ARRAYS OPEN A LOT OF DOORS

ADAM SWEENEY
CS 211

INTRODUCTION

- No new syntax
- Having arrays in our tool belt opens a lot of new possibilities

AGENDA

- Partially filled arrays
- Sorting arrays
- Searching arrays
- Multi-dimensional arrays

PARTIALLY FILLED ARRAYS

HALF FULL, OR HALF EMPTY?

- There is no rule that arrays must always be full
- We also want to handle our data, even if data set sizes fluctuate
- We can declare large arrays

LARGE ARRAYS

- Advantages
  - A bit tougher to go out of bounds
  - Lets us “choose” the size of our data set
- Disadvantages
  - Wastes memory
    - Arrays are always fully allocated, any unused elements are “wasted” space
  - More maintenance (*read: code*) needed to utilize
- Verdict?
  - Generally, a partially filled array is a great idea
  - Just don’t go too crazy on the capacity

USING PARTIALLY FILLED ARRAYS

- Up to now, size and capacity have been interchangeable
- We now must distinguish between them
  - Size refers to the number of 'active' elements in an array
  - Capacity refers to the maximum number of elements that can be held
- We will care about the size and capacity when adding elements to the array
- We will only care about the size for everything else

SORTING ARRAYS

SORTING IS A BIG TOPIC

- There are literal volumes written on sorting techniques
- Over half of CS 560 was spent on various sorting algorithms (for me)
- Some links to help visualize different sorting techniques
  - http://panthema.net/2013/sound-of-sorting/
  - https://www.toptal.com/developers/sorting-algorithms/
- We will cover one simple method

SELECTION SORT

- Look at every element of the array and find the first value
- Swap the element with the smallest value with element 0
- Repeat the process, minus element 0
  - Search the array starting from element 1
  - Swap the next minimum with element 1
- And so on until the array is sorted

SEARCHING ARRAYS

A COUPLE METHODS

- Brute force
  - Check every element to see if the value matches
- Binary search
  - Requires a sorted array
  - Check the middle element, if it’s a match, you’re done
  - If not, compare the value of the middle element against the value you’re searching for
  - Choose the half of the array that should contain your value, and check its midpoint
  - And so on, until you find the value

COMPARISON OF SEARCHES

- Assuming an array with ‘n’ elements
- Brute force
  - At most, you must make n comparisons
  - On average n/2 comparisons
- Binary search
  - At most, $\log_2 n$ comparisons
- Example, an array with 10,000 elements
  - Brute force: 10,000 at worst, 5,000 average comparisons
  - Binary search: 14 comparisons at worst

MULTI-DIMENSIONAL ARRAYS

2D+

- We can declare arrays with as many dimensions as needed
  - Might get hard to visualize beyond 3
- C++ doesn’t understand anything beyond 1D
- A 2D (or higher) array is seen by the compiler as an array of arrays

DECLARATIONS

- C-array
  - `int matrix[5][5];`
    - Simple enough
    - [ROWS][COLUMNS]
- std::array
  - `std::array<std::array<int, 5>, 5> matrix;`
    - A little messy looking
    - std::array<type, COLUMNS>, ROWS> name;
- When using either array in code, the first index is the number, the second is the column

PASSING TO FUNCTIONS

- C-array
  - `void foo_func(int matrix[][5], int size);`
    - All dimensions beyond the first MUST be specified
- std::array
  - `void foo_func(std::array<std::array<int, 5>, 5> matrix);`
    - That never gets easier to type