

Memory and Pointers

What ideas/concepts/topics of interest are you thinking about for your final project?

pollev.com/cs106bpolls



What ideas/concepts/topics of interest are you thinking about for your final project?



Roadmap

C++ basics

User/client

vectors + grids

stacks + queues

sets + maps

Core
Tools

testing

algorithmic
analysis

recursive
problem-solving

Object-Oriented
Programming

Implementation

arrays

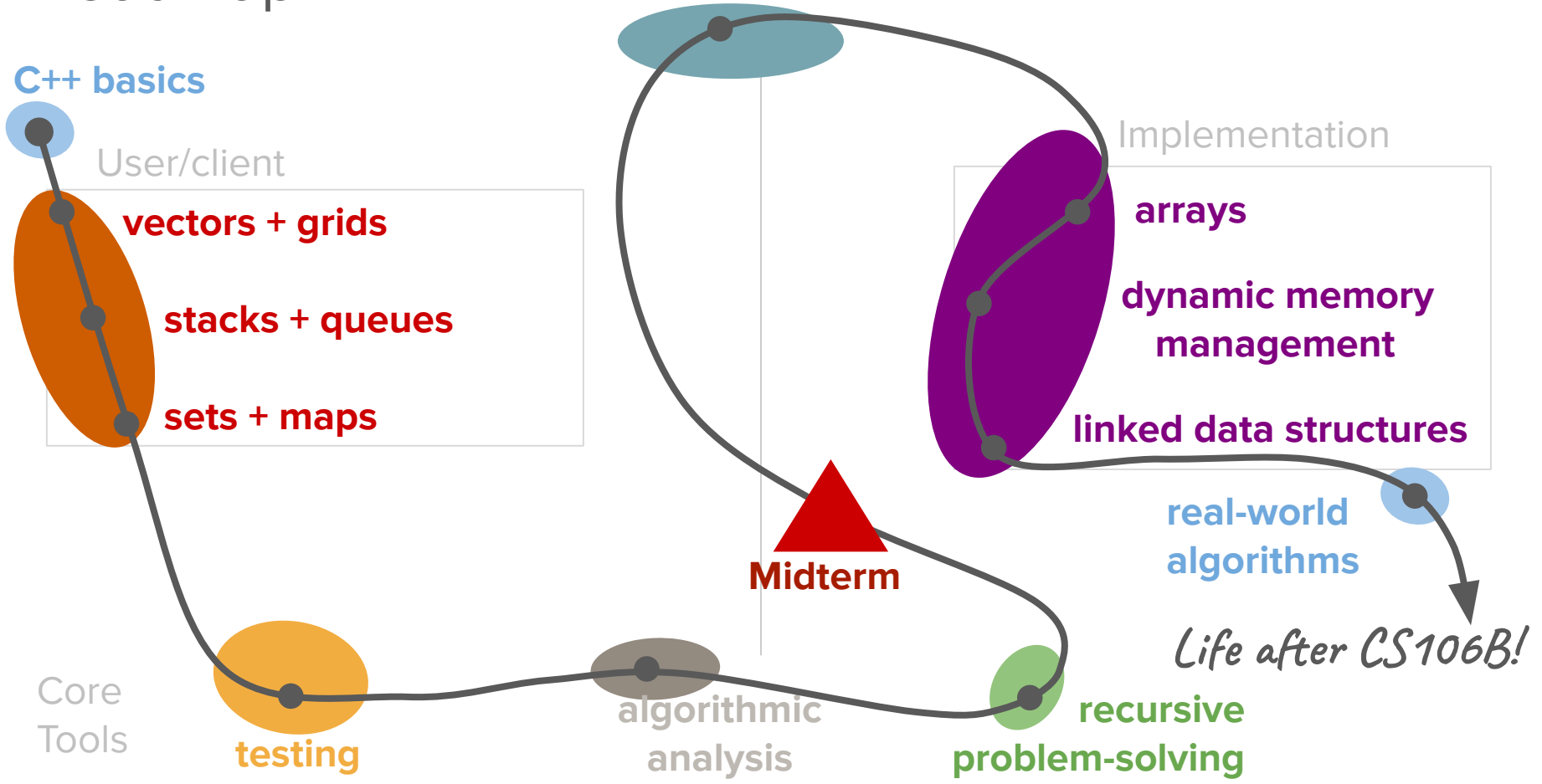
dynamic memory
management

linked data structures

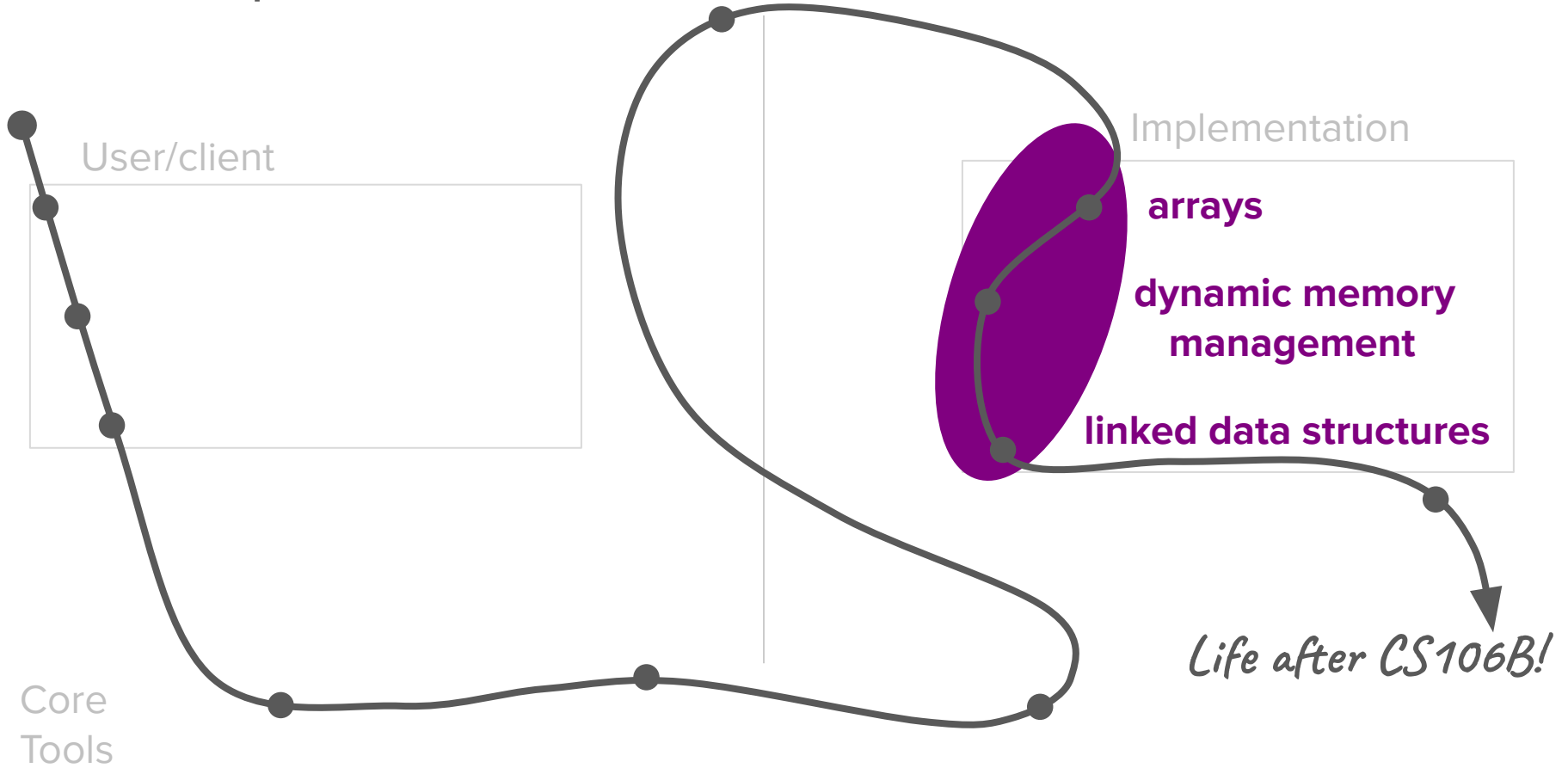
real-world
algorithms

Life after CS106B!

Midterm



Roadmap



Today's question

How is a computer's
memory system organized?

How can we navigate and
directly manipulate
computer memory in C++?

Today's topics

1. Review (Priority Queues and Heaps)
2. Computer Memory
3. Pointed Points on Pointers

Review

[priority queues and heaps]

Implementing ADT Classes

- The first step of implementing an ADT class (as with any class) is answering the three important questions regarding its public interface, private member variables, and initialization procedures.
- Most ADT classes will need to store their data in an underlying array. **The organizational patterns of data in that array may vary**, so it is important to illustrate and visualize the contents and any operations that may be done.
- The paradigm of "growable" arrays allows for fast and flexible containers with dynamic resizing capabilities that enable storage of large amounts of data.

Levels of abstraction

What is the interface for the user?

(Priority Queue)



How is our data organized?

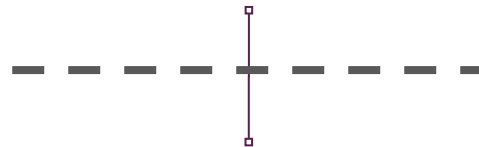
(sorted array, binary heap)



What stores our data?

(arrays)

Abstract Data Structures



Data Organization Strategies



Fundamental C++ Data Storage

What is a priority queue?

- A queue that orders its elements based on a provided “priority”
- Like regular queues, you cannot index into them to get an item at a particular position.
- Useful for maintaining data sorted based on priorities
 - Emergency room waiting rooms
 - Different airline boarding groups (families and first class passengers, frequent flyers, boarding group A, boarding group B, etc.)
 - Filtering data to get the top X results (e.g. most popular Google searches or fastest times for the Women’s 800m freestyle swimming event)

Supported operations

- **enqueue(priority, elem)**: inserts **elem** with given **priority**
- **dequeue()**: removes the element with the highest priority from the queue
- **peek()**: returns the element with the highest priority in the queue (no removal)
- **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

What is a binary heap?

- A heap is a tree-based structure that satisfies the *heap property* that **parents have a higher priority than any of their children.**
- Additional properties
 - **Binary:** Two children per parent (but no implied orderings between siblings)
 - **Completely filled** (each parents must have 2 children) except for the bottom level, which gets populated from **left to right**
- Two types → which we use depends on what we define as a “higher” priority
 - **Min-heap:** smaller numbers = higher priority (closer to the root)
 - **Max-heap:** larger numbers = higher priority (closer to the root)

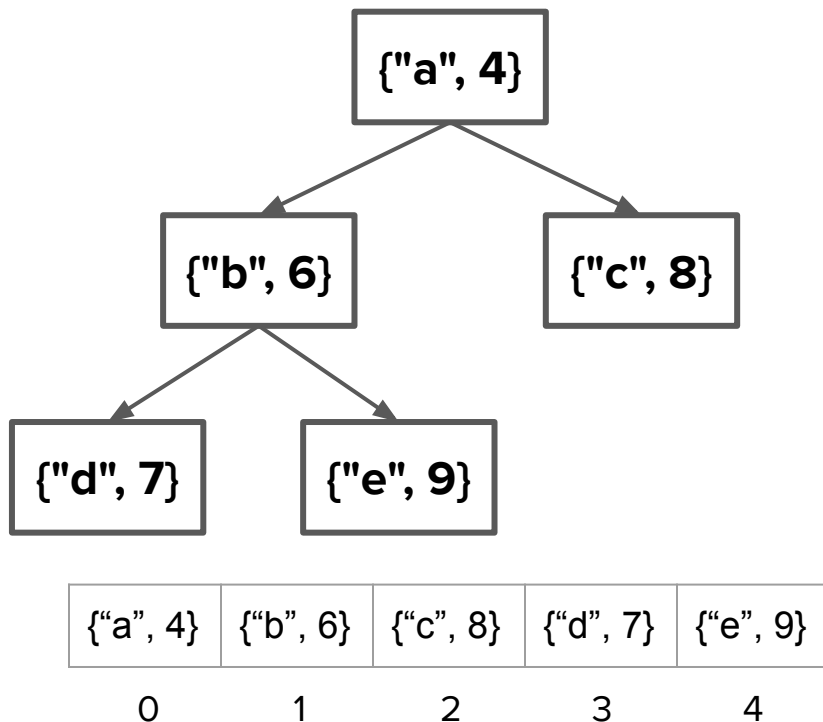
Binary heaps + implementation

Node: i

Left child: $2*i + 1$

Right child: $2*i + 2$

Parent: $(i-1) / 2$



Node index: 0

Left child: 1

Right child: 2

Parent: N/A

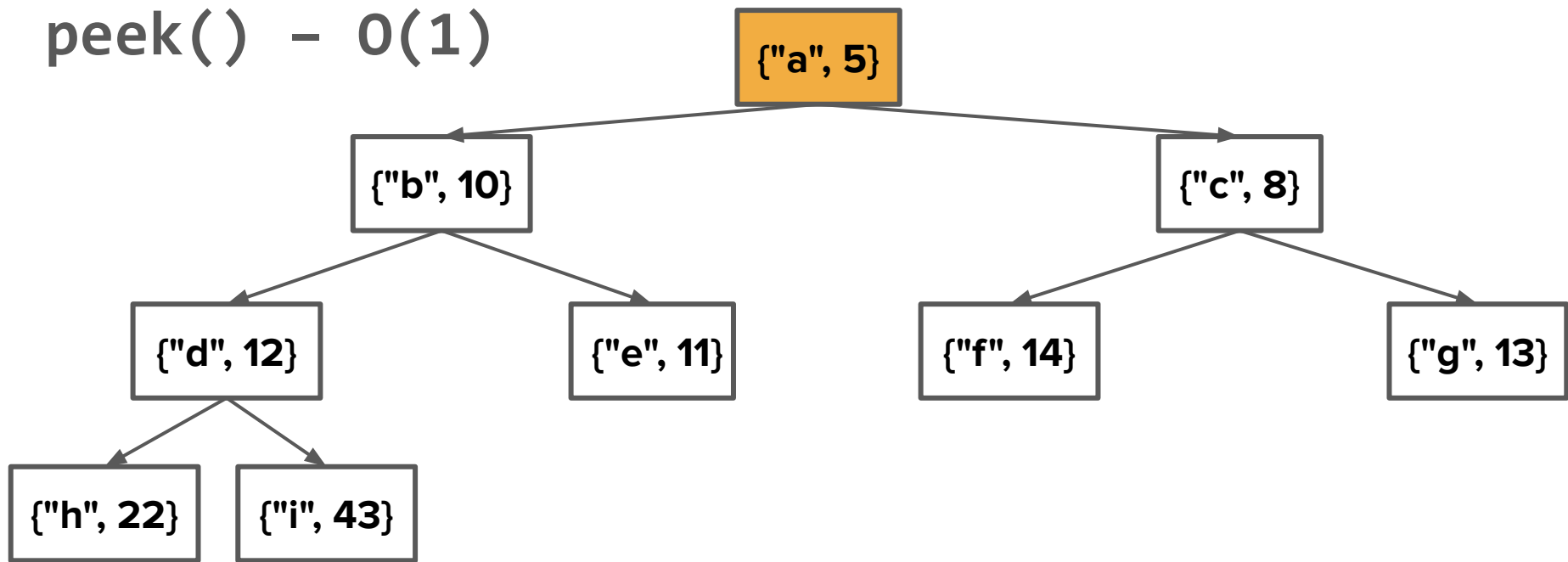
Node index: 1

Left child: 3

Right child: 4

Parent: 0

peek() - O(1)



{"a", 5}	{"b", 10}	{"c", 8}	{"d", 12}	{"e", 11}	{"f", 14}	{"g", 13}	{"h", 22}	{"i", 43}	?	?	?
0	1	2	3	4	5	6	7	8	9	10	11

enqueue()

- Add the new element into the first empty slot in the array.
- Bubble up to regain the *heap property*!
- Runs in time **$O(\log n)$**

dequeue()

- Remove the minimum element: the root of the tree.
- Replace the root with the “last” element in our tree (last level, farthest right) since we know that location will end up empty.
- Bubble down to regain the *heap property*!
- Runs in time **$O(\log n)$**

Summary

- **Priority queues** are queues ordered by **priority** of their elements, where the **highest priority** elements get dequeued first.
- **Binary heaps** are a good way of organizing data when creating a priority queue.
 - Use a min-heap when a smaller number = higher priority (what you'll use on the assignment) and a max-heap when a larger number = higher priority.
- There can be multiple ways to implement the same abstraction! For both ways of implementing our priority queues, we'll use **arrays** for data storage.

Levels of abstraction

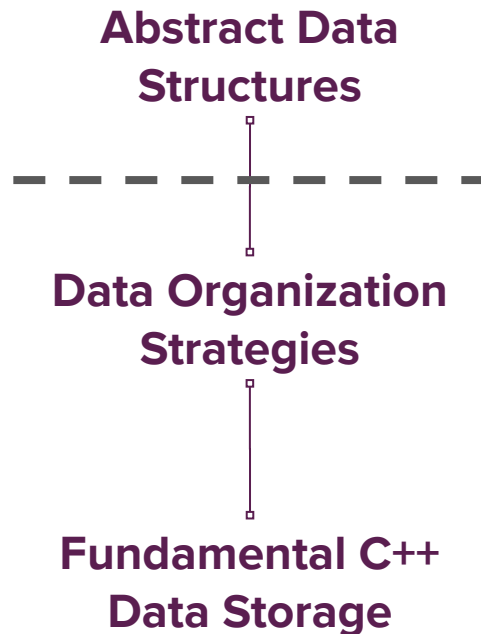
What is the interface for the user?
(Vector, Set, Priority Queue, etc.)



How is our data organized?
(sorted array, binary heap, tree, etc.)



What stores our data?
(arrays, linked lists, etc.)



Levels of abstraction

What is the interface for the user?
(Vector, Set, Priority Queue, etc.)

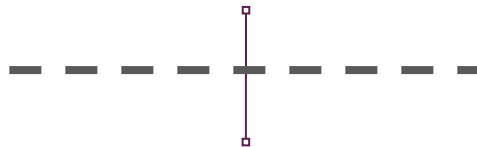


How is our data organized?
(sorted array, binary heap, tree, etc.)



What stores our data?
(arrays, **linked lists, etc.**)

Abstract Data Structures



Data Organization Strategies



**Fundamental C++
Data Storage**

How is a computer's memory system organized?

Levels of abstraction

**Abstract Data
Structures**

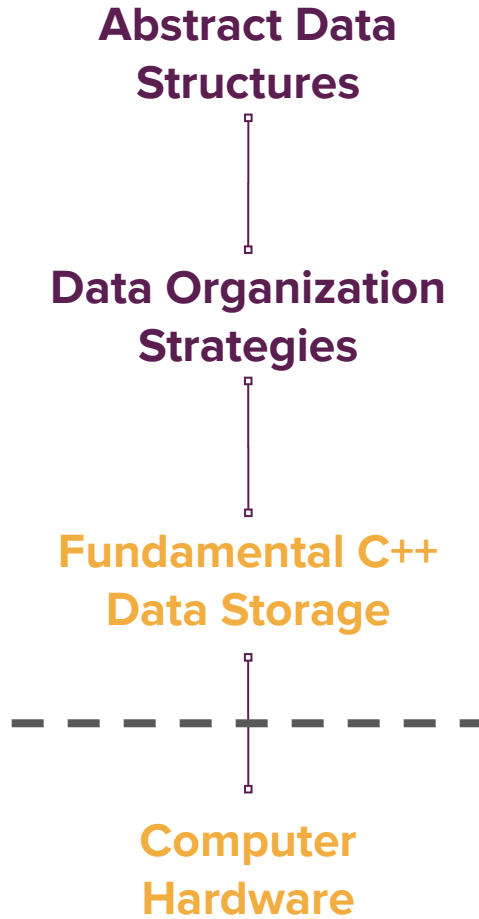


**Data Organization
Strategies**



**Fundamental C++
Data Storage**

Levels of abstraction



The Hardware/Software Boundary

What is computer memory?

- A computer is a real, physical machine made up of many different components. We refer to these components as the computer's **hardware**.

What is computer memory?

- A computer is a real, physical machine made up of many different components. We refer to these components as the computer's **hardware**.
- The computer programs we write are referred to as **software**. These allow us to send specific instructions to the computer's hardware to do calculations, store information, etc.

What is computer memory?

- A computer is a real, physical machine made up of many different components. We refer to these components as the computer's **hardware**.
- The computer programs we write are referred to as **software**. These allow us to send specific instructions to the computer's hardware to do calculations, store information, etc.
- 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 usually refer to when we talk about "computer memory."
 - C++ gives us a variety of fundamental ways to access computer hardware from our code.

What is computer memory?

- A computer is a real, physical machine made up of many different components. We refer to these components as the computer's **hardware**.
- The computer programs we write are called **software**. These allow us to send specific instructions to the hardware to do calculations, store information, etc.
- 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 usually refer 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 the 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.

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 even more interesting data storage and organization techniques (beyond arrays).

Why is computer memory important?

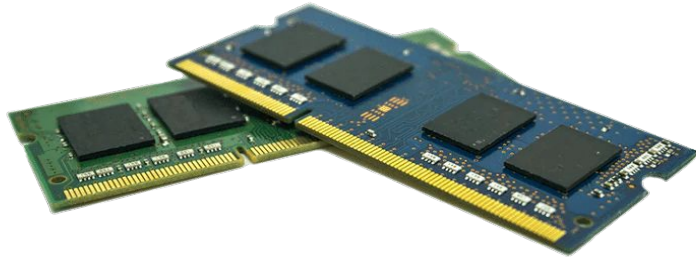
- We've already seen the power and importance of being able to **dynamically allocate arrays** and use these as **data storage fundamentals** for ADT classes.
- Being able to directly work with computer memory opens up the doors to even more interesting data storage and organization techniques (beyond 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?

- Let's build a mental model of how data is organized in computer memory.

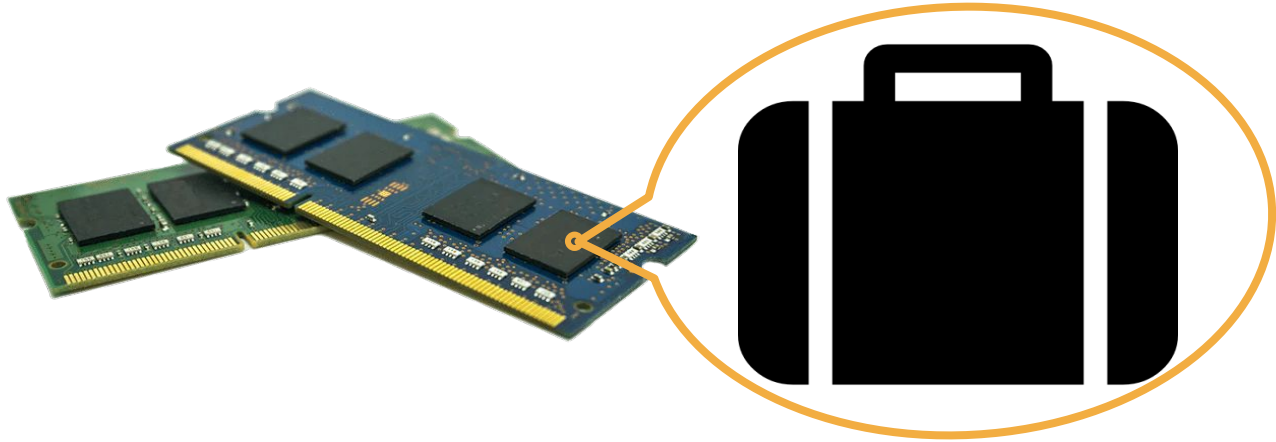
How is computer memory organized?

- Let's build a mental model of how data is organized in computer memory.



How is computer memory organized?

- Let's build a mental model of how data is organized in computer memory.



How is computer memory organized?

- Let's build a mental model of how data is organized in computer memory.
- Memory can be thought of as a giant collective pool of boxes (or suitcases, to stay on thematic trend) in which we can store information.

How is computer memory organized?

- Let's build a mental model of how data is organized in computer memory.
- Memory can be thought of as a giant collective pool of boxes (or suitcases, to stay on thematic trend) in which we can store information.
- **Question:** How can we communicate with the computer to find exactly which box we want to access/store information in?

How is computer memory organized?

- Let's build a mental model of how data is organized in computer memory.
- Memory can be thought of as a giant collective pool of boxes (or suitcases, to stay on thematic trend) in which we can store information.
- **Question:** How can we communicate with the computer to find exactly which box we want to access/store information in?
 - **Key Idea:** Each box can be located using the computer's internal organization system, in which each box has an associated numerical location, called a **memory address**.

How is computer memory organized?

- Let's build a mental model of how data is organized in computer memory.
- Memory can be thought of as a giant collective pool of boxes (or suitcases, to stay on thematic trend) in which we can store information.
- **Question:** How can we communicate with the computer to find exactly which box we want to access/store information in?
 - **Key Idea:** Each box can be located using the computer's internal organization system, in which each box has an associated numerical location, called a **memory address**.
 - Just like a normal address, this value tells us where the box is located!

Memory Addresses

```
string pet = "cat";
```

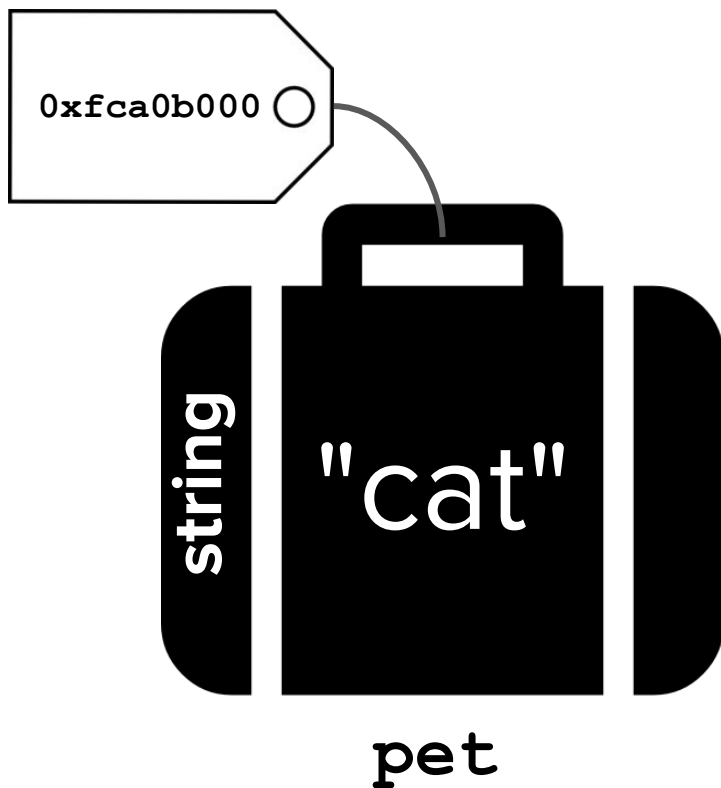
Memory Addresses

```
string pet = "cat";
```



Memory Addresses

```
string pet = "cat";
```



Memory Addresses

```
string pet = "cat";
```

The *memory address* of `pet` is `0xfca0b000`. This special numerical value acts as the unique identifier for this variable across the entire pool of the computer's memory.

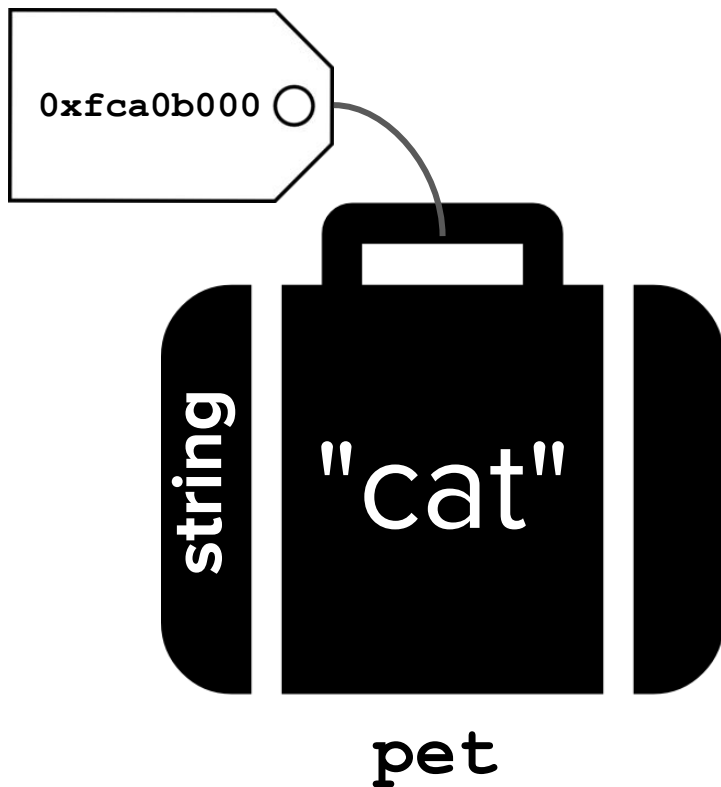


Memory Addresses

```
string pet = "cat";
```

How is this value determined?

The computer (operating system) determines the address, not you!

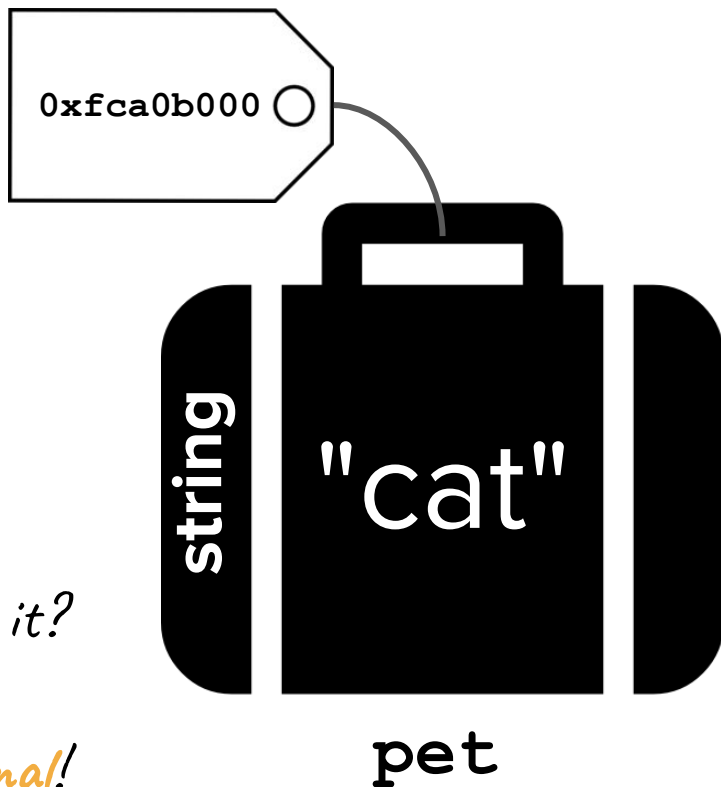


Memory Addresses

```
string pet = "cat";
```

Is that really a number? Why is preceded by 0x and have letters in it?

*Let's talk (briefly) about **hexadecimal**!*



The Hexadecimal Number System

- Normally, we represent numbers using the decimal (base-10) number system.
 - Each place value represents a factor of ten (ones, tens, hundreds, etc.), and there are 10 digits.

The Hexadecimal Number System

- Normally, we represent numbers using the decimal (base-10) number system.
- In computer systems, there a number of factors that make it 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.), and there are 16 possible "digits" for each place value.
 - Since there are only 10 numerical digits (0-9), this system also uses the letters a to f as "digits."
 - **0 1 2 3 4 5 6 7 8 9 a(10) b(11) c(12) d(13) e(14) f(15)**

The Hexadecimal Number System

- Normally, we represent numbers using the decimal (base-10) number system.
- In computer systems, there a number of factors that make it more convenient to express numbers using the **hexadecimal (base-16) number system**.
- The prefix **0x** is used to communicate that a number is being expressed in hexadecimal.

The Hexadecimal Number System

- Normally, we represent numbers using the decimal (base-10) number system.
- In computer systems, there a number of factors that make it more convenient to express numbers using the **hexadecimal (base-16) number system**.
- The prefix **0x** is used to communicate that a number is being expressed in hexadecimal.
- In the end, remember that the specific address values have no special meaning to us, since they're always generated by the computer. This is mostly just a fun aside!

Memory Organization Summary

- 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.

Memory Organization Summary

- 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.

Pointers are a type of variable that allow us to store memory addresses directly, giving us more power over the computer.

Announcements

Announcements

- Midterm regrade requests are due **tomorrow by 11:59pm**.
- Assignment 2 revisions are due **tomorrow at 11:59pm**.
- The final project proposals are due **Sunday, July 24 at 11:59pm**. We are excited to read through the exciting ideas!
 - You can expect feedback around a week from the deadline.
- Assignment 4 is due next **Tuesday, July 26 at 11:59pm**.
 - YEAH materials are now on Ed and Canvas.
 - Make sure to get started early! Debugging dynamic memory can be tricky.

How can we navigate and
directly manipulate computer
memory in C++?

How can we navigate and
directly manipulate computer
memory in C++?

Pointers!

Pointers

- A pointer is a data type that allows us to work directly with computer memory addresses.

Pointers

- A pointer is a data type that allows us to work directly with computer memory addresses.
- Just like all other variables, pointers take up space in memory and store specific values.

Pointers

- A pointer is a data type that allows us to work directly with computer memory addresses.
- Just like all other variables, pointers take up space in memory and store specific values.
- **A pointer always stores a memory address**, telling us where in the computer's memory to look for a certain value.

Pointers

- A pointer is a data type that allows us to work directly with computer memory addresses.
- Just like all other variables, pointers take up space in memory and store specific values.
- **A pointer always stores a memory address**, telling us where in the computer's memory to look for a certain value.
- In doing this, they quite literally "point" to another location on your computer.

What is a pointer?

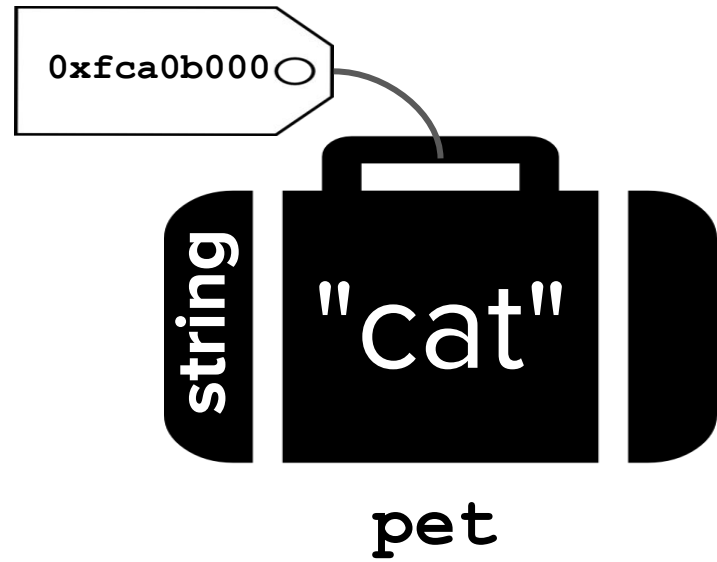
A memory address!

Moving beyond arrays

- We've already worked with pointers in the context of dynamically allocated arrays.
- However, pointers don't have to point to only arrays!
- We'll see the power of pointers when we learn about linked data structures that use non-contiguous dynamic memory.

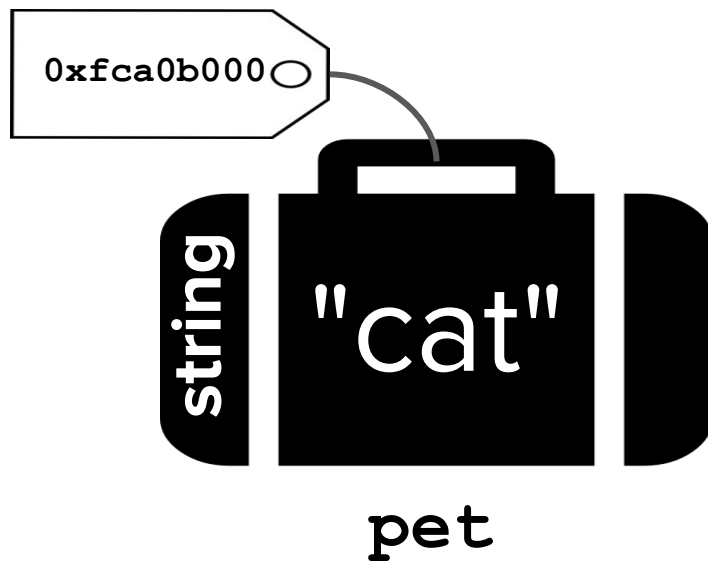
Introduction to Pointers

```
string pet = "cat";
```



Introduction to Pointers

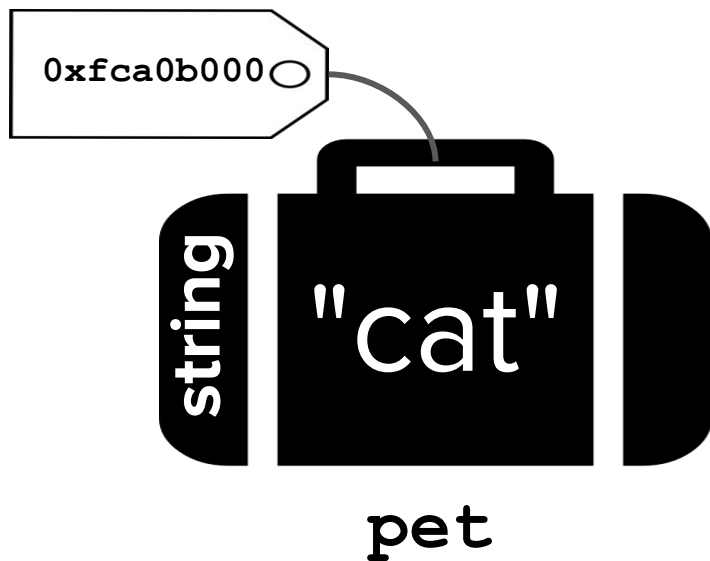
```
string pet = "cat";  
string* petPointer = addressOf(pet);
```



Introduction to Pointers

```
string pet = "cat";  
string* petPointer = addressOf(pet);
```

This "function" doesn't really exist but we'll resolve that soon enough!



Introduction to Pointers

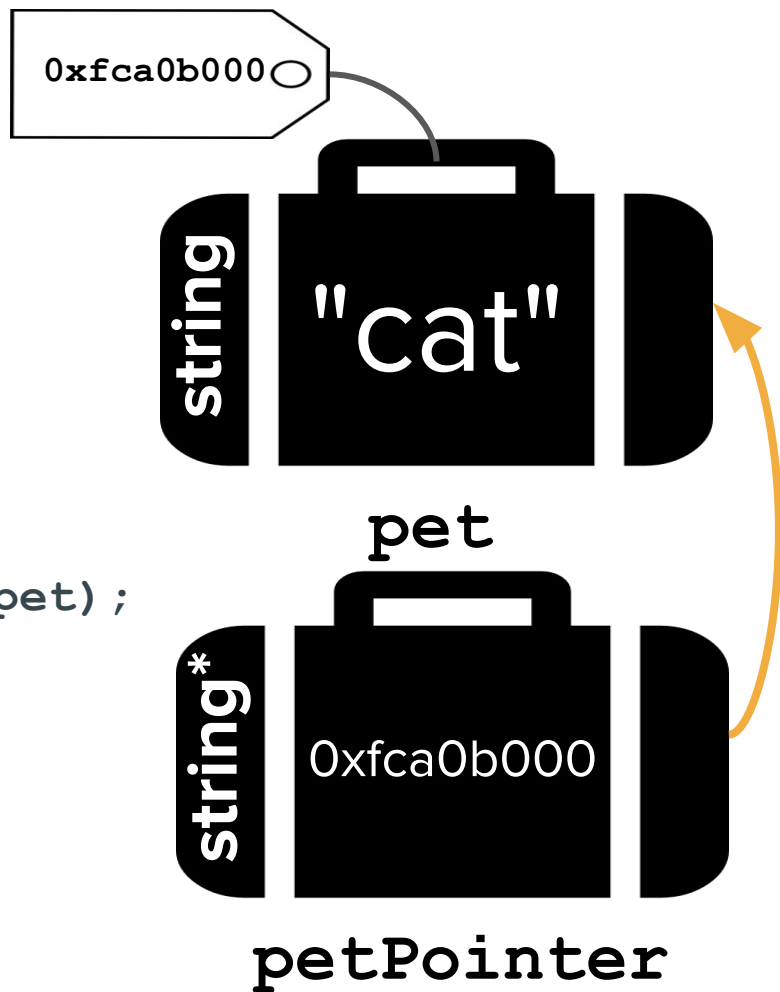
```
string pet = "cat";  
string* petPointer = addressOf(pet);
```



Introduction to Pointers

```
string pet = "cat";  
string* petPointer = addressOf(pet);
```

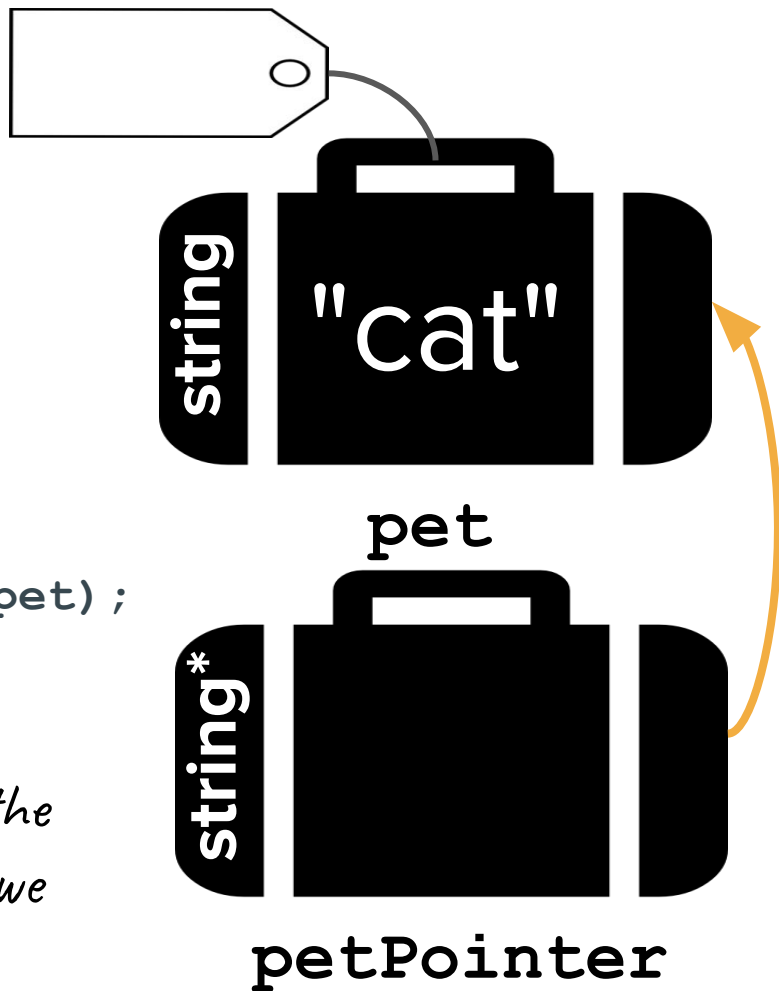
We generally use an arrow to "point" from a pointer to the variable it points to.



Introduction to Pointers

```
string pet = "cat";  
string* petPointer = addressOf(pet);
```

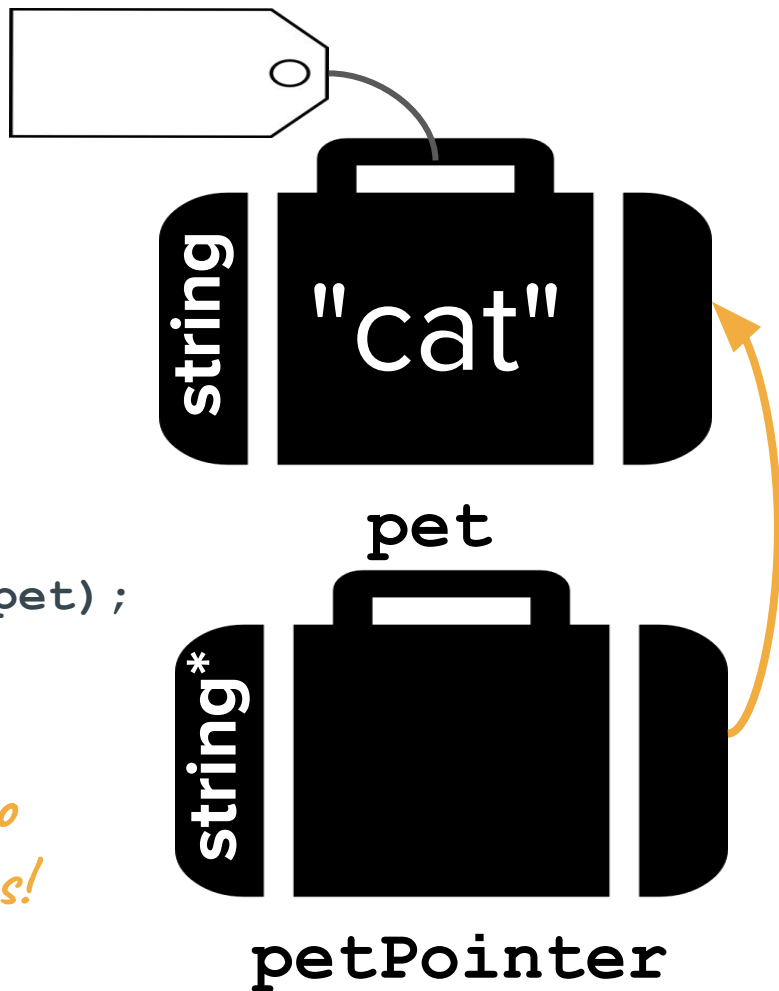
In fact, the specific memory address values don't actually matter. It is just the associated pointer/pointee relationship we care about.



Introduction to Pointers

```
string pet = "cat";  
string* petPointer = addressOf(pet);
```

This visual relationship is key to understanding pointers. The best way to learn pointers is to draw lots of pictures!



What is a pointer?

A memory address!

Pointer Syntax

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!
- Let's 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* petPtr; // declare a pointer to a string
```

```
int* agePtr; // declare a pointer to an int
```

```
char* letterPtr; // declare a pointer to a char
```

Pointer Syntax (Part 1)

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

```
string* petPtr; // declare a pointer to a string
```

```
int* agePtr; // declare a pointer to an int
```

```
char* letterPtr; // declare a pointer to a char
```

- **Important Note:** The type for `petPtr` is `string*` and not `string`. **The type “pointer to T,” denoted `T*` is different from the type of the pointee itself (T).**

Pointer Syntax (Part 2)

- When initializing a pointer, we can use the **& (ampersand)** operator to get the address of the variable that we want to point to

Pointer Syntax (Part 2)

- When initializing a pointer, we can use the **& (ampersand)** operator to get the address of the variable that we want to point to

```
string pet = "cat";
```

```
string* petPointer = addressOf(pet);
```

Pointer Syntax (Part 2)

- When initializing a pointer, we can use the **& (ampersand)** operator to get the address of the variable that we want to point to

```
string pet = "cat";  
string* petPointer = &pet;
```

Pointer Syntax (Part 2)

- When initializing a pointer, we can use the **& (ampersand)** operator to get the address of the variable that we want to point to

```
string pet = "cat";  
string* petPointer = &pet;
```

- Note: This is **not** the same as using a reference parameter. Same symbol but different meanings! Oh C++...

Pointer Syntax (Part 2)

- When initializing a pointer, we can use the **& (ampersand)** operator to get the address of the variable that we want to point to

```
string pet = "cat";  
string* petPointer = &pet;
```

- Note: This is **not** the same as using a reference parameter. Same symbol but different meanings! Oh C++...
- By the way: **You should never need to do this in code you write in CS106B!** You'll use it more in CS107, but if you find yourself using it in this class, reconsider your reason for using it.

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, a.k.a. dynamic memory).

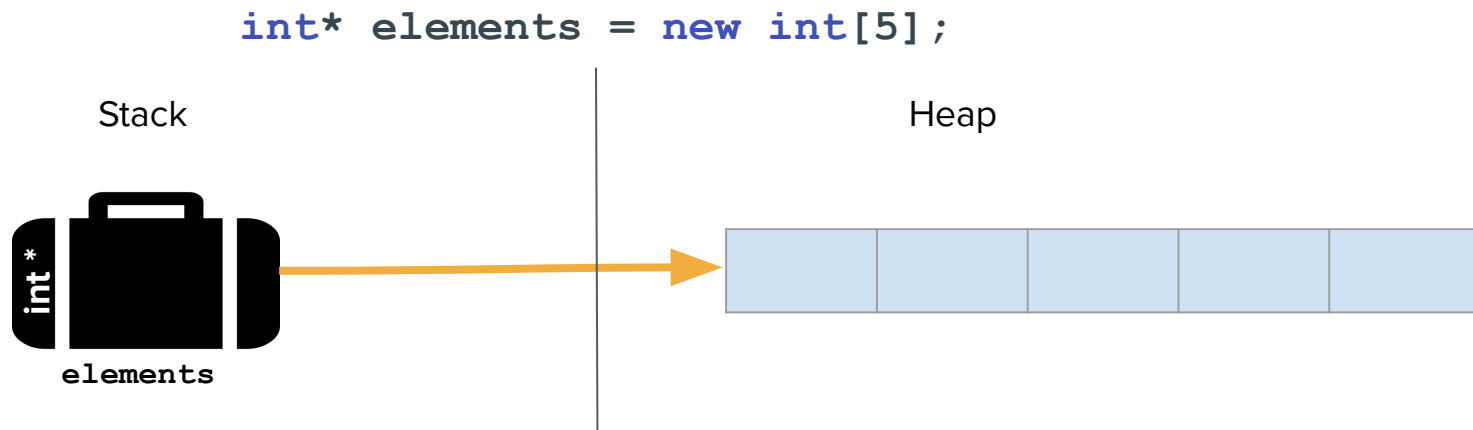
Pointer Syntax (Part 3)

- Pointers are necessary to store the value generated by the **new** keyword (which is just a memory address on the heap, a.k.a. dynamic memory).
- We're familiar with this in the context of arrays:

```
int* elements = new int[5];
```

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, a.k.a. dynamic memory).
- We're familiar with this in the context of arrays:



Pointer Syntax (Part 3)

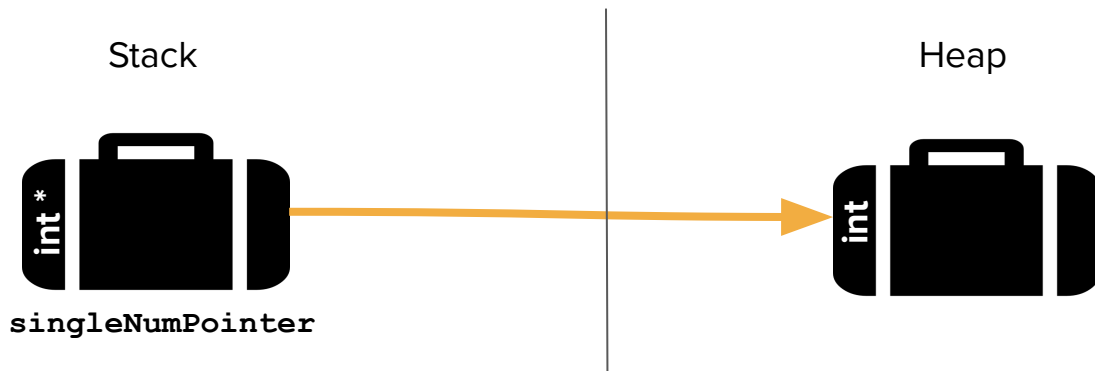
- Pointers are necessary to store the value generated by the **new** keyword (which is just a memory address on the heap, a.k.a. dynamic memory).
- But C++ also allows us to dynamically allocate space for just a single variable

```
int* singleNumPointer = 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, a.k.a. dynamic memory).
- But C++ also allows us to dynamically allocate space for just a single variable

```
int* singleNumPointer = 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, a.k.a. dynamic memory).
- But C++ also allows us to dynamically allocate space for just a single variable

```
int* singleNumPointer = new int;
```

- The usefulness of this will become apparent starting next week when we start our discussion of linked data structures.

Aside: Freeing dynamic memory with pointers

- If you allocate memory using the **new[]** operator (e.g. `new int[137]`), you have to free it using the **delete[]** operator.

```
delete[] ptr;
```

- If you allocate memory using the **new** operator (e.g. `new int`), you have to free it using the **delete** operator.

```
delete ptr;
```

- **Make sure to use the proper deletion operation.** Mixing these up leads to bad, undefined behavior!

Pointer Syntax (Part 4)

- To read or modify the variable that a pointer points to, we use the ***** (**asterisk**) operator to **dereference the pointer**.

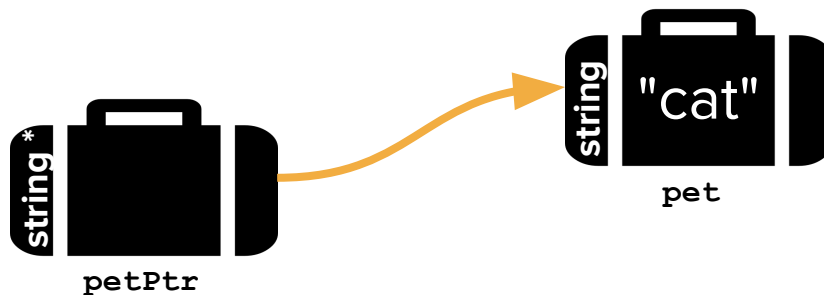
Pointer Syntax (Part 4)

- To read or modify the variable that a pointer points to, we use the * (**asterisk**) operator to **dereference the pointer**.
- Dereferencing a pointer involves following the arrow to the memory location at the end of the arrow **and** then reading or modifying the value stored there.

Pointer Syntax (Part 4)

- To read or modify the variable that a pointer points to, we use the ***** (asterisk) operator to **dereference the pointer**.
- Dereferencing a pointer involves following the arrow to the memory location at the end of the arrow and then reading or modifying the value stored there.

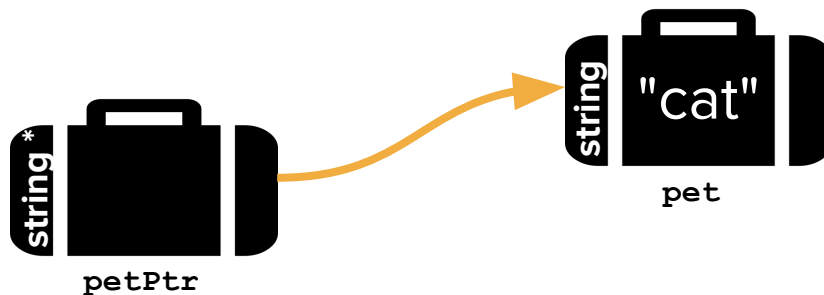
```
string* petPtr;  
string pet = "cat";  
petPtr = &pet;  
cout << *petPtr << endl;
```



Pointer Syntax (Part 4)

- To read or modify the variable that a pointer points to, we use the ***** (asterisk) operator to **dereference the pointer**.
- Dereferencing a pointer involves following the arrow to the memory location at the end of the arrow and then reading or modifying the value stored there.

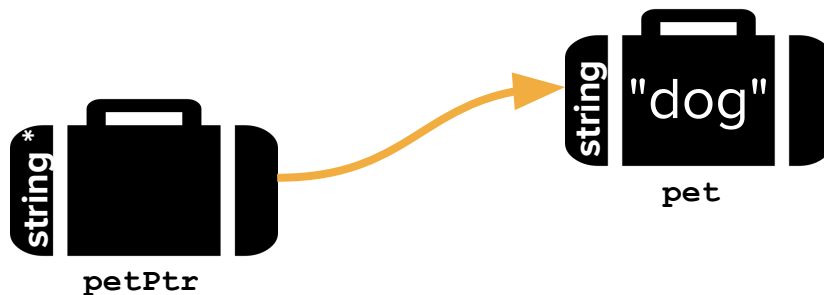
```
string* petPtr;  
string pet = "cat";  
petPtr = &pet;  
cout << *petPtr << endl;  
*petPtr = "dog";
```



Pointer Syntax (Part 4)

- To read or modify the variable that a pointer points to, we use the ***** (asterisk) operator to **dereference the pointer**.
- Dereferencing a pointer involves following the arrow to the memory location at the end of the arrow and then reading or modifying the value stored there.

```
string* petPtr;  
string pet = "cat";  
petPtr = &pet;  
cout << *petPtr << endl;  
*petPtr = "dog";
```



Pointer Tips

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 Tips (Part 1)

- What do we do if we want to declare/initialize a pointer variable but we don't yet have anything to point it at?

Pointer Tips (Part 1)

- What do we do if we want to declare/initialize a pointer variable but we don't yet have anything to point it at?

```
string* petPtr;
```

Pointer Tips (Part 1)

- What do we do if we want to declare/initialize a pointer variable but we don't yet have anything to point it at?

```
string* petPtr;
```



*This is dangerous
and unpredictable!*

Pointer Tips (Part 1)

- What do we do if we want to declare/initialize a pointer variable but we don't yet have anything to point it at?
- To ensure that we can tell if a pointer has a valid address or not, set your declared pointer equal to the special value **`nullptr`**, which means "no valid address."

Pointer Tips (Part 1)

- What do we do if we want to declare/initialize a pointer variable but we don't yet have anything to point it at?
- To ensure that we can tell if a pointer has a valid address or not, set your declared pointer equal to the special value `nullptr`, which means "no valid address."

```
string* petPtr = nullptr;
```



This allows for safe, consistent behavior. No arrow means no valid address.

Pointer Tips (Part 2)

- How can we tell if a pointer is safe to use (dereference)?

Pointer Tips (Part 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`**!

Pointer Tips (Part 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 printPetName(string* petPtr) {  
    if (petPtr != nullptr) {  
        cout << *petPtr << endl; // prints out the value pointed to by petPtr  
                                   // if it is not nullptr  
    } else {  
        cout << "petPtr is not valid!" << endl;  
    }  
}
```

Positively Practical Pointer Practice

Getting Practice with Pointers

- The little boxes (suitcases) and arrows that we draw to show the state of the memory are so, so important to understanding what is happening.
- **Always draw box and arrow diagrams when working with pointers!**
- As with most things, the best way to build an understanding of pointers is to practice, practice, practice!
 - The published code project for today has a bunch of pointer examples. We strongly recommend reading the code, predicting the output, and then running the code to confirm your predictions!
 - To finish off lecture today, we'll work through a couple of the examples together, building up diagrams as we go.

Pointer Practice (Part 1)

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



Pointer Practice (Part 1)

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

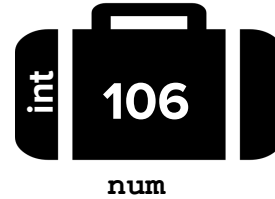
```
int num = 106;
```



Pointer Practice (Part 1)

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

```
int num = 106;
```

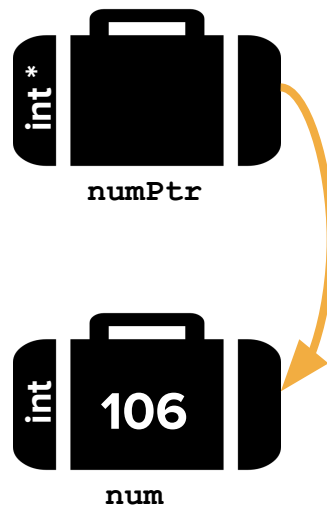


Pointer Practice (Part 1)

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

```
int num = 106;
```

```
numPtr = &num;
```



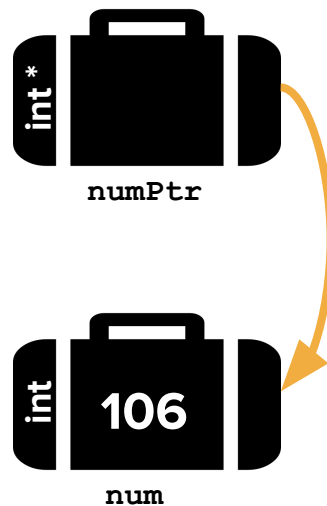
Pointer Practice (Part 1)

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

```
int num = 106;
```

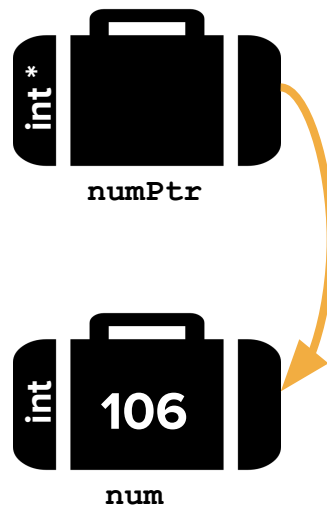
```
numPtr = &num;
```

*At this point, we say that
numPtr “points to” num*



Pointer Practice (Part 1)

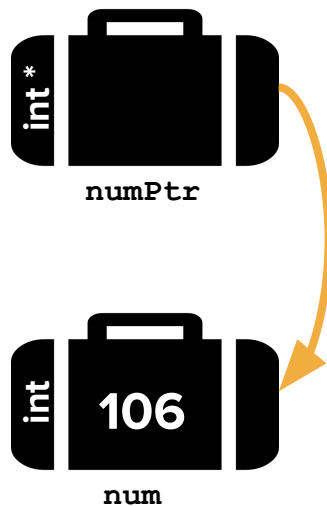
```
int* numPtr = nullptr;  
  
int num = 106;  
  
numPtr = &num;  
  
cout << *numPtr << endl;
```



Pointer Practice (Part 1)

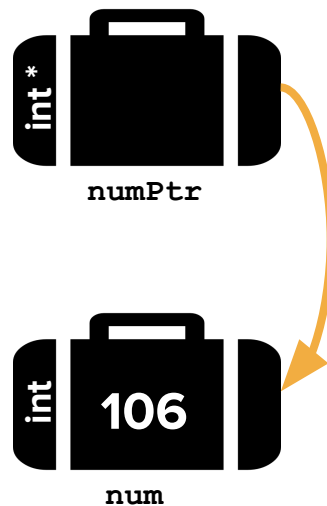
```
int* numPtr = nullptr;  
  
int num = 106;  
  
numPtr = &num;  
  
cout << *numPtr << endl; // 106
```

*By dereferencing **numPtr** we can print out the value of the variable that it points to.*



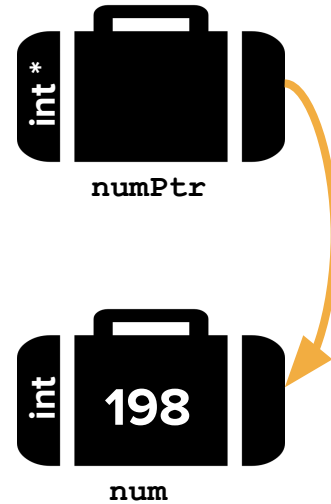
Pointer Practice (Part 1)

```
int* numPtr = nullptr;  
  
int num = 106;  
  
numPtr = &num;  
  
cout << *numPtr << endl;  
  
*numPtr = 198;
```



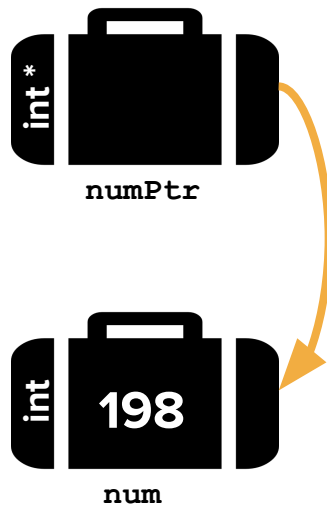
Pointer Practice (Part 1)

```
int* numPtr = nullptr;  
  
int num = 106;  
  
numPtr = &num;  
  
cout << *numPtr << end;  
  
*numPtr = 198;
```



Pointer Practice (Part 1)

```
int* numPtr = nullptr;  
  
int num = 106;  
  
numPtr = &num;  
  
cout << *numPtr << end;  
  
*numPtr = 198;
```



Dereferencing `numPtr` also allow us to modify the value of the variable/memory it points to.

What is a pointer?

A memory address!

Pointer Practice (Part 2)

- What is the output of the following code snippet?

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

pollev.com/cs106bpolls

What is the output of the code snippet?

"hello" gets printed

gets printed

The program throws an error because you're trying to dereference and print nullptr.

The program crashes.



Pointer Practice (Part 2)

- What is the output of the following code snippet?

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

Console



```
***  
*** 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. |
```

Pointer Practice (Part 2)

- What is the output of the following code snippet?

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



Pointer Practice (Part 2)

- What is the output of the following code snippet?

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



Pointer Practice (Part 2)

- What is the output of the following code snippet?

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

*When you dereference a `nullptr`,
you encounter a **segmentation
fault**, and the program crashes!*



Pointer Practice (Part 2)

- What is the output of the following code snippet?

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

Takeaway: Always use a nullptr check before dereferencing a pointer.



Pointer Practice (Part 3)

- What is the output of the following snippet of code?

```
string* strPtr1 = nullptr;  
string* strPtr2 = nullptr;
```

```
string s = "hello";  
strPtr1 = &s;  
strPtr2 = strPtr1;
```

```
*strPtr1 = "goodbye";
```

```
cout << *strPtr1 << " "  
      << *strPtr2 << endl;
```

Attendance ticket:

<https://tinyurl.com/pointerPractice>

Please don't send this link to students who are not here. It's on your honor!

Fun with Binky

A CS106A Throwback

- Some of you in this class may have taken CS106A with Nick Parlante, one of the intro CS lecturers here at Stanford.
- Back in 1999, he created a stop-motion claymation video starring a character named Binky that has been a staple of explaining pointers in intro CS classes at Stanford ever since.
- We've worked hard today and covered a lot of new material, so let's finish off by enjoying this fun throwback video...



`*x = 42;`

Summary

Memory and Pointers

- 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 especially useful as a tool to store the location of dynamically allocated memory (both arrays and individual elements) acquired on the heap with **new**.
- The **dereference operator** allows us to access and modify the memory pointed to by a pointer.

What's next?

Roadmap

Object-Oriented Programming

C++ basics

User/client

vectors + grids

stacks + queues

sets + maps

Implementation

arrays

dynamic memory management

linked data structures

real-world algorithms

Life after CS106B!

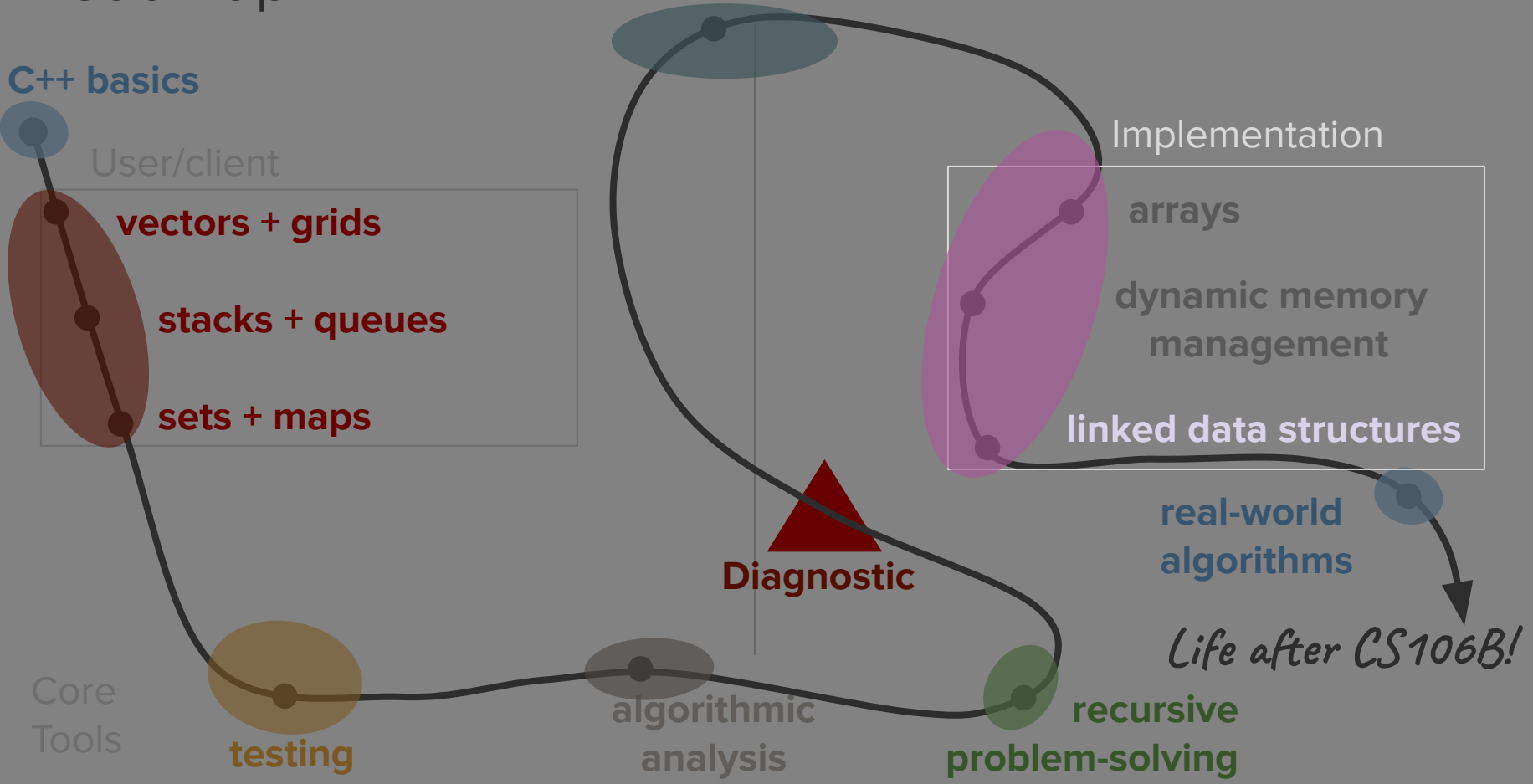
Core Tools

testing

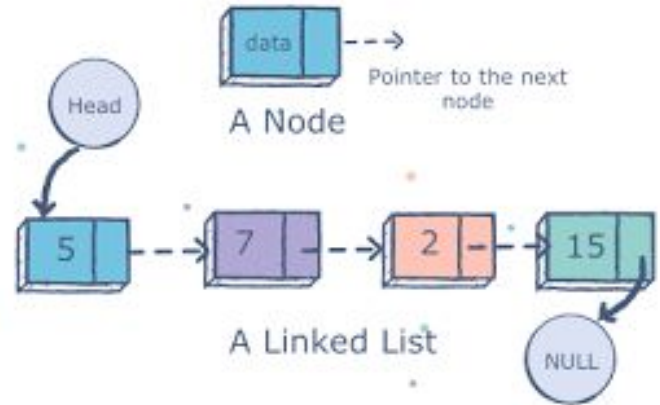
algorithmic analysis

recursive problem-solving

Diagnostic

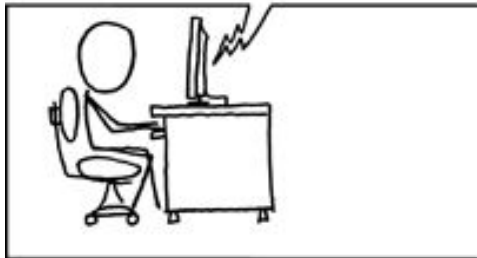


Introduction to Linked Lists



```
prev ->next = toDelete ->next;  
delete toDelete;
```

```
// if only forgetting were  
// this easy for me.
```



```
assert "It's going to be okay.";
```

