**Lecture 12 - Chapter 6: Arrays – Mon Oct 2 or Tues Oct 3**

**Announcements**

Reading:

* Chapter 6

Assignments:

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

**Today’s Goals**

1. Arrays cont.
2. Character Arrays
3. Static Local Arrays
4. Passing Arrays to Functions

**Today’s Terminology 5 min**

**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 NOT runtime error in C
* 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

**Arrays**

**Example**

* Define an array that holds 5 integers

// Create an array that can store 5 integers

**int** numberList[5];

0 1 2 3 4

* Each element in the array is an **int**
* The size of the array cannot be changed
* The variable **numberList** is a reference to the entire array

**Accessing Array Elements**

* To access the individual elements within an array use "array index" notation
  + arrayNameVariable [index]

numberList[0]

Array access is

through an index

**for** (size\_t i = 0; i < 5; i++) {

numberList[i] = 0;

}

* Arrays are zero based
  + First element in array - numberList[0]
  + Last element in array - numberList[4]
* For loops generally used with arrays since we know how many times the loop will occur
  + Example: assign the numbers 0 to 4 to numberList
  + Note: The type **size\_t** should be used for an array index (represents an unsigned integral type)

// Assign the numbers 0 to 4 to numberList array

**for** (size\_t i = 0; i < 5; i++) {

numberList[i] = i;

**printf** ("numberList[%u] = %d\n", i, numberList[i]);

}

**Displays**

numberList[0] = 0

numberList[1] = 1

numberList[2] = 2

numberList[3] = 3

numberList[4] = 4

* To manually assign values to our numberList array we could write code like this:

// Manually initialize array

numberList[0] = 3;

numberList[1] = 34;

numberList[2] = 44;

numberList[3] = 81;

numberList[4] = 7;

* Note: if you try to access an element outside the range of an array C does not create a runtime error but you are causing problems since writing over memory.

**Array Default Values**

* Like other types, when an ***array*** is created, it is filled with garbage values NOT zero!

**int** numberList[5];

// Showing the values for a numeric array are filled with garbage

**for** (size\_t i = 0; i < 5; i++) {

**printf** ("numberList[%u] = %d\n", i, numberList[i]);

}

**Displays (will vary on each system)**

numberList[0] = 2130567168

numberList[1] = 0

numberList[2] = 2686716

numberList[3] = 2031061624

numberList[4] = 2686916

* Should always initialize arrays with an appropriate default value. Here is one way to do so:

**int** numberList[5];

// Initializing array elements to zero

**for** (size\_t i = 0; i < 5; i++) {

numberList[i] = 0;

**printf** ("numberList[%u] = %d\n", i, numberList[i]);

}

**Array Initializers**

* Shorthand notation that combines **definition and initialization** into one statement
* With scalar types, we've written statements like:

// Initialize scalar values with one statement

**int** myData = 19;

**double** taxRate = 0.2;

**char** middleInitial = 'W';

Here we declare variables and assigned values (initialized).

* With arrays, we can write a statement like:

// Array initializer – define and initialize in one statement

**int** numberList[5] = {3, 34, 44, 81, 7};

**for** (size\_t i = 0; i < 5; i++) {

**printf** ("numberList[%u] = %d\n", i, numberList[i]);

}

Here we defined and initialized an array!

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

// Fast way to initialize all elements to zero

**int** numberList[5] = {0};

* If you leave off the array size, the number of initializers determines the array’s size

**Examples of Processing Arrays**

// Use this array

**int** numbers[5];

* Initializing array with input values

**for** (size\_t i = 0; i < 5; i++) {

**puts** ("Enter a numeric value");

**scanf** ("%d", &numbers[i]);

}

* Initializing array with random values between 0 and 100

**for** (size\_t i = 0; i < 5; i++) {

numbers[i] = **rand**() % 100;

}

* Displaying array

**for** (size\_t i = 0; i < 5; i++) {

**printf** ("numbers[%d] = %d\n", i, numbers[i]);

}

* Summing all elements

**int** sum = 0;

**for** (size\_t i = 0; i < 5; i++) {

sum = sum + numbers[i];

}

**printf** ("Sum = %d\n", sum);

* Finding the largest element

Note loop starts at 1 not zero

**int** largest = numbers[0];

**for** (size\_t i = 1; i < 5; i++) {

**if** (numbers[i] > largest) {

largest = numbers[i];

}

}

**printf** ("Largest = %d\n", largest);

**Examples of Processing Arrays in one Program**

**#include** <stdio.h>

**#include** <stdlib.h>

**#define** LENGTH 5 // Defines a constant value – used to represent size of array

**int** **main**(void) {

// Examples of processing array

**int** numbers[LENGTH];

// 1. Initializing array with input values

**for** (size\_t i = 0; i < LENGTH; i++) {

**puts** ("Enter a numeric value");

**scanf** ("%d", &numbers[i]);

}

// 2. Displaying array

**for** (size\_t i = 0; i < LENGTH; i++) {

**printf** ("numbers[%d] = %d\n", i, numbers[i]);

}

// 3. Summing all elements

**int** sum = 0;

**for** (size\_t i = 0; i < 5; i++) {

sum = sum + numbers[i];

}

**printf** ("Sum = %d\n", sum);

// 4. Finding the largest element

**int** largest = numbers[0];

**for** (size\_t i = 1; i < 5; i++) {

**if** (numbers[i] > largest) {

largest = numbers[i];

}

}

**printf** ("Largest = %d\n", largest);

} // main

**Displays**

Initialize array with input values

Enter a numeric value

8

Enter a numeric value

2

Enter a numeric value

7

Enter a numeric value

64

Enter a numeric value

45

Displaying array

numbers[0] = 8

numbers[1] = 2

numbers[2] = 7

numbers[3] = 64

numbers[4] = 45

Sum all elements of an array

Sum = 126

Find the largest element in the array

Largest = 64

**Rules for Arrays**

* Once created, an array's size is fixed
* All elements in an array will be the same type
* To reference an item in an array use "bracket" notation
* Attempting to access an element with an index outside the range of the array
  + **DOES NOT cause a runtime error as it does in Java**
* These do not work in C

**printf** ("Length of array is %d\n", numberList.length);

**printf** ("Length of array is %d\n", numberList.max);

**Character Arrays**

**Strings**

* In languages like Java, strings are a data type
* In C, there is no string data type but strings are represented by an array of characters

**Defining Character Arrays**

* Define a “string” in C

// Create an array that can store 5 characters

**char** string[5];

**Examples**

* Define and initialize array with a string

// Create an array and initialize with a string

**char** string1[] = "Hello World"; // Complier determines size of the array based

// on string length. Here length of string is // 11 but there is one more character all

// strings contain – the null character – so

// array length is 12

* Define and initialize an array with an initializer list

// Initialize array with initializer list

// Hard way of getting values into array!

**char** string2[] = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd'};

* Initializing array with input values

**char** string3[20];

**puts** ("Enter your first name"); Note: use %s - did not include & before string3

**scanf** ("%19s", string3);

* Display array representing a string

**printf** ("%s\n", string3); Outputs characters until the Null is reached

**Notes**

* The string-terminating character is the null character
* Null is represented by **‘\0’**
* Don’t forget to include space in your array for the null character
* When using character arrays in scanf you **DO NOT** include the address operator (&)
* The array name is the **address** of the first element in the array
* **scanf** reads characters until a space, tab, newline, of end of file indicator is reached
* Trying to put more values into your array than there is space allocated can
  + Program to crash, or
  + Cause buffer overflow

Both are issues!

**Static Local Arrays and Automatic Local Arrays**

**Recall from Chapter 5**

* **Automatic Identifier**
  + Exist only within the block defined
  + A function’s local variables – have **auto** by default so no need to include keyword
* **Static Local Identifier**
  + Exist from the time program beings until program terminated
  + Static local variables in functions
    - Exist for program duration
    - Only visible in function
    - Retain value between function calls

**Automatic Local Arrays**

* Local variables by default are automatic
* Memory for variable is created in function and destroyed when function completes
* This can be expensive for arrays if function is called often

**Static Local Arrays**

* Array is created once
* Defining array as static causes all elements to be initialized to zero by default – but only once!
* Array is only visible in the function
* Array retains values between calls
* Increases performance

Figure 6.11 for an example!

**Passing Arrays to Functions**

**Passing Arrays**

* Chapter 5 we learned how to pass **ints**, **floats**, and **doubles** to functions

**int** number1 = 7;

**int** number2 = 82;

*swap*(number1, number2);

.

.

.

// Function to swap two integers values

**void** **swap** (**int** num1, **int** num2) {

...

} // swap

* We can also pass arrays to methods

// Define an array to hold 10 integer values

**int** numbers1[10];

**int** numbers2[10];

swap(numbers1, numbers2, 10);

.

.

.

// Function to swap two integer arrays

**void** **swap** (**int** numbers1[], **int** numbers2[]) {

...

} // swap arrays

* Note: in the function call when passing arrays, you **DO NOT** use array notation in the argument list

**Correct:** swap(numbers1, numbers2, 10);

**Incorrect:** swap(numbers1[], numbers2[], 10);

**Incorrect:** swap(numbers1[10], numbers2[10]);

**Passing Scalars vs. Passing Arrays (The Difference)**

* Scalars - C uses ***pass by value***
* Arrays - C uses ***pass by reference***
* Pass by value
  + Means a copy ***of the value/variable*** is sent to the function
  + The original variable outside function is not changed by anything that happens inside the function
  + Below **number1** and **number2** in main are **not** affected by the what happens in **swap** function

**int** **main**(**void**) {

**int** number1 = 7;

**int** number2 = 82;

swap (number1, number2); **// num1=7 & num2=82 after swap function called**

} // main

// Function to swap two integers values

**void** **swap** (**int** num1, **int** num2) {

// Swap the values

**int** temp = num1;

num1 = num2;

num2 = num1;

} // swap

* Pass by reference
  + Means the ***address to the array*** is sent to the function
  + In C, the name of the array evaluates to the **address** of the 1st element in the array
  + The original array outside the method **is** affected by changes inside the function

**Pass by Reference Example**

* Example of passing an array to a function and the effects

**int** **main**(**void**) {

// Passing arrays to methods - showing effects of pass by reference

**int** array1[LENGTH] = {0, 1, 2, 3, 4};

**int** array2[LENGTH] = {100, 101, 102, 103, 104};

**puts** ("array1 BEFORE swap method");

**for** (size\_t i = 0; i < LENGTH; i++) {

**printf** ("array1[%d] = %d\n", i, array1[i]);

}

**puts** ("array2 BEFORE swap method");

**for** (size\_t i = 0; i < LENGTH; i++) {

**printf** ("array2[%d] = %d\n", i, array2[i]);

}

// Now swap the values in the arrays so

// array1 contains the values from array2 and

// array2 contains the values from array1

arraySwap(array1, array2, LENGTH);

**puts** ("array1 AFTER swap method");

**for** (size\_t i = 0; i < LENGTH; i++) {

**printf** ("array1[%d] = %d\n", i, array1[i]);

}

**puts** ("array2 AFTER1 swap method");

**for** (size\_t i = 0; i < LENGTH; i++) {

**printf** ("array2[%d] = %d\n", i, array2[i]);

}

} // end main

// Swaps the values of two integer arrays

// Input: Two integer arrays

// Return: None

**void** **arraySwap** (**int** numbers1[], **int** numbers2[], size\_t size) {

**int** tempArray[size];

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

tempArray[i] = numbers1[i];

}

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

numbers1[i] = numbers2[i];

}

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

numbers2[i] = tempArray[i];

}

} // swap

**We can combine these for loops and write one concise loop:**

**void** **arraySwap** (**int** array1[], **int** array2[], size\_t size) {

**int** tempArray[size];

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

tempArray[i] = numbers1[i];

numbers1[i] = numbers2[i];

numbers2[i] = tempArray[i];

}

} // swap

**Displays**

array1 BEFORE swap method

array1[0] = 0

array1[1] = 1

array1[2] = 2

array1[3] = 3

array1[4] = 4

array2 BEFORE swap method

array2[0] = 100

array2[1] = 101

array2[2] = 102

array2[3] = 103

array2[4] = 104

array1 AFTER swap method

array1[0] = 100

array1[1] = 101

array1[2] = 102

array1[3] = 103

array1[4] = 104

array1 AFTER swap method

array2[0] = 0

array2[1] = 1

array2[2] = 2

array2[3] = 3

array2[4] = 4

**Trace of Example**

* The main function is where the code starts
* Two integer arrays are defined and initialized

**int** array1[LENGTH] = {0, 1, 2, 3, 4};

**int** array2[LENGTH] = {100, 101, 102, 103, 104};

* The **arraySwap** function is called
  + The address to array1 and address to array 2 are sent to the function

array1 1

Contents of array 1

array 2

Contents of array2

arraySwap(array1, array2, LENGTH);

* Inside the arraySwap function the values of the arrays are swapped
  + A local array called **tempArray** is created
  + Swap occurs
  + arraySwap ends and control returns to main
* The arrays in main now have NEW values
  + array1's address (or starting memory location) was sent to arraySwap
  + array2's address (or starting memory location) was sent to arraySwap
  + arraySwap was manipulating the actual arrays NOT copies!

**Notes**

* When an element of an array is passed to a function, it is passed by value
* When an entire array is passed to a function, it is passed by reference by default
* Done for performance reasons – if passed by value – huge arrays would consume lots of memory & time!
* It is possible to pass arrays by value – will learn about in Chapter 10
* If you do not want a function to be able to modify the incoming array
  + Use type qualifier **const**
  + Place next to array name in parameter list of function
  + Turns array into a constant array, it now cannot be modified

**void** **arraySwap** (**const** **int** numbers1[], **const** **int** numbers2[], size\_t size) {

**int** tempArray[size];

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

tempArray[i] = numbers1[i];

numbers1[i] = numbers2[i]; Complier error because **const** array!

numbers2[i] = tempArray[i]; Complier error because **const** array!

}

} // swap

See Figure 6.13 for an example

* Go over this figure to understand difference of passing entire array vs passing single element