**Name:** Fabiola INGABIRE

on 13 jun 2024

Email: [fabiolaingabire02@gmail.com](mailto:fabiolaingabire02@gmail.com)

**ROBOTIC TEAM**

**WEEK1: Assignment**

**1.1: Change value via a pointer**

**Write a program to define an integer x and two pointers to integer.**

**Both pointers should point to the integer x.**

**Change the integer value via one pointer and read it back via the other pointer.**

**Documentation for Code: Variable Pointers**

This document explains the code that demonstrates the concept of variable pointers in C.

**Functionality:**

The code explores how pointers can be used to access and modify the value of a variable indirectly.

**Explanation:**

**1. Variable Declaration:**

- An integer variable `x` is declared and initialized with the value 20.

**2. Pointer Declaration:**

- Two integer pointers, `p` and `d`, are declared. Pointers store memory addresses, not actual values.

**3. Assigning Address to Pointer:**

- The address of the variable `x` is assigned to both pointers `p` and `d` using the `&` (address-of) operator. This means `p` and `d` now point to the same memory location where the value of `x` is stored.

**4. Accessing Value using Pointer:**

- The `printf` statement `printf("The value of pointer p is %d\n",\*p);` retrieves the value stored at the memory location pointed to by `p`. Since `p` points to `x`, it prints the value of `x` (which is 20).

**5. Printing Address:**

- The `printf` statement `printf("The adress of x is %p\n",p);` prints the address stored in the pointer `p`. This address is the same as the memory location where the value of `x` is stored.

**6. Modifying Value through Pointer:**

- The line `\*d=30;` modifies the value stored at the memory location pointed to by `d`. Since `d` also points to `x`, this effectively changes the value of `x` to 30.

**7. Printing Updated Value:**

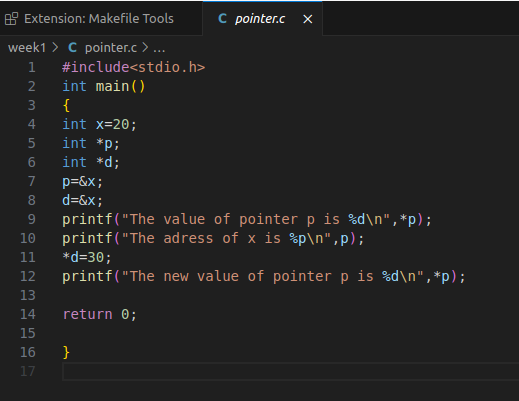
- The final `printf` statement `printf("The new value of pointer p is %d\n",\*p);` again accesses the value through pointer `p`. Since `p` and `d` point to the same memory location where `x` is stored (which has been modified to 30), this prints the new value of `x` (which is now 30).

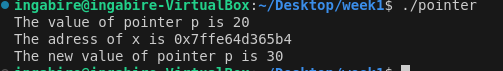
**Key Points:**

- Pointers store memory addresses.

- The `\*` operator deference a pointer, meaning it accesses the value stored at the memory location pointed to by the pointer.

- Modifying the value through a pointer changes the original variable's value.

**OUTPUT:**

****

**1.2 Value swap**

**When you want to return a value from a function, you can simply return that value.**

**What happens when you need to return more than one value? In that case you can use**

**a pointer type parameter.**

**Write a function that can swap two integer values.**

**The function is called swapValues(), the following code snippet explains what it does:**

**int value1 = 35;**

**int value2 = -97;**

**swapValues(….,…….);**

**// now value1 equals -97 and value2 equals 35.**

**Documentation for Code: Swap Values Function**

This document explains the code that demonstrates a function for swapping the values of two integer variables in C.

**Functionality:**

The code defines a function `swapvalues` that takes pointers to two integer variables as arguments and swaps their values.

**Explanation:**

**1. swapvalues` Function:**

- This function is declared with the name `swapvalues`, taking two integer pointers (`int\*a` and `int\*b`) as arguments.

**2. Temporary Variable:**

- Inside the function, an integer variable `temp` is declared.

**3. Swapping Values:**

- The swapping process is achieved using the following steps:

- The value stored at the memory location pointed to by `a` is assigned to `temp`. This effectively stores the original value of `value1` in `temp`.

- The value stored at the memory location pointed to by `b` is assigned to the memory location pointed to by `a`. This copies the value of `value2` into `value1` (indirectly through `a`).

- Finally, the value stored in `temp` (which is the original value of `value1`) is assigned to the memory location pointed to by `b`. This completes the swap by placing the original value of `value1` into `value2`.

**4. Function Call:**

- In the `main` function, two integer variables `value1` and `value2` are initialized with values 35 and -97, respectively.

- The `swapvalues` function is called, passing the addresses of `value1` and `value2` using the `&` (address-of) operator (`swapvalues(&value1,&value2);`). This provides the function with a way to access and modify the original variables.

**5. Printing Before and After Swap:**

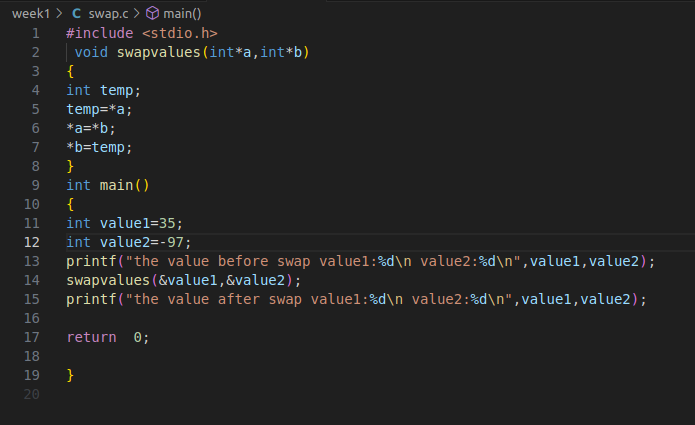
- `printf` statements are used to print the values of `value1` and `value2` before and after the swap function call. This demonstrates that the function successfully swapped the values of the original variables.

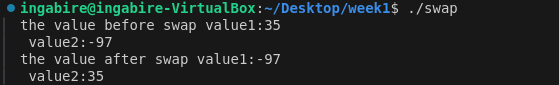
**Key Points:**

- The `swapvalues` function operates on the memory locations of the variables passed to it (using pointers).

- The temporary variable `temp` is used to hold the original value of one variable while the other variable's value is copied.

- Passing addresses using `&` allows the function to modify the original variables.

**OUTPUT:**

****

**1.3 : Pointer arithmetic on array**

**Write a small program that declares an array of 5 integers and declare a pointer that**

**points to the last element of the array. Write a loop that traverses the array via the**

**pointer in reverse direction.**

**This means that a loop such as this is not allowed:**

**for (int i = 4; i >= 0; i--)**

**{**

**printf("%d\n", array[i]);**

**}**

**In other words: you are not allowed to use a loop counter, you can only use the pointer**

**you have.**

**Documentation for Code: Printing Array Elements in Reverse Order**

This document explains the code that demonstrates how to print the elements of an array in reverse order using a pointer and a loop in C.

**Functionality:**

The code iterates through an array of integers in reverse order, accessing and printing each element's value using a pointer.

**Explanation:**

**1.Array Declaration:**

- An integer array `a` of size 5 is declared and initialized with values {12, 30, 40, 50, 60}.

**2. Pointer Initialization:** - An integer pointer `p` is declared and assigned the address of the last element in the array (`a[4]`). This means `p` initially points to the memory location where the value 60 is stored.

**3. `while` Loop:** - A `while` loop is used to iterate through the array elements in reverse order. The loop condition `p >= a` ensures that the loop continues as long as the pointer `p` points to a memory location within the array's boundaries (i.e., not less than the address of the first element `a`).

**4. Printing Element Value:** - Inside the loop:

- The value stored at the memory location pointed to by `p` is dereferenced using `\*p`, and this value is printed using `printf("the Value is:%d\n",\*p);`. This effectively prints the elements in reverse order because `p` starts at the last element and decrements in each iteration.

**5. Decrementing Pointer:**

- After printing the value, the pointer `p` is decremented using `p--`. This moves `p` one memory location backward, pointing it to the next element in the array (towards the beginning).

**6. Loop Termination:**

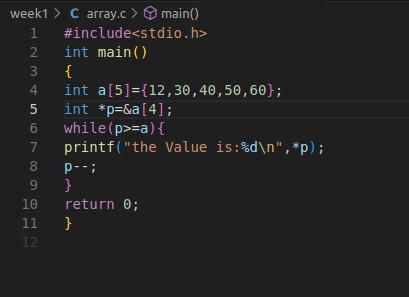
- The loop continues iterating as long as the condition `p >= a` remains true. Once `p` reaches a memory location less than the address of the first element `a`, the loop terminates.

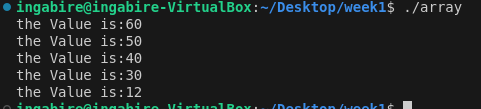
**Key Points:**

- The pointer `p` is used to traverse the array in reverse order by starting at the end and moving backwards.

- The loop condition ensures the pointer stays within the valid memory range of the array.

- Dereferencing the pointer `\*p` accesses the value stored at the memory location it points to.

****

****

**OUTPUT:**

**1.4 Generic array add**

**Write a function that can summarize a specific number of values in an array of**

**doubles and return the result.**

**Documentation for Code: Array Summarization Function**

This document explains the code that demonstrates a function for calculating the sum and count of elements in a double-precision floating-point array in C.

**\*\*Functionality:\*\***

The code defines a function `summarize\_array` that takes a double array and its size as arguments and returns a structure containing the sum of the array elements and the number of elements.

**Explanation:**

**1. `Summary` Structure:**

- A structure named `Summary` is defined with two members:

- `sum`: A double-precision floating-point variable to store the sum of array elements.

- `count`: An integer variable to store the number of elements in the array.

**2. `summarize\_array` Function:**

- This function is declared with the name `summarize\_array`, taking two arguments:

- `arr`: A double array representing the data to be summarized.

- `size`: An integer representing the number of elements in the array.

**3. Initializing `Summary`:**

- Inside the function, a variable of type `Summary` named `summary` is declared.

- The `sum` member of `summary` is initialized to 0.0.

- The `count` member of `summary` is set to the value of `size` (the number of elements in the array).

**4. Calculating Sum:**

- A `for` loop iterates through the elements of the array `arr`.

- Inside the loop:

- The value of the current element `arr[i]` is added to the `sum` member of the `summary` structure using `summary.sum += arr[i]`.

- This accumulates the sum of all elements in the array.

**5. Returning Structure:**

- After the loop completes, the function returns the `summary` structure, which now contains the calculated sum and the original count (number of elements).

**6. `main` Function:**

- The `main` function creates a double array `arr` with sample values.

- It calculates the size of the array using `sizeof(arr) / sizeof(arr[0])`.

- The `summarize\_array` function is called with the `arr` and its size as arguments.

- The returned `summary` structure is stored in the variable `summary`.

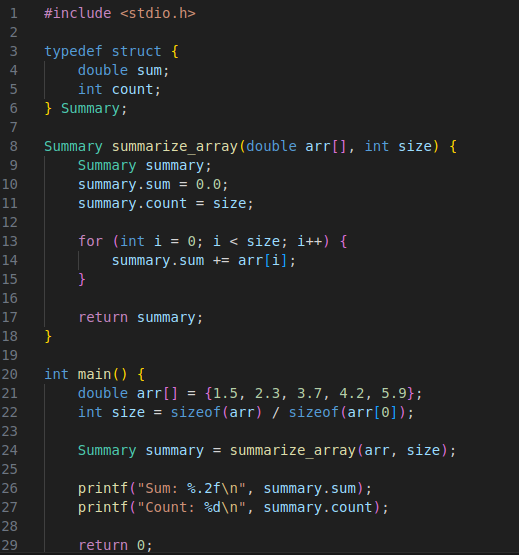
- Finally, the `sum` and `count` members of the `summary` structure are printed using `printf` statements.

**Key Points:**

- The `summarize\_array` function encapsulates the logic for calculating the sum and count.

- The `Summary` structure is a user-defined data type that groups related data (sum and count) together.

- The function returns a structure to provide both calculated values (sum and count) in a single unit.

****

****

**OUTPUT:**

**1.6 Generic array add (part 2)**

**Another way of writing the function in the previous assignment would be by returning**

**the result via a parameter. A reason to do this is if you need to return more than one**

**thing. For example: what would arrayAdd() do if it is called with a NULL pointer? It**

**has no way of telling the caller that an error occurred.**

**Please change the arrayAdd() function according this new specification.**

**Documentation for Code:** Array Summarization Function with Error Handling

This document explains the code that demonstrates a function for calculating the sum and count of elements in a double-precision floating-point array in C, with error handling capabilities.

**Functionality:**

The code defines a function `summarize\_array` that takes a double array, its size, and a pointer to a `Summary` structure as arguments. It calculates the sum and count, stores them in the provided structure, and returns an integer code indicating success or failure.

**Explanation:**

**1. `Summary` Structure (same as previous example):**

- A structure named `Summary` is defined with two members:

- `sum`: A double-precision floating-point variable to store the sum of array elements.

- `count`: An integer variable to store the number of elements in the array.

**2. `summarize\_array` Function:**

- This function is declared with the name `summarize\_array`, taking three arguments:

- `arr`: A double array representing the data to be summarized.

- `size`: An integer representing the number of elements in the array.

- `summary`: A pointer to a `Summary` structure where the results will be stored.

**3. Error Checking:**

- The function starts by checking if the provided array (`arr`) or the `summary` structure pointer (`summary`) is NULL (indicates invalid memory addresses).

- If either is NULL, it returns -1 to signal an error condition.

**4. Initializing `Summary`:**

- Assuming no errors are detected:

- The `sum` member of the `\*summary` structure (accessed using the pointer) is initialized to 0.0.

- The `count` member of the `\*summary` structure is set to the value of `size` (the number of elements in the array).

**5. Calculating Sum:**

- A `for` loop iterates through the elements of the array `arr`.

- Inside the loop:

- The value of the current element `arr[i]` is added to the `sum` member of the `\*summary` structure using `(\*summary).sum += arr[i]`.

- This accumulates the sum of all elements in the array.

**6. Returning Success:**

- After the loop completes, the function returns 0 to indicate successful execution.

**7. `main` Function (similar with error handling):**

- The `main` function creates a double array `arr` with sample values.

- It calculates the size of the array using `sizeof(arr) / sizeof(arr[0])`.

**8. `Summary` Declaration and Function Call:**

- A `Summary` structure variable `summary` is declared.

- The `summarize\_array` function is called with the `arr`, its size, and the address of the `summary` variable (`&summary`) as arguments. This passes the memory location of `summary` to the function.

**9. Error Handling in `main`:**

- The return value (`result`) from the `summarize\_array` function is checked.

- If `result` is 0 (success):

- The calculated `sum` and `count` are printed from the `summary` structure.

- If `result` is not 0 (error):

- An error message is printed to indicate the failure.

**10. Returning from `main`:**

- The `main` function returns 0 to signal successful program termination.

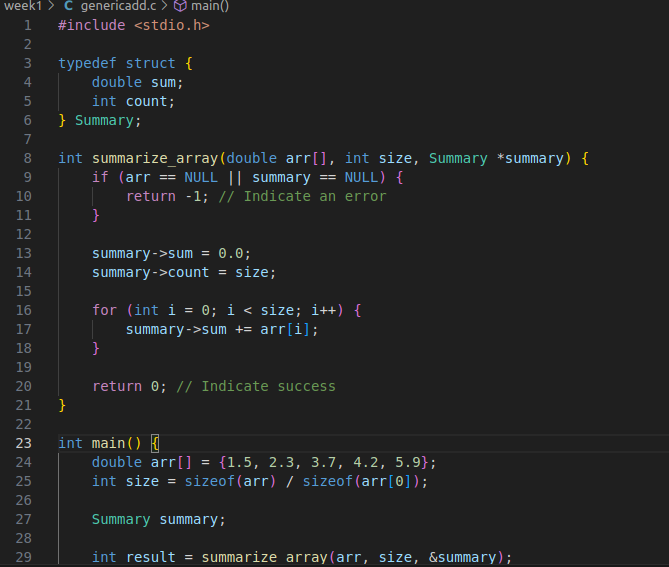
**Key Points:**

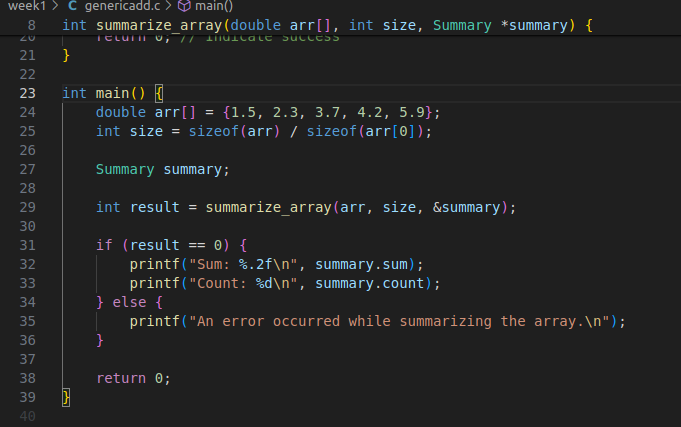
- The `summarize\_array` function now handles potential errors by checking for NULL pointers.

- It returns an error code (-1) to indicate issues and 0 for success.

- The `**main`** function incorporates error handling by checking the return value from **`summarize\_array`** and taking appropriate actions.

- Passing a pointer to the `Summary` structure allows the function to modify the original data structure in **`main`.**

****

**OUTPUT:**

****