Chapter 6 – Array Concepts

As mentioned before, arrays make accessing a lot of data more efficient than allocating individual variables. A very common thing to do with lists is to sort them. We frequently have alphabetical lists, but we also sort lists numerically such as standings in sports. We will learn 2 simple algorithms for generating sorted lists: bubble sort and insertion sort. It is important to understand how efficient the algorithm is to sort and to access. There are more efficient sorting algorithms, but they are complicated to implement, so we will simply use them as libraries after we learn about data structures.

We start with insertion sort. We will sort in ascending order, but you can just as easily do it in descending order. An important property for sorting is the ability to sort in-place; this means that no excessive memory is needed to execute the algorithm. Fortunately, insertion sort has this property.

Here is a small sample list, but the concept extrapolates to larger lists. Insertion sort takes each element from left to right and inserts it into the proper place to the left. "8" is already in place. "16" is already in place because it comes after "8". "-3" is not in place, so it needs to be placed before "8". We move "16" to the 3rd spot, then we move "8" to the 2nd spot. We keep repeating until move the last "4" into its correct spot. Below is the list after each major iteration (we don't iterate over the shaded area):

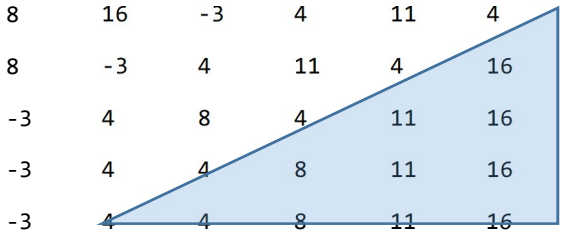
8	16	-3	4	11	4
-3	8	16	4	11	4
-3	4	8	16	11	4
-3	4	8	11	16	4
-3	4	4	8	11	16

Recall that the first array element has an index of 0 and the last array element has an index of *n-1*. So for the above list, it is 5.

> **Exercise 6.1: To implement this algorithm, you will have to know how to iterate backwards through a list. Download Example_6_1 if you need to see how to do this. All you have to do is to insert the correct condition to complete the sort. Notice that the example will simply reverse the list with no condition set. The above iterations is what you output should look like. Be careful about the indices that you use; it is very easy to be off by "1". You can use the *printList* method provided or you can make your own.**
>
> **Exercise 6.2: Change exercise 6.1 to implement a descending sort.**

Notice how many comparisons we have to make to fully sort the list. We have 1+2+3+4+5 = 15 or (6*5)/2 = 15. Notice that there are 6 columns and 5 rows! If we say the list size is ***n***, then we have $\frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$. For efficiency, we only consider the highest power and ignore the coefficient, so we say the efficiency is order *n* squared, $O(n^2)$. The best sort routines have an efficiency of $O(n*lg(n))$, lg is log base 2. So insertion sort is generally only used on small lists, $n < 100$. If we look at a list size of 100, the number of comparisons is 100*100 versus 100*7 respectively, which makes the insertion sort 14 times slower. At a list size of 1000, the number of comparisons is 1000*1000 versus 1000*10, which makes the insertion sort 100 times slower! This shows that insertion sort is definitely not the way to go for very large lists.

Bubble sort has the same efficiency as insertion sort. It also has the in-place property. It is implemented with 2 for loops which means $O(n^2)$. We will use the same sample list. Below is the list after each major iteration (we don't iterate over the shaded area):



Exercise 6.3: To implement this algorithm, you will need a nested loop. It doesn't matter if the outer loop counts down or up. The inner loop needs to count from 1 to the last bubble. The inner loop will check the current element with the previous element to see if a swap is necessary. I am being vague on purpose because the explanation should help with how the create the sort. Be very careful about how you count and check your result with each line. For a bonus mark, use a swap flag to exit the algorithm early.

Exercise 6.4: Use a sorting algorithm of your choice. Look back to Example_5_2 if you need. Generate 40 random numbers between 0 and 199. Then sort the list. You decide how the output should look.

A common operation is to look for a name in the list, but we are starting with just numbers for ease. An obvious way to do this is starting at the beginning of the list and checking each item until we get to the end. This has an efficiency of order *n*, *O(n)*. An example of this is Exercise 5.3: the *IndexOfString* method.

Exercise 6.5: Use your code from Exercise 6.4. Get a number from the user to search for. Write a method to search for this number; it should return the index number. Once the number to search for is less than the current element, you can terminate the search since the list is in order. If the number is not found, return -1 for the index number; this is a common return value for not finding something. Your output should say whether the number was found or not; if found, also print out the index number. Part of your output should be printing the random list at the beginning; this is so the search can be verified for correctness. You should test searching for the first and last numbers in the list.

You might think that *O(n)* is efficient. However, there is an algorithm that can search with an efficiency of *O(lg(n))*; this is the binary search algorithm. So for a list size of 100, the number of comparisons is 100 vs 7 respectively; so again we have a multiple of 14 times slower. And for a list size of 10000 is 10000 vs 14; this is a multiple of over 700 times slower! So, how do we implement the binary search? It's a matter of using the high-low game strategy; this means we need variables for the bottom and top (left and right) of the list. We will demonstrate a search for 48 with the following list:

```
0 15 7: -30 -25 -6 -3 4 11 15 [28] 31 32 34 37 39 48 51 61

8 15 11:       -30 -25 -6 -3 4 11 15 28 31 32 34 [37] 39 48 51 61

12 15 13:      -30 -25 -6 -3 4 11 15 28 31 32 34 37 39 [48] 51 61
```

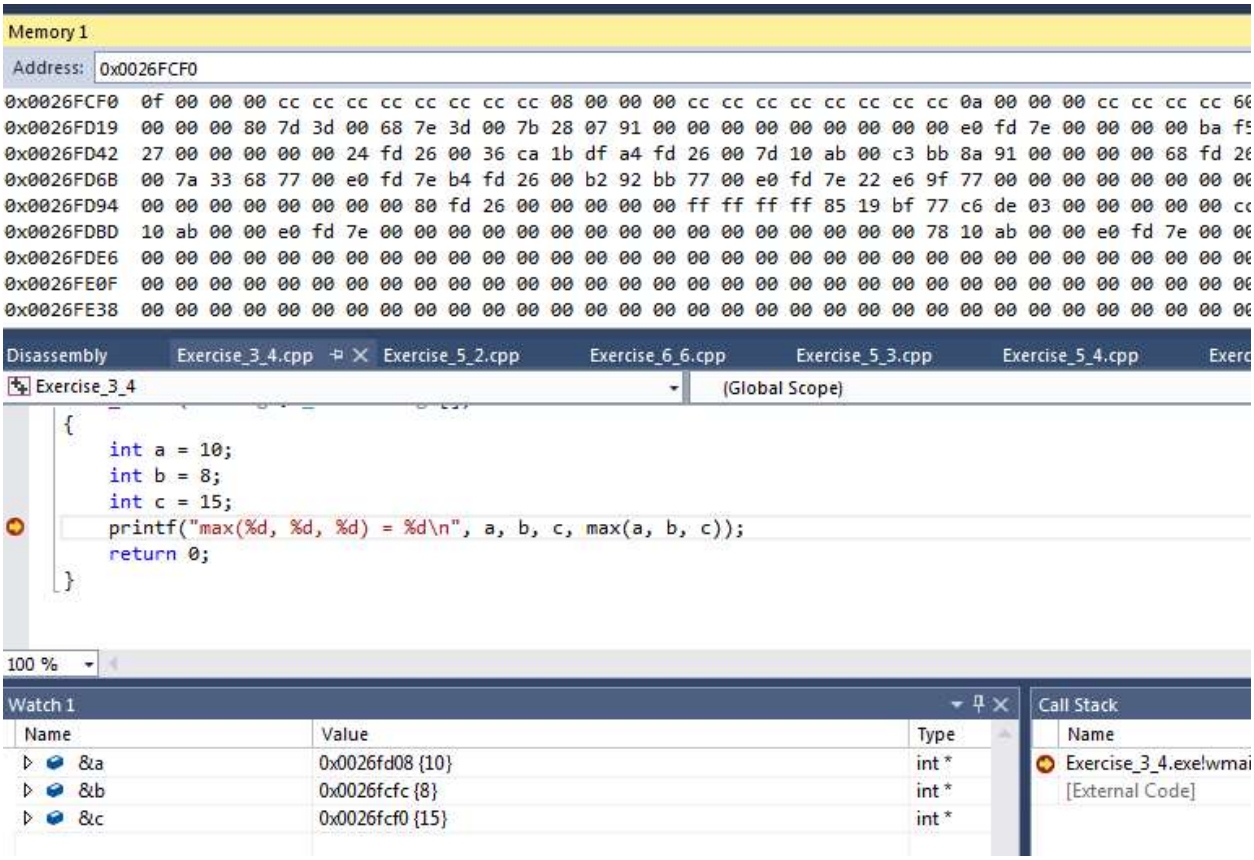
Now we will demonstrate a search for -7, a number not in the list:

```
0 15 7: -30 -25 -6 -3 4 11 15 [28] 31 32 34 37 39 48 51 61

0 6 3:          -30 -25 -6 [-3] 4 11 15 28 31 32 34 37 39 48 51 61

0 2 1:          -30 [-25] -6 -3 4 11 15 28 31 32 34 37 39 48 51 61

2 2 2:          -30 -25 [-6] -3 4 11 15 28 31 32 34 37 39 48 51 61

2 1 2:          stop since bottom index > top index
```

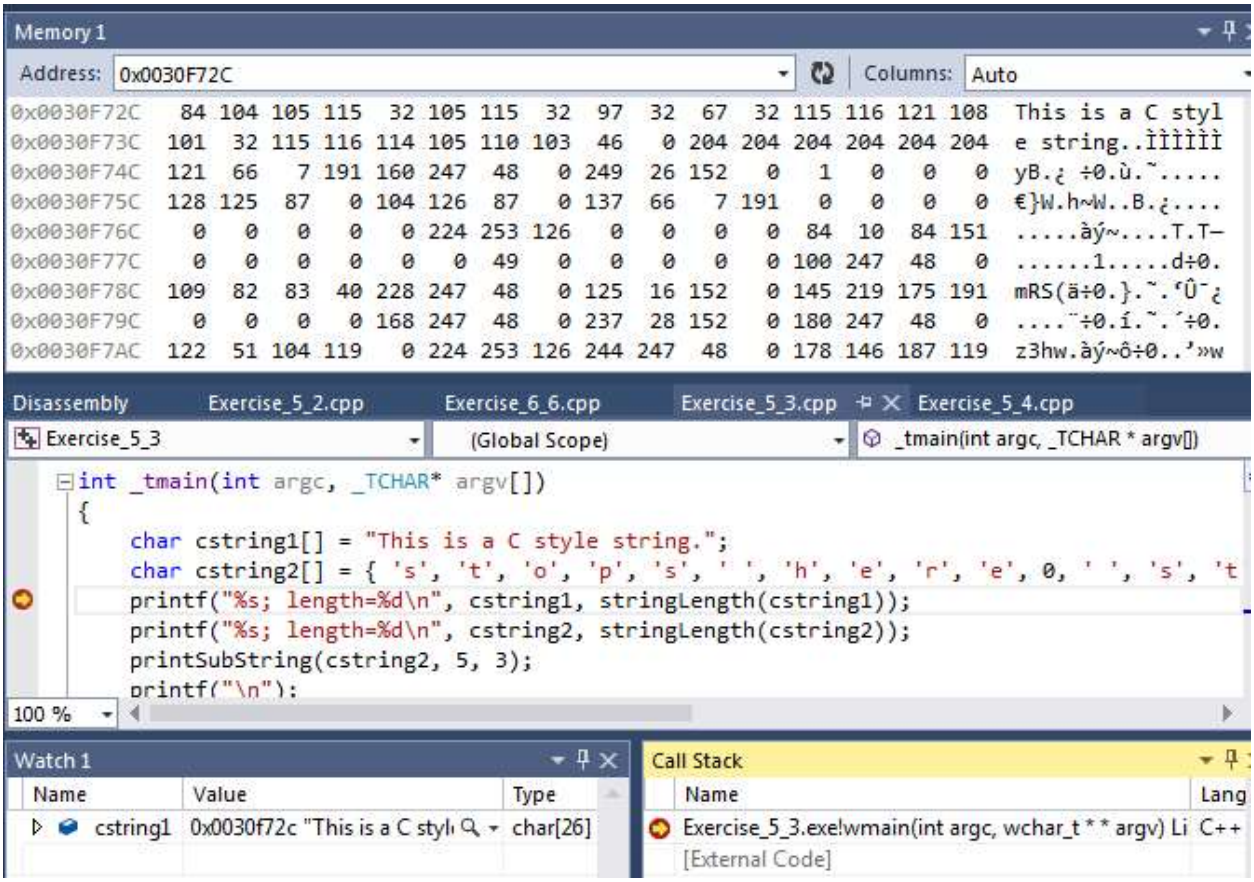
Exercise 6.6: Use Exercise 6.5, but implement the binary search. Add a variable to count the number of comparisons made doing the search and add it to the printout. Sample printout:

```
37, 47, 62, 64, 72, 74, 77, 78, 79, 81, 84, 88, 91, 93, 94, 98, 98, 103…
Enter number to search for (q to quit): 61
Number of searches: 5
61 is not in the list.
Enter number to search for (q to quit): 62
Number of searches: 4
The first of occurrence of 62 is at index 2
Enter number to search for (q to quit):
```

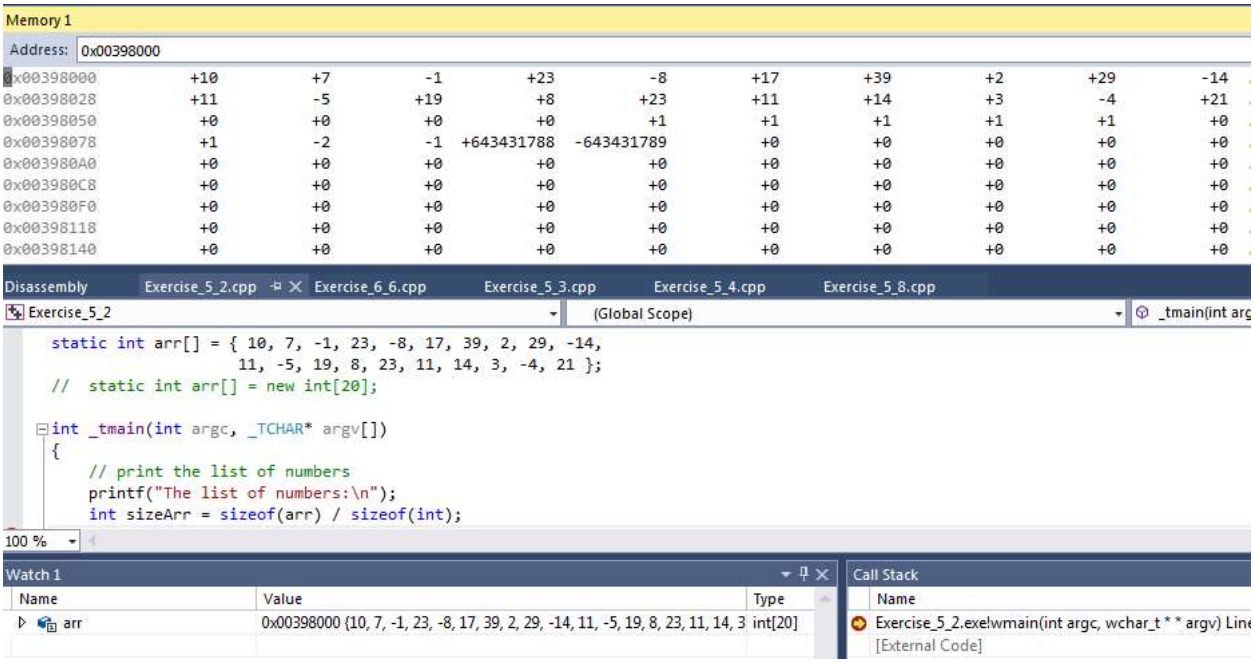
Now that you've worked with arrays for a bit, let's see how it is implemented in memory. When you declare individual variables, they can be placed in any order, so not necessarily the order that you have declared them. And they may not even be in a contiguous block. For example:



When you allocate an array, it is guaranteed to be contiguous:



To access any character, we use *cstring1[index]*. The math to calculate the memory address is simple since 1 character takes up 1 byte. The equation is the ***address of cstring1 + index***. This shows why the first array element is always 0 rather than 1.



When we allocate an array that has elements larger than a byte, accessing elements is not much more difficult. For the above example of an ***int*** array whose element size if 4 bytes, we calculate the address of any element using the ***address of arr + index*4***. Fortunately, multiplying by 4 is an extremely simple operation for a computer. In fact, multiplying by any power of 2 is extremely easy. That's why native data types have a size that is a power of 2. Let's look at an example:

```
Decimal              Binary
3*4 = 12             11*100 = 1100
5*4 = 20             101*100 = 10100
13*4 = 52            1101*100 = 110100
10*4 = 40            1010*100 = 101000
```

So, hopefully you noticed the pattern; we simply add 2 zeroes. The CPU does this with an operation called shift bits left. This is akin to us multiply by 10; we don't actually do any arithmetic, we just add a zero to the end. If we multiply by 1000, we simply add 3 zeroes.

In conclusion, arrays are very efficient for accessing any element.

Libraries

Now that you have a full appreciation of sort arrays, let's look at how to use the libraries to sort. In Java, it is straight forward; all you have to do is call Arrays.sort with any array as the parameter. To search for an element, you call Array.binarySearch with the array and element as the parameters. You can download Example_6_2. Here is a code snippet:

```
import java.util.Random;
import java.util.Arrays;

int list1[] = new int[40];
int listSize = list1.length;
Random rand = new Random();
for (int i = 0; i < listSize; i++)
        list1[i] = rand.nextInt(200000);
```

```
long startTime = System.nanoTime();
Arrays.sort(list1);
estimatedTime = System.nanoTime() - startTime;
System.out.println("Number of microseconds for Library sort " + estimatedTime/1000);
index = Arrays.binarySearch(list1, a);
```

We need to check the performance of routines. You have to make multiple runs to get a reasonable answer; the reason is a multi-tasking operating system interrupts your code therefore increasing your run-time. We do this with a routine called System.nanoTime(); it isn't necessarily measuring nanoseconds, but it will get the best timer available in the system. If you are clever, you'll be asking "won't the counter wrap-around since it is counting nanoseconds?" Well…. long is 64 bits, so we have 2^{63} which is almost 10^{19}. Divide by nanoseconds and we still have 10^{10}. Divide this by the number of seconds in a day, and we still get 10^{5}; that is a lot of days!

Exercise 6.7: Use Exercise 6.5 and 6.6. Increase the array size to 10000 and make another the same size. Generate random numbers from 0 to 200000 to fill both arrays with the same values. Do your sort and display the number of microseconds. Then do the Library sort and display the number of microseconds. Display the last 20 numbers of the array. Then do a linear search, binary search, and library search of one of the last 20 numbers; print the number of nanoseconds for each of the searches. Sample printout:

```
81985, 161637, 61536, 49118, 58145, 99113, 163282, 84909, 95798, …

Number of microseconds for bubble sort 285195

1, 38, 38, 52, 68, 78, 100, 128, 142, 185, 220, 241, 283, 317, 355, …
199611, 199611, 199621, 199624, 199626, 199657, 199688, 199703, …

Number of microseconds for Java Library sort 8189

199611, 199611, 199621, 199624, 199626, 199657, 199688, 199703, …
Enter number to search for (q to quit): 199611

Number of nanoseconds for linear search 835329

The first of occurrence of 199611 is at index 9980

Number of nanoseconds for binary search 9371

The occurrence of 199611 is at index 9980

Number of nanoseconds for Library search 20080

The occurrence of 199611 is at index 9980
Enter number to search for (q to quit): 199620

Number of nanoseconds for linear search 854069

199620 is not in the list.

Number of nanoseconds for binary search 4908

199620 is not in the list.
```

```
Number of nanoseconds for Library search 4908

199620 is not in the list.
Enter number to search for (q to quit):
```

Exercise 6.8: Sorry, no hints for this one. You will perform a histogram of rolling a pair of dice. Make sure you roll 2 dies and add them together. Perform 360000 rolls then printout the results as a table of numbers. In the comments, state what are the expected probabilities of rolling each number as a fraction of 36. Test your histogram with 10 rolls first that way you know your code is working.

Now we revisit the deck of cards problem.

Exercise 6.9: Create an array to hold the whole deck and initialize it as a new ordered deck. Print out the deck. Shuffle the deck by looping through every card and randomly pick a card to switch positions with. Print out the deck. Finally, ask the user for how many hands to deal out for BlackJack. Determine the winning hand(s). Remember to do some error checking and create re-usable functions. Sample printout:

```
Ordered deck:
Ace of Spades, 2 of Spades, 3 of Spades, …, King of Clubs.
Shuffled deck:
5 of Hearts, Queen of Diamonds, 7 of Spades, 9 of Spades, Jack of Clubs, 8 of Hearts, …
How many hands to deal? 3
5 of Hearts, 9 of Spades = 14
Queen of Diamonds, Jack of Clubs = 20
7 of Spades, 8 of Hearts = 15
Winner(s): Hand 2
```