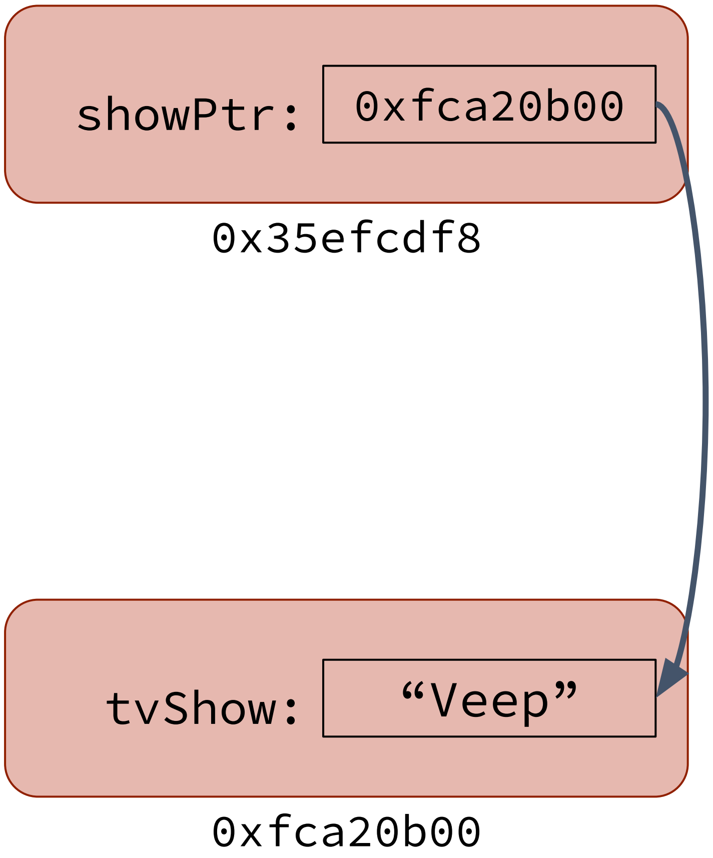
0x35efcdf8

tvShow: “Veep”

0xfca20b00

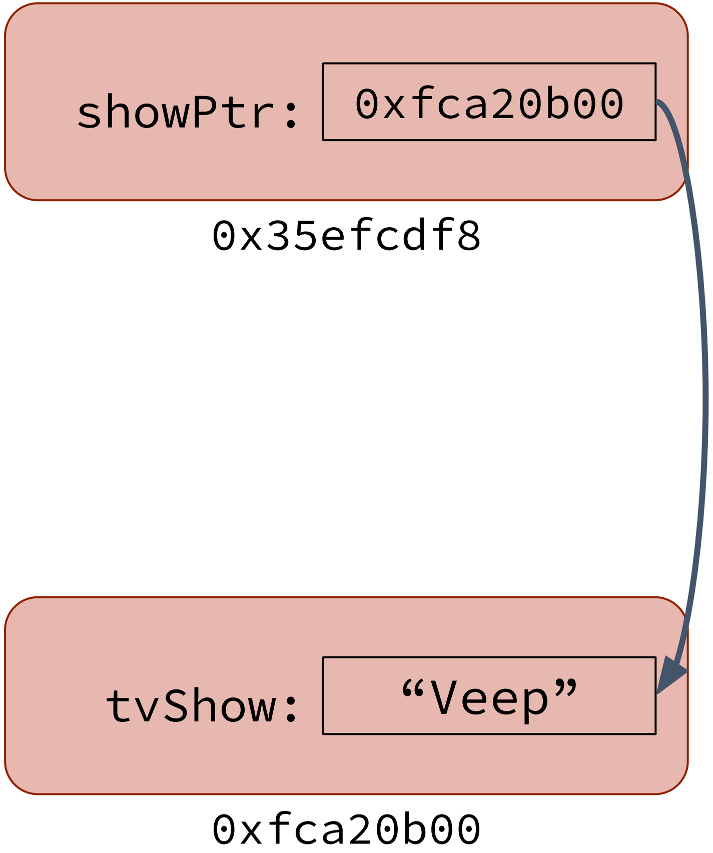
Pointer Syntax, Part 2

```
string* showPtr;
string tvShow = “Veep”;
showPtr = &tvShow;
```



Pointer Syntax, Part 2

```
string tvShow = “Veep”;
string* showPtr = &tvShow;
```



Pointer Syntax, Part 3

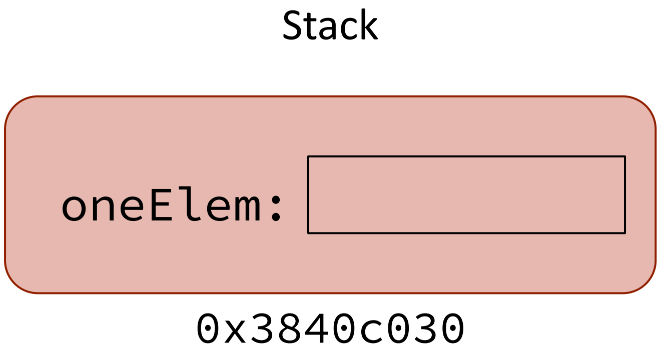
- Pointers are necessary to store the value generated by the new keyword (which is just a memory address on the heap).

```
int* oneElem = new int;
```

Pointer Syntax, Part 3

- Pointers are necessary to store the value generated by the new keyword (which is just a memory address on the heap).

```
int* oneElem = new int;
```



Stack

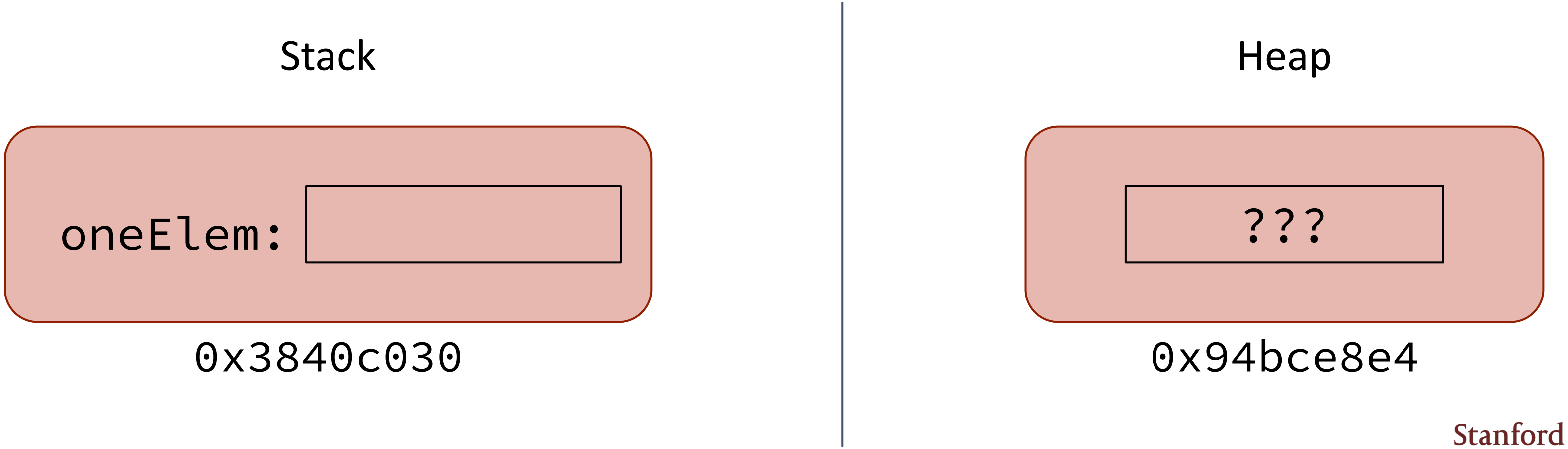
oneElem:

0x3840c030

Pointer Syntax, Part 3

- Pointers are necessary to store the value generated by the new keyword (which is just a memory address on the heap).

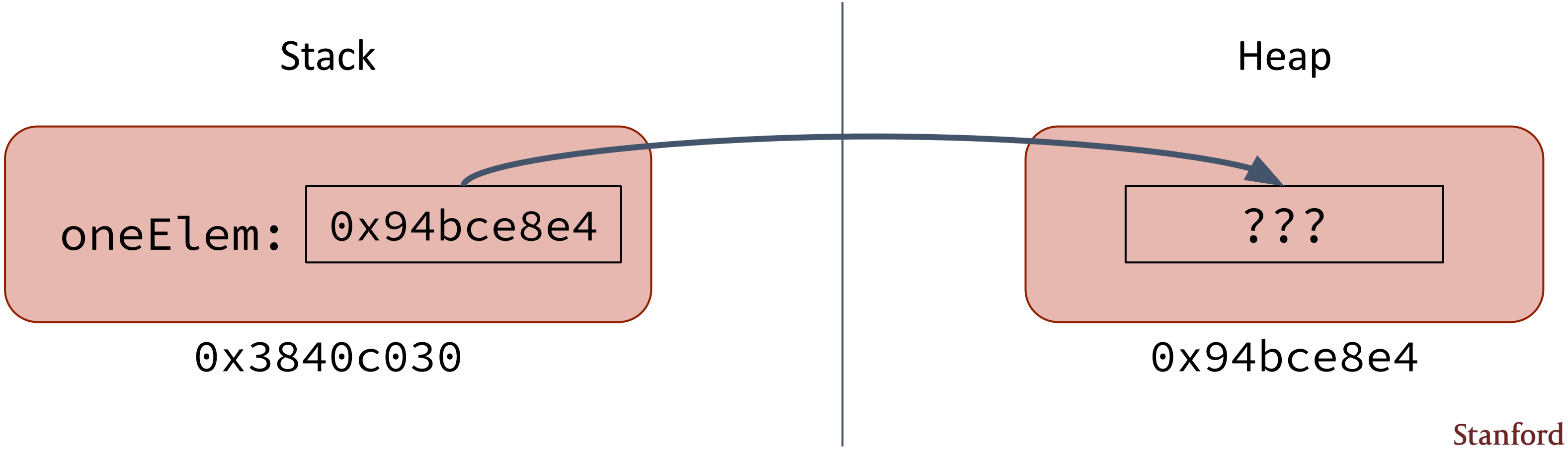
```
int* oneElem = new int;
```



Pointer Syntax, Part 3

- Pointers are necessary to store the value generated by the new keyword (which is just a memory address on the heap).

```
int* oneElem = new int;
```

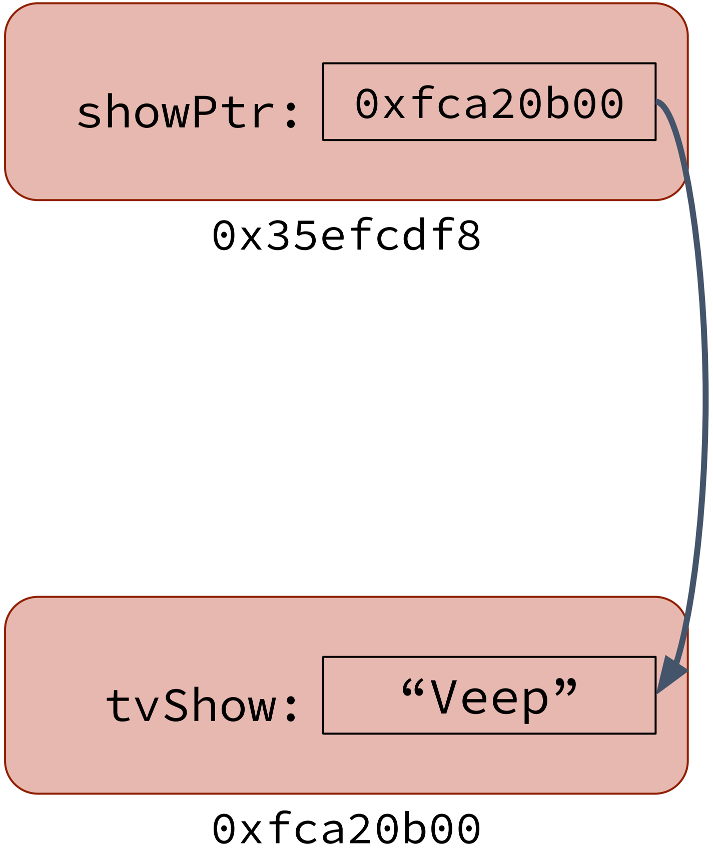


Pointer Syntax, Part 4

- To read or modify the variable that a pointer points to, we use the * (asterisk) operator (in a different way than before!)
- Known as **dereferencing the pointer**
- Follow the arrow to the memory location at the end of the arrow and then read or modify the value stored there

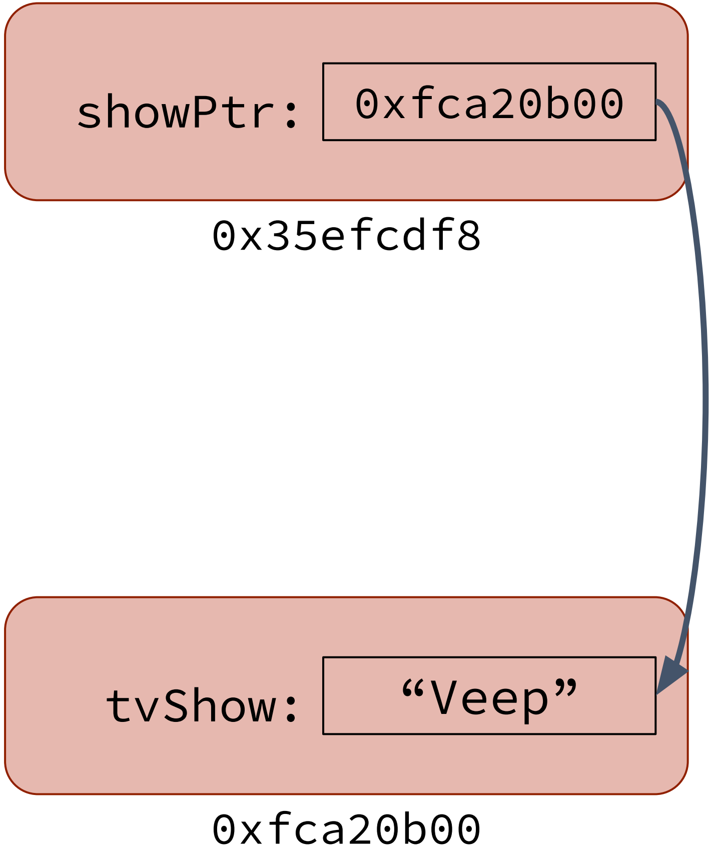
Pointer Syntax, Part 4

```
string tvShow = “Veep”;
string* showPtr = &tvShow;
```



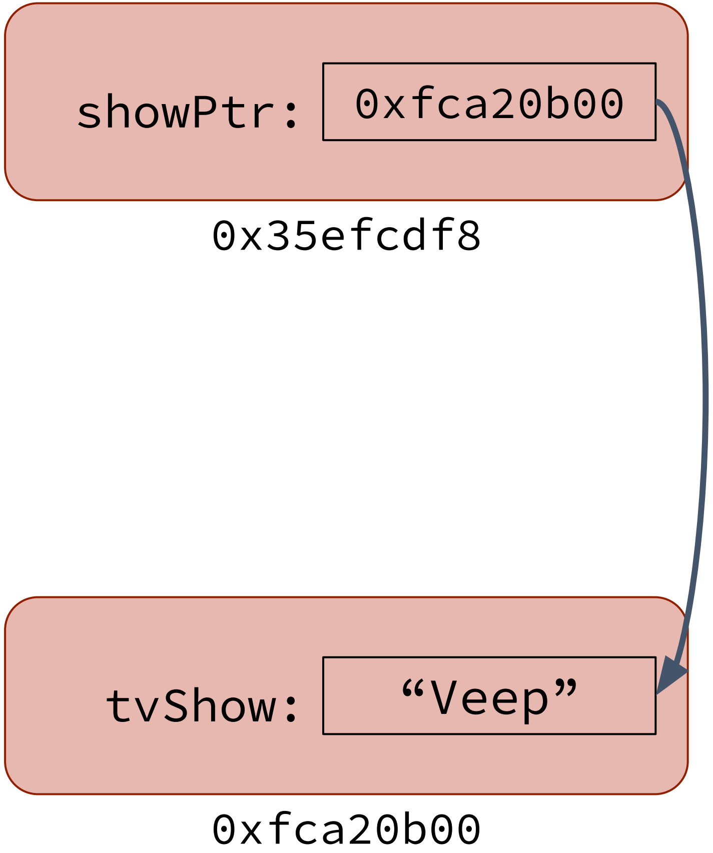
Pointer Syntax, Part 4

```
string tvShow = “Veep”;
string* showPtr = &tvShow;
cout << *showPtr << endl;
```



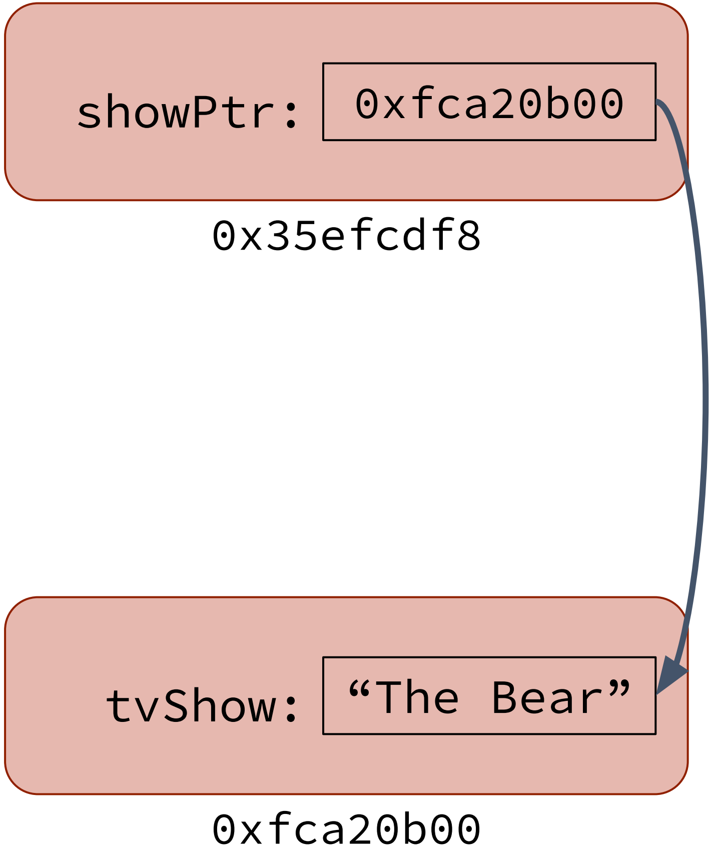
Pointer Syntax, Part 4

```
string tvShow = “Veep”;
string* showPtr = &tvShow;
cout << *showPtr << endl;
*showPtr = “The Bear”;
```



Pointer Syntax, Part 4

```
string tvShow = “Veep”;
string* showPtr = &tvShow;
cout << *showPtr << endl;
*showPtr = “The Bear”;
```



showPtr: 0xfca20b00

0x35efcdf8

tvShow: “The Bear”

0xfca20b00

What is a pointer?

A memory address!!

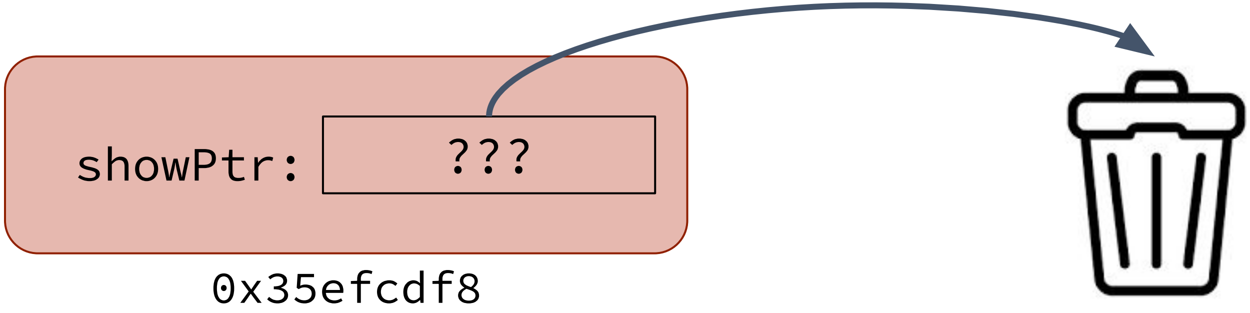
Pointer Tips

- Working with pointers and direct memory access can be very tricky!
- You must always be hyper-vigilant about what is pointing where and what pointers are valid before trying to dereference them
- Here are a couple helpful tips to keep in mind when working with pointers

Pointer Tip #1

- When we declare/initialize a pointer but don't have anything to point it at yet, that can be dangerous and unpredictable

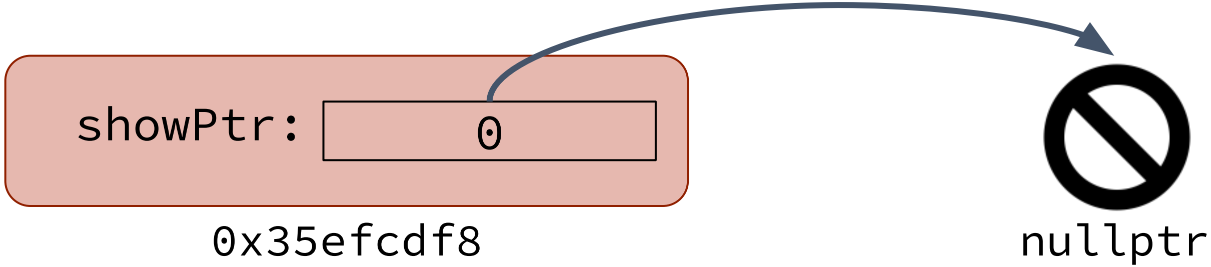
```
string* showPtr;
```



Pointer Tip #1

- When we declare/initialize a pointer but don’t have anything to point it at yet, that can be dangerous and unpredictable
- To ensure that we can tell if a pointer has a valid address or not, set your declared pointer to `nullptr`, which means "no valid address"
    - `nullptr` is C++ is actually just `0`

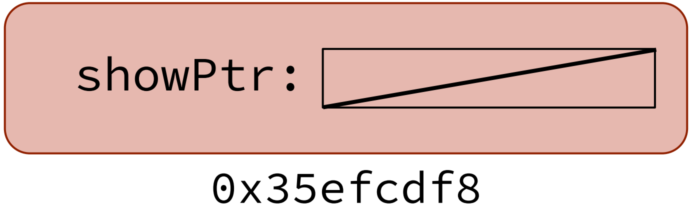
```
string* showPtr = nullptr;
```



Pointer Tip #1

- When we declare/initialize a pointer but don’t have anything to point it at yet, that can be dangerous and unpredictable
- To ensure that we can tell if a pointer has a valid address or not, set your declared pointer to `nullptr`, which means "no valid address"
  - `nullptr` is C++ is actually just `0`

```
string* showPtr = nullptr;
```



showPtr:

0x35efcdf8

Pointer Tip #2

- How can we tell if a pointer is safe to use (dereference)?
- If you are unsure if your pointer holds a valid address, you should check for `nullptr`!

```
void printShowName(string* showPtr) {
    if (showPtr != nullptr) {
        cout << *showPtr << endl; // prints out the value pointed to by showPtr
                                 // if it is not nullptr
    } else {
        cout << "showPtr is not valid!" << endl;
    }
 }
```

Pointer Practice

Draw diagrams!

What is a pointer?

A memory address!!

Practice #1

- What type does this pointer point to?
- What should we draw?

```
int* numPtr = nullptr;
```

Practice #1

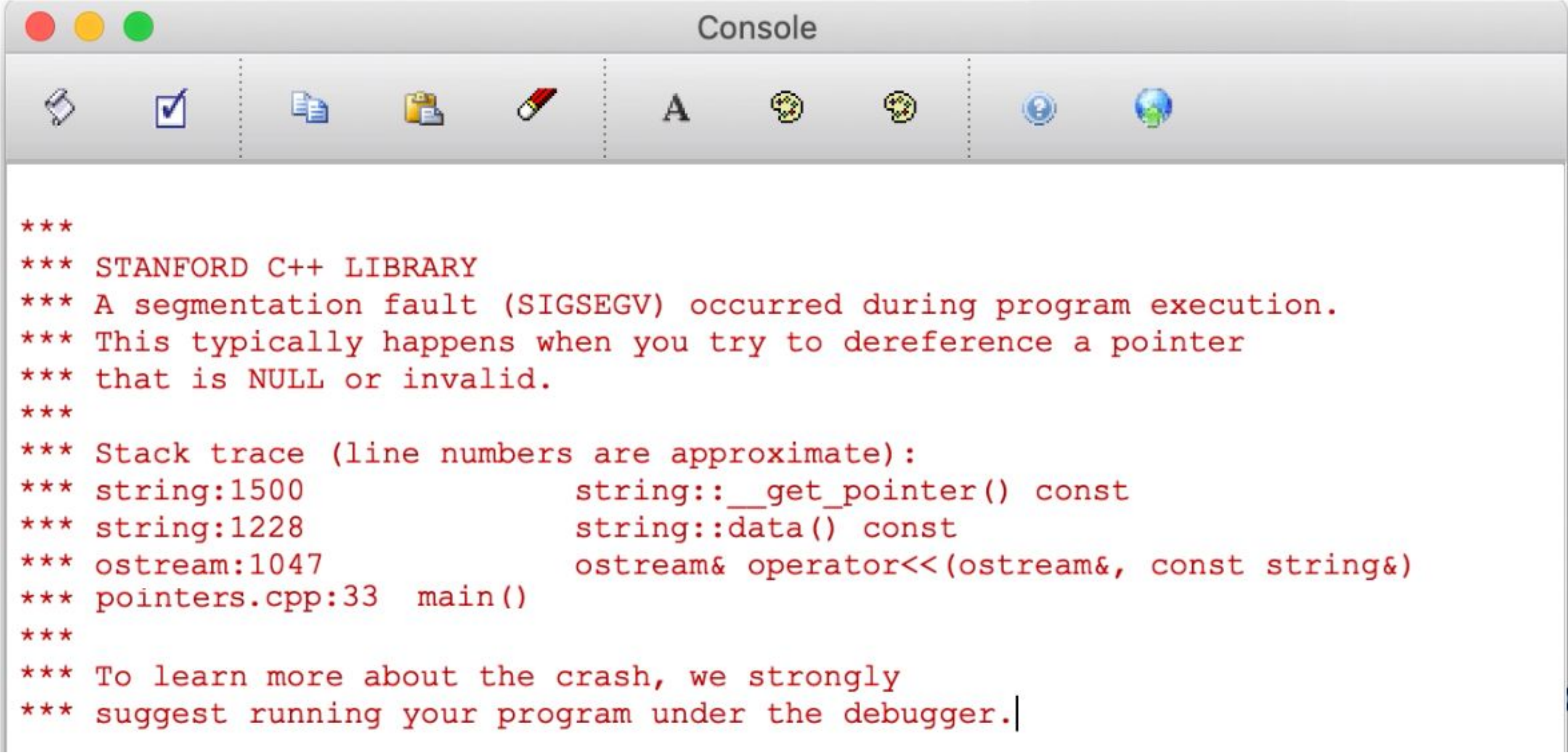
- Trace through this code with a diagram
- What is the output?

```
int* numPtr = nullptr;
int num = 16;
numPtr = &num;
cout << *numPtr << end;
*numPtr = 198;
```

Practice #2

- Trace through this code with a diagram
- What is the output?

```
string* sPtr = nullptr;
string s = "hello";
cout << *sPtr << endl;
```



```
***
*** STANFORD C++ LIBRARY
*** A segmentation fault (SIGSEGV) occurred during program execution.
*** This typically happens when you try to dereference a pointer
*** that is NULL or invalid.
***
*** Stack trace (line numbers are approximate):
*** string:1500              string::__get_pointer() const
*** string:1228              string::data() const
*** ostream:1047             ostream& operator<<(ostream&, const string&)
*** pointers.cpp:33  main()
***
*** To learn more about the crash, we strongly
*** suggest running your program under the debugger.
```

Practice #2

- How can we fix this code?

```
string* sPtr = nullptr;
string s = "hello";
cout << *sPtr << endl;
```

Practice #2

- How can we fix this code?

```
string* sPtr = nullptr;
string s = "hello";
if (sPtr != nullptr) {
  cout << *sPtr << endl;
}
```

```
string* sPtr = nullptr;
string s = "hello";
sPtr = &s;
cout << *sPtr << endl;
```

Practice #3

- What is the output?

```
string* sPtr1 = nullptr;
string* sPtr2 = nullptr;
string s = "hello";
sPtr1 = &s;
cout << *sPtr1 << endl;

sPtr2 = sPtr1;
cout << *sPtr2 << endl;

*sPtr1 = "goodbye";
cout << *sPtr1 << endl;
cout << *sPtr2 << endl;
```

Check out the Lecture 18 Code

Copy Constructor

Challenge Problem

Binky!

Recap

- All variables in a computer program are stored in computer memory and can each be uniquely identified by their numerical memory address
- Pointers are a special type of variable that store memory addresses
- Pointers are essential to store the location of dynamically allocated memory acquired on the heap
- The dereference operator allows us to access and modify the memory pointed to by a pointer