PLAYBOOK NOTES: RECURSION – PART 3

The following is the code for the next function we will trace:

```
285  void PositiveNumberCount(int myArray[], int arrayCount, int initialArrayPosition, int* pPositiveNumberCount)
286  {
287      // Loop until array position reaches size of the array
288      if (initialArrayPosition < (arrayCount - 1))
289      {
290          PositiveNumberCount(myArray, arrayCount, initialArrayPosition + 1, pPositiveNumberCount);
291      }
292
293      if (myArray[initialArrayPosition] > 0)
294      {
295          // Increment positive number count
296          (*pPositiveNumberCount)++;
297      }
298  }
```

The function call from the main looks like this:

```
85      int myArray[] = { 2, 0, 1, -5 };
86
87      int numberOfPositiveNumbers = 0;
88
89      PrintDivider(stdout, DASH, 80);
90      CenterString(stdout, "2nd Recursive Function: Array function - Printed to the screen", 80);
91      PrintDivider(stdout, DASH, 80);
92
93      PositiveNumberCount(myArray, 4, 0, &numberOfPositiveNumbers);
94
95      fprintf(stdout, "Number of positive integers in the array: %d\n", numberOfPositiveNumbers);
```

You can see from the picture above that there are 4 integers in the array.

You can also see that we are passing in a 0, this is the starting position of the array.

When the array position is equal to the array size, we have reached our base case.

This function does not produce output so our trace table will look like this:

Function:	Parameter(s)	Next Line to Execute:

To start, we will set a break point at line 93. (Program execution has paused at line 93).

Our call stack will look like this:

Function:	Parameter(s)	Next Line to Execute:
main	None	93

At line 93, there is a call to our function PositiveNumberCount passing in the following values:

- myArray – {2, 0, 1, -5}
- 4 – The size of the array
- 0 – The starting position of the array
- `&numberOfPositiveNumbers – The address to a variable that hold the count of positive numbers.`

NOTE: Since the parameters, myArray, the size of the array, and the address for the positive number count are not relevant to the base case, these parameters will not be represented on the call stack.

Next, we step into the function and now our call stack looks like this:

First function call: The call stack looks like this

Function:	Parameter(s)	Next Line to Execute:
`PositiveNumberCount`	position = 0	Top of the function
main	None	95

The next line of code is an if-statement that checks to see if position (0) is less than the size of the array minus 1 (3). This is true, so we call the function PositiveNumberCount again, except this time we pass in position + 1.

Second function call: The call stack looks like this

Function:	Parameter(s)	Next Line to Execute:
`PositiveNumberCount`	position = 1	Top of the function
`PositiveNumberCount`	position = 0	293
main	None	95

The next line of code is an if-statement that checks to see if position (1) is less than the size of the array minus 1 (3). This is true, so we call the function PositiveNumberCount again, again we will pass in position + 1.

Third function call: The call stack looks like this

Function:	Parameter(s)	Next Line to Execute:
`PositiveNumberCount`	position = 2	Top of the function
`PositiveNumberCount`	position = 1	293
`PositiveNumberCount`	position = 0	293
main	None	95

The next line of code is an if-statement that checks to see if position (2) is less than the size of the array minus 1 (3). This is true, so we call the function PositiveNumberCount again, again we will pass in position + 1.

Fourth function call: The call stack looks like this

Function:	Parameter(s)	Next Line to Execute:
`PositiveNumberCount`	position = 3	Top of the function
`PositiveNumberCount`	position = 2	293
`PositiveNumberCount`	position = 1	293
`PositiveNumberCount`	position = 0	293
main	None	95

The next line of code is an if-statement that checks to see if position (3) is less than the size of the array minus 1 (3). This is false, so we are now at our base case. The function continues by checking to see if myArray[3] is greater than 0. If it is, the counter will be incremented.

The function PositiveNumberCount is now finished and comes off the stack.

Our call stack looks like this:

Function:	Parameter(s)	Next Line to Execute:
`PositiveNumberCount`	position = 2	293
`PositiveNumberCount`	position = 1	293
`PositiveNumberCount`	position = 0	293
main	None	95

The next function at the top of the stack finishes executing. This function will check to see if myArray[2] is greater than 0. If it is, the counter will be incremented.

The function PositiveNumberCount is now finished and comes off the stack.

Our call stack looks like this:

Function:	Parameter(s)	Next Line to Execute:
`PositiveNumberCount`	position = 1	293
`PositiveNumberCount`	position = 0	293
main	None	95

The next function at the top of the stack finishes executing. This function will check to see if myArray[1] is greater than 0. If it is, the counter will be incremented.

The function PositiveNumberCount is now finished and comes off the stack.

Our call stack looks like this:

Function:	Parameter(s)	Next Line to Execute:
`PositiveNumberCount`	position = 0	293
main	None	95

The next function at the top of the stack finishes executing. This function will check to see if myArray[0] is greater than 0. If it is, the counter will be incremented.

The function PositiveNumberCount is now finished and comes off the stack.

Now we are back in the main and line 95 is ready to execute.

This line prints the result to the screen:

```
Number of positive integers in the array: 2
```