Programming Abstractions

CS106B

Cynthia Bailey Lee
Julie Zelenski

Topics:

- **Review: Pointers**
  - Including two quick new ideas:
    - dereference
    - null pointer
- **Main topic of today: Link Nodes**
  - What is a struct?
  - `LinkNode` struct
  - Chains of link nodes
  - `LinkNode` operations

REVIEW: Pointers

MEMORY ADDRESSES AND POINTERS

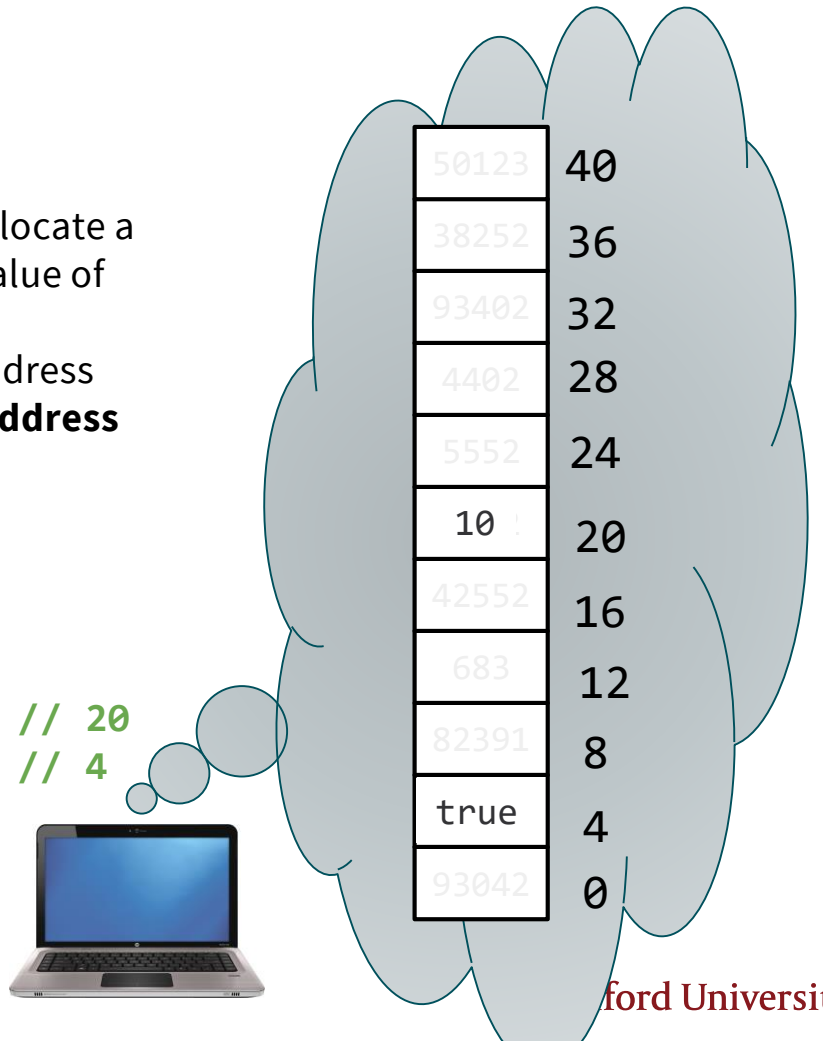
Address-of operator &

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 get the value of a variable's address using the & operator.

```
int candies = 10;
bool kitkat = true;
cout << &candies << endl;    // 20
cout << &kitkat << endl;     // 4
```



Pointer type

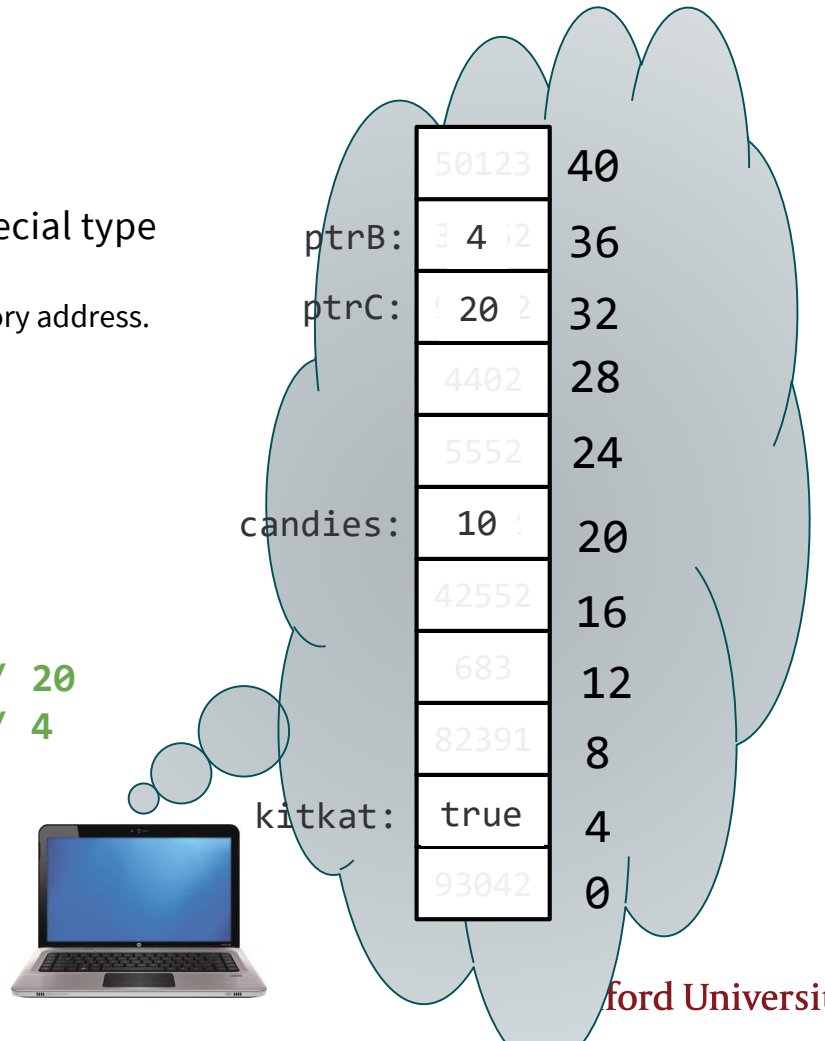
You can store memory addresses in a special type of variable called a **pointer**.

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

You can declare a pointer by writing *(The type of data it points at)**

- e.g. `int*`, `string*`

```
int candies = 10;
bool kitkat = true;
cout << &candies << endl;    // 20
cout << &kitkat << endl;     // 4
int* ptrC = &candies;
bool* ptrB = &kitkat;
```



Dereference and Null Pointer

TWO QUICK NEW IDEAS RELATED TO MEMORY ADDRESSES AND POINTERS

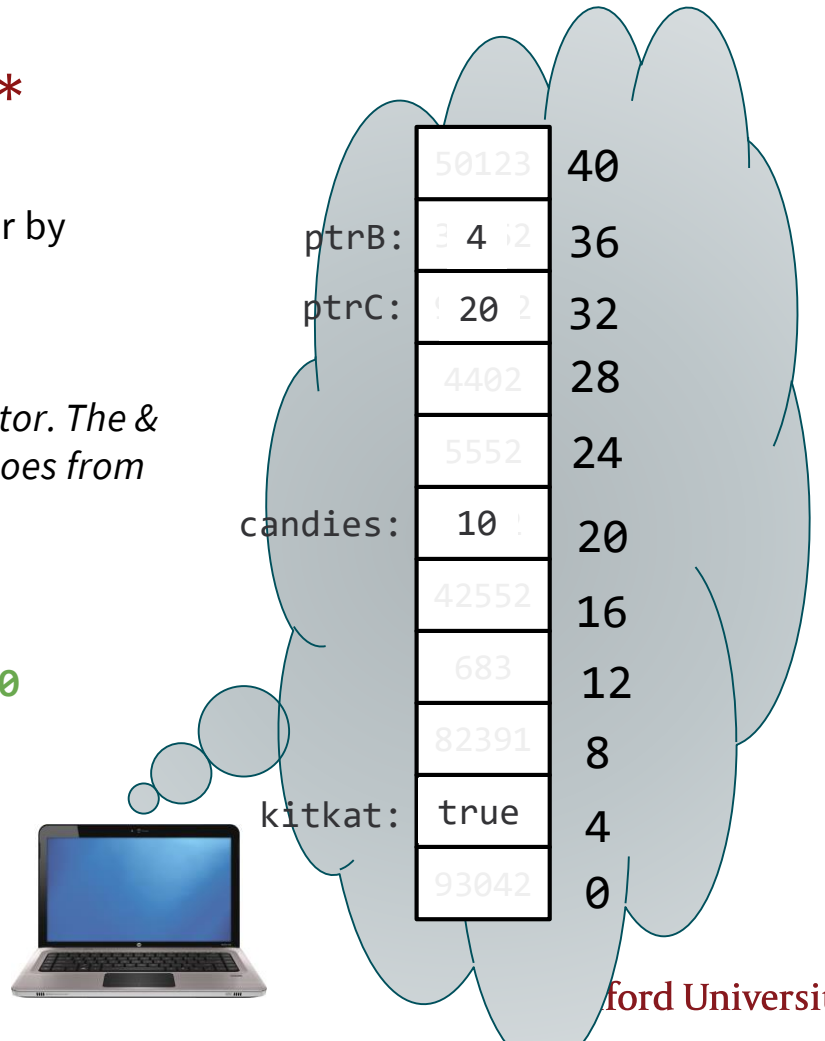
The dereference operator *

You can follow ("**dereference**") a pointer by writing
variable_name

*This is sort of the "inverse" of the & operator. The & goes from value to address, and the * goes from address to value.*

```
int candies = 10;
bool kitkat = true;
cout << &candies << endl;    // 20
cout << &kitkat << endl;     // 4
int* ptrC = &candies;
bool* ptrB = &kitkat;

cout<< ptrC << endl; // 20
cout<< *ptrC << endl; // 10
```



Null Pointer

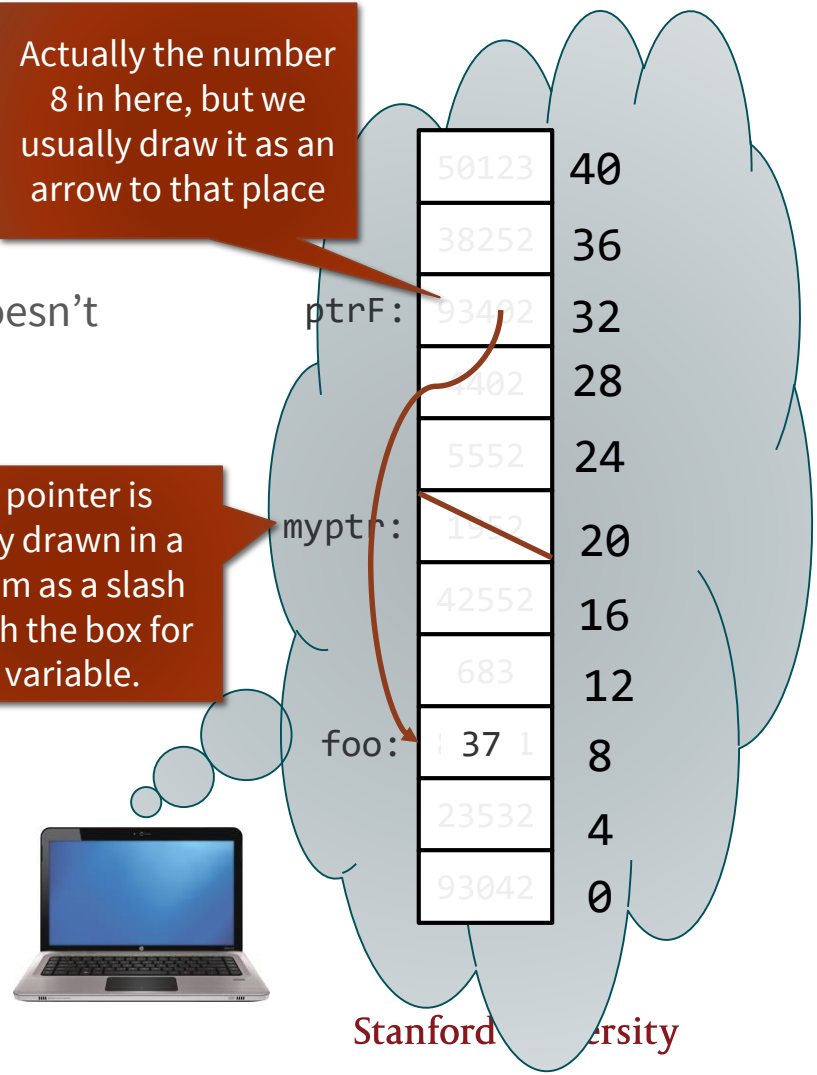
- When we want a variable with a pointer type "blank," we set it to be a "null pointer"
  - A special C++ built-in value that means it doesn't point to any valid memory address
  - Useful for initialization or sentinel value

- Example:

```
int  foo   = 37;
int* ptrF  = &foo;
int* myptr = nullptr;
…
if (myptr == nullptr) {
    cout << "haven't assigned an actual
            value to myptr yet!" << endl;
}
```

Actually the number 8 in here, but we usually draw it as an arrow to that place

Null pointer is usually drawn in a diagram as a slash through the box for the variable.

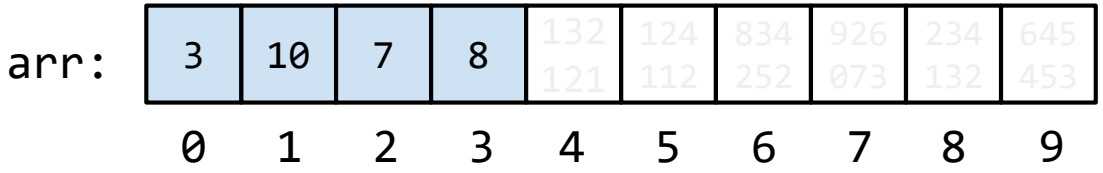


Array Performance

LIMITATIONS OF THE ARRAY, AND A MORE FLEXIBLE ALTERNATIVE

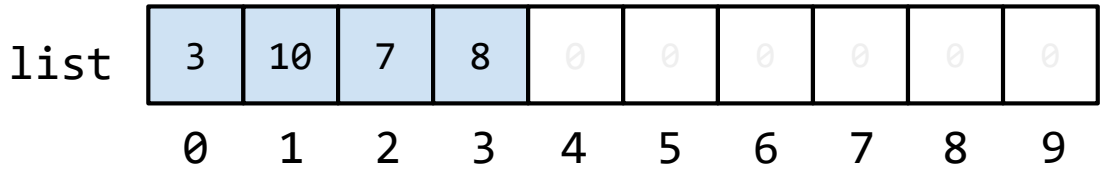
Arrays

What are arrays good at? What are arrays bad at?




EXIT

Array Performance



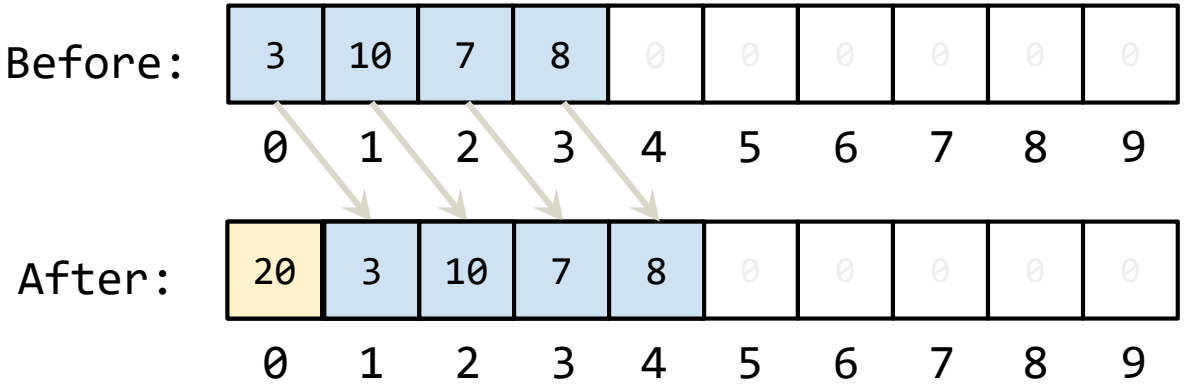
What are the most annoying operations on a tightly packed row of theater seats, or a tightly packed book shelf, etc?

Insertion - **O(n)**
Deletion - **O(n)**
Lookup (given index/memory address) - **O(1)**

Let's brainstorm ways to improve insertion and deletion....

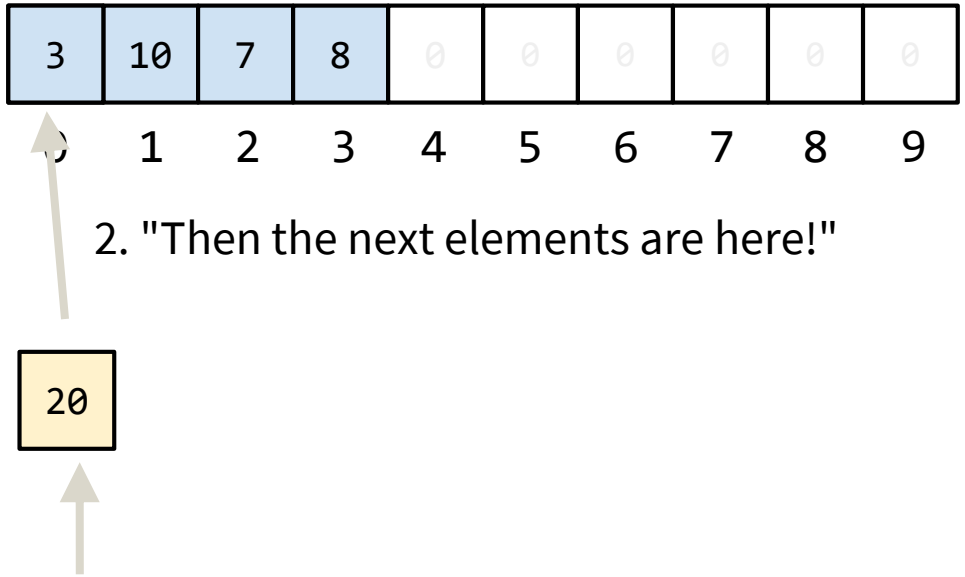
Add to front

What if we were trying to add an element "20" at index 0?



Add to front

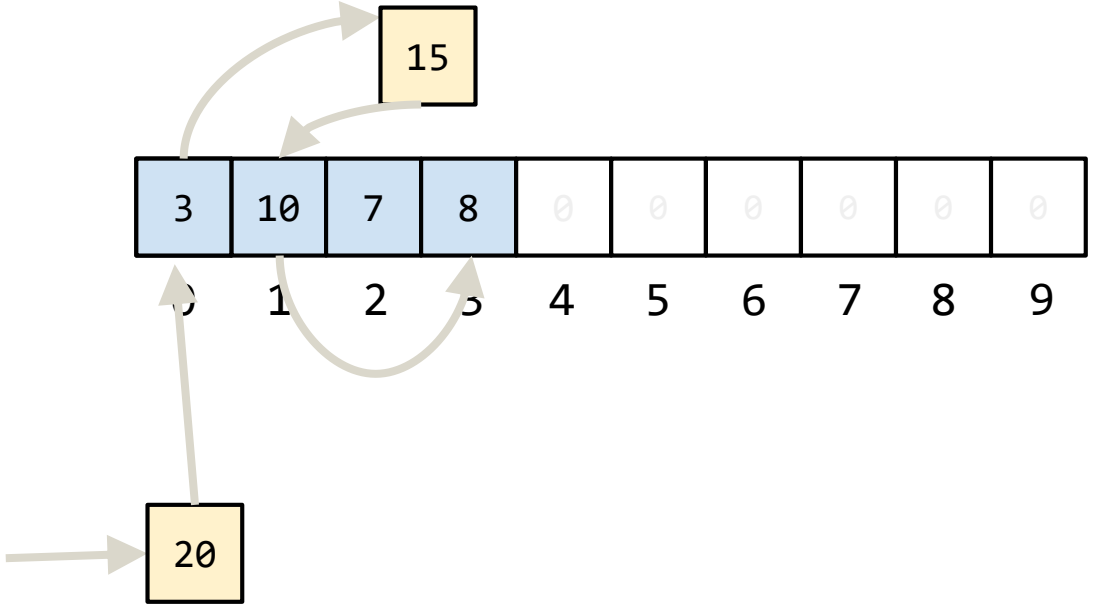
Wouldn't it be nice if we could just do something like:



More operations

Now we add 15 as a new 3rd element, and remove the 7:

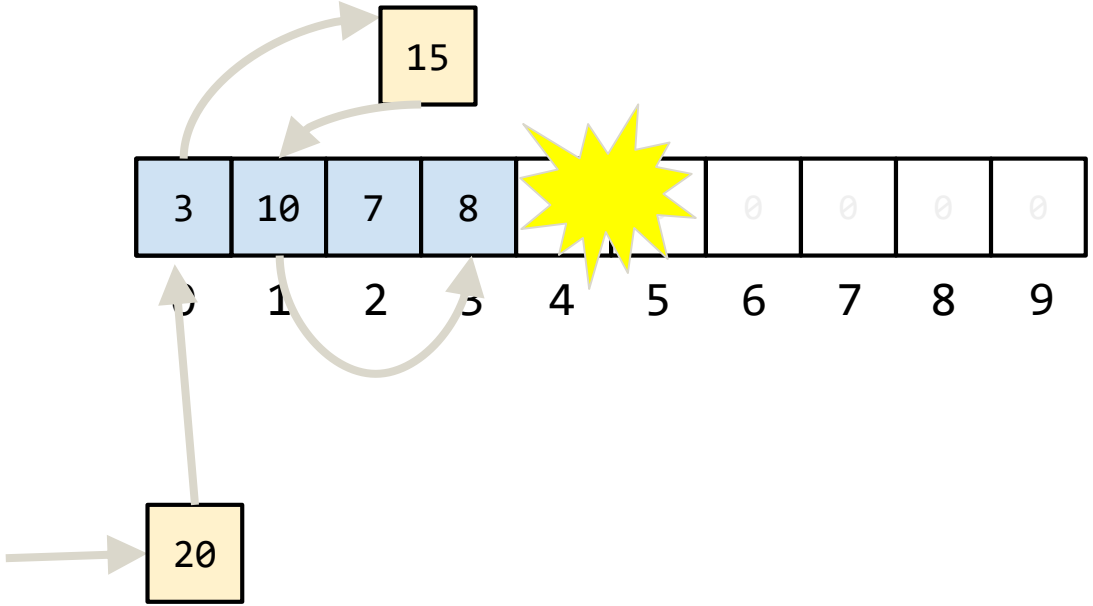
Arrows everywhere! (but no scooting over in those array buckets/seats, at least…)



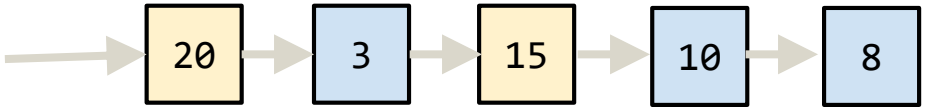
More operations

Now we add 15 as a new 3rd element, and remove the 7:

Arrows everywhere! (but no scooting over in those array buckets/seats, at least…)



This is a list of linked nodes!



- A list of linked nodes (or a linked list) is composed of interchangeable nodes
- Each element is stored separately from the others (vs contiguously in arrays)
- Elements are chained together to form a one-way sequence using pointers
- Edits are easier than an array in that no “scooting over” is needed!

Linked Nodes

A GREAT WAY TO EXERCISE YOUR POINTER UNDERSTANDING

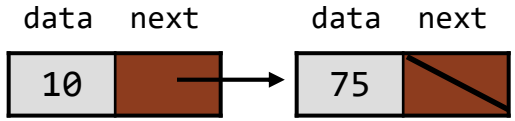
The `LinkNode` Struct

- To enable each bucket of the more flexible array alternative to both hold a value *and* tell you where to look for the next value, we need a `struct` with two fields:

```
struct LinkNode {
   int data;
   LinkNode* next;
};
```

  - `data`: the data being stored (what would be in the array)
  - `next`: a pointer to the next node `struct` in the sequence (or `nullptr` if this is the end of the sequence)

- The result is a chain that looks like this:



Wait, hold on, what's a struct??

C/C++ STRUCT TYPES

C/C++ `struct`: Like a lightweight class

```
struct LinkNode {
   int data;
   LinkNode* next;
};
```

- Like a class, but simpler—just a collection of some variables together into a new type
  - *A holdover from C, before the idea of objects (that combine variables and methods together)*
- **Example:** You can declare a variable of this type in your code, and use "." to access fields:

```
LinkNode node;
node.data = 20;
node.next = nullptr;
cout << "The data in the LinkNode is: " << node.data << endl;
```

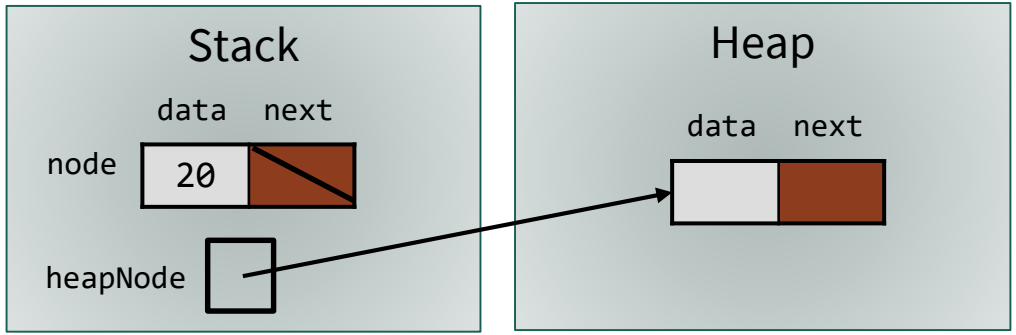
C/C++ `struct` and pointers

```
struct LinkNode {
   int data;
   LinkNode* next;
};
```

- Just like arrays or really any type of variable, you can put structs on the heap by calling "new"
- **Example:**

```
LinkNode node; // This LinkNode (both fields) is on the STACK
node.data = 20;
node.next = nullptr;
cout << "The data in the LinkNode is: " << node.data << endl;

LinkNode* heapNode = new LinkNode; // Both fields of this one are on the HEAP
// Now we want to set the data field to 6 and next field to nullptr how do we do that?
```



The -> dereference operator

```
struct LinkNode {
   int data;
   LinkNode* next;
};
```

- Just like arrays or really any type of variable, you can put structs on the heap by calling "new"
- **Example:**

```
LinkNode node; // This LinkNode (both fields) is on the STACK
node.data = 20;
node.next = nullptr;
cout << "The data in the LinkNode is: " << node.data << endl;

LinkNode* heapNode = new LinkNode; // Both fields of this one are on the HEAP
// Now we want to set the data field to 6 and next field to nullptr how do we do that?
(*heapNode).data = 6; // Dereference to follow pointer to struct, then access field
heapNode->data = 6;   // Since above syntax is clunky, we use this -> instead!
heapNode->next = nullptr;
cout << "The data in the LinkNode is: " << heapNode->data << endl;
```

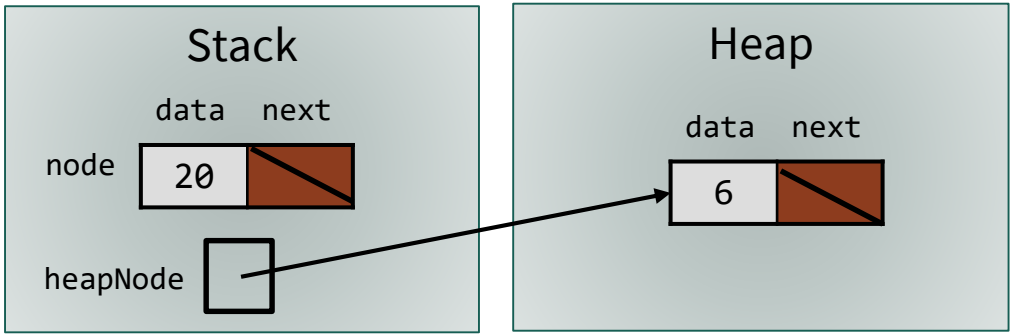
You should basically forget you ever saw `(*heapNode).data`! ☺
Remember `heapNode->data`, we will use it all the time!

Memory diagram: `struct` and pointers

```
struct LinkNode {
   int data;
   LinkNode* next;
};
```

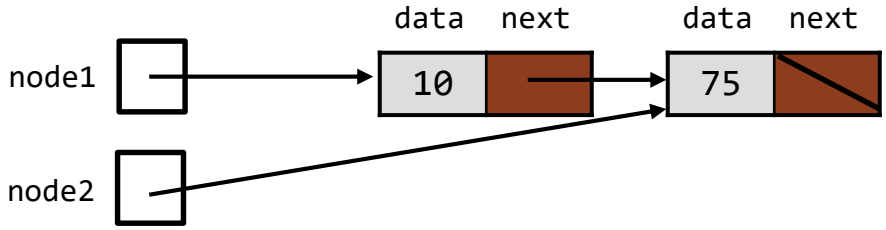
```
LinkNode node; // This LinkNode (both fields) is on the STACK
node.data = 20;
node.next = nullptr;

LinkNode* heapNode = new LinkNode; // Both fields of this one are on the HEAP
heapNode->data = 6;
heapNode->next = nullptr;
cout << "The data in the LinkNode is: " << heapNode->data << endl;
```



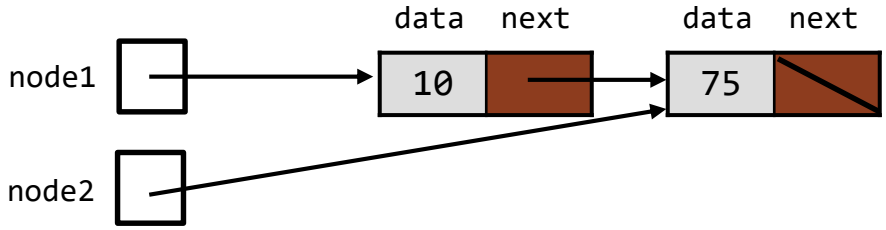
Ok, now back to Linked Nodes

Your Turn: finish the code to match the picture




```
LinkNode* node1 = new LinkNode;
node1->data = 10;
LinkNode* node2 = new LinkNode;
node2->data = 75; // YOUR TURN: complete the code to make picture
```

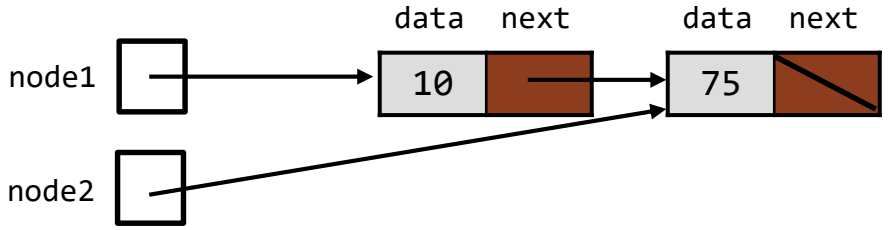
Your Turn: finish the code to match the picture




```
LinkNode* node1 = new LinkNode;
node1->data = 10;
LinkNode* node2 = new LinkNode;
node2->data = 75; // YOUR TURN: complete the code to make picture

node1->next = node2;     // needed to connect node1 and node2
node2->next = nullptr;   // needed to indicate no more nodes after this
```

Your Turn: finish the code to match the picture

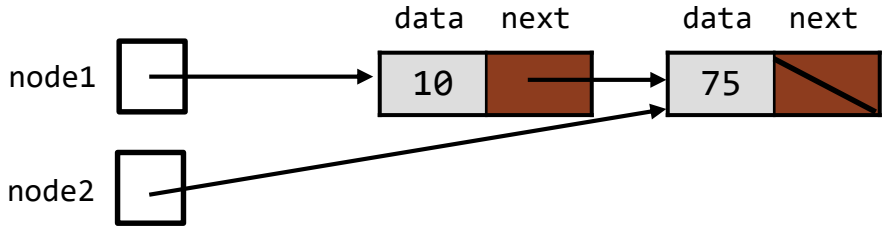



```
LinkNode* node1 = new LinkNode;
node1->data = 10;
LinkNode* node2 = new LinkNode;
node2->data = 75; // YOUR TURN: complete the code to make picture

node1->next = node2;     // needed to connect node1 and node2
node2->next = nullptr;   // needed to indicate no more nodes after this
```

IMPORTANT: ASSIGNMENT OPERATOR WITH POINTERS
When assigning one pointer to another, we are making the two pointers *point to the same destination*. We are *not* making the one on the right point to the one on the left as its destination.

Your Turn: finish the code to match the picture

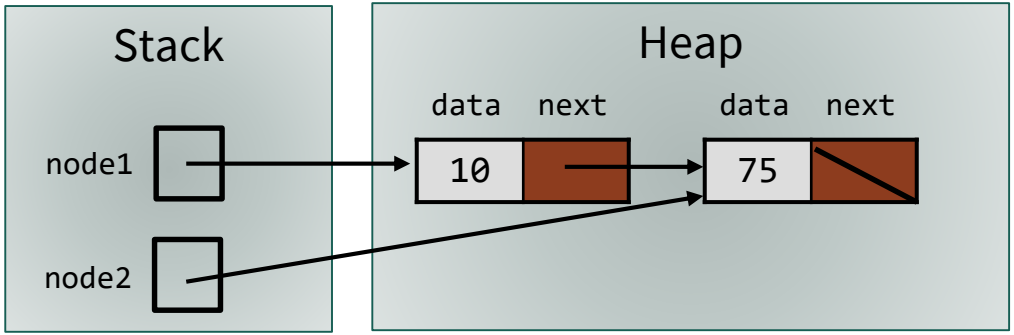



```
LinkNode* node1 = new LinkNode;
node1->data = 10;
LinkNode* node2 = new LinkNode;
node2->data = 75; // YOUR TURN: complete the code to make picture

node1->next = node2;      // needed to connect node1 and node2
node1->next->next = nullptr;   // alternate way edit node2!
```

Alternate solution! After node1 and node2 are joined, we don't really need the pointer variable named node2 anymore. We can modify node2's next field *through* node1!

Your Turn: finish the code to match the picture




```
LinkNode* node1 = new LinkNode;
node1->data = 10;
LinkNode* node2 = new LinkNode;
node2->data = 75;

node1->next = node2;
node1->next->next = nullptr;
node2 = nullptr;
```

Review/Reminder: the variables `node1` and `node2` are local variables, so they'll be stored in the **stack** part of memory. The nodes themselves will be stored in the **heap** part of memory, since we got them from `new`.

FIRST RULE OF LINKED NODE/LISTS CLUB:

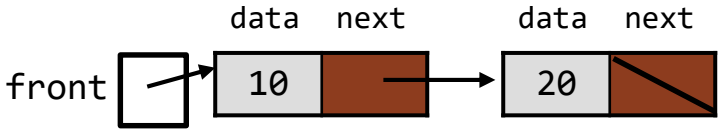
DRAW A PICTURE OF LINKED LISTS

Do no attempt to code linked nodes/lists without pictures!

List code example: Draw a picture!

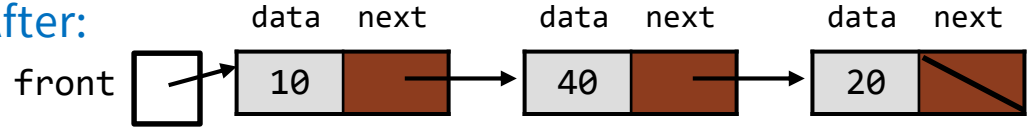
```
struct LinkNode {
  int data;
  LinkNode* next;
};
```

Before:

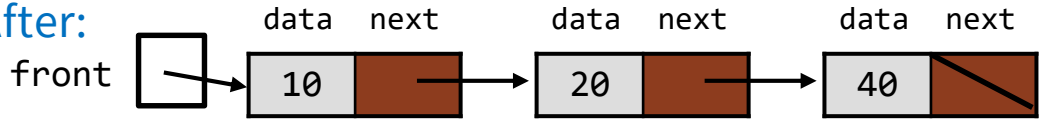



```
front->next->next = new LinkNode;
front->next->next->data = 40;
```

A. After:



B. After:



C. Using `next` that is `nullptr` gives an error

D. Other/none/more than one