Programming Abstractions

CS106B

Cynthia Lee
Topics:

- **Pointers**
  - What is a pointer?

- **Link Nodes**
  - LinkNode struct
  - Chains of link nodes
  - LinkNode operations
Pointers

Taking a deeper look at the syntax of that array on the heap
Memory is a giant array

`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**.
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.

```cpp
int candies = 10;
bool kitkat = true;
cout << &candies << endl; // 20
cout << &kitkat << endl;  // 0
```
Address-of operator &

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*`

```cpp
int candies = 10;
bool kitkat = true;
cout << &candies << endl; // 20
cout << &kitkat << endl; // 0
int* ptrC = &candies;
bool* ptrB = &kitkat;
```
Dereference operators * and ->

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

*variable_name

Remember that if what we find at the destination is a struct, we dereference AND access a field of the struct at once with the struct dereference operator ->

```cpp
int candies = 10;
bool kitkat = true;
cout << &candies << endl; // 20
cout << &kitkat << endl; // 0
int* ptrC = &candies;
bool* ptrB = &kitkat;

cout<< ptrC << endl; // 20
cout<< *ptrC << endl; // 10
```
Dereference operators * and ->

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

*variable_name

Remember that if what we find at the destination is a struct, we dereference AND access a field of the struct at once with the struct dereference operator ->

int candies = 10;
bool kitkat = true;
cout << &candies << endl; // 20
cout << &kitkat << endl; // 0
int* ptrC = &candies;
bool* ptrB = &kitkat;
int** ptr_to_ptrC = &ptrC;
cout<< ptrC << endl; // 20
cout<< *ptrC << endl; // 10
Linked Nodes

Another important application of pointers

We’ll start by looking at a limitation of the array
Arrays

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

arr:

```
0  1  2  3  4  5  6  7  8  9
3 10  7  8 132 124 834 926 234 645
121 112 252 073 132 453
```
Memory is a giant array...

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?

Before:

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>10</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

After:

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>3</td>
<td>10</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Add to front

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

1. "Start here instead!"
2. "Then the next elements are here!"
More operations

Now we add to the front again, and remove the 7 at index 2:
Arrows everywhere! (but no scooting over in those array buckets/seats, at least…)
More operations

Now we add to the front again, and remove the 7 at index 2:
Arrows everywhere! (but no scooting over in those array buckets/seats, at least…)

```
 0 1 2 3 4 5 6 7 8 9
```

```
3 10 7 8
```

15

20

Stanford University
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
Linked Node

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

- We can chain these together in memory:

```
LinkNode* node1 = new LinkNode;  // YOUR TURN: complete the code to make picture
node1->data = 75;
node1->next = NULL;
LinkNode* node = new LinkNode;
node->data = 10;
node->next = node1; // correct answer
```
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!

Before:

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

A. After:

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

B. After:

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

C. Using “next” that is NULL gives error

D. Other/none/more than one
List code example: Draw a picture!

Before: front

Write code that will put these in the reverse order.