

## Week 1 Assignment on C Programming

Name: Alain Niganze

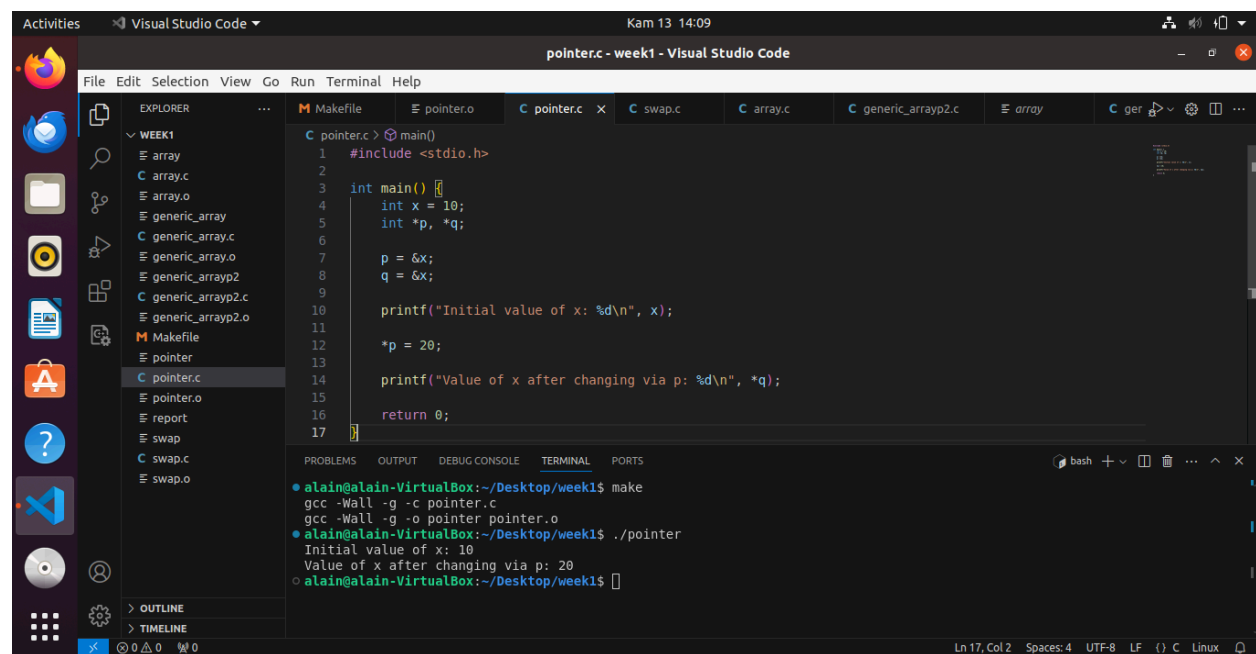
Email: niganzealain@gmail.com

Date: 13 june 2024

### Summary of Activities:

#### 1. Exercise 1: Change value via a pointer

- **Objective:** The goal of this exercise was to demonstrate how pointers can be used to change and read the value of an integer. Specifically, it involved defining an integer `x` and two pointers to that integer, modifying `x` via one pointer, and then reading the modified value via the other pointer.
- **Explanation:** In the `pointer.c` program, I initialized an integer `x` and two pointers `p` and `q`. Both pointers were set to point to `x`. by then changed the value of `x` through the pointer `p` and printed the updated value using pointer `q`. This exercise highlighted how multiple pointers can reference the same memory location and how changes through one pointer are reflected when accessed via another.



The screenshot shows the Visual Studio Code interface with the file `pointer.c` open. The code defines a `main` function that initializes an integer `x` to 10, declares two pointers `p` and `q`, and sets both to point to `x`. It then prints the initial value of `x` (10), changes the value of `x` through pointer `p` to 20, and prints the value of `x` through pointer `q` (20). The terminal output shows the successful compilation and execution of the program, confirming the change in value.

```
1 #include <stdio.h>
2
3 int main() {
4     int x = 10;
5     int *p, *q;
6
7     p = &x;
8     q = &x;
9
10    printf("Initial value of x: %d\n", x);
11
12    *p = 20;
13
14    printf("Value of x after changing via p: %d\n", *q);
15
16    return 0;
17 }
```

```
alain@alain-VirtualBox:~/Desktop/week1$ make
gcc -Wall -g -c pointer.c
gcc -Wall -g -o pointer pointer.o
alain@alain-VirtualBox:~/Desktop/week1$ ./pointer
Initial value of x: 10
Value of x after changing via p: 20
alain@alain-VirtualBox:~/Desktop/week1$
```

Figure1: Shows with combined the code and result on pointer function see in terminal.

#### 2. Exercise 2: Value swap

- **Objective:** The objective here was to create a function that swaps the values of two integers using pointers. This approach is necessary when a function needs to return more than one value.
- **Explanation:** The `swap.c` program defined a function `swapValues` that takes two integer pointers as parameters. Within the function, the values pointed to by these pointers are swapped using a temporary variable. This technique is essential for in-place modification of the variables passed to the function. The main function demonstrates this by swapping the values of two integers and printing the results before and after the swap.

The screenshot shows the Visual Studio Code editor with the file `swap.c` open. The code defines a function `swapValues` that takes two integer pointers and swaps their values using a temporary variable. The `main` function initializes two integers, `value1` (35) and `value2` (-97), prints their initial values, calls `swapValues`, and prints their values after the swap. The terminal at the bottom shows the output of the program: "Before swap: value1 = 35, value2 = -97" and "After swap: value1 = -97, value2 = 35".

```

1 #include <stdio.h>
2
3 void swapValues(int *a, int *b) {
4     int temp = *a;
5     *a = *b;
6     *b = temp;
7 }
8
9 int main() {
10     int value1 = 35;
11     int value2 = -97;
12
13     printf("Before swap: value1 = %d, value2 = %d\n", value1, value2);
14
15     swapValues(&value1, &value2);
16
17     printf("After swap: value1 = %d, value2 = %d\n", value1, value2);
18
19     return 0;
20 }

```

Terminal Output:

```

After swap: value1 = -97, value2 = 35
alain@alain-VirtualBox:~/Desktop/week1$

```

- Figure2:shows the code snipped on swap function  
And here is the screenshot thst show the output after testing our code from make file , see.....

This screenshot shows the same Visual Studio Code environment, but the terminal now displays the output of the `make` command. It shows the compilation process using `gcc` and the execution of the resulting binary. The output of the program is the same as in the previous screenshot, confirming the swap function works correctly.

```

alain@alain-VirtualBox:~/Desktop/week1$ make
gcc -Wall -g -c swap.c
gcc -Wall -g -o swap swap.o
alain@alain-VirtualBox:~/Desktop/week1$ ./swap
Before swap: value1 = 35, value2 = -97
After swap: value1 = -97, value2 = 35
alain@alain-VirtualBox:~/Desktop/week1$

```

- Figure3:after making the compilation we see that on this screen we achieve our desirable output.
- 

### 3. Exercise 3: Pointer arithmetic on array

- **Objective:** The task was to traverse an array in reverse order using a pointer without employing a loop counter. This exercise aimed to practice pointer arithmetic.
- **Explanation:** In `array.c`, an array of 5 integers was declared along with a pointer that initially pointed to the last element of the array. A while loop was then used to traverse the array in reverse by decrementing the pointer. Each value was printed as the pointer moved from the end to the beginning of the array. This exercise demonstrated how to manipulate and navigate arrays using pointers.

The screenshot shows the Visual Studio Code editor with the file `array.c` open. The code in the editor is as follows:

```

1 #include <stdio.h>
2
3 int main() {
4
5     int array[5] = {19, 28, 83, 94, 115};
6
7     int *ptr = &array[4];
8
9
10
11     while (ptr >= array) {
12         printf("%d\n", *ptr);
13         ptr--;
14     }
15     return 0;
16 }

```

The terminal output at the bottom shows the compilation and execution of the program:

```

gcc -Wall -g -c array.c
gcc -Wall -g -o array array.o
alain@alain-VirtualBox:~/Desktop/week1$ ./array
115
94
83
28
19
alain@alain-VirtualBox:~/Desktop/week1$

```

- Figure4:shows the both code and result from the terminal.

### 4. Exercise 4: Generic array add

- **Objective:** This exercise involved writing a function to sum a specified number of values in an array of doubles and return the result. This task was meant to showcase basic array manipulation and function return values.
- **Explanation:** In `generic_array.c`, a function `summarize` was defined to take an array of doubles and the number of values to sum. The function iterates through the specified number of elements, calculating the total sum, and returns this sum. The main function provides an example array and calls the `summarize` function, printing the result.

```

1  #include <stdio.h>
2
3
4  double summarize(double *array, int num_values) {
5      double sum = 0.0;
6      for (int i = 0; i < num_values; i++) {
7          sum += array[i];
8      }
9      return sum;
10 }
11
12 int main() {
13
14     double values[5] = {1.0, 2.5, 3.5, 4.0, 5.5};
15
16     int num_values = 3;
17
18
19     double result = summarize(values, num_values);
20     printf("The sum of the first %d values is: %.2f\n", num_values, result);
21
22     return 0;
23 }
24
25

```

- Figure5:show the code the array of doubles and return the result  
So now let us drop down the result to see is this code above is really works .

```

alain@alain-VirtualBox:~/Desktop/week1$ make
make: Nothing to be done for 'all'.
alain@alain-VirtualBox:~/Desktop/week1$ make
gcc -Wall -g -c generic_array.c
gcc -Wall -g -o generic_array generic_array.o
alain@alain-VirtualBox:~/Desktop/week1$ ./generic_array
The sum of the first 3 values is: 7.00
alain@alain-VirtualBox:~/Desktop/week1$

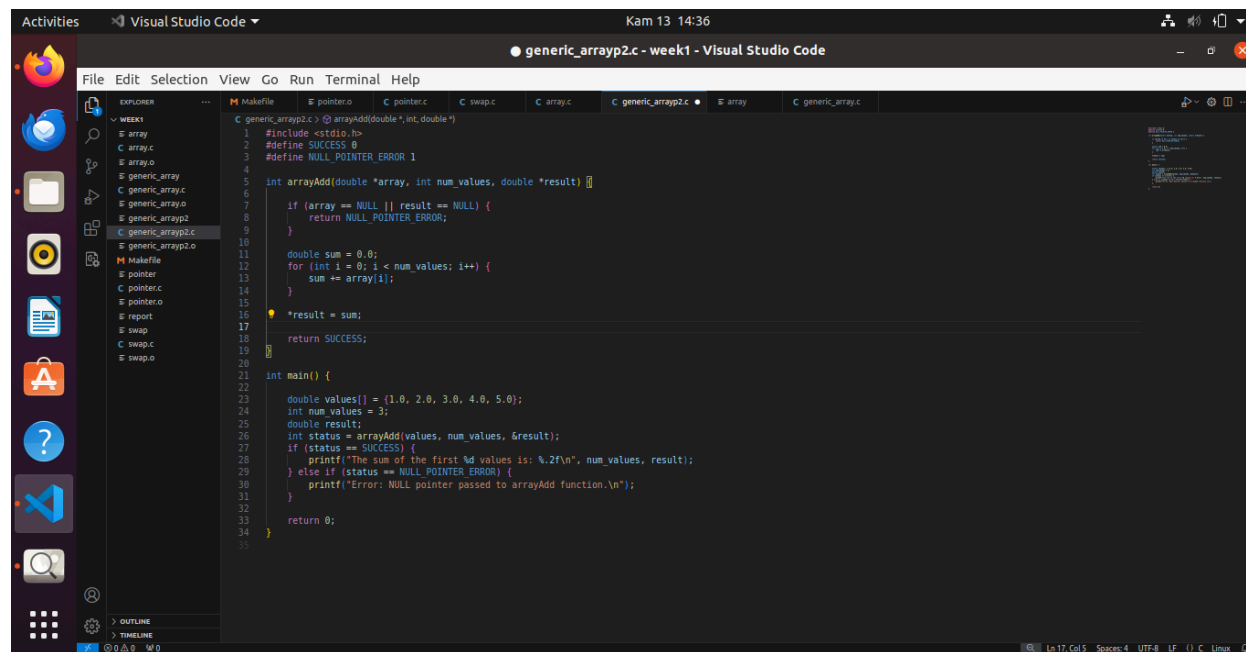
```

- Figure6: shows actually result from array with functionality of pointer.

## 5. Exercise 5: Generic array add (part 2)

- **Objective:** This exercise aimed to modify the previous function to return the result via a parameter and handle errors like NULL pointers. This approach is useful when a function needs to return multiple results or indicate errors.

- **Explanation:** The `generic_arrayp2.c` program introduced an enhanced version of the previous function called `arrayAdd`. This function accepts an additional parameter to store the result and checks for NULL pointers to handle potential errors. If no errors are found, it sums the specified number of values in the array and stores the result in the provided memory location. The main function demonstrates the usage of `arrayAdd`, checking the returned status and printing the result or an error message accordingly.



```
1 #include <stdio.h>
2 #define SUCCESS 0
3 #define NULL_POINTER_ERROR 1
4
5 int arrayAdd(double *array, int num_values, double *result) {
6
7     if (array == NULL || result == NULL) {
8         return NULL_POINTER_ERROR;
9     }
10
11     double sum = 0.0;
12     for (int i = 0; i < num_values; i++) {
13         sum += array[i];
14     }
15
16     *result = sum;
17
18     return SUCCESS;
19 }
20
21 int main() {
22
23     double values[] = {1.0, 2.0, 3.0, 4.0, 5.0};
24     int num_values = 5;
25     double result;
26     int status = arrayAdd(values, num_values, &result);
27     if (status == SUCCESS) {
28         printf("The sum of the first %d values is: %.2f\n", num_values, result);
29     } else if (status == NULL_POINTER_ERROR) {
30         printf("Error: NULL pointer passed to arrayAdd function.\n");
31     }
32
33     return 0;
34 }
```

- Figure 7 shows the enhancement of code from exercise number 4 with passed the result as third parameter.

Let us now see what we can in terminal as the result

```
1 #include <stdio.h>
2 #define SUCCESS 0
3 #define NULL_POINTER_ERROR 1
4
5 int arrayAdd(double *array, int num_values, double *result) {
6     if (array == NULL || result == NULL) {
7         return NULL_POINTER_ERROR;
8     }
9
10    double sum = 0.0;
11    for (int i = 0; i < num_values; i++) {
12        sum += array[i];
13    }
14
15    *result = sum;
16    return SUCCESS;
17 }
18
19
20 int main() {
21    double values[] = {1.0, 2.0, 3.0, 4.0, 5.0};
22    int num_values = 3;
23
24    ...
25 }
```

```
alain@alain-VirtualBox:~/Desktop/week1$ make
gcc -Wall -g -c generic_array2.c
gcc -Wall -g -o generic_array2 generic_array2.o
alain@alain-VirtualBox:~/Desktop/week1$ ./generic_array2
The sum of the first 3 values is: 6.000
alain@alain-VirtualBox:~/Desktop/week1$
```

## Reflection:

- **Challenges Faced:** Some challenges included understanding and correctly implementing pointer arithmetic and ensuring proper handling of edge cases like NULL pointers.
- **Learnings:** Through these exercises, I gained a deeper understanding of how pointers work in C, how to manipulate arrays using pointers, and how to handle multiple return values and errors in functions.
- **Next Steps:** I plan to continue exploring more advanced topics in C programming, such as dynamic memory allocation, linked lists, and file handling, to build a stronger foundation.