**Lecture 13 - Chapter 6: Arrays – Wed Oct 4 or Thurs Oct 5**

**Announcements**

Reading:

* Chapter 6

Assignments:

* Graded: Assignment #4
* Due: Assignment #5
* Assign: Assignment #6 - due on **Oct 18** (MW class) or **Oct 19** (TR class) **(no late assignments accepted)**

**Today’s Goals**

1. Assignment #4 feedback
2. Multi-Dimensional Arrays
3. Sorting Arrays

**Assignment #4 (Gas Pump) Feedback**

* Missing pseudocode
  + Lose 15 pts for not including
* Whitespace
  + Add it to code to increase readability
* Constants
  + Lots of people not using constants
  + Start looking for places where constants make sense and create them
* Use of break in if statements – remember not to do this type of unstructured coding
* Not using while loops when user enters invalid values
  + Like for the discount card, user should enter a ‘Y’ or ‘N’
  + If they enter anything else should handle this in a loop – loop until input is valid

**printf** ("Do you have a discount card? (Y/N) ");

**scanf** (" %c", &yesNoInput);

// Validate discount card input

**while** (**toupper**(yesNoInput) != 'Y' && **toupper**(yesNoInput) != 'N' && **toupper**(yesNoInput) != 'Q') {

**printf** ("Invalid input, do you have a discount card? (Y/N) ");

**scanf** (" %c", &yesNoInput);

}

* Not using a for loop to simulate the pumping of gas
* People forgot to put the space before the %c so had issues
* Using ASCII codes in if statements
  + If (yesNoInput == 89) don’t do this!
  + If (yesNoInout == ‘Y’) do this!!

**Today’s Terminology**

**Terminology**

* Array
  + A data structure used to store a collection of values that are all the same type
* Index
  + Refers to a specific element within an array
  + Must be an integer or integer expression
  + The **first element** in an array is at index 0, the second element is at index 1, etc.
* Indexed Variable
  + Used to reference each element in an array
* Array Initializer
  + A statement where you define and initialize an array
* Out of Bounds Error
  + Attempting to access an element with an index outside the range of the array is a runtime error
* Off by One Error
  + Mistakenly referencing the first element in an array with the index 1, not index 0
* Linear Search
  + Searching an array for a specific key value by starting at beginning and comparing the key with each element sequentially.
* Binary Search
  + Searching a pre-sorted array by splitting the array in half on each comparison.
* Bubble Sort
  + Combination of searching and sorting. During each pass, the largest element is moved to its proper location in the array.
* Big O Notation
  + A way to describe the performance of an algorithm

**Multi-Dimensional Arrays**

**Purpose**

* We’ve seen that an array is a container that provides a way to store more than one value
* Arrays can have more than one dimension!
* One-dimensional arrays – model linear collections of elements
* Two-dimensional arrays – model a matrix or a table

**Defining Two-Dimensional Arrays**

* General form

**datatype arrayName [row][column];**

Any type (int, float, double, etc.) The 2 dimensions of the array

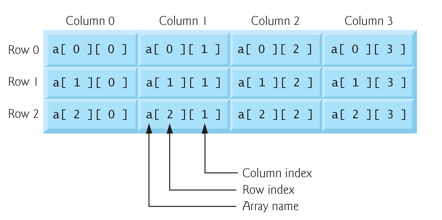
All elements within the array will be this type

**Example**

* Define an array that models a table with 3 rows and 4 columns

**int** a[3][4];

* To access individual elements within a 2-dimensional array use "array index" notation with 2 dimensions
  + arrayName [row][column]



* Remember arrays are zero based
  + First element in array - table[0][0]
  + Last element in array - table[2][3]
* Nested for loops are used with a 2-dimensional array

// Display the values in the 3x4 array

**for** (size\_t row = 0; row < 3; row ++){

**for** (size\_t column = 0; column < 4; column++) {

**printf** ("Table[%d][%d] = %d\n", row, column, table[row][column]);

}

**puts** ("");

}

* Remember array **default** values, like other types, are garbage values NOT zero!
* Should always initialize arrays with an appropriate default values
  + Example using initializer when defining the array

// Array initializer – define and initialize in one statement

// This table has 3 rows and 4 columns = 12 elements

**int** table[3][4] = {{3,2,5,2}, {34,23,1,45}, {44,81,7,45}};

Initialize row #0 Initialize row #1 Initialize row #2

* If there are fewer initializers than elements, the remaining elements are initialized to zero.

// You could write it this way as well but not as clear!

**int** table[3][4] = {3, 2, 5, 2, 34 23, 1, 45, 44, 81, 7, 45};

**How Table Array Will Look**

Table[0][0] = 3

Table[0][1] = 2

Table[0][2] = 5

Table[0][3] = 2

Table[1][0] = 34

Table[1][1] = 23

Table[1][2] = 1

Table[1][3] = 45

Table[2][0] = 44

Table[2][1] = 81

Table[2][2] = 7

Table[2][3] = 45

// Fast way to initialize the 12 elements to zero

**int** table[3][4] = {0};

// In this case not enough initializers so set remaining elements to 0

**int** table3[3][4] = {{3,2,5,2}, {34,23}, {1}};

**How Table Array Will Look**

Table[0][0] = 3

Table[0][1] = 2

Table[0][2] = 5

Table[0][3] = 2

Table[1][0] = 34

Table[1][1] = 23

Table[1][2] = 0

Table[1][3] = 0

Table[2][0] = 1

Table[2][1] = 0

Table[2][2] = 0

Table[2][3] = 0

* With one-dimensional arrays
  + If you leave off the array size, the number of initializers determines the array’s size

**int** numberList[5] = {2, 4, 6}; // Values at index 3 and 4 are zero!

* With two-dimensional arrays
  + If you leave off both dimensions – does not work
  + Must specify one dimension – complier needs it to determine memory locations

// YOU CANNOT DO THIS! First index not required but subsequent indices are

**int** table[][] = {{3,2,5,2}, {34,23,1,45}, {44,81,7,45}};

// This works

**int** table[][COLUMNS] = {{3,2,5,2}, {34,23,1,45}, {44,81,7,45}};

**Examples of Processing Arrays**

// Use these constants

**#define** ROWS 3

**#define** COLUMNS 4

// Use this array

**int** table[3][4];

* Initializing array with input values

**for** (size\_t row = 0; row < ROWS; row++) {

**for** (size\_t column = 0; column < COLUMNS; column++) {

**printf** ("Enter value for table[%d][%d]\n", row, column);

**scanf** ("%d", &table[row][column]);

}

}

* Displaying array

**for** (size\_t row = 0; row < ROWS; row ++){

**for** (size\_t column = 0; column < COLUMNS; column++) {

**printf** ("Table[%d][%d] = %d\n", row, column, table[row][column]);

}

}

* Summing all elements

**int** tableSum = 0;

**for** (size\_t row = 0; row < ROWS; row++) {

**for** (size\_t column = 0; column < COLUMNS; column++) {

tableSum +=table3[row][column];

}

}

**printf** ("Sum of all elements = %d\n", tableSum);

* Review Figure 6.22

**Sorting Arrays**

**Sorting**

* Organizing values into ascending or descending order
* In computer programming a common activity is to write code to sort an array
* There are lots of sorting algorithms
  + Selection sort
  + Bubble sort
  + Quick sort
  + Merge sort
  + etc.

**Bubble Sort**

* Only sort we will look at – bubble sort (sinking sort)
* Smallest values bubble to top, largest values sink to bottom
* Process
  + Compare successive neighboring pairs
  + If the pair is not in order, its values are swapped
  + If the pair is in order, the values remain unchanged
  + Make several passes through array until sorted

**Notes**

* Easy algorithm to code but slow
* Not suitable for large arrays
* **On each pass**, successive pairs of elements are compared
* On 1st pass – largest value is guaranteed to sink to bottom
* On 2nd pass – second largest is guaranteed to sink to bottom, etc.
* Performance Big O Notation
* Best case– List already sorted
  + One pass is all that is needed
  + Since # comparisons is (n -1) = **O(n)**
* Average and worst-case complexity are **O(n2)** 
  + In worse case you need to make n-1 passes!
  + 1st pass – compare n-1 pairs
  + 2nd pass – compare n-2 pairs

…

* + (n-1)th pass – compare 1 pair (no further processing required)
  + 5 elements could take:
    - 1st pass - (n-1) = 4 comparisons
    - 2nd pass - (n-2) = 3 comparisons – no need to compare last element
    - 3rd pass - (n-3) = 2 comparisons - no need to compare last two elements
    - 4th pass - (n-4) = 1 comparisons

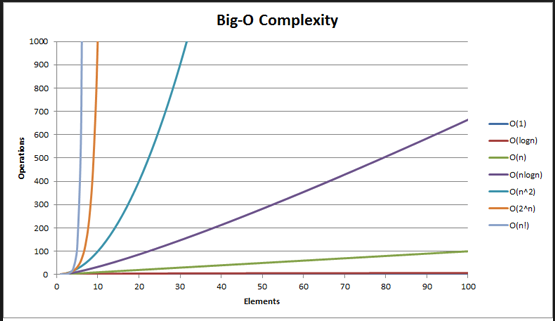
Total number of comparisons: (n-1) + (n-2) + … + 2 + 1 = 4 + 3+ 2 + 1 = 10

This is not 52 = 25?

No, but as **n** grows the impact becomes more

and more and it tends toward **n2**

* O(n2)
  + Performance directly proportional to the square of the number of elements in array



**Visually**

* 1st pass – watch largest number sink to bottom

int array[6] = {2, 9, 5, 4, 8, 1];

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 9 | 5 | 4 | 8 | 1 |  | Compare array[0] & array[1] – 2 & 9 are already sorted – do nothing |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 9 | 5 | 4 | 8 | 1 |  | Compare array[1] & array[2] – 9 & 5 are NOT sorted – swap |

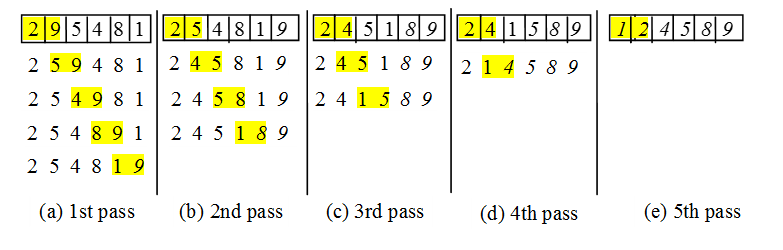
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 5 | 9 | 4 | 8 | 1 |  | Compare array[2] & array[3] – 9 & 4 are NOT sorted – swap |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 5 | 4 | 9 | 8 | 1 |  | Compare array[3] & array[4] – 9 & 8 are NOT sorted – swap |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 5 | 4 | 8 | 9 | 1 |  | Compare array[4 ]& array[5] – 9 & 1 are NOT sorted – swap |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 5 | 4 | 8 | 1 | 9 |  | First pass complete! Largest value in correct location. **(6 - 1 = 5 comparisons)** |

* All 5 passes



**Code for Bubble Sort**

* Pseudocode – 1st draft

If array[0] > array[1]

Then swap array[0] and array[1]

If array[1] > array[2]

Then swap array [1] and array[2] Need to do this for (size-1) pairs

… Need 5 comparisons

If array[4] > array[5]

Then swap array[4] and array[5]

* Pseudocode – 2nd draft

If array[0] > array[1]

temp = array[0]

array[0] = array[1]

array[1] = temp

Lots of repeating so need a loop!

if array[1] > array[2]

temp = array[1]

array[1] = array[2]

array[2] = temp

…

…

if array[4] > array[5]

temp = array[4]

array[4] = array[5]

array[5] = temp

* Pseudocode – 3rd draft Need to make (size-1) = 5 passes through array

Can do 0 to 5 on loop counter or 1 to 6

for pass = 1 to size

for each pair of elements Need (size – 1) pair comparisons each pass

if array[i] > array[i+1] to compare adjacent elements

temp = array[i]

array[i] = array[i+1]

array[i+1] = temp

end for each pair of elements

end for pass

* C Code – Figure 6.15 and Figure 6.16

