CS 106X, Lecture 18
Arrays and Trees

reading:
Programming Abstractions in C++, Chapters 11.3, 16.1
Plan For Today

• Arrays
• Announcements
• Trees
Learning Goals

- Understand the difference between Vectors and arrays
- Understand how trees use pointers to represent data in useful ways
Plan For Today

- Arrays
- Announcements
- Trees
int myArray[6];

(type name length)

(2 12 15 6 -1 4)

(on the stack)
Arrays

```
int *myArray = new int[6];
```

(on the heap)
void makeArrays() {
    int x = 42;
    int y = 61;
    int a1[3];
    int* a2 = new int[3];
    ...
}

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>42</td>
<td>-5</td>
<td>17</td>
</tr>
</tbody>
</table>
void makeArrays() {
    int x = 42;
    int y = 61;
    int a1[3];
    int* a2 = new int[3];
    delete[] a2;
    ...
}

```
<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>42</td>
<td>-5</td>
<td>17</td>
</tr>
</tbody>
</table>
```
Heap Initialization

```cpp
int *myArray = new int[6];
```
Heap Initialization

int *myArray = new int[6]();

slower!
Arrays As Parameters

```c
void myFunction(int arr[]) {
    ...
}

int main() {
    int myArray[6];
    myFunction(myArray);
    return 0;
}
```
C++ arrays have no members!
  - No size field
  - No helpful methods like `indexOf`

```c++
int arraySize = 6;
int myArray[arraySize];

for (int i = 0; i < arraySize; i++) {
  myArray[i] = ...;
}
```
Array Length

• C++ arrays have no members!
  – No size field
  – No helpful methods like `indexOf`

```cpp
void myFunction(int arr[], int size) {
    
}

int main() {
    int arraySize = ... 
    int myArray[arraySize]; 
    myFunction(myArray, arraySize);
}
How Vector/Stack works

- Vectors and Stacks contain internal **arrays** to store elements.
- The array is created with some extra space (it is an “unfilled array”), and is replaced with a larger array when space runs out.

```cpp
Vector<int> v;
v.add(42);
v.add(-5);
v.add(17);

Stack<int> s;
s.push(42);
s.push(-5);
s.push(17);
```

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>42</td>
<td>-5</td>
<td>17</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>size</td>
<td>3</td>
<td>capacity</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Outgrowing Our Shell

Image from Freshwater and Marine Image Bank: https://commons.wikimedia.org/wiki/File:FMIB_52669_Pagurus_bernhardus_the_hermit-crab-retouched.svg
Let's implement a stack using an unfilled array.

- We'll call it **ArrayStack**. It will be very similar to the C++ Stack.

- Its behavior:
  - `push(value)`
  - `pop()`
  - `peek()`
  - `isEmpty()`
  - `toString()`
  - `operator <<`

- The stack's size will be the number of elements added to it so far.
  - The actual array length ("capacity") in the object may be larger. We'll start with an array of **length 10** by default.
class ArrayStack {
  public:
    ...

  private:
    int elements[INITIAL_CAPACITY];
    ...
};
class ArrayStack {
public:
...

private:
    int *elements;
...
};
ArrayStack::ArrayStack() {
    elements = new int[INITIAL_CAPACITY]();
    ...
}
Plan For Today

- Arrays
- Announcements
- Trees
Announcements

• Congratulations on finishing the midterm!
• Midterms will be graded and handed back by Monday
• Section Leading Application due tonight at midnight! See cs198.stanford.edu
Announcements

- WiCS Casual Dinner **Tues. 11/6 5-7 in Gates 403!** Join me!

Stanford Women in Computer Science

**CASUAL DINNER**

Tuesday, November 6th from 5-7 PM at Gates 403

Come have dinner with CS students and faculty. Everyone is welcome, especially students just starting out in CS!
Plan For Today

• Arrays
• Announcements
• Trees
Trees
Trees

Random expansion from sentence.txt grammar for symbol "<s>"
Trees

Trees
// Example student solution

function run() {
  // move then loop
  move();

  // the condition is fixed
  while (notFinished()) {
    if (isPathClear()) {
      move();
    } else {
      turnLeft();
    }
  }

  // redundant
  move();
}

This is a figure in an academic paper written by a recent CS106 student!
Trees

- **Tree**: a directed, acyclic structure of linked nodes
  - *Directed*: has one-way links between nodes
  - *Acyclic*: No path wraps back around to the same node twice (aka only one unique path from a node to any other node).

- Recursive!
  - A tree is either empty, or
  - A root node that contains **data** and zero or more **subtrees**.
Trees
Trees
Trees
Trees

branches
Trees
Trees
Trees

siblings
Trees

Level 1

Level 2

Level 3

Level 4
Trees

subtree
Height = 4 -> longest path from the root to any node, in # nodes
TreeNode

// A TreeNode is one node in a binary tree of integers.
struct TreeNode {
    int data;       // data stored at this node
    TreeNode* left; // pointer to left subtree
    TreeNode* right; // pointer to right subtree

    // Constructs a node with the given data and links.
    TreeNode(int data, TreeNode* left, TreeNode* right) {
        this->data = data;
        this->left = left;
        this->right = right;
    }

    bool isLeaf() const {
        return left == nullptr && right == nullptr;
    }
};
A basic tree node object stores data and pointers to left/right
- Multiple nodes can be linked together into a larger tree
Common Tree Operations

- print
- height
- size
- contains
- deleteTree
- Write a function named `print` that accepts a tree node pointer as its parameter and prints the elements of that tree, one per line.
  - A node's left subtree should be printed before it, and its right subtree should be printed after it.

  - Example: `print(root);`

```
29
41
6
17
81
9
40
```
void print(TreeNode* node) {
    // (base case is implicitly to do nothing on NULL)
    if (node != nullptr) {
        // recursive case: print left, center, right
        print(node->left);
        cout << node->data << endl;
        print(node->right);
    }
}
• **traversal**: An examination of the elements of a tree.
  – A pattern used in many tree algorithms and methods

• Common orderings for traversals:
  – **pre-order**: process root node, then its left/right subtrees
  – **in-order**: process left subtree, then root node, then right
  – **post-order**: process left/right subtrees, then root node
Traversals

- pre-order: 17 41 29 6 9 81 40
- in-order: 29 41 6 17 81 9 40
- post-order: 29 6 41 81 40 9 17
Recap

• Arrays
• Announcements
• Trees

Next time: more trees!