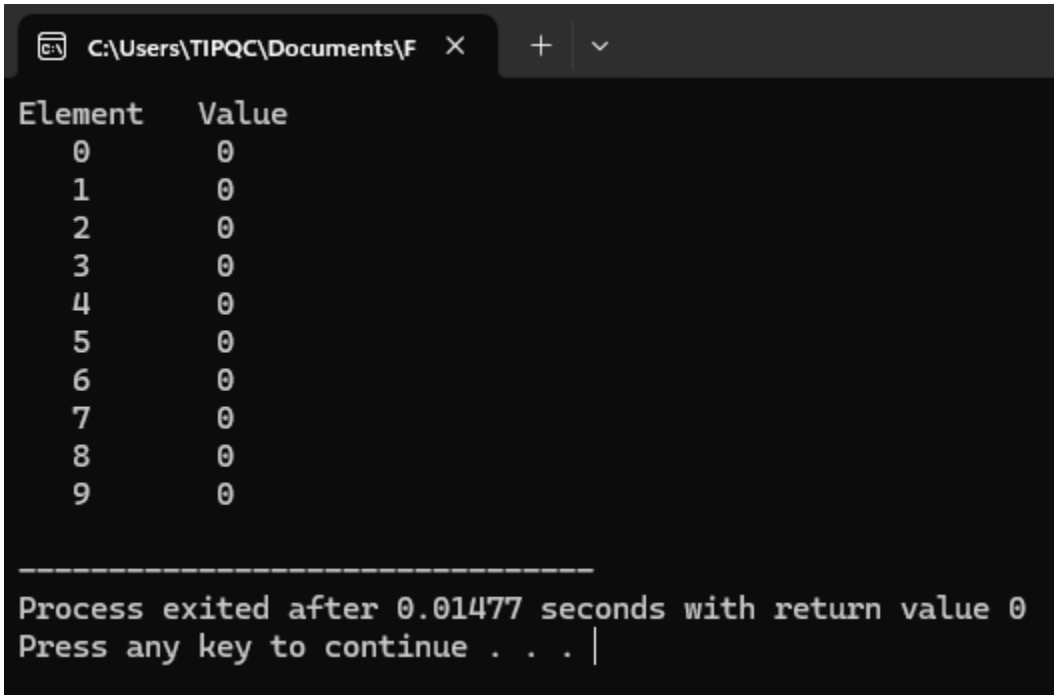
Activity No. 4.2	
Hands-On: Arrays	
Course Code: CPE007	**Program:** Computer Engineering
Course Title: Programming Logic and Design	**Date Performed: 09/13/25**
Section: CPE11S1	**Date Submitted: 09/13/25**
Name(s): James Daniel M. Verano	**Instructor: Engr. Jimlord M. Quejado**
6. Output	

Example 1:

Code:

```cpp
#include <iostream>
using namespace std;

int main() {
    int n[10];

    // Initialize array elements to 0
    for (int i = 0; i < 10; i++) {
        n[i] = 0;
    }

    cout << "Element   Value" << endl;

    // Print index and value
    for (int i = 0; i < 10; i++) {
        cout << "    " << i << "         " << n[i] << endl;
    }

    return 0;
}
```

Output:



Explanation on how does the code work:

In the first for loop function line codes, it is to declare the array elements and index value to 0 up to 9, and once reached, the index and value of *n[i]* that was declared, is fixed. Declaring *n[i]* to 0, means from 0, from the starting loop if i with an index of 0 *(i = 0)*, as long as *[i]* is LESS THAN 10, INCREMENT *[i]*.

For visuals:
i = 0, n[i] = 0;
i = 1, n[i] = 0;
i = 2, n[i] = 0;
i = 3, n[i] = 0;
. . . . .

And on the second for loop, which is the same index declaration, meaning that, from index 0 to 9, we will be functioning the inside code. Which is the printing of the index value, which was *[i]* and the value assigned to *n[i]*.

Example 2:

Code:

```cpp
#include <iostream>
using namespace std;

int main() {
    int n[10] = {32, 27, 64, 18, 95, 14, 90, 70, 60, 37};

    cout << "Element  Value" << endl;

    for (int i = 0; i < 10; i++) {
        cout << "    " << i << "       " << n[i] << endl;
    }

    return 0;
}
```

Output:

```
Element  Value
    0       32
    1       27
    2       64
    3       18
    4       95
    5       14
    6       90
    7       70
    8       60
    9       37

--------------------------------
Process exited after 0.01669 seconds with return value 0
Press any key to continue . . .
```

Explanation on how does the code work:

In this code, it's just the same as the code from the previous example, the difference is *n[i]*. *n[i]* now has element value, which we would only need to print them out using the same for loop code, with the *[i]* index. In the previous code, we created a for loop function to add a value to *n[10]*, the variable that was declared as an array with element value. Since in

this case, n[10] now has an element value, we would not need to loop *n[10]* to add element value but proceed to the case of assigning the *[i]* index value to each of the elements in the array of *n[10]*, and starts the printing of the index value and *n[i]* which is the assigned array element.

for visuals:
i = 0, n[i] = [0]; (First array element value)
i = 1, n[i] = [1]; (Second array element value)
i = 2, n[i] = [2]; (Third array element value)
. . . . .

Example 3:

Code:

```
#include <iostream>
using namespace std;

#define SIZE 12

int main() {
    int a[SIZE] = {1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45};
    int total = 0;

    for (int i = 0; i < SIZE; i++) {
        total += a[i];
    }

    cout << "Total of array element values is " << total << endl;
    return 0;
}
```

Output:

```
Total of array element values is 383

--------------------------------
Process exited after 0.009969 seconds with return value 0
Press any key to continue . . .
```

Explanation on how the code works:

In this code, we can see that, we have a declared variable, *total = 0,* which is going to be used as a value input for the addition of each element array, and *a[SIZE]* that has quite an amount of array element value. In this example, our intended outcome is to add all element values, and print it out. As just the same, from the previous examples, we will be using the same for loop function in the printing and adding value. The difference is, we will add a functioning code, wherein the array element values will be added to each other. As we can see, there's a new phrase, *i < SIZE* in the for loop function, but it is just the same, as *i < 10*. It is just that it's declared in a variable where a value is declared, *SIZE = 12*, it is just the same as, *i < 12 (SIZE)*. So, nothing changes. Inside the for loop function, there is a functioning code, which is declared as, *total += a[i]*. This code function adds all element values that are declared to *[i]*. We can say that the amount that we declared are 12 array element values to 0 - 11 index value since the value of SIZE is 12.
Thus, *total += a[i]* implies adding the value of the assigned index values to *a[SIZE]* Interpreted as, *a[i]*.

For visuals:

i = 0, a[SIZE] / a[12], assign a[i] = a[0]. first array element value.
i = 1, a[1]. Second array element value.
i = 2, a[2]. Second array element value.

[AND]
total += Means, total = a[0] + a[1];

[WITH NUMBER VALUE]
total = 1 + 3; // 1 and 3 is assigned to a[0] and a[1].
since now, we have a value of 4 in total, we will be adding that total variable to the next array value, and assign it back to total.
total = total + a[2];
total = total + a[3];
. . . . .
// and keep looping it till it reaches to the desired for loop condition function which was i < SIZE.

7. Supplementary Activity

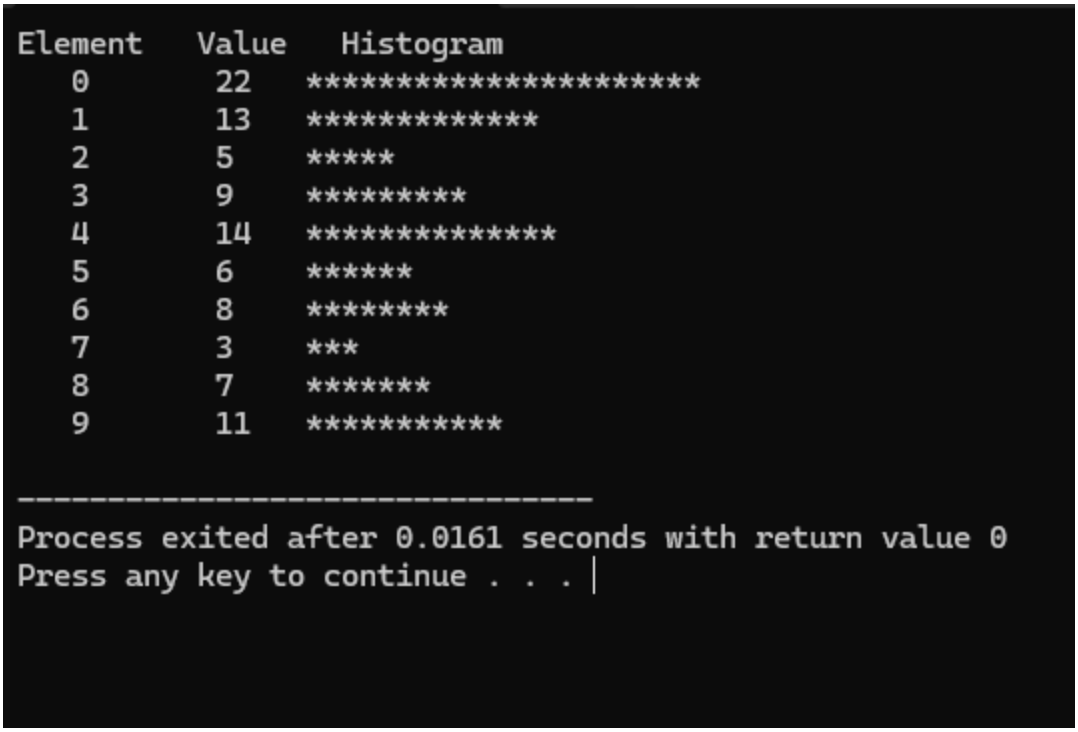
Example 1:

Code:

```cpp
#include <iostream>
using namespace std;

int main() {
    int value[10] = {19, 3, 15, 7, 11, 9, 13, 5, 17, 1};
    char histogram = '*';

    cout << "Element   Value   Histogram" << endl;

    // Print index and value
    for (int i = 0; i < 10; i++) {
        cout << "    " << i << "         " << value[i] << "     ";
        for (int h = 0; h < value[i]; h++) {
            cout << histogram;
        }
        cout << endl;
    }

    return 0;
}
```

Output:

```
Element   Value   Histogram
   0       19    ********************
   1       3     ***
   2       15    ***************
   3       7     *******
   4       11    ***********
   5       9     *********
   6       13    *************
   7       5     *****
   8       17    *****************
   9       1     *

--------------------------------
Process exited after 0.01962 seconds with return value 0
Press any key to continue . . .
```



Explanation on how the code works:

In this example, we can see that we need to print out 3 sections, the index number, the array value, and the array's value histogram. In the previous, output activity example, the print out of the index number and the array value, will undergo the same procedure of, assigning of index value to the array elements, and the print out of it. I'll be more likely to focus on how the print out of the array's value histogram works. Slight recap, we know that the first for loop function, declared as, *for (int i = 0; i < 10; i++)* is the assigning of index value and the identity the amount of loop this function will be looping. Now, on the second for loop, we can see that we are more focused on the histogram. In the second for loop, declared as, *for (int h = 0; h < value[i]; h++)*, This means that, if h is LESS THAN the value declared in the variable *value[i]*, we will keep looping and printing it. so, hypothetically, if h = 0 and *value[i]* = 12, we will keep printing the *char variable declared in histogram. Don't forget to add an endl; INSIDE the first for loop parenthesis, at the very end of the parenthesis tag, BUT NOT INSIDE THE SECOND FOR LOOP PARENTHESIS AREA.* or else the * will print line by line.

for visuals:
h = 0 < value[0] = 12 = 0 < 12, [true] = print * = *
h = 1 < value[0] = 12 = 1 < 12, [true] = print * = **
h = 2 < value[0] = 12 = 2 < 12, [true] = print * = ***
h = 3 < value[0] = 12 = 3 < 12, [true] = print * = ****
. . . . .

Example 2:

Code:

```cpp
#include <iostream>
using namespace std;

#define RESPONSE_SIZE 40

int main() {
    int responses[RESPONSE_SIZE] =
    { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
    1, 6, 3, 8, 6, 10, 3, 8, 2, 7,
    6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
    5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
    string rp = "response";
    string student = "students";
    int COUNTER_COUNT = 11;
    int counter[COUNTER_COUNT] = {0} ;

    for (int i = 0; i < RESPONSE_SIZE; i++) {
        ++counter[responses[i]];
    }

    cout << "-------------------------\n Response Summary: \n-------------------------\n";

    for(int i = 1; i < COUNTER_COUNT; i++) {
        cout << rp << " " << i << " " << ": " << counter[i] << " " << student << endl;
    }
    return 0;
}
```

Output:

```
-------------------------
 Response Summary:
-------------------------
response 1 : 2 students
response 2 : 2 students
response 3 : 2 students
response 4 : 2 students
response 5 : 5 students
response 6 : 11 students
response 7 : 5 students
response 8 : 7 students
response 9 : 1 students
response 10 : 3 students

--------------------------------
Process exited after 0.1677 seconds with return value 0
Press any key to continue . . .
```

Explanation of how the code works:

In this example, all the previous examples are a combination of this output. As we can see, we have 2 for loop functions, let us identify them one by one. First and common of all, is the declarations. The *responses[RESPONSE_SIZE]* is the declaration for the array's elements. As we can see, *#define RESPONSE_SIZE 40* is just the same as declaring, *RESPONSE_SIZE = 40;*.We have 2 strings declaration, which is *student & rp*, for us to print out the word "students" and "response", to accurately follow the examples desired output structure: "response 1: " 2 " Students ".

Next is, the most used declaration inside the for loop functions, the *COUNTER_COUNT* and *counter[COUNTER_COUNT]*. The *COUNTER_COUNT*, is declared as the variable that will be going through the responses from 1 to 10. WHEREIN, BUT Index 0 will not be counted in the process as it will still be assigned. Why 11? in the process of for loop function, we know that *[i]* assigning always starts to 0. Thus, we need it to be 11 so we can reach the value *i = 10*. Now, the *counter[COUNTER_COUNT]* is assigned to 0 since this is the assigning of a counting variable on the index value where it was assigned.

for visuals:
counter[COUNTER_COUNT] = counter[0] = 0;
counter[COUNTER_COUNT] = counter[1] = 0;
counter[COUNTER_COUNT] = counter[2] = 0;
. . . . .

the *counter[COUNTER_COUNT]* counts the amount of 1's 2's 3's that are inside the array elements. Also one of the reasons why it was assigned to 0, as its starting value.

So now, the first for loop function focuses on the counting of similar array element values, and incrementing their assigned counter variable to display the total amount of responses in 1 - 10.

Now, in the second for loop function, we can see it is no longer equal to 0, since we know that our starting counting variable that contains the responses 1 - 10 is 1 to 10. So, we will start at *i = 1* to see how many responses our variable was assigned to each counting variable, which was the*[COUNTER_COUNT]*.

for visuals:
i = 1, COUNTER_COUNT = counter[i] = counter[1] // The amount of 1 in the array's element.
i = 2, COUNTER_COUNT = counter[i] = counter[2]
i = 3, COUNTER_COUNT = counter[i] = counter[3]
i = 4, COUNTER_COUNT = counter[i] = counter[4]
. . . . .

And the rest of the "*Cout*" line code, is just for spacing and design structure for an organized and cleaner visual in the compiler.

8. Conclusion

In all the explanation and code examples, with different interpretation and intended outcomes, with all the different coding expressions I have established and researched to successfully debug it, I have analyzed a deeper understanding and visualized more to how line code expressions may differ to each other even with a slightest difference. This has significantly impacted my analysis on creating an explanatory procedure and visuals, to furtherly examine how the code works, and how it was structurized to work. The deeper you try to analyze, examine, and display on how your code works, the more you would understand where the assigning, looping, and creation of a new looping function and conditional statement starts.

9. Assessment Rubric