ASSIGNMENT 1 REPORT

1.1 Change value via a pointer

Objective:

This question aimed to test the understanding of pointers in C programming and how to modify values indirectly through pointers. The objective was to define an integer variable x and two pointers to integers, where both pointers point to x. Then, changing the integer value via one pointer and reading it back via the other pointer.

Explanation:

In the `main` function, an integer variable **`x`** is declared and initialized with a value of **10**. Two integer pointers, **`ptr1`** and **`ptr2`**, are declared and initialized to hold the **address of `x`**. This means **`ptr1`** and **`ptr2`** are pointing to the memory location where **`x`** is stored.

The initial value of x is printed using `printf`. Then, the value of x is changed from 10 to 20 by dereferencing **`ptr1`** and assigning 20 to it **(`*ptr1 = 20;`).** Since **`ptr2`** also points to the same memory location as **`ptr1`**, the change made through **`ptr1`** affects the value of x as well.

Finally, the new value of x is printed using `ptr2`. Even though the value of x was changed via `ptr1`, `ptr2` also reflects the updated value because both pointers point to the same memory location, which contains the updated value.

```c
#include <stdio.h>

int main() {
    int x = 10; //Define an integer x
    int *ptr1 = &x; // Define a pointer ptr1 pointing to x
    int *ptr2 = &x; // Define another ptr2 pointing to x

    printf("initial value of x: %d\n", x);
    // Change the value of x via ptr1
    *ptr1 = 20;

    // Read the value of x via ptr2
    printf("new value of x via ptr2: %d\n", *ptr2);


    return 0;
}
```

OUTPUT:

```
ingabire@ingabire-lenovo:~/Desktop/assignment1$ ./ChangeValue
initial value of x: 10
new value of x via ptr2: 20
ingabire@ingabire-lenovo:~/Desktop/assignment1$
```

1.2 Value swap

Objective:

This question aimed to demonstrate the usage of pointers to swap values of two variables in C. The objective was to write a function `swapValues` that swaps the values of two integer variables.

Explanation:

I starts by including the standard input-output library to utilize functions like **`printf`** and **`scanf`**. And the core functionality lies in the **`swapValues`** function, which takes two integer pointers as arguments and swaps the values they point to. Inside `main()`, two integer variables **`value1`** and **`value2`** are declared. My program prompts the user to input values for **`value1`** and **`value2`**, then displays these values before the swap. After calling the `swapValues` function with the addresses of **`value1`** and **`value2`**, it prints the swapped values.

ValueSwaap.c > ...

```c
#include <stdio.h>

// Function to swap two integer values
void swapValues(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int value1, value2;

    printf("Enter the first integer: ");
    scanf("%d", &value1); // Allowing user to input value1

    printf("Enter the second integer: ");
    scanf("%d", &value2); // Allowing user to input value2

    printf("Before swap: value1 = %d, value2 = %d\n", value1, value2);

    // Call swapValues function
    swapValues(&value1, &value2);

    printf("After swap: value1 = %d, value2 = %d\n", value1, value2);

    return 0;
}
```

OUTPUT:

```
ingabire@ingabire-lenovo:~/Desktop/assignment1$  ./ValueSwaap
Before swap: value1 = 35, value2 = 12
After swap: value1 = 12, value2 = 35
ingabire@ingabire-lenovo:~/Desktop/assignment1$
```

1.3 Pointer arithmetic on array

Objective:

This question aimed to test understanding of pointer arithmetic and traversing arrays using pointers instead of loop counters. Simply the codes helps to traverse an array of integers in reverse order using a pointer.

Explanation:

I have begin the program by declaring an array of five integers, **`array`**, initialized with the values {10, 20, 30, 40, 50}. A pointer, **`ptr`**, is then declared and initialized to point to the last element of the array, **`array[4]`.**

The core functionality of the program is a **`while`** loop that iterates as long as the pointer **`ptr`** is greater than or equal to the address of the first element of the array. Inside the loop, the value pointed to by **`ptr`** is printed using **`printf`**. After printing the current value, **`ptr`** is decremented using the **`--`** operator to move the pointer to the previous element of the array.

This loop continues until **`ptr`** has traversed all the way to the first element of the array. When **`ptr`** is no longer pointing to an element within the array bounds, the loop exits. Finally, the program returns 0, indicating successful execution.

PointerArthmetic.c > main()

```c
#include <stdio.h>

int main () {
 int array[5] = {10,20,30,40,50};// declare array of 5 integers
 int *ptr = &array[4]; //declare a pointer that points to the last element of the array


 // traverse the array in reverse using the pointer

 while (ptr >= array) {
  printf("%d\n",*ptr);//print the  value pointed to by ptr
   ptr--;  // move the pointer to the previous element

   }

   return 0;
   }

```

OUTPUT:

```
ingabire@ingabire-lenovo:~/Desktop/assignment1$ ./PointerArthmetic
50
40
30
20
10
ingabire@ingabire-lenovo:~/Desktop/assignment1$
```

1.4 Generic array add

Objective:

This question aimed to implement a function that sums up a specific number of values in an array of doubles and returns the result.

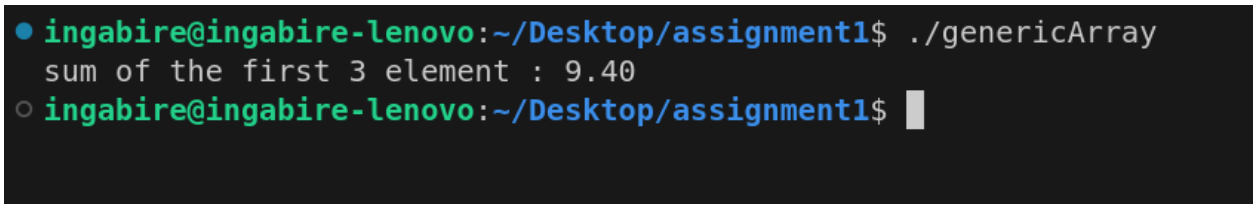
Explanation:

I begins with the inclusion of the standard input-output library via **`#include <stdio.h>`**, enabling the use of functions like **`printf`** for output. The core functionality is encapsulated in the **`sumArray`** function, which takes a pointer to a double array and an integer representing the number of elements to sum. Inside this function, a variable **`sum`** is initialized to **`0.0`** to store the cumulative sum. A **`for`** loop iterates through the array up to the specified number of elements, adding each element to **`sum`**. Once the loop completes, the function returns the total sum.

In the **`main`** function, an array of doubles named **`array`** is declared and initialized with five values: **`1.5, 4.2, 3.7, 8.4,`** and **`2.5`**. An integer **`numberOfElements`** is set to **`3`**, indicating that only the first three elements of the array should be summed. The **`sumArray`** function is called with **`array`** and **`numberOfElements`** as arguments, and its result is stored in the **`result`** variable. This variable holds the sum of the first three elements of the array. The **`printf`** function then outputs the result, formatted to two decimal places for clarity, displaying the message "Sum of the first 3 elements: 6.60". The program concludes with **`main`** returning **`0`**, signifying successful execution.

```c
#include <stdio.h>
// function to sum a specific number of values in an array  and answer will be double
double sumArray(double *array, int numberOfElements) {

double sum = 0.0;

for (int i =0; i < numberOfElements; i++) {
    sum += array[i];
}
return sum;
}

int main() {
    double array[]= {1.5, 4.2, 3.7, 8.4,2.5};

    int numberOfElements = 3;

    double result = sumArray(array,numberOfElements);

    printf("sum of the first %d element : %.2f\n", numberOfElements,result);
}
```

OUTPUT:



1.6 Generic array add (part 2)

Objective:

This question aimed to modify the **`arrayAdd`** function to return the result via a parameter to handle error cases.

This arrayAdd function is the sumArray function in the above question.

Explanation:

It includes necessary error codes, **SUCCESS, ERROR_NULL_POINTER,** and **ERROR_INVALID_NUMBER_OF_ELEMENTS**, defined using the **`#define`** preprocessor directive. The **`arrayAdd`** function is designed to sum elements of a double array and return the result via a pointer, with error checking for null pointers and invalid element counts.

The **`arrayAdd`** function begins by checking if the input array pointer is **NULL**, returning **ERROR_NULL_POINTER** if true, to prevent null pointer dereferencing. It then checks if the number of elements to sum is less than or equal to zero, returning **ERROR_INVALID_NUMBER_OF_ELEMENTS** if true, to ensure valid input. If both checks pass, it initializes a **`sum`** variable to **`0.0`** and iterates over the specified number of elements in the array, accumulating their sum. The resulting sum is stored in the location pointed to by the **`result`** pointer, and the function returns **SUCCESS** to indicate successful execution.

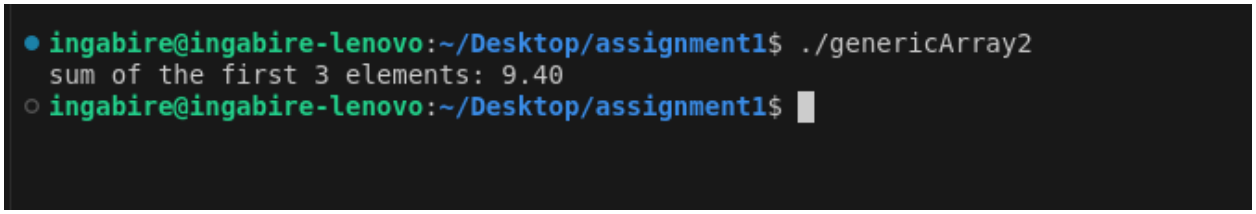
In the **`main`** function, an array of doubles, **`array`**, is declared and initialized with five values: **`1.5, 4.2, 3.7, 8.4,`** and **`2.5`**. An integer **`numberOfElements`** is set to **3**, indicating that only the first three elements should be summed. A **`result`** variable is also declared to store the sum. The **`arrayAdd`** function is called with **`array`, `numberOfElements`**, and the address of **`result`**.

The return status of **`arrayAdd`** is checked using **`if-else`** statements. If the status is **SUCCESS**, the program prints the sum of the first three elements, formatted to two decimal places. If the status is **ERROR_NULL_POINTER**, it prints an error message indicating a null pointer was

provided. If the status is **`ERROR_INVALID_NUMBER_OF_ELEMENTS`**, it prints an error message indicating an invalid number of elements. Finally, the **`main`** function returns **`0`**, indicating successful program termination.

```c
#include <stdio.h>
 #define SUCCESS 0
 #define ERROR_NULL_POINTER -1
 #define ERROR_INVALID_NUMBER_OF_ELEMENTS -2


 //function to sum a specific number of values in an array of doubles
 int arrayAdd(double *array, int numberOfElements, double *result) {
    if(array == NULL) {
        return ERROR_NULL_POINTER;//error if array pointer is null
    }
    if (numberOfElements <= 0) {
        return ERROR_INVALID_NUMBER_OF_ELEMENTS;//error if number of element is invalid
    }

    double sum = 0.0;
    for (int i =0; i < numberOfElements; i++) {
        sum += array[i];
    }
    *result = sum;//store the result in provided pointer
    return SUCCESS;// indicate success
 }


int main(){
    double array[] = {1.5, 4.2, 3.7, 8.4,2.5};
    int numberOfElements =3;
    double result;

    int status = arrayAdd(array, numberOfElements, &result);

    if(status == SUCCESS) {
        printf("sum of the first %d elements: %.2f\n", numberOfElements, result);
    } else if (status == ERROR_NULL_POINTER) {
        printf("error: null pointer provided.\n");
    } else if (status == ERROR_INVALID_NUMBER_OF_ELEMENTS) {
        printf("error: invalid number of element.\n");
    }

    return 0;
    }
```

OUTPUT:


ingabire@ingabire-lenovo:~/Desktop/assignment1$ ./genericArray2
sum of the first 3 elements: 9.40
ingabire@ingabire-lenovo:~/Desktop/assignment1$